

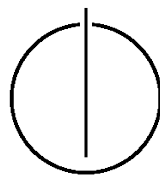
FAKULTÄT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

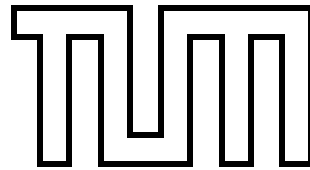
Bachelorarbeit in Wirtschaftsinformatik

# **A Visual Tool for Conflict Resolution in EA Repositories**

Tobias Schrade







FAKULTÄT FÜR INFORMATIK

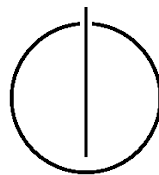
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelorarbeit in Wirtschaftsinformatik

A Visual Tool for Conflict Resolution in EA Repositories

Ein visuelles Tool für die Konfliktlösung in EA  
Repositories

Author: Tobias Schrade  
Supervisor: Prof. Dr. Florian Matthes  
Advisor: Sascha Roth  
Date: December 15, 2013





I assure the single handed composition of this bachelor thesis only supported by declared resources.

Ich versichere, dass ich diese Bachelorarbeit selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 15.12.2013

Tobias Schrade



---

## Abstract

During the merge of different states, of enterprise architecture (EA) models, it is almost inevitable that conflicts occur. These conflicts often can not be solved automatically and need further input from multiple EA stakeholders. In this vein, it is necessary to provide users with context information such that they grasp an understanding of both, local and global implications of the conflict. Commonly, the conflicts are shown in a list, making solving them a difficult and time-consuming task, because the context information of the conflict is lost. Visualizing these conflicts in a model view would enhance this task by adding exactly this context information.

In this thesis, we provide the design and implementation for an interactive conflict management dashboard. Its goals are to facilitate the conflict resolution. We add relevant context information and collaboration facilities to empower end-users to solve model conflicts. In order to cope with complexity of an EA model, i.e. over one thousand instances of more than 50 different types, we present concepts and implementation of a filter incorporated in the interactive conflict management dashboard.





# Contents

<b>Abstract</b>	<b>vii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Model Conflicts . . . . .	1
1.2 Outline . . . . .	3
<b>2 Foundations</b>	<b>5</b>
2.1 Conceptual Framework for Interactive Visualizations . . . . .	5
2.2 NodeJS Server . . . . .	5
2.3 TogetherJS Application . . . . .	5
<b>3 Related Work</b>	<b>9</b>
3.1 View Type . . . . .	9
3.2 Conflict Highlighting . . . . .	11
3.2.1 Conflict Icons . . . . .	11
3.3 Conflict Meta Information . . . . .	12
3.4 Visible Versions . . . . .	12
3.5 Resolve Multiple Conflicts Support . . . . .	12
3.6 Helicopter View . . . . .	12
3.7 Concept Matrix . . . . .	13
<b>4 Possible Use-Cases for a Conflict Resolution Visualization</b>	<b>17</b>
4.1 Resolve Multiple Conflicts . . . . .	17
4.2 Resolve One Conflict . . . . .	17
4.3 Solve Conflict Collaboratively . . . . .	17
4.4 Forward Conflict . . . . .	19
4.5 Postpone Conflict . . . . .	19
4.6 Create Conflict . . . . .	19
4.7 Use Case Matrix . . . . .	19
<b>5 Design of the Prototype Visualization</b>	<b>21</b>
5.1 Dashboard . . . . .	21
5.2 Instance Overview Layer . . . . .	23
5.3 Instance Detail Layer . . . . .	24
5.4 Info Boxes . . . . .	26
5.5 Filter . . . . .	27
<b>6 Implementation Details about the Prototype Visualization</b>	<b>29</b>
6.1 Conflict Task Interface . . . . .	29
6.2 Visualization Framework . . . . .	31
6.3 Implemented Framework Extensions . . . . .	33
6.3.1 Stateful Update . . . . .	33
6.3.2 Compensable Interaction . . . . .	36
6.3.3 Broadcasting with NodeJS . . . . .	37
6.3.4 Solve Conflict . . . . .	39

6.3.5	Learning and Batch-Solving . . . . .	41
6.4	Visualization Implementation . . . . .	43
6.4.1	Implementation Data-Structure . . . . .	43
6.4.2	Implementation Data-Binding . . . . .	45
6.4.3	Viewpoint Generation . . . . .	46
6.4.4	Instance- and Type-Filter . . . . .	48
<b>7</b>	<b>Conclusion and Outlook</b>	<b>53</b>
7.1	Summary of the Work . . . . .	53
7.2	Critical Reflection and Outlook . . . . .	54
	<b>Appendix</b>	<b>55</b>
	<b>Bibliography</b>	<b>55</b>

# List of Figures

1.1	Update Update Conflict Time-line	2
1.2	Delete Update Conflict Time-line	2
1.3	Create Update Conflict Time-line	3
2.1	Conceptual Visualization Framework	6
2.2	Google Trends for NodeJS	6
3.1	Conflicts in Model View Example	10
3.2	Conflicts in Tree View Example	10
3.3	Merge in KDiff3 Example	11
4.1	Conflict Visualization Use-Cases	18
5.1	Dashboard Mock Up	22
5.2	Instance Overview Layer Mock Up with Hover	23
5.3	Instance Overview Layer Mock Up with Info Box	24
5.4	Instance Detail Layer Mock Up	25
5.5	Instance Detail Layer Mock Up with Info Box	25
5.6	Example Info Boxes	26
5.7	Example iTunes Smart Playlist Filter	27
5.8	Example Type and Instance Filter	28
6.1	Conflict Task Interface Class Diagram	30
6.2	Visualization Framework Class Diagram	32
6.3	Problem of the Update Function	34
6.4	Solution to the Problem of the Update Function	34
6.5	Update Function Class Diagram	35
6.6	Update Function Sequence Diagram	35
6.7	Compensable Interaction Example Object Diagram	36
6.8	Compensable Interaction Class Diagram	37
6.9	Broadcasting Functions JavaScript Code	37
6.10	Configuration Broadcast Sequence Diagram	38
6.11	Visualization Broadcast Sequence Diagram	39
6.12	Solve Conflict Interaction JavaScript Code	40
6.13	Solve Conflict Sequence Diagram	41
6.14	Learning Hash Maps Structure Diagram	43
6.15	Data-structure of the Visualization Implementation	44
6.16	Create SchemaNodes Pseudo Code	46
6.17	Instance Filter Class Diagram	49
6.18	Instance Filter Java Implementation	49
6.19	Instance Filter Flow Diagram	51



# List of Tables

3.1	Different Conflict Icons Examples . . . . .	11
3.2	Related Work Concept Matrix Part One . . . . .	14
3.3	Related Work Concept Matrix Part Two . . . . .	15
4.1	Interactive Visualization Use Case Matrix . . . . .	19
7.1	Use Case Analysis Matrix . . . . .	54



# 1 Introduction

In an ever-changing world enterprises are forced to adapt and change not only their business processes but also the infrastructure behind these processes on a frequent basis. These changes require exact planning which is a task of the enterprise architecture (EA) management. Fields of EA management are not only knowing the as is state but also designing the planned state and performing the transformation. A difficult part is the transformation and merge of the as is state and the planned state. It is inevitable that conflicts, which cannot be solved automatically, occur during the merge. In this case further input from the enterprise architects is required. The solution of the conflicts can become an error prone and time consuming task, as there is not appropriate tool support as of now. The current state of the art is, that the conflicts are just presented in a list. This representation has many downsides, like missing context information and doesn't meet the requirements, defined in [RHM<sup>+</sup>13]. These requirements are for instance a function to solve multiple conflicts at once and that the mechanisms to solve a conflict must be intuitive for both, data owners and EA stakeholders. [FAB<sup>+</sup>11] also states that this conflict resolution must provide functions to solve the conflicts collaboratively, involving the data owners and EA stakeholders. Currently there is no application meeting all these requirements. Motivated from these requirements we formed the following research question:

Q1: How to provide (interactive) visual means to communicate and resolve model conflicts?

This central research question is further divided into two subquestions, finally leading to a prototype implementation of a visualization. The two subquestions are:

Q1.1: Which (EA) visualizations are scalable for large ( $\leq 1K$  Objects) EA models?

This means, that we will be looking for visualizations, can handle real live data of a company. Due to the fact that a company has a lot of different applications and processes, their models are become big with easily over one thousand objects. When you have so many different objects, models often get unclear and confusing and the conflict resolution wont be intuitive anymore. In this work we are looking for functionalities, to decrease the number of objects displayed and only visualizing the objects and details the user currently needs to see.

Q1.2: How can a visualization support collaborative solution of model conflicts?

This question mainly addresses the collaboration part, mentioned by [FAB<sup>+</sup>11]. As collaboration is very important throughout the process of solving conflicts, we will be looking for solutions to this subquestion. Collaboration will also be an important feature of our prototype.

## 1.1 Model Conflicts

This section will introduce the conflicts, which could occur during the merge of two or more models. As there are many different combinations of conflicts, we will only explain the three most important ones in detail.

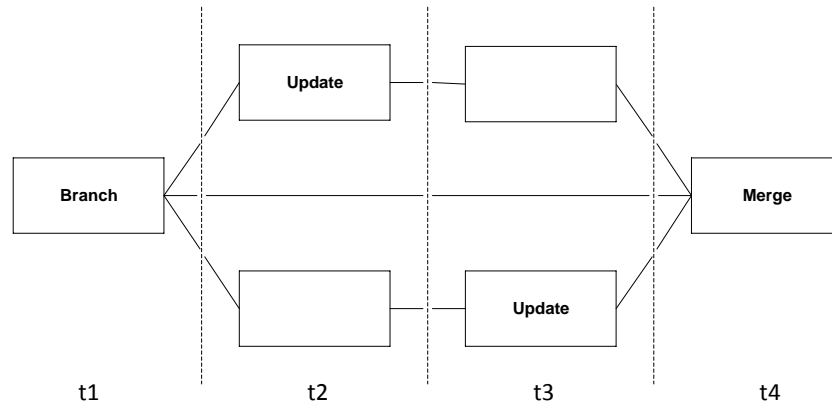


Figure 1.1: Update Update Conflict Time-line

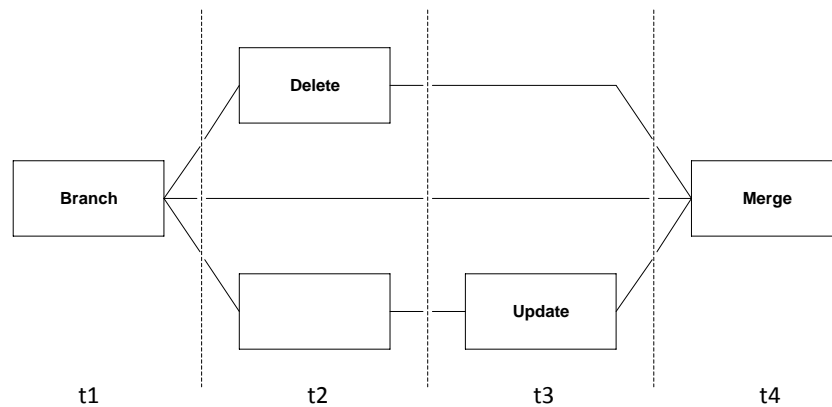


Figure 1.2: Delete Update Conflict Time-line

### Update Update Conflict

The update update conflict is the most often occurring of all conflicts. Figure 1.1 shows the time-line which actions need to happen to generate an update update conflict. At t1, the initial object gets branched, this is the so called base version of the object. At point t2, one version gets updated, for example its name gets changed and at t3 the name of the other version gets updated as-well. During the merge in t4, the algorithm now can not decide, whether version one or two is correct. To solve this conflict, further input is required.

### Delete Update Conflict

A Delete Update Conflict is a conflict, which occurs when a instance is deleted in one branch and updated in another. The time-line leading to this conflict is shown in Figure 1.2. T1 again represents the time, where the branches are created. At t2 the object gets deleted in the first branch and at t3 the same objects gets updated. As it is not possible to merge these two operations in t4, because you can not update a deleted object, a conflict was created.



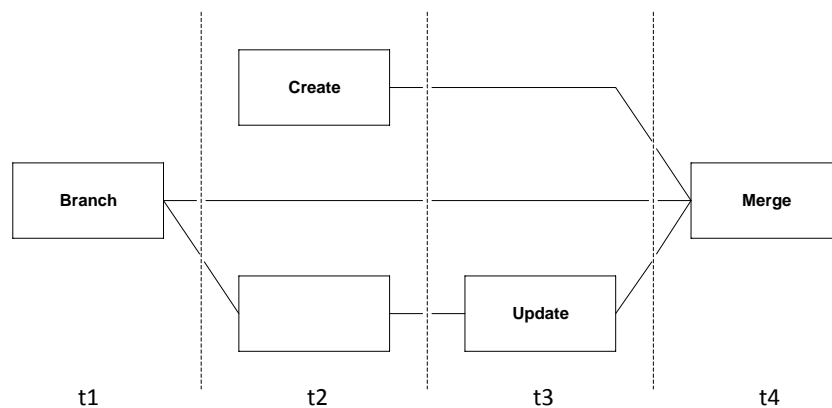


Figure 1.3: Create Update Conflict Time-line

### Create Update Conflict

The third and last conflict described is a create update conflict. This conflict is not as common as the other two, but still an important conflict for model merges. Its time-line is shown in Figure 1.3. T1 is again the point of the branch. At t2 a new object is created, for example with the name "Application 4". In t3, the name of this object gets changed to "Application 4" as-well. During the merge of these two branches, the algorithm can not decided, which Application 4 is the correct one and therefore has to leave this open for the user to solve it manually. A create update conflict was created.

## 1.2 Outline

The outline of this work is as followed. It is divided up into 6 chapters, all aiming towards the prototype implementation to give an answer to the central research question, presented above.

Chapter one is about the motivation behind the work and giving an introduction to model conflicts.

After this, chapter two gives information about the foundations of this work. On the one hand the foundations about interactive visualizations in section 2.1 and on the other hand an introduction to NodeJS and TogetherJS in sections 2.2 and 2.3.

Chapter three presents the results of a literature and application study about related work.

The fourth chapter then presents all use-cases which might be possible with an interactive tool for enterprise architecture visualization. At the end of the chapter an overview of the use-cases implemented in the prototype and the chapter, where the implementation is further described, is given.

Chapter five then presents the mock-ups of the prototype and the functionality they provide. Besides the three main layers of our visualization, the layout of different info-boxes, which are the main tool to solve a conflict and the way the filter looks like are described.

Chapter six is the main part of the work and describes different implementation details, starting with the interface to the conflict tasks in section 6.1 and an introduction to the used graphical framework in 6.2, it also explains the implemented framework extensions to add functionality(6.3) and the actual implemented Java code for the visualization itself in section 6.4.

To end the work there is a summary and a critical analysis written in chapter seven.



## 2 Foundations

This chapter introduces the foundations for our work, including a framework for interactive visualizations and used technologies like NodeJS server.

### 2.1 Conceptual Framework for Interactive Visualizations

In this section we will introduce the framework behind our implementation and give some background information about it. The basis for our mode is presented in the paper "Towards a Conceptual Framework for Interactive Enterprise Architecture Management Visualizations" by Schaub et al. [SMR12]. The paper itself introduces a framework, outlines the requirements for an interactive visualization of an enterprise architecture model and applies the first two points in a prototype implementation. Figure 2.1 is based on a illustration from this paper and represents a part of the framework introduced by the authors.

This framework is adaptable to many different viewpoints which are determined by the view model, the visualization model, the view interaction model and the visual interaction model. For our visual tool for conflict resolution, we have a set viewpoint with defined possible interactions(visual interaction model) and symbols (visualization model).

The data model is the source of all data available and the data interaction model restricts access to this data depending on the role the user has. This is both already implemented in the platform we used for our visualization. The information model represents the information of how the data in the data model is structured and the interaction model describes all possible interactions on this data.

The symbolic model and the symbolic interaction model together are the actual visualization with the symbolic model representing the symbols, like rectangles or images and the symbolic interaction model representing the possible interactions on this very symbols. This interactive view is generated with the help of the data saved in the viewpoint and the data stored in the data model with the access rights from the data interaction model.

Further information on the model is given by Schaub et al. in their paper [SMR12].

### 2.2 NodeJS Server

NodeJS is a server which is based on JavaScript and it supports web-sockets via JavaScript natively. There is also an api called socket.io[Rau], which provides all the needed functionality for a fluent server-client communication for a Java implementation. According to Google Trends, NodeJS is rated as breakthrough and the latest state of the art[Incb](cf. Figure ??). Due to these facts we choose NodeJS as our broadcasting server. The implementation and further details about NodeJS can be found in section 6.3.3.

### 2.3 TogetherJS Application

TogetherJS [Lab] is a tool also implemented in the visualization, to enhance the broadcasting function.It embeds a chat client into the visualization and by connecting to the TogetherJS server you can chat, also with voice and video (if your browser supports it), with the colleagues about the

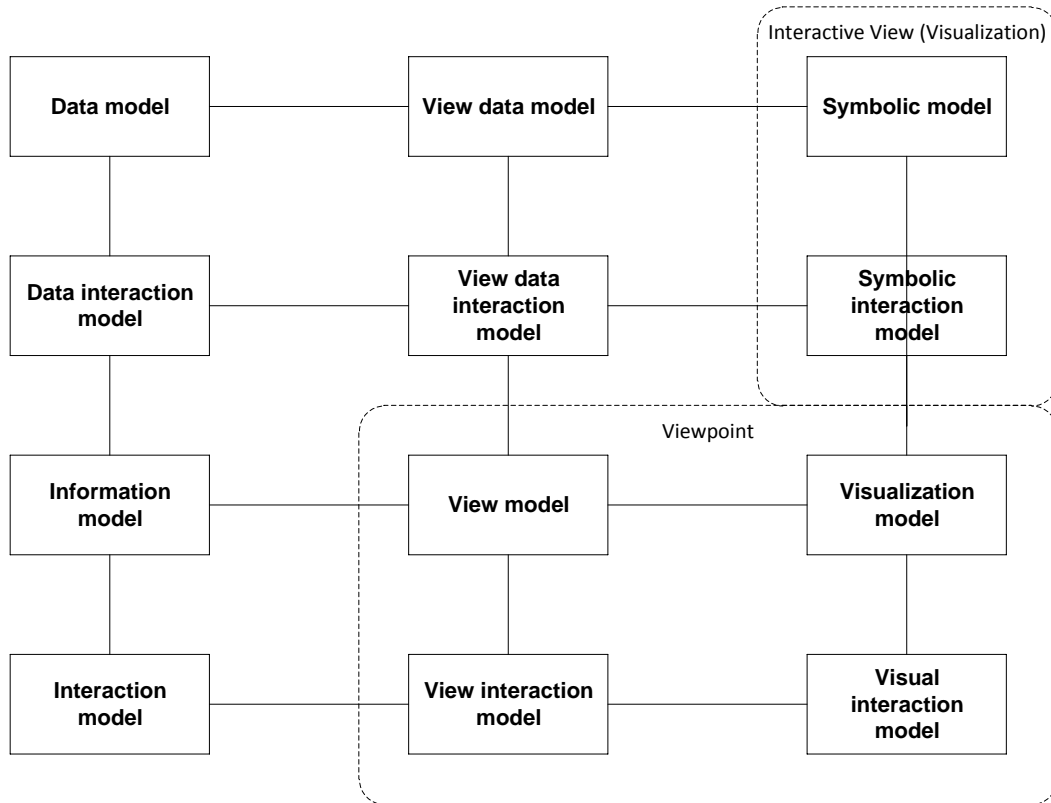


Figure 2.1: Conceptual Visualization Framework based on [SMR12]

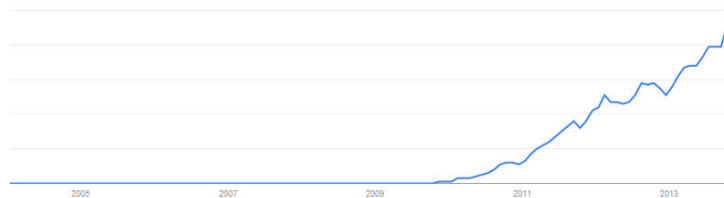


Figure 2.2: Google Trends for NodeJS

conflicts. This enables discussion about the conflicts without a platform change and speeds up the process of conflict-solving. In our visualization, we use the server provided by the TogetherJS platform, as this is sufficient for research purposes.



## 3 Related Work

In this chapter, we present the findings of our literature and application study in the field of merge conflict visualizations. First we tried to only analyze model merge specific programs but, due to the fact that we had too less outcome, we included text and code merge programs in the research as-well. This ensures a more general approach on merge conflict visualizations and many aspects of text and code merge can be applied on a model merge visualization, too.

We analyzed a total of 10 different programs and articles which deal with different kind of merges and visualizations of the merge conflicts. Namely these are:

- Model Merge
  - ConflictVisaulization for Evolving UML Models [BSWK12]
  - EMF Compare [Fou]
  - Toad Data Modeler [Inca]
  - SiLift [KKOS12]
  - VisualParadigm for UML [Par]
  - Eclipse Model Repository [UL]
  
- Text Merge
  - KDiff3 [Eib]
  - MS Outlook [Mic]
  - P4Merge [Per]
  - DiffMerge [Sou]

In the following sections we will introduce the different aspects we analyzed and present the outcome of the research.

### 3.1 View Type

There are three main view types, model view, tree view and text view, whereas text view is used by all code and text merge applications and the model merge applications use either model or tree view. Brosch et al. [BSWK12] states, that for model merge you should use the model view, because this ensures, that the modeler is still in the environment he is used to and therefore he is probably able to resolve the conflicts more efficiently and with a better rate of success than in a different view. This statement also fosters the requirements to usability defined by [RHM<sup>+</sup>13]. The obvious downside of the model view is, that it can become confusing when there are a lot of conflicts in a big model. As you can see in Figure 3.1. This is a reason why other programs like Toad Data Modeler [Inca] use a tree view of the conflicts, but as you can see these trees also get really big and difficult to handle for bigger models (Figure 3.2).

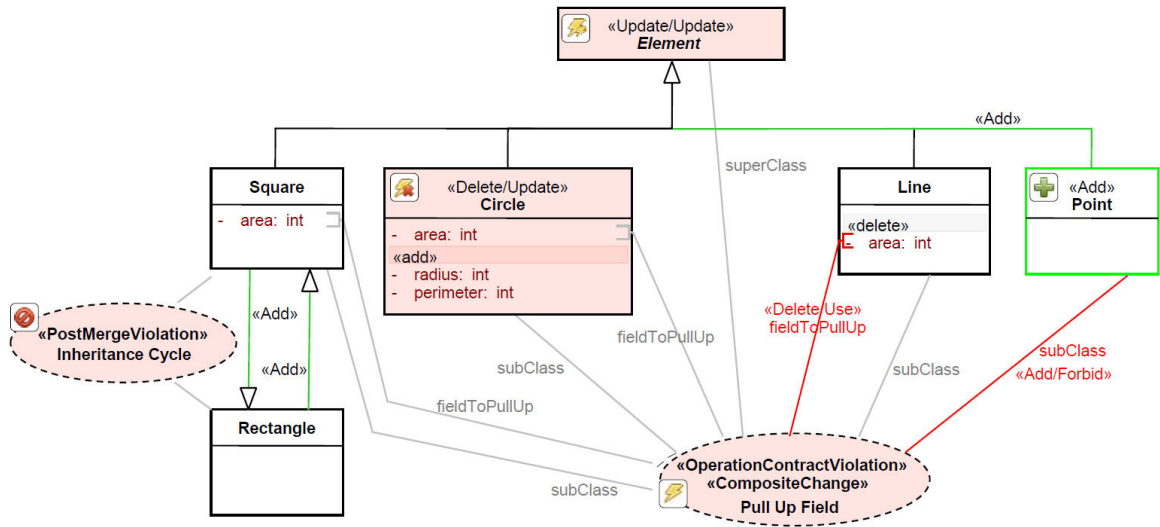


Figure 3.1: Conflicts in Model View Example[KKL<sup>+</sup>10]

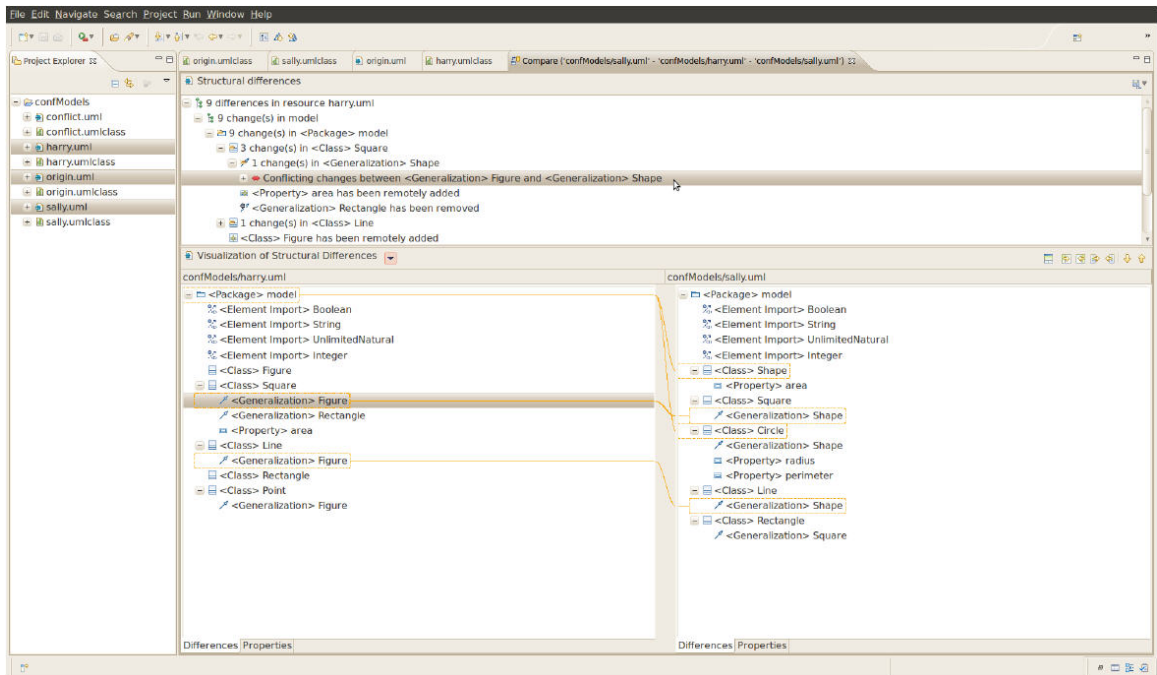


Figure 3.2: Conflicts in Tree View Example[KKL<sup>+</sup>10]



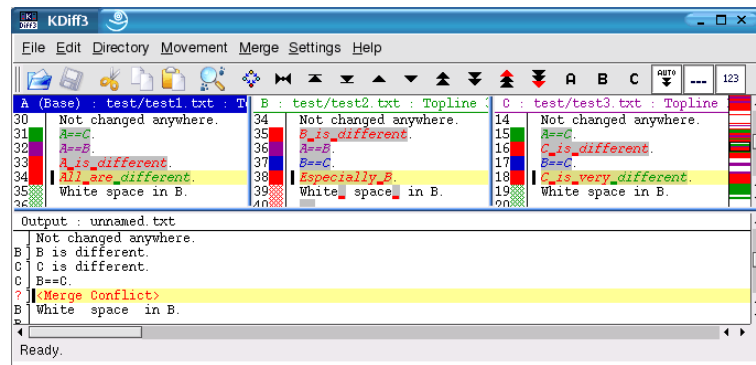


Figure 3.3: Merge in KDiff3 Example[Eib]

Table 3.1: Different Conflict Icons Examples

Article / Application	Icons		
	Update Update Conflict	Update Delete Conflict	Create
Visual Paradigm			
Conflict Visualization for Evolving UML Models			
Toad Data Modeler			
EMF Compare	Blue border around attribute		

## 3.2 Conflict Highlighting

All of the analyzed products use some kind of highlighting, where as color coding (marks a type of conflict in the same color all the time) and specific icons for every conflict are the two main kinds of highlighting used. Where most programs use color coding some use icons in addition to it, too. For color coding there are two different approaches identifiable. On the one hand the color is chosen according to the type of the conflict as used by Brosch et al. [BSWK12] and on the other hand the color chosen according to the branches which are in conflict and the ones which are not, used by KDiff3 [Eib].

The type of conflict based coding often uses green for create (conflicts), blue or red for update conflicts and red for delete conflicts. An example therefore is again Figure 3.1.

Figure 3.3 displays an example merge in KDiff3. As you can see here, green indicates that version A and C are the same, purple indicates A and B and blue B and C. Red stands for the version, which is in a conflict with every other version. It stands out that the branch based color coding is only used by code and text merges and not for model merges.

### 3.2.1 Conflict Icons

Conflict icons are a kind of highlighting which is used by many model programs ([BSWK12], [Fou], [Inca], [Par]) to support the "normal" color coding described above. The specific icons, these program use are shown in table 3.1 (Icons copied from the different programs). As you can see the icon colors used in the application Toad Data Modeler correlate with the colors used for the normal color coding, this makes it easier to classify the different conflicts even if the user does not know either what the symbol or the color means. Another positive aspect about icons is, that you have more freedom in picking symbols than picking colors for highlighting due to the limitations of clearly distinguishable colors available. Therefore you are able to design icons which clearly mark a specific conflict and the user is most likely able to recognize the type of the conflict faster.

### 3.3 Conflict Meta Information

Conflict meta information means any additional information about the conflict like the users, who committed the conflicting changes and the according timestamps. Two out of the ten chosen programs [BSWK12] and [Eib] provide an easy and direct access to conflict meta information. The other platforms either just speak of "Base" "Local" "Server" versions without further meta information, the user has to look it up somewhere else (e.g. [Per]) or do not include real versioning like [UL] but just provide a tool to compare two documents.

In terms of collaboration it is inevitable to provide processed meta information about the conflicts [BSW<sup>+</sup>09]. [BSWK12] safes a link to the corresponding editors for every conflict, so the user has an easy access to this important information.

### 3.4 Visible Versions

In this section, we analyzed which versions/branches are shown in the merge and conflict resolution view of the different programs. Is it either the local version, the server side version, the base version or a combination of them. In the end we also checked, if there is a preview of the merged version. All the text-merge applications (except MS Outlook) show all versions at the same time, including the merge preview. They all have the same basic structure of the merge view. The three merging versions in one row side by side and the preview on the bottom, as shown in the earlier example of KDiff3 in Fig. 3.3. The more interesting question for us was, how do the model merge applications handle the different versions and what do they show in case of a conflict. In this field every application has his own solution. Outstanding in this section are Brosch et al. [BSWK12] and Visual Paradigm for UML [Par]. As mentioned earlier, they show the conflicts in the model view, and they include all changes directly and mark the conflicting ones with color coding, icons and annotations (cf. Fig. 3.1). Visual Paradigm for UML during the merge process first shows all conflicts in a tree view of the diagram. By selecting one conflict, it is shown in a small model view. In a separate box the user can find the local, server and base version and he can select one of them to solve the conflict. An advantage of this conflict presentation is, that even when there are lots of conflicts at the same time the user is able to filter them easily via the tree view and then concentrate on a single conflict, but the downside of this is, that you don't see the context in which the specific conflict is directly.

### 3.5 Resolve Multiple Conflicts Support

In this part, we checked, if the tool provides a feature to solve multiple conflicts in one batch, for instance by selecting a branch as the correct version for all conflicts or solve all conflicts by always selecting the latest change, no matter in what branch they occurred. This is a main requirement to a visualization to resolve merge conflicts in EA models, according to [FAB<sup>+</sup>11]. 60% of all analyzed tools provide such a feature, but with small differences between them. For the text merge tools it is usually a button for every branch and by pressing one of these buttons you solve all conflicts accordingly to the selected branch. None of the tools provides a function to batch-solve with other options than selecting the same branch all the time.

### 3.6 Helicopter View

A helicopter view is a small overview of the whole model or text, preferably with the conflicts marked for faster localization and solving. None of the analyzed model merge tools provide such a function, but all text merge programs (except MS Office for the merge of contacts) do. For text it is

not really a helicopter view, but just a small color indication besides the scrollbar with the conflicts marked in the colors according to the color coding of the given program.

### **3.7 Concept Matrix**

On the following two pages show all findings of our research condensed in two tables (Table 3.2 and 3.3). In there are all the aspects mentioned in this chapter and some additional concepts, which are self-explanatory.

Table 3.2: Related Work Concept Matrix Part One

Model Merges:	View Type	Conflict Highlighting						Conflict Meta Information	Specific Icons for every Conflict Type	Backup Copy/Change Log
		C	R	U	D	M	U			
Conflict Visualization for Evolving UML Models	model	green	-	red	red	-	-	+	-	
EMF Compare	model	green	-	blue	red	-	-	+	-	
Toad Data Modeler	tree	green	-	blue	red	-	-	+	+	
SiLift	model,tree	red	-	red	red	-	-	-	-	
Visual Paradigm for UML	model,tree	-	-	-	-	-	-	+	-	
Eclipse Model Repository	tree	green	-	blue	red	-	-	-	-	
Other mergers:										
KDiff3	text			Version Coloring	Version Coloring		User,Time	-	-	
MS Outlook	text	-	-	so	-	-		-	+	
P4Merge	text		Version Coloring		so			-	-	
DiffMerge	text		Version Coloring	Version Coloring				-	-	

Table 3.3: Related Work Concept Matrix Part Two

	Version Shown at a time	Merged Version Visible	Batch accept one version	Total No. of conflicts	Overview / Helicopter View	Undo Button
Model Merges:						
Conflict Visualization for Evolving UML Models	BV,B1,B2	+	-	-	-	+
EMF Compare	B1,B2	-	+	-	-	-
Toad Data Modeler	B1,B2	-	+	+	-	+
SiLift	B1,B2	-	-	-	-	-
Visual Paradigm for UML	BV,B1,B2	-	-	-	-	+
Eclipse Model Repository	BV,B1,B2	-	+	-	-	-
Other mergers:						
KDiff3	BV,B1,B2	+	+	-	+	-
MS Outlook	BV,B1	+	+	-	-	-
P4Merge	BV,B1,B2	+	-	+	+	+
DiffMerge	BV,B1,B2	+	+	-	+	+



## 4 Possible Use-Cases for a Conflict Resolution Visualization

In this chapter we will describe the use-cases, our visualization aims to offer. An overview over all possible use-cases is shown in Figure 4.1. We will explain the use-cases from top to bottom in the diagram, explaining the "includes" cases in the section of the one, where they are included. At the end of this chapter we provide an overview table of all uses cases and the degree to which they can be fulfilled with the help of the visualization.

### 4.1 Resolve Multiple Conflicts

Resolve Multiple Conflicts is the function to resolve more than one conflict with only one click. Our implementation provides a learning algorithm and on the basis of its findings it proposes a solution strategy. When the user accepts this strategy, it will solve all remaining conflicts at once. Further information about the learning algorithm is written in section 6.3.5. Resolving multiple conflicts of course includes resolving a single conflict. This is explained in the following part.

### 4.2 Resolve One Conflict

Resolving a conflict includes many sub use-cases, as shown by the diagram. Resolving a conflict means selecting the right solution and apply it to the conflict with the outcome that the conflict is solved. We distinguish three types of conflicts. Schema, instance and schema/instance conflicts. The schema conflicts can be solved directly on the dashboard as stated in section 5.1. For solving a instance conflict, the user has to go one step deeper into the visualization. Instance conflicts and their possible solutions are showed in the Instance Overview Layer and the Instance Detail Layer, as shown in the sections 5.2 and 5.3. As a schema/instance conflict is both, a conflict on schema and instance level, the user can solve it like an schema conflict and like an instance conflict. The details about the implementation of solving a conflict are given in section 6.3.4

The conflict summary is an overview over all conflicts, this summary is given by the dashboard and especially by the status bars, added to the different types, because they show how many instances of this type do have a conflict.

If the user wants to drill down a conflict the instance detail layer(section 5.3) provides the needed functionality. The user is able to see the associated instances and therefore additional information, which supports the process of solving a conflict.

### 4.3 Solve Conflict Collaboratively

The possibility to solve a conflict collaboratively is of high importance for the visualization, as it highly increases the quality of the solutions and prevents wrong decisions, due to consultation of colleagues. The two main functions hereby are on the one hand broadcasting, so every user can see the exact same visualization and on the other hand discussing the visualization via an embedded chat-client. Broadcasting is further explained in section 6.3.3 and the chat client in section 2.3. The use-case, which is currently not able with the help of the visualization is to comment a conflict.

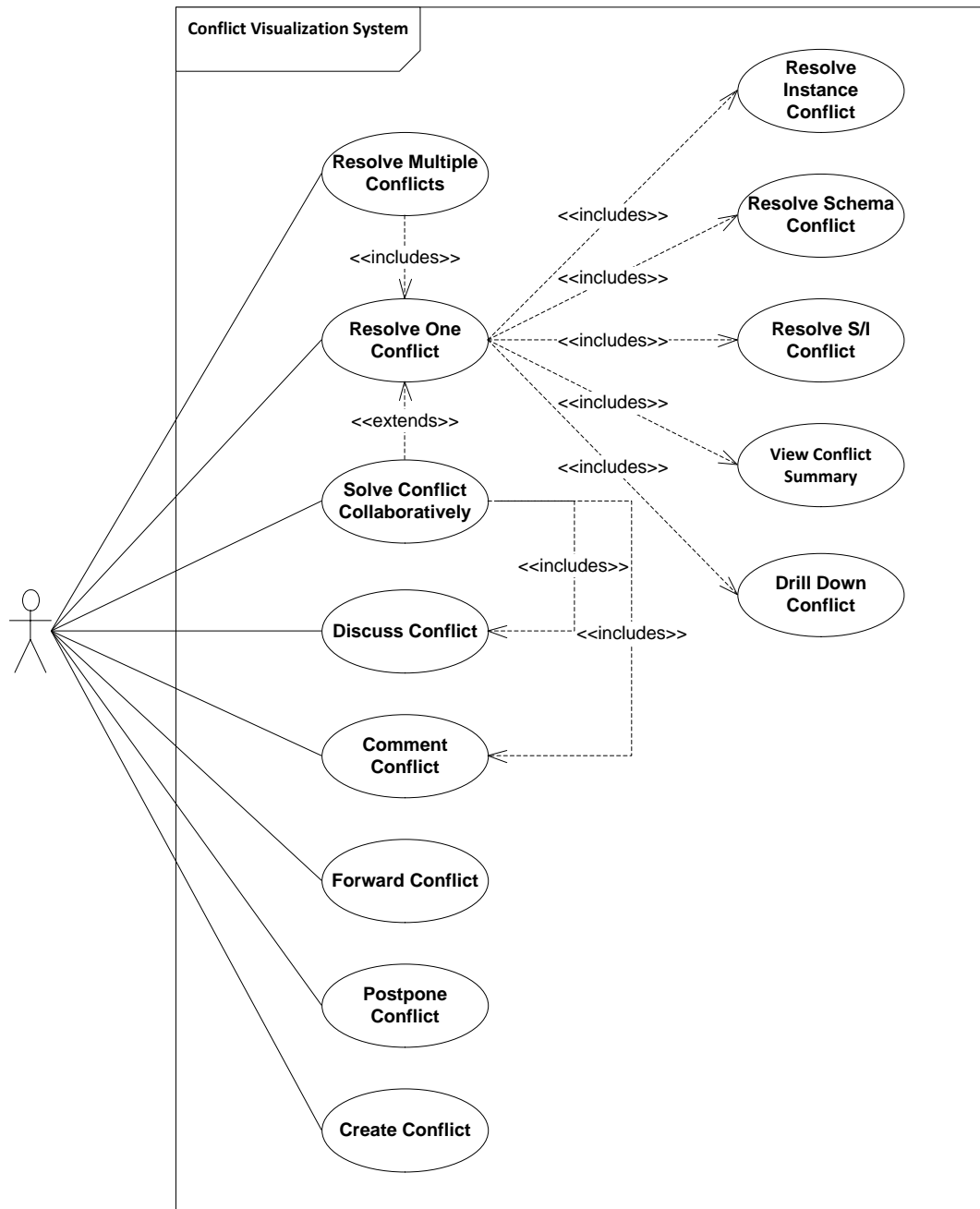


Figure 4.1: Conflict Visualization Use-Cases



Table 4.1: Interactive Visualization Use Case Matrix

Use-Case	Further Explanation in Chapter
Resolve Multiple Conflicts	6.3.5
Resolve One Conflict	6.3.4
Resolve Instance Conflict	5.2, 5.3
Resolve Schema Conflict	5.1
Resolve Schema/Instance Conflict	5.1, 5.2, 5.3
View Conflict Summary	5.1
Drill Down Conflict	5.3
Solve Conflict Collaboratively	6.3.3, 2.3
Discuss Conflict	2.3
Comment Conflict	-
Forward Conflict	4.4
Postpone Conflict	4.5
Create Conflict	-

## 4.4 Forward Conflict

Forwarding is currently not a function offered by the visualization, if the user wants to forward a conflict, he has to open the conflict detail page, where he is able to forward the conflict to another user.

## 4.5 Postpone Conflict

Postponing conflicts is not part of the visualization, the user is able to ignore a conflict for a certain time, but he is not able to set a new due date for it. However postponing is possible in the detailed task view a conflict, when the user has the needed rights.

## 4.6 Create Conflict

Conflicts are only created by the merge algorithm and there is no functionality implemented, which allows the user to create his own conflicts. The visualization is a tool to solve given conflicts, but not to create new ones.

## 4.7 Use Case Matrix

Table 4.1 provides an overview over all use-cases and the chapter, where the implementation or the design of the function is further described.



## 5 Design of the Prototype Visualization

This chapter is all about the design and mock ups of the implemented visualization and some of its functions. The visualization, and therefore this chapter, is divided into three layers. Layer 1 is called dashboard. Layer 2 the first instance layer with all instances of one type, called instance overview layer and layer 3 the second instance view with a single instance and the instances associated to it. All views are UML-like models, so it should be easy for the user to understand the different elements of the visualization. The layers and can be closed by clicking the small "X" in the respective top right corner.

### 5.1 Dashboard

The dashboard is the first view, which is showed, when the visualization is opened. It provides a lot of information at a glance. The basis of this layer is a class-diagram with all defined types and their relationships in the given workspace as classes and associations. The class "Application" in Figure 5.1 will be described in detail, to explain the notation. "Application (32)" is the name and the number of instances of this type, existing in the workspace. "Product Name (32): Text [1,1]" is the first defined (indicated by the green text color) attribute of an application. In this case 32 (all) instances have this attribute defined. This number is especially important, when the attribute is not defined, because then it is interesting how many instances use the attribute even-though it is not defined. Examples therefore are the attribute "Revision" (derived) or "Query Language" of "Database" (undefined). The difference between derived and undefined is, that derived are attributes, which over 50% of the instances of this type use. "Text" means that there is a value constraint on the attribute and only text is allowed as value. Other possible constraints are for instance "Date", "Number", "Enumeration" or "No type constraint". The last information given about a single attribute is its multiplicity. It can reach from [\*,\*] (any number), [1,\*] (at least one) to [1,1] (exactly one). The associations between the classes get created for every attribute with a link constraint to a page of a specific type. For our type Application, it is the attribute "creates" with a link constraint to pages of the type "Data Object". Schema conflicts are also marked on the dashboard, with a red exclamation mark sign, which will always be the sign for a conflict. Clicking on it will open a conflict info-box, where the conflict can be solved. More information about the info-box can be found in section 5.4. The last thing worth mentioning is the red/green bar in the bottom of every class. This bar denotes how many instances of a type are afflicted with at least one conflict. In our case, there are eight instances with conflicts, which is 25%. From the dashboard you can show different second layers, one for each type, by double-clicking the respective type.

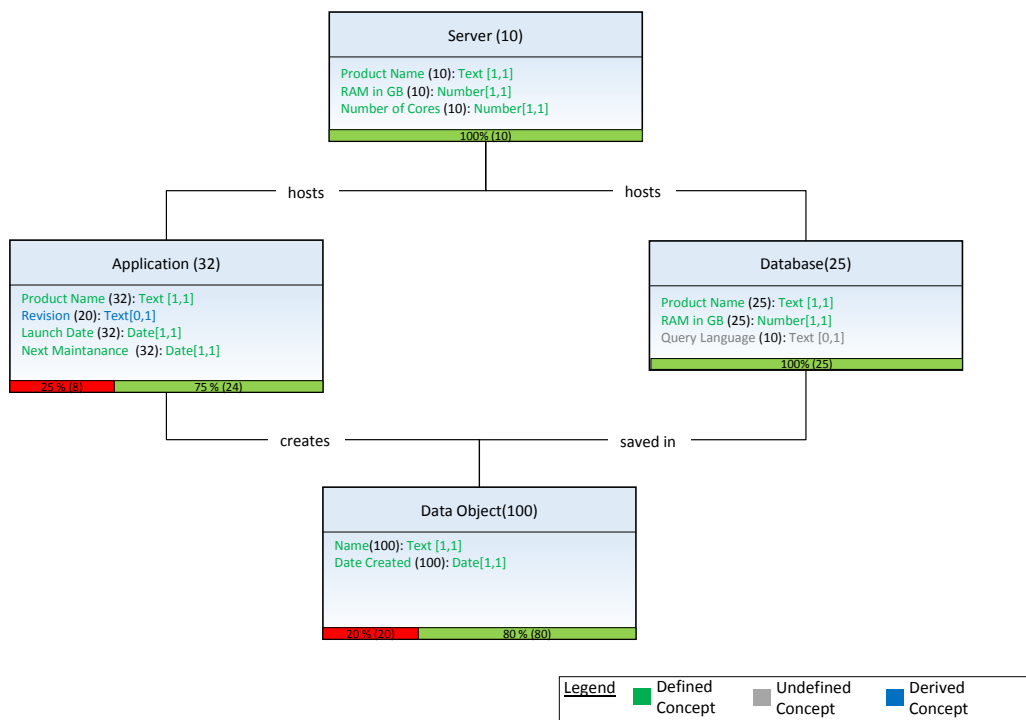


Figure 5.1: Dashboard Mock Up

## 5.2 Instance Overview Layer

The second layer is all about an overview of the instances of a selected type. In the first view in Figure 5.2 the user can see all instances of the type "Data Object" in a minimized view, to make sure, that as many instances as possible are viewed at the same time and to support big numbers of instances. Conflicts linked to an instance are marked with the red exclamation mark, as shown in the symbol of "Data Object 1". If more information about the instances is required, it is possible to hover them. By doing so a box with detailed information will appear, structured like an object in an UML-object diagram. The name is the name of the instance/page in Tricia and all attributes have their values shown. Conflicting attributes are highlighted in red and marked with the exclamation mark. If the user wants to know more about the conflict he can click the exclamation mark symbol and a detailed info-box will appear, which is shown in Figure 5.3. This box is the main tool to solve a conflict and it's information is adjusted depending on the conflict. In this case a update-update conflict for a value of an attribute is shown. In one branch the "Name" - Attribute got changed from "Table1" to "Table5" and in the second branch the name is changed to "Table7". The third value is the one from the base version. For every value, there is the option to click the user-symbol to get to the Tricia page of the last editor to get his or here contact details, if more information is necessary. With the help of the check the user can accept one version as the correct one and solve the conflict. By doing so, the data on the server will be updated and the visualization will reload, but this conflict will not be shown anymore, as it is marked as solved.

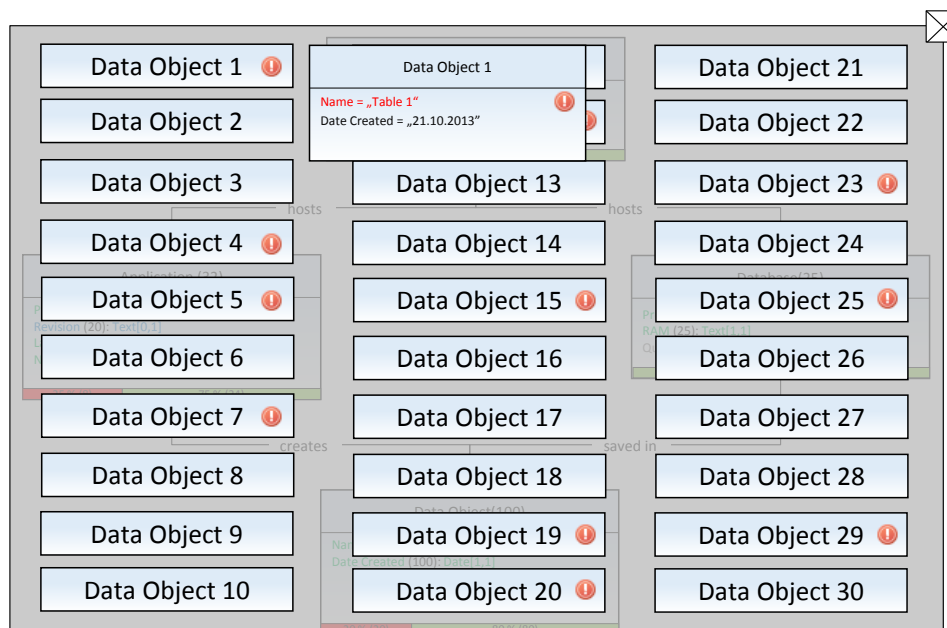


Figure 5.2: Instance Overview Layer Mock Up with Hover

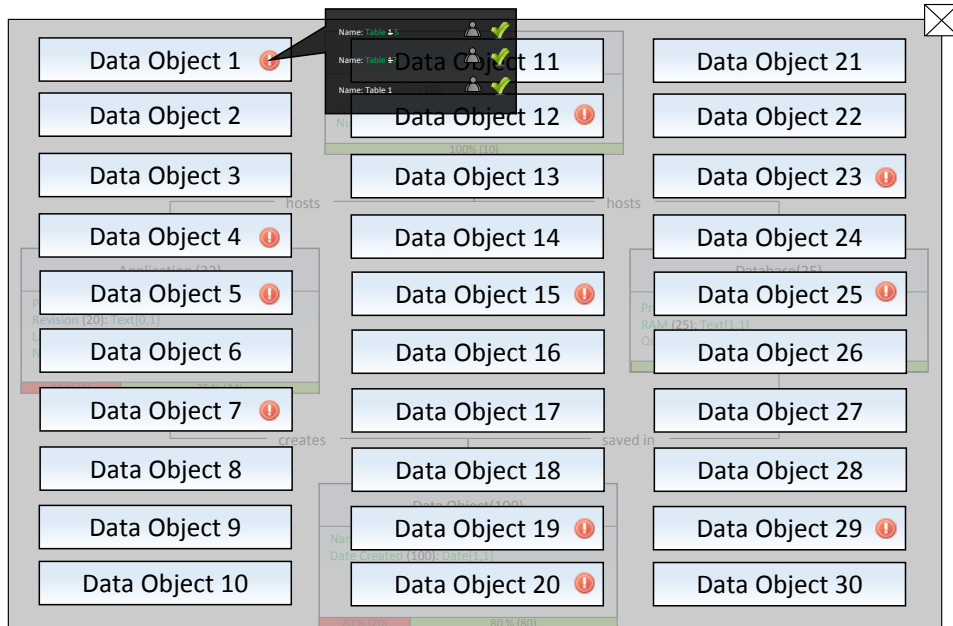


Figure 5.3: Instance Overview Layer Mock Up with Info Box

### 5.3 Instance Detail Layer

The third is the last layer in the visualization and represents a single page with all other pages directly associated to it and is shown by double-clicking a instance in the instance overview layer. It is called instance detail layer, as it represents further details about a single instance. Like in layer two all pages are represented as an object in a UML diagram as shown in Figure 5.4. All conflicts of the shown pages will be added to the visualization as-well, again marked with the red exclamation mark. It is possible to solve all marked conflicts like in layer three with the help of the conflict info box (Figure 5.5). The third layer helps the user to get additional context information of the conflicts and therefore it is more likely, that the conflict can be solved correctly without further research like asking the last editors of the models.

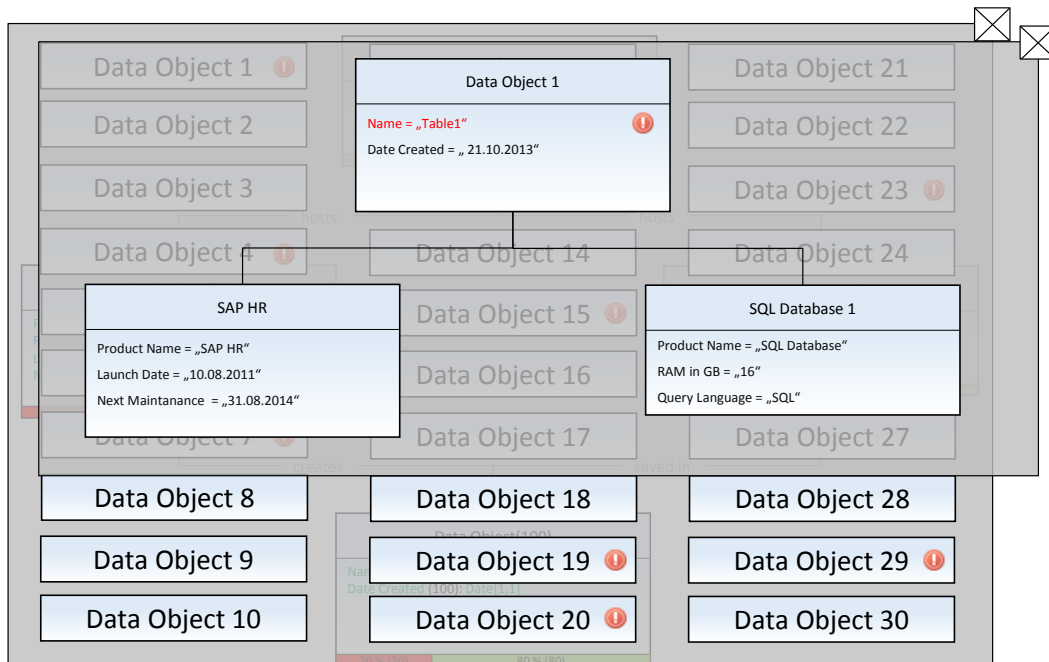


Figure 5.4: Instance Detail Layer Mock Up

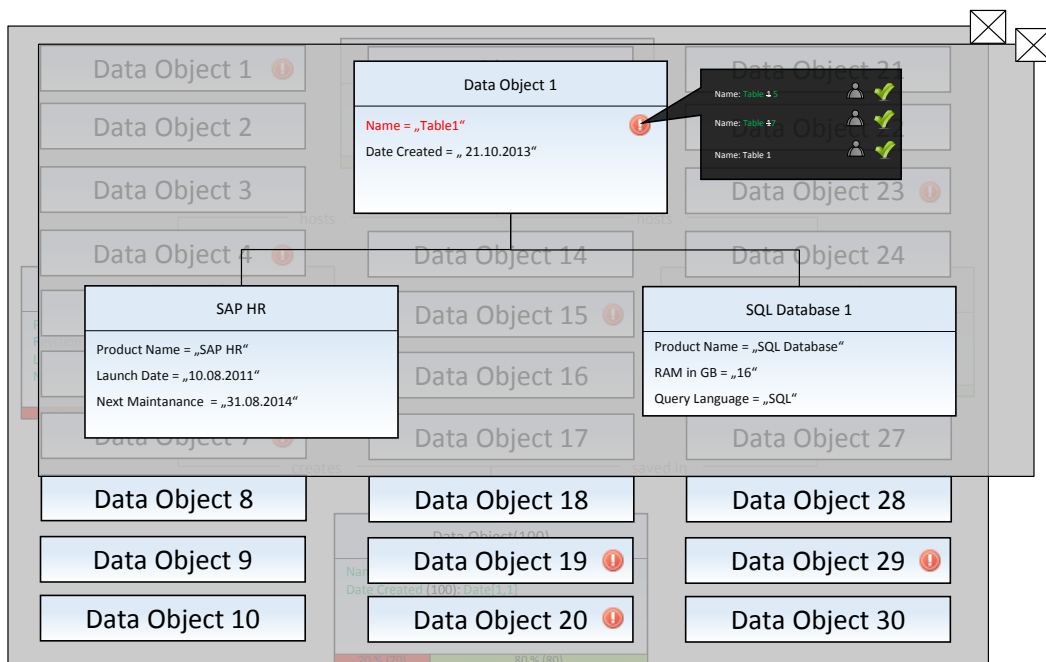


Figure 5.5: Instance Detail Layer Mock Up with Info Box

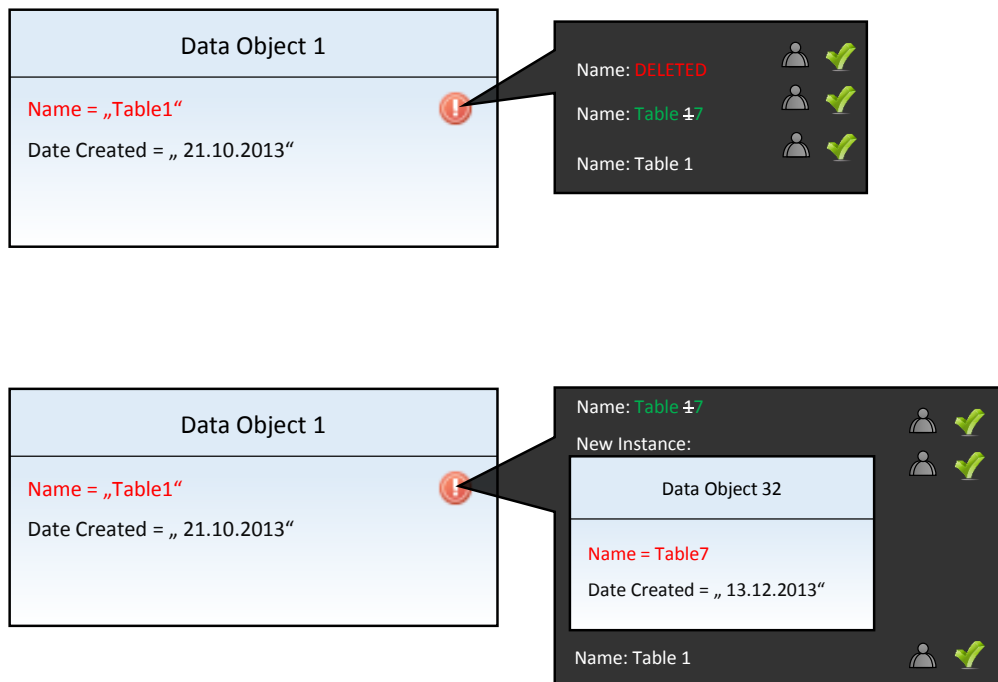


Figure 5.6: Example Info Boxes

## 5.4 Info Boxes

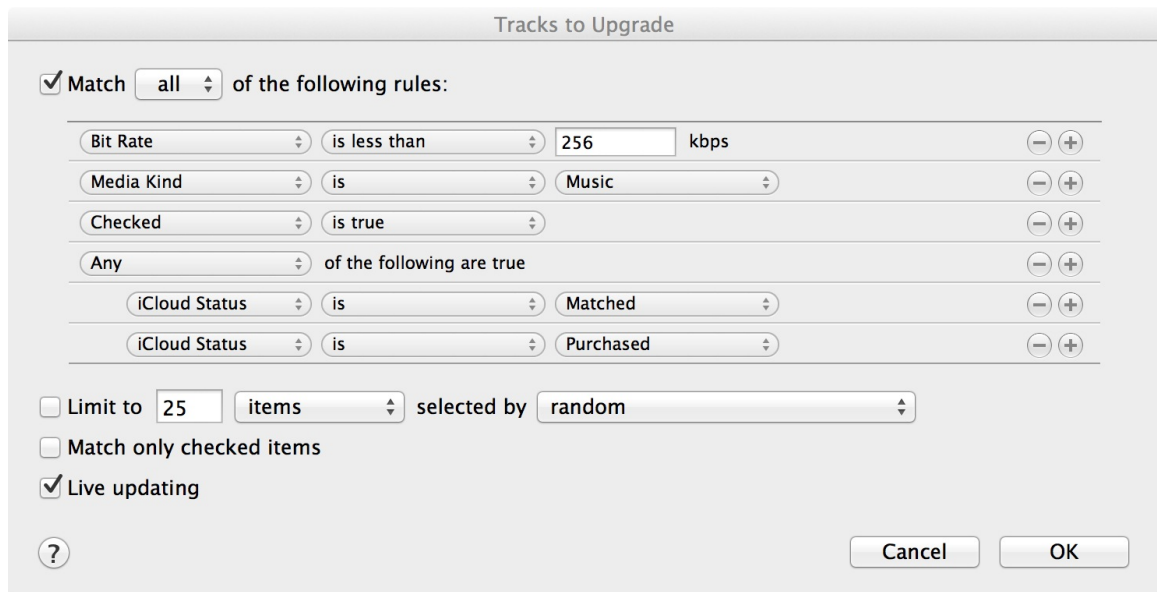
The info boxes are the main tool to solve conflicts and also the main tool to get information about the conflicts. Therefore it is very important, that they are customized for the specific type of conflict. The only part, which is the same for every info-box is the right part. There is always a small user symbol, to get information about the last user and the check, to accept the version as the solution of the conflict. In the following part we will show some example info-boxes and the idea behind the layout of their specific content.

In the examples above, Figure 5.3 and 5.5, the info box represented a value update-update conflict on object level. An update conflict always shows the old and the new value combined. First the name of the attribute which is in conflict and then the old value where the deleted part of the old value is stroke through and written in white and the part, which is still part of the value written in green. This is followed by the new value, which was not already included in the old one, also written in green. So if someone only wants to know the new value, he or she can just read the green letters. This is, how every update conflict for values is shown.

When there is a conflict, including a delete operation as one version, the line for this version will just show the name of the attribute or instance and the word "DELETED" in red letters, as there is no more information needed.

One of the more complex conflicts is, when a total new instance is created and an other instance got his name changed, so both instances have the same name, which results in a create-update conflict. When this happens the info-box shows a node symbol of the new instance like in the instance detail layer, so the user can see every attribute and its values at a glance. Examples of the info-boxes described above are shown in Figure 5.6



Figure 5.7: Example iTunes Smart Playlist Filter<sup>1</sup>

## 5.5 Filter

The filter is a very important part of the visualization, because it enables the user to handle big amount of data by only showing the relevant data in the visualization. As we made pretty good experience with the iTunes smart playlist filter, we decided to implement a similar feature. An example iTunes smart playlist filter is shown in Figure 5.7<sup>1</sup>. The most important feature is the possibility to create sub-filters, meaning that you can filter a given subset even further and then connect the result to other filters. In Figure 5.7 there are the three main filters, "Bit Rate", "Media Kind" and "Checked" and the sub-filter "iCloud Status". The three main filters are concatenated with an "and" operation because the filtered music has to match all of them, but for the sub-filter the any rule applies, so all objects having the "iCloud Status" set to "Matched" or "Purchased" will match it. As logical expression this particular filter is written as:  $\text{Bit Rate} < 256 \wedge \text{Media Kind is Music} \wedge \text{Checked is true} \wedge (\text{iCloud Status is Matched} \vee \text{iCloud Status is Purchased})$ . These sub-filters can be stacked several times to generate complex logical expressions, which are still quite easy to read and understand, because of the table layout of the filter in the user interface.

Figure 5.8 shows the filter graphical user interface as implemented in our visualization. In the following we will explain its elements in detail. The initial version of the filter contains one filter row for every type possibly shown in the visualization. The check-box indicates, whether the type is shown in the filtered visualization or whether it's filtered out, so by clicking the checkbox the user can filter out a type with all its instances with one click. The next element is the the type name followed by the basic filter elements. The first drop-down menu contains all attributes of the type, defined, undefined and derived. Depending on the type selected and it's constraint, whether the type's value is a number, a text, a date or not specified, the next drop-down contains comparison operations. For numbers, these are "<", ">", "=" or "Not Null". For text "contains", "starts with", "ends with", "=" and "not null" are possible comparisons and for dates they are "before", "after", "=" and "not null". When the type is not specified alls comparison operations are available, as no exclusion can be made. The next field is the input field for the number or string which will be

<sup>1</sup>Image source: <http://4.bp.blogspot.com/-IR9qeVFJx9I/UACaHbiq81I/AAAAAAAAABuE/GjExGAADZIE/s1600/Smart%2BPlaylist%2B-%2BTracks%2Bto%2BUgrade.jpg>; Last Accessed: 08.12.2013

<input checked="" type="checkbox"/>	Server	RAM in GB ▼	">" ▼	128	+ - ▼
	and ▼	Number of Cores ▼	">" ▼	16	+ - ▼
	or ▼	RAM in GB ▼	">" ▼	256	+ - ▼
	and ▼	Number of Cores ▼	">" ▼	32	+ - ▼
<input checked="" type="checkbox"/>	Application	Product Name ▼	contains ▼	String	+ - ▼
<input checked="" type="checkbox"/>	Database	Product Name ▼	contains ▼	String	+ - ▼
<input checked="" type="checkbox"/>	Data Object	Name ▼	contains ▼	String	+ - ▼

Figure 5.8: Example Type and Instance Filter

compared with the attributes values of the selected type. Only instances, which match the filter will be shown in the visualization. The three buttons in the end are there to either add or delete a filter line (including sub-filters) or to add a sub-filter.

In the example filter in Figure 5.8 all types are shown, but there is an instance filter for the instances of the type "server". The main filter consists of two filter rules, which are concatenated with an "or" (line 1 and 3 of the filter). Both of these filter rules have an additional sub-filter, which has to be matched by the instances as well ("and" as link operation). As an logical expression, this filter can be read as follows:  $(\text{Ram in GB} > 128 \wedge \text{Number of Cores} > 16) \vee (\text{RAM in GB} > 256 \wedge \text{Number of Cores} > 32)$ . The visualization will now show all instances of the type server with more than 128 GB RAM and 16 cores or more than 256 GB RAM and 32 cores. For the instances of the types Application, Database and Data Object no filter is entered and therefore every instance will be visualized. It is also possible to add more than one sub-filter and to add one or more sub-filters to the sub-filter, as the filter structure is fully recursive. Further information about the implementation of the filter is given in section 6.4.4.

# 6 Implementation Details about the Prototype Visualization

This section explains a lot of implementation details of our visualization. Starting with the explanation of the interface to conflict tasks and an introduction to the visualization framework, the framework extensions implemented by us and the implementation of the visualization itself are described.

## 6.1 Conflict Task Interface

Figure 6.1 shows the classes implemented by Björn Kirschner, which are used as interface between his merge algorithm and our visualization. As some of these classes are relevant for the visualization, we will explain the important classes in short. Namely these classes are: `MergeWithSpaceData`, `ModelConflictTask`, `ConflictListEntry + ChangeSet` and `Change`, they are marked in green in the diagram.

### **MergeWithSpaceData**

`MergeWithSpaceData` is the location where all information about a single merge is saved. The `PageSpaces`, which are merged, the `LearningData` and all `MergeConflictTasks`, which were created during the initial merge are stored there. The `LearningData` is important for learning and batch-solving, explained in detail in chapter 6.3.5.

### **ModelConflictTasks**

A `ModelConflictTask` is the central element shown in our visualization. There are different `ModelConflictTasks` depending on the type of the conflict, as they have slight differences. The information saved in an `ModelConflictTask` is mainly used by the `InfoSymbol` for the creation of the info-boxes (See chapter 6.4.3). Also important about a `ModelConflictTask` is the link to the `ConflictListEntries`.

### **ConflictListEntry + ChangeSet**

A `ConflictListEntry` has all conflicting `ChangeSets` and all `Changes` linked to it. Whereas a `ChangeSet` consists of `Changes` and provides additional information like the last editor and the timestamp when the change was carried out. A `ConflictListEntry` saves the old and the new value set in the change depending on what kind of change is linked to it.

### **Change**

A `Change` is the modular action, captured by the merge algorithm. In our case it has several subclasses, depending on the type of the change. Many changes together are centralized in one `ChangeSet`. Originally the change has stored the information, which attribute or page got changed, and what was the old and what is the new value.

All these classes together, besides the data about types and pages saved in the wiki, provide the basis data for our conflict visualization.

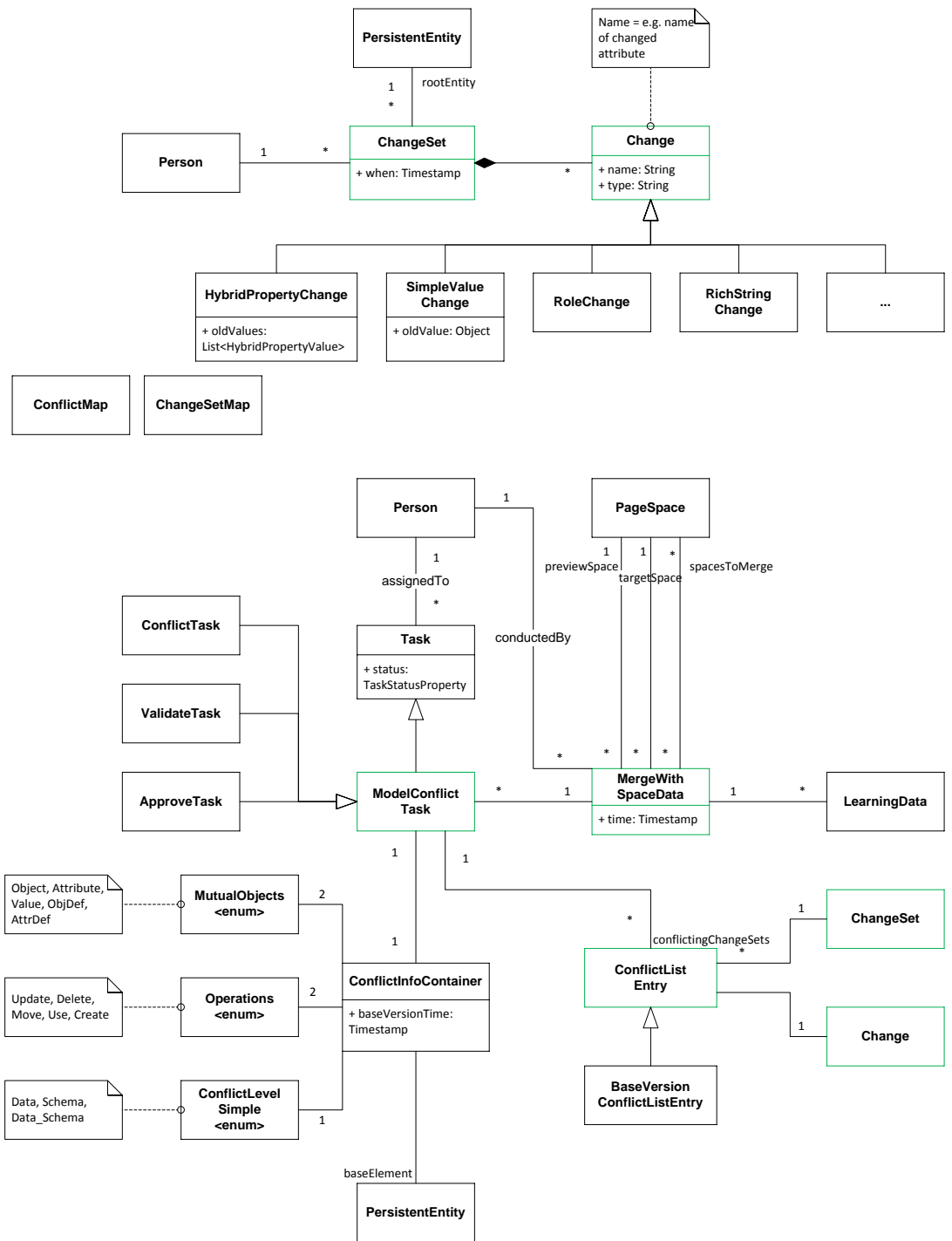


Figure 6.1: Conflict Task Interface Class Diagram[Kir]

## 6.2 Visualization Framework

The framework I am using to generate the visualizations is the RaphaelJS framework with modifications implemented by the sebis chair at Technical University Munich. Figure 6.2 shows all classes the framework provides, which are relevant for my visualizations. All classes marked in green are newly added and the blue ones are modified and will be explained in detail later. In the following, we will explain some of the standard classes and where they are used.

Every Object is a so called VisualizationObject with an unique id and a content, which can be set individually. Moreover it has also a boolean flag visited, which is important to avoid endless loops during the serialization, for instance when there are loop dependencies between different visualization objects. From here it splits up into the two main parts. "Symbols" on the one hand and "Interactions" on the other. For all three mentioned classes the composite pattern is implemented, so there is a subclass CompositeVisualizationObject/Symbol/Interaction, which can contain several VisualizationObjects/Symbols/Interactions. I will now first explain some of the subclasses of symbol and after that the different used interactions.

Every Symbol has a x- and y-position and height and width, determining the exact position and size of the symbol on the paper. The sub-class PlanarSymbol introduces attributes like color, border color, opacity and border width. As shown in the diagram, there are various sub-classes of PlanarSymbol. The most used ones in our implementation are: Rectangle, Line, Text, Image. All have their class specific attributes. Rectangle only introduces a round amount to round the corners of a rectangle. In the class Line it is possible to set the start and end arrow. An object of the class Text, of course, has the displayed text and its font as attributes. Furthermore the vertical and horizontal alignment, the tool-tip text (which will be showed by hovering the text) and a boolean attribute strike-through, which has to be set to true, if the text should be displayed as stricken out, can be set. The last class worth mentioning for this thesis is Image. It is used, to insert a image from an external source. This is done quite easily, by just creating an image object with the URL of the image, which should be displayed. These were all symbol classes worth mentioning and we will now explain some of the interactions.

The interactions are a bit more complicated as the symbols as they provide way more functionality. We will have a deeper look into the classes Click, Show, Hide and Update. The most important thing about interactions is, that every interaction has a so called visObject, which is a VisualizationObject, the interaction is linked to, it has different meanings, depending on the interaction. A click has, besides the visObject, the attributes onClickAction and doubleclick. Doubleclick is a boolean which determines if the click has to be a double-click or not. An onClickAction is an interaction, which will be triggered, when the click is executed. The visObject for click is the object, which has to be clicked to trigger the onClickAction. Show and Hide do exactly what their name implicates. When executed they either show or hide their visObject. An object of the class update is used, if the user wants to reload the a whole visualization, for instance when the data in the background has changed and he wants to have the new data displayed in your visualization. There is a more detailed explanation of the update interaction in section 6.3.1.

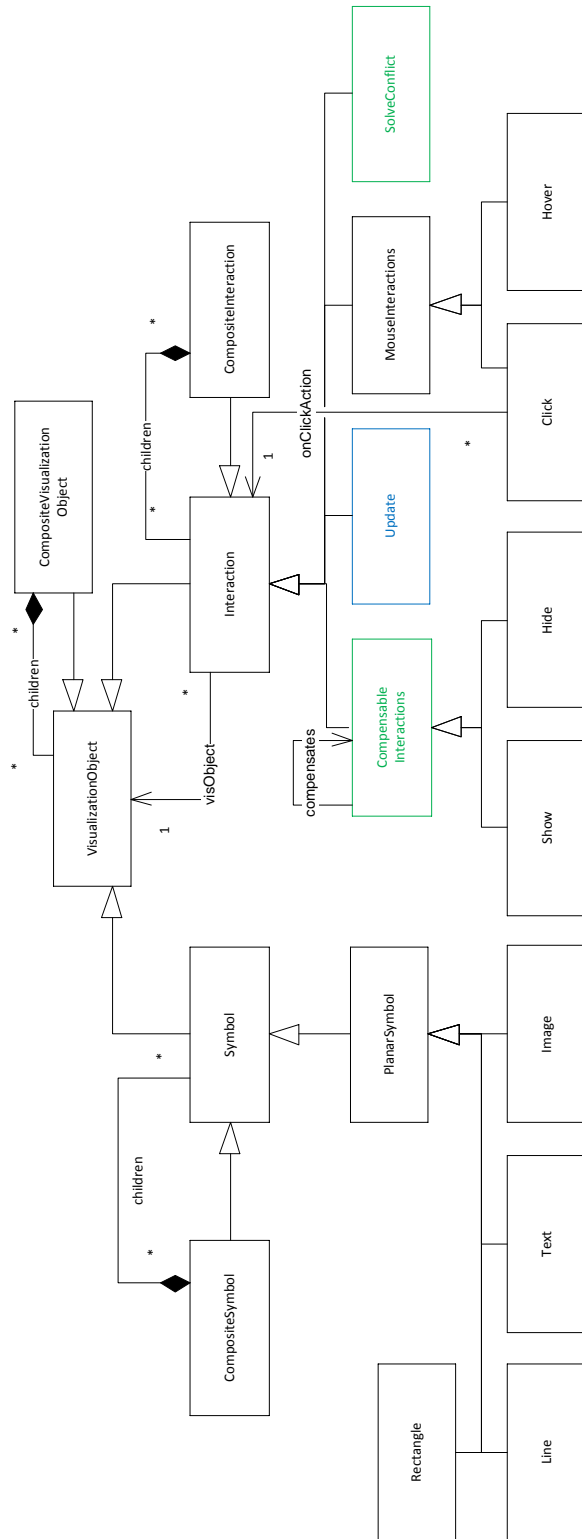


Figure 6.2: Visualization Framework Class Diagram

## 6.3 Implemented Framework Extensions

This section introduces the extensions we implemented into the graphical framework, to support our needed functionalities.

### 6.3.1 Stateful Update

In this section, we describe the new update function, which is an extended version of the normal update provided by the framework. With the help of an update, as it is implemented by the framework, you can reload your visualization with new data, but the update is stateless, which means all user interaction with the visualization is lost after the update. In the implementation updates are used when the user solves a conflict and the data on the server-side changes, so the model does not represent an outdated version of the data or when a filter is applied to the visualization. Because you want to continue at the same position where you have been before updating the visualization, we implemented a data-structure to save all executed clicks on the visualization and just re-execute them after the update. This data-structure and its behavior will be explained in the following part.

All visualization objects, including objects of the type "Click", of the visualization have a unique id, but saving this id would not help, due to the fact, that after an update all ids change and the old ids are worthless as there is no relation between the old and the new ids. To solve this problem another identifier for the clicks is needed. In the visualization every click-able symbol is related to a specific page on the server, and every page has a unique URL. This URL obviously doesn't change after an data update and there is only one object representing a page/URL per layer of the visualization. So we pass this URL through the data-binding (as URI) to the click-able symbol (as content), as you can see in the example in Figure 6.3. This ensures, that every object has a unique identifier, which is persistent through all updates of the visualization.

For a multi layered visualization, this data-structure is not powerful enough, as both interactions in Figure 6.3 share the same page and because of this have the same URL saved, which then can't be used to find the exact object, which was triggered. Due to this fact the data-structure had to be extended in some point. In the final implementation the interaction itself also has a content, which is a number representing the respective visualization layer. This is shown in Fig. 6.4.

Figure 6.5 shows a class diagram based model of the data-structure as it is implemented in the framework. Every executed relevant interaction gets saved in the configuration at its unique position, namely this is "paper.config.interactions[Symbol.content]click[Click.content]". "paper.config.interactions" is the base position, which is the same for every interaction, "Symbol.content" is the URL of the page represented by the symbol, "click" indicates that the saved interaction is a click, and "Click.content" is the layer which includes this interaction. The identifier click has no use but making sure that the update function can be extended even further with other interactions as clicks when needed. During the re-rendering of the visualization the program checks for every interaction, whether it is already saved in the configuration and if it is, it gets executed directly. With this implementation the state of the shown visualization will be the same after the update as it was before, but with the changed data.

These were the important facts about the data-structure, now we will have a look at the sequence diagram of the update and the interactions between visualization server and client, as shown in Figure 6.6. The call "getVisualization" and its response is the standard call to get the visualization from the server. The browser then passes the visualization and its configuration to the JavaScript engine. Now the Engine starts to log all user interactions and saves them in the configuration (config\* = config + tracked user interaction). The now following calls are all in a loop as they are the same for every executed update. First the update function of the server with the modified configuration, config\*, is called. The server will render the new visualization and send it and the config\* back as response. JavaScript will now replace the old visualization and after doing so, the config\* will be applied to it. This means all stored interactions will be executed and thereby the user will see the same state of the visualization as before the update. After this is done, the engine

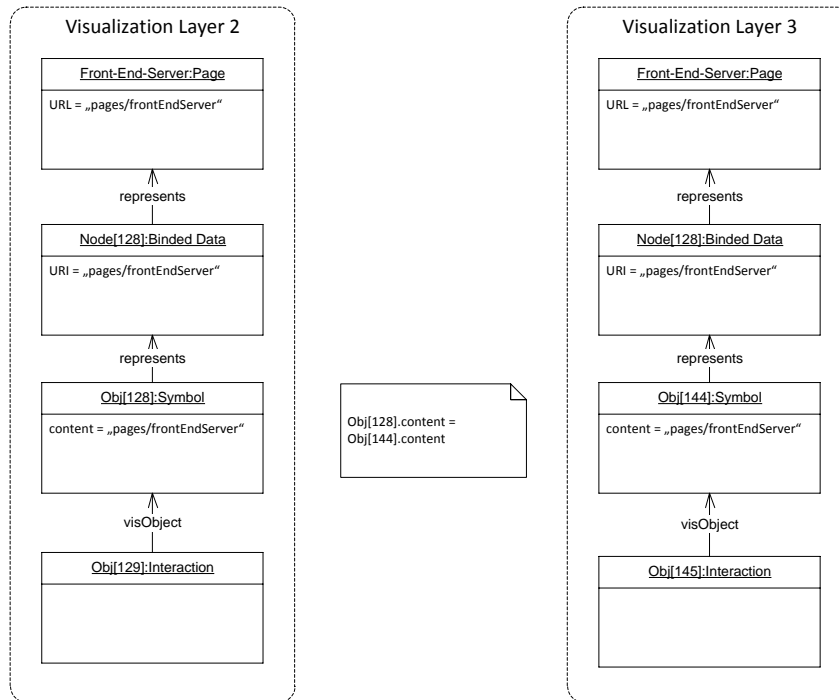


Figure 6.3: Problem of the Update Function

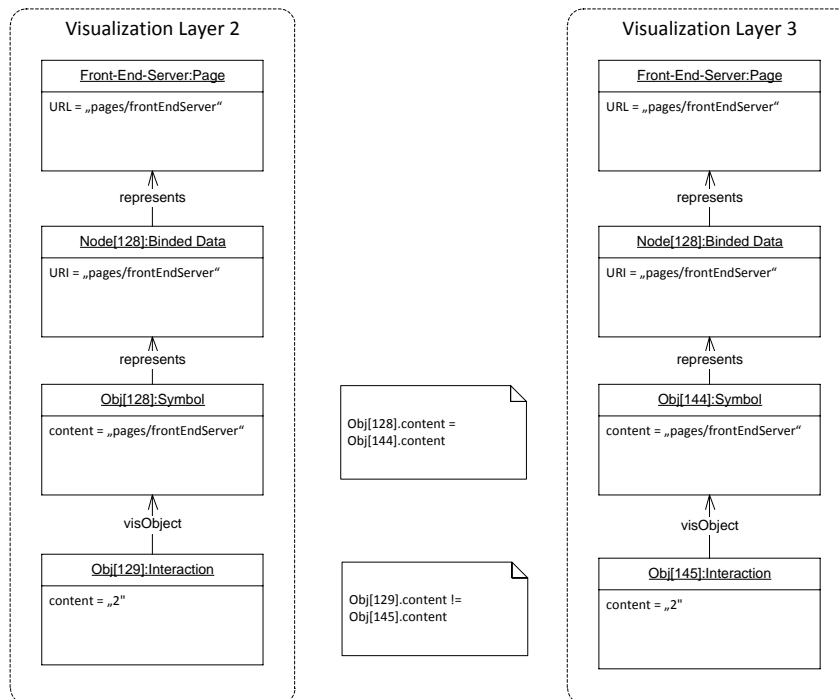


Figure 6.4: Solution to the Problem of the Update Function



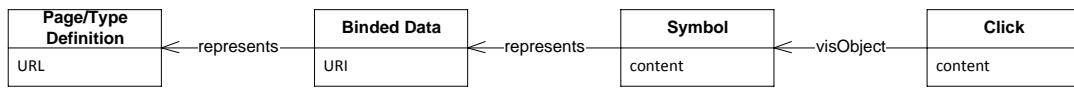


Figure 6.5: Update Function Class Diagram

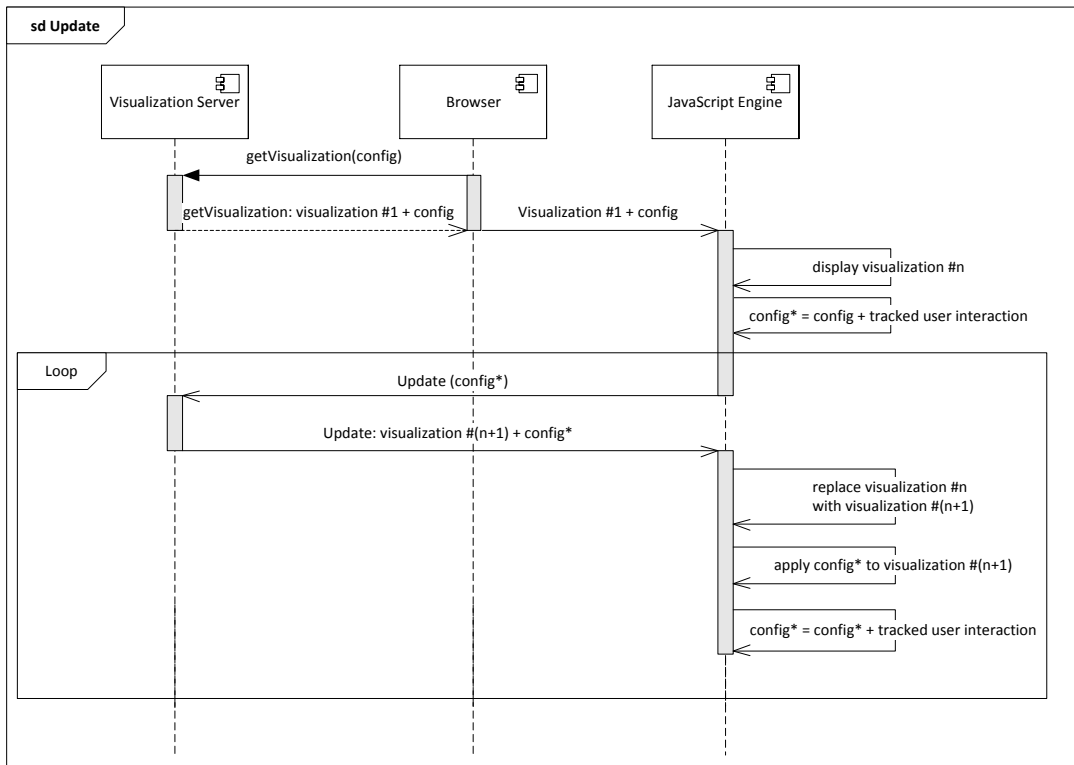


Figure 6.6: Update Function Sequence Diagram

will continue tracking and saving all user interactions and save them in the configuration. It has to be added, that this tracking can occur more than once, but the loop is not included in the diagram, due to overview reasons. Calling the update function again will result in a restart of the loop at the update call.

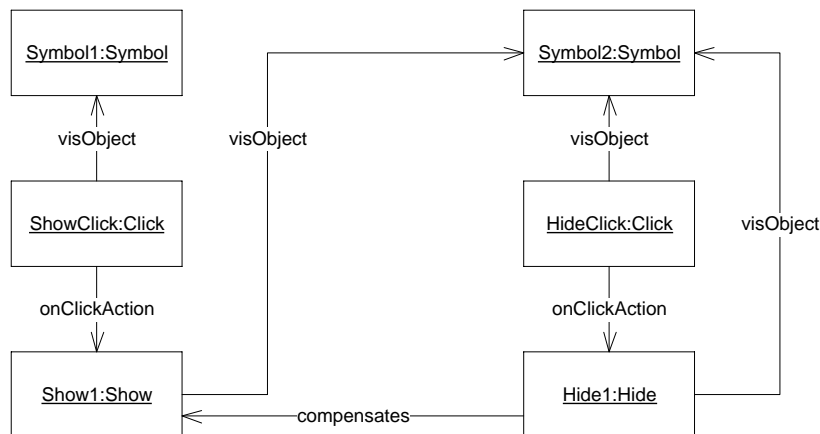


Figure 6.7: Compensable Interaction Example Object Diagram

### 6.3.2 Compensable Interaction

In the previous section we introduced the new update function. To make the new update work properly there is another extension of the framework required. There are interactions, which make others undone. Two such interactions are for example Hide and Show. In the following these types of interactions are called compensable interactions. In the following we will describe the implementation of these compensable interactions. Figure 6.7 displays an example with two interactions which compensate each other. When the user clicks on Symbol1, Symbol2 will be shown and a click on Symbol2 hides it again. In this case Show1 and Hide1 compensate each other. Therefore it is possible, that the configuration saves the interactions several times, as the user could hide and re-open Symbol2 as often as he wants. Another inefficiency is, that after an update the hide and show actions will be executed with no visual effect at the end. To prevent this, Children of CompensableInteraction got an association to another compensable interaction. In the case showed in Figure 6.7 Hide1 has a association to Show1. As it is not possible to save the compensates attribute on both interactions, due to serialization issues, We decided to always safe the compensates attribute on the action, which will be executed after the other.

Figure 6.8 shows a part of the new framework class diagram, with the new introduced class CompensableInteraction with its subclasses Hide and Show.

Because of this new data-structure it is now possible to remove compensated interactions from the paper.configuration. After every execution of a compensable interaction the associated interaction now gets deleted out of the paper.configuration. When there is an interaction deleted, the compensable interaction itself wont be added to the configuration to prevent the record of useless data. For further work also a Draggable interaction could become a compensable interaction, because it is possible to first drag and drop an object into a new container and then put it back to the old one, but as this is not part of our visualization we decided to leave the implementation out.

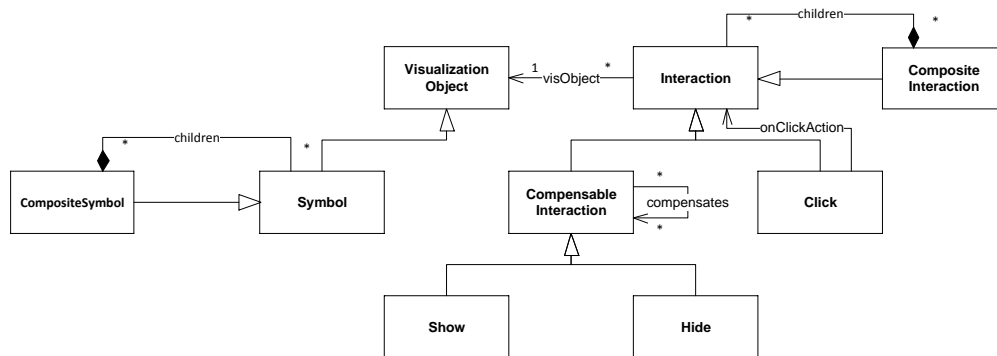


Figure 6.8: Compensable Interaction Class Diagram

```

1 //Code on client side, to connect to the server and send an update message
2 paper.socket = io.connect('http://localhost:8080');
3 paper.socket.emit('update', paper.config);
4
5 //Code on sever side, to broadcast the update message
6 socket.on('update', function (data) {
7   socket.broadcast.emit('update', data);
8 });
9
10 //Code on client side, to handle the update message
11 paper.socket.on('update', function (data) {
12   new Update($id$, data)();
13 });
  
```

Figure 6.9: Broadcasting Functions JavaScript Code

### 6.3.3 Broadcasting with NodeJS

In the now following section, we will describe the use and implementation of a NodeJS[Incc] server for broadcasting, which leads towards real time collaboration. To configure NodeJS, just a small piece of JavaScript code is needed. This code determines which actions the server will take, when receiving a certain type of message or when a client tries to connect to the server. The handshake is done completely automatically and doesn't need further configuration. For our visualization NodeJS is used as a broadcasting server, to emit new configurations and visualizations. The broadcasting of actual configurations and visualizations enables the feature to work collaboratively in (almost) real-time. Clients can connect to the server by activating real-time collaboration and will then trigger and receive broadcast messages. Therefore only small adjustments in the code have to be made, by adding code, like shown in Figure 6.9, Line1-3. Paper.socket is the actual socket to reach the NodeJS server and gets defined in the first row. The second row emits a "Update" message, and sends the actual configuration of the visualization enclosed to the message. On the server side we configure, what happens when it receives an update message. In this case it will take the message and broadcast it to the clients, Line 5-8. To make the client listen to the message, line 10-13 get implemented. In this case the client will take the received data and perform an update of the visualization (Line 12).

To further explain the behavior of the broadcast, Figure 6.10 shows the sequence diagram of the

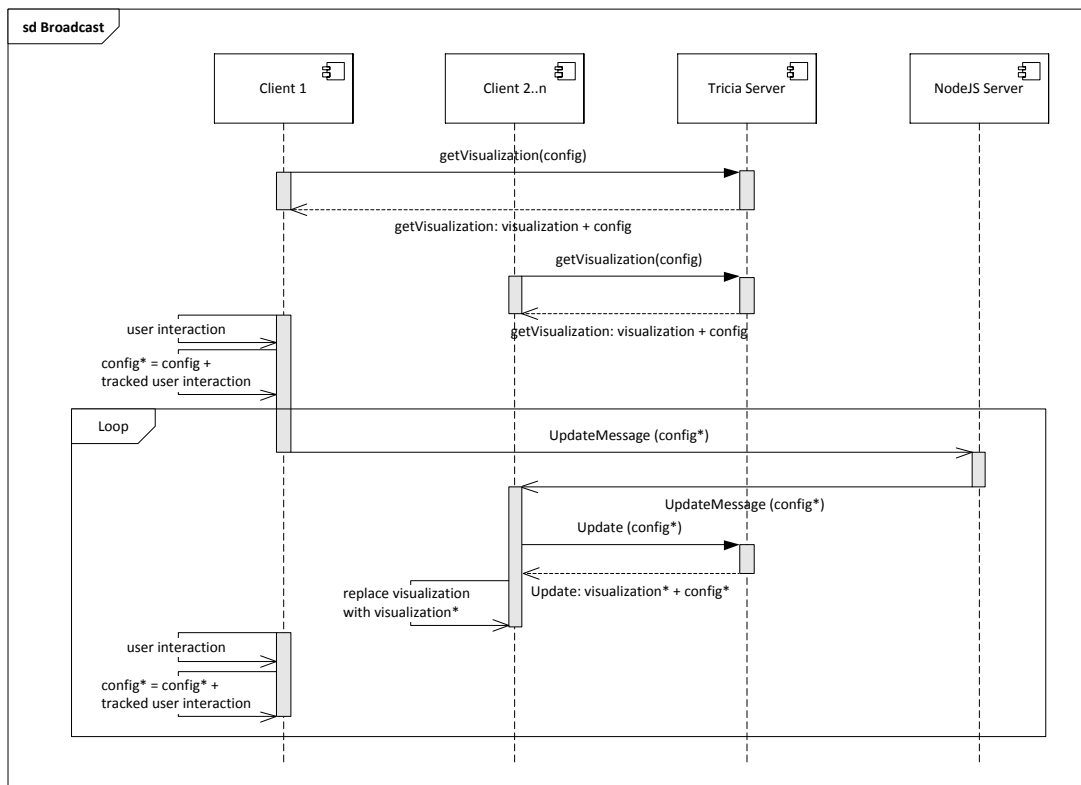


Figure 6.10: Configuration Broadcast Sequence Diagram

of a simple broadcast. Simple, because there are no data changes on the server, the case with change of server-sided data will be discussed later. The part before the loop is for the sake of completeness as these are the calls and responses, which have to be made to get the initial visualization. The important part is the one within the loop. After every change of the configuration, for instance by showing a new object, this loop gets triggered. The client, that performed the interaction sends out an "UpdateMessage" with the new configuration ( $config^*$ ) to the NodeJS server. The server then broadcasts this message to all connected clients. After an incoming UpdateMessage, the client updates his visualization by triggering an update as described in chapter 6.3.1.

This implementation has room for improvement. The biggest problem is, that for a high number of connected clients  $N$  the load of the visualization server will increase drastically, because it has to create  $N-1$  visualizations. Another downside is the latency and therefore the time it will take, until every client received the new visualization, because of all the interactions between the clients, the visualization and the NodeJS server. For the next problem, a broadcast with data update, this problem has been solved. This time, the client calls the server to update data, e.g. when a conflict is solved. After doing so, the server automatically generates a new visualization and emits it via a DataUpdate message to the NodeJS server, which then starts to broadcast this message to all connected clients. The client triggering the DataUpdate automatically connects to the NodeJS server, although real time collaboration might be deactivated, to get the latest visualization as well. All clients will, on receiving a DataUpdate, replace their visualization with the new one, attached to the incoming message. With this version, the server created the visualization only once, no matter

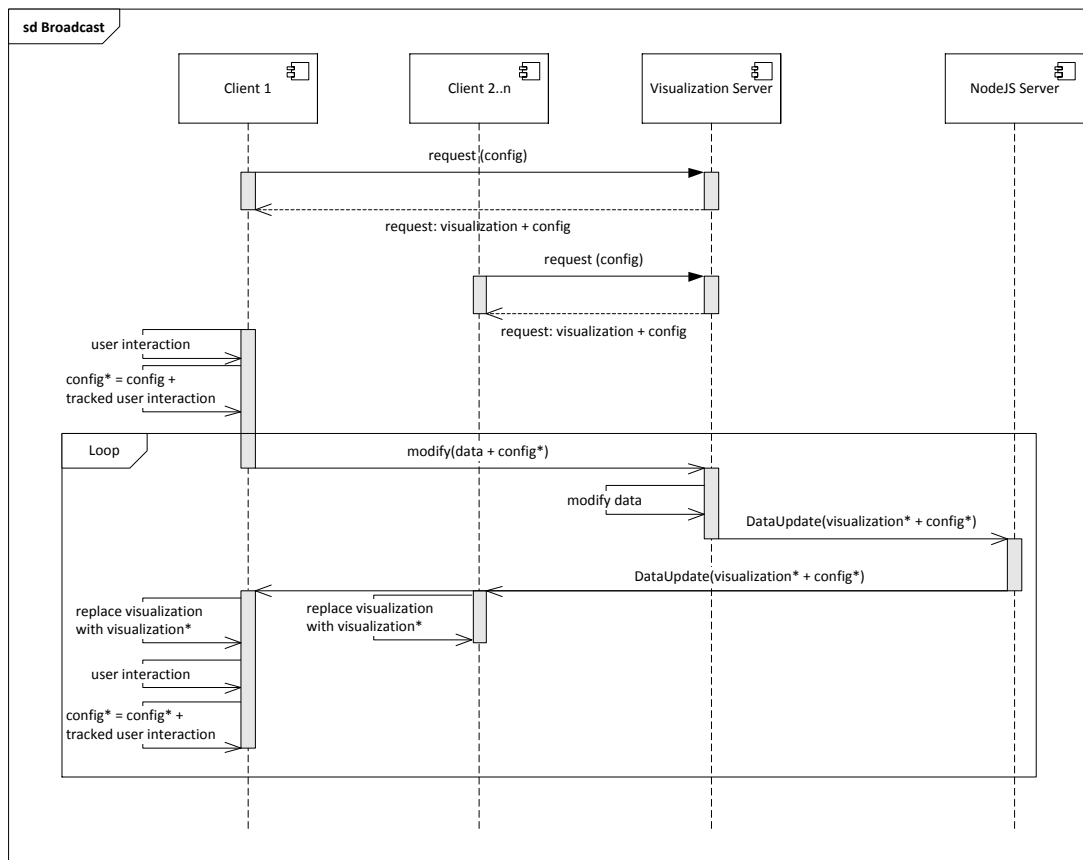


Figure 6.11: Visualization Broadcast Sequence Diagram

how many clients are connect to the NodeJS and the latencies are way lower, due to the fact, that the clients only receive one message, already including the visualization and don't have to call the visualization server for a new visualization themselves. The downside of the version is the problem with the rights, every client will get the visualization generated with the rights of the first client. This could include objects, which are usually hidden to the other clients or the other way round. So with this implementation the rights system is lost. This process is illustrated in Figure 6.11.

### 6.3.4 Solve Conflict

SolveConflict is basically an extended version of the update interaction, with special functionality to solve a given model conflict task. Every SolveConflict object needs a conflict list entry, which will be the selected solution for the model conflict task it is linked to. In the following we will explain the JavaScript code behind this interaction. All line numbers will refer to the line of code in Figure 6.12.

The code is divided into four main parts. In part one from line 1 to 5 a needed variable is declared and a configuration is made. In the second block from line 7-15 the solve conflict handler, implemented by Björn Kirschner is called with an ajax call. The parameters for the call are the chosen solution (conflictListEntry) and the actual configuration of the visualization. With this information

```
1 SolveConflict = function(visualizationId , paper , conflictListEntry) {
2   return function() {
3     console.log("SolveConflict");
4     var node = jQuery("#viewpoint" + visualizationId);
5     paper.config.extension = "json";
6
7     $.ajax({
8       url : _context_ + "conflictTask/solveConflict" ,
9       type : "POST" ,
10      dataType : "html" ,
11      data : "id=" + conflictListEntry + "&config=" + JSON.stringify(paper.config) ,
12      context : document.body ,
13      success : function(data) {
14      }
15    });
16
17    paper.socket.on('dataupdate' , function (data) {
18      console.log('dataupdate')
19      node.attr('disabled' , true);
20      node.replaceWith(data);
21      node.attr('disabled' , false);
22    });
23
24    if (paper.config.learning == null || paper.config.learning){
25      $.ajax({
26        url : _context_ + "conflictTask/solveConflictLearning" ,
27        dataType : "html" ,
28        data : "id=" + conflictListEntry ,
29        context : document.body ,
30        success : function(data) {
31          data = jQuery.parseJSON( data );
32          if ((data.Strategy[0] != "No Strategy" )){
33            createLearningPopUp(data , paper)
34          }
35        }
36      });
37    };
38  }
39 };
```

Figure 6.12: Solve Conflict Interaction JavaScript Code

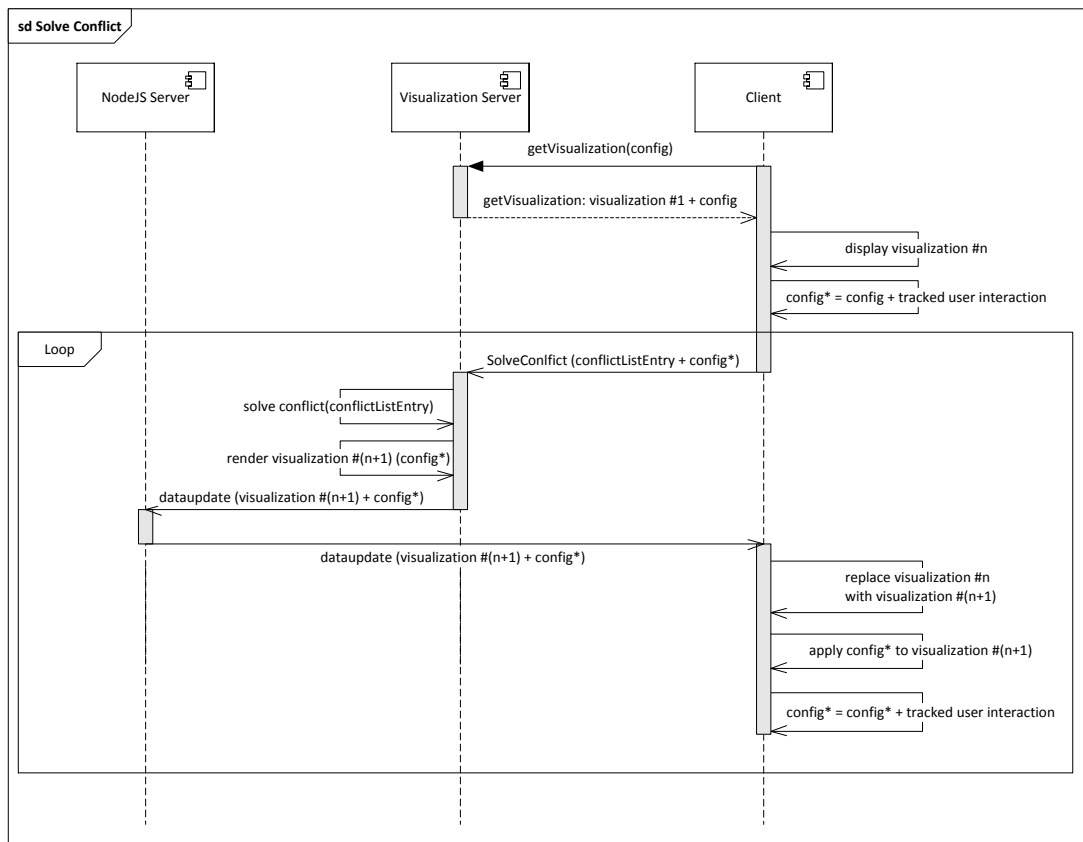


Figure 6.13: Solve Conflict Sequence Diagram

the solve conflict handler will solve the conflict and, after that, generate a new visualization with the help of the given configuration. The visualization will be sent to the NodeJS server and broadcasted to all connected clients, when broadcasting is enabled (More information about broadcasting in Section 6.3.3). If broadcasting is not enabled for our client, it will listen manually to the broadcasting socket and wait for the new visualization. This is implemented in lines 17 to 22. The last block (24-39) is for the intelligent learning, which will be explained in detail in section 6.3.5. Here the solve conflict learning handler is called, as long as learning is not deactivated for the current visualization. If the learning handler replies with an solution strategy a window will pop up explaining the strategy and asking, whether the user wants to solve all conflicts according to the given strategy. In a reduced (without learning) sequence diagram the solve process looks like this:

### 6.3.5 Learning and Batch-Solving

For the implementation of a learning algorithm not only the visualization framework needed new functionalities, but also the visualization server needed new handlers and a new data-structure to store meta information about the already solved conflicts. First we will explain the new data-structure and which information is collected, afterwards the new handlers and last but not least, when they get called by the visualization.

### Data-structure

As explained earlier in section 6.1, every merge has a page where all important data about the merge is saved, the so called MergeWithSpaceData. This central location is the best to save the data we need for our learning algorithm as it is linked to every conflict task of the merge. A new Entity of the type LearningData gets linked to the MergeWithSpaceData for every solved conflict. LearningData contains all the important data the learning algorithm needs to find one of the three supported strategies in the users solve behavior. The first pattern could be, that the user solves every conflict by selecting the same space as solution - the "Space Strategy". Another one could be, that always the change of a specific user, no matter in which space, is the selected version - the "Last Editor Strategy". The third tracked strategy is, that always the latest or the oldest change is the correct one (all changes in the time between oldest and latest are ignored) - the "Time Of Change Strategy". These three attributes are, besides a time-stamp, when the conflict got solved and the user, who solved it, the data, that gets saved in a LearningData object. The solve-time is important, because the learning handler will then only respect the learning data objects, which where created the same day. And the user is important because the handler only looks for conflicts solved by the user, which is currently working with the visualization.

### Handler

There are two new handlers, which will be explained in the following. The first one is the SolveConflictLearningHandler and the second one the BatchSolveConflictHandler. The first generates objects of the type LearningData and analyzes the already existing LearningData Objects. Therefore it only needs a ConflictListEntry (CLE), as the last editor of the object and the chosen page and therefore it's space are directly linked to it. A bit more complicated is the calculation, whether the solution is the latest or the oldest change, or was made anytime in between these two. For this calculation the handler collects all possible solutions (CLEs) from the task linked to the chosen CLE and compares their time-stamps. With the collected data, a new LearningData object gets generated and linked to the MergeWithSpaceData. After this is done, all LearningData objects of the given merge will be analyzed. Three hash maps get initialized, one for every strategy, as shown in Figure 6.14. For every LearningData object the value of the respective space, last editor or last time edited gets increased by one. For last time edited, the key "Between" is ignored, because most of the changes are somewhere in between the latest and oldest and it is not considered as a strategy picking them. The threshold determines, when the handler sends a recommendation for a certain key, because a strategy has been detected. It is calculated based on the number of total conflict tasks linked to a merge. When there are less than 16 conflicts, the threshold is 4, between 17 and 100 conflicts it is 25% and for numbers above 100 it is always 25. This ensures that, at low numbers of conflicts there wont be a recommendation too early and when there are really many conflicts, it won't be too late. If any value is above the threshold, the handler will return the discovered strategy, the key, whose value exceeded the threshold and the id of the MergeWithSpaceData object.

The second handler is for batch solving all remaining conflicts a merge. This handler needs four parameters to work properly. Namely these are: The learning strategy, the MergeWithSpaceData object, the key, which was returned by the learning handler, and the actual configuration of the visualization. The handler iterates through all open conflict tasks of the merge. For every tasks it checks all possible CLEs until the one matching the strategy and the key is found. When the strategy is last time edited, it is a bit more complicated, because this information is not saved in the CLEs so it has to iterate through all CLEs to find the one with the oldest and the one with the latest time-stamp. Now that the solution of the actual conflict is found, the batch conflict handler calls the "normal" conflict handler to solve the conflict with the found solution and continues with the next task until all tasks are solved. After this, the handler calls the VisualizationProcessor to generate a new visualization with the solved conflicts and sends it to the NodeJS server, which broadcasts it



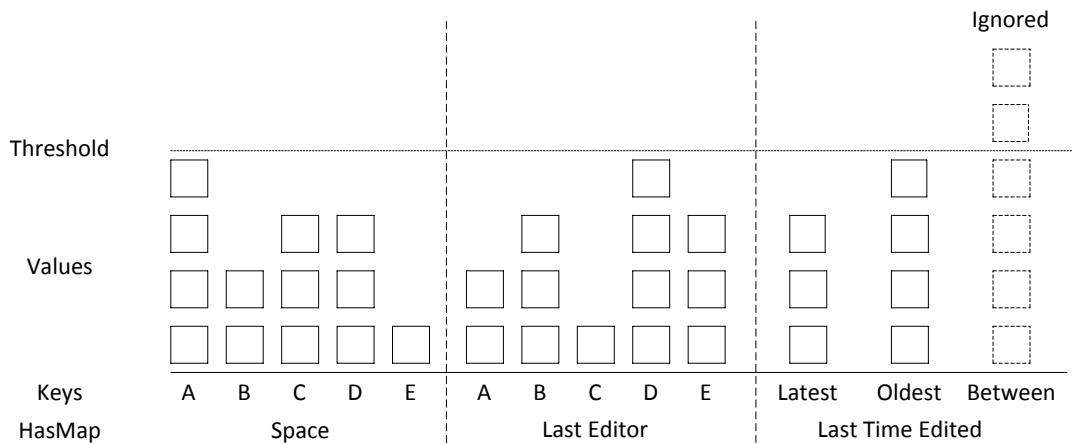


Figure 6.14: Learning Hash Maps Structure Diagram

to all connected clients.

### Handler Calls

The learning handler simply gets called, whenever a `SolveConflict` is executed. The function has all needed variables for the handler and passes them as parameters. When the learning handler detects a strategy, a pop-up will be created. The user now has the option to accept the recommended strategy and by doing so, it will automatically start the batch solving handler to solve all remaining conflicts. When the user doesn't want to accept the strategy he can either just click the pop-up away or disable learning completely. By doing so the `LearningHandler` won't be called again, before he re-opens the whole visualization.

## 6.4 Visualization Implementation

In this section the implementation of the visualization itself is explained, including the data-structure, the data-binding and the generation of the viewpoint. At the end we also describe the implementation details of the type- and instance-filter as it is a central function of the visualization.

### 6.4.1 Implementation Data-Structure

This section is about the data-structure and its methods behind the whole visualization and how they interact with each other. Figure 6.15 shows all important classes, which were implemented for the visualization. The gray classes were implemented by the sebis chair and we used them, therefore they were added to the overview. The following part will first describe the different classes to handle the data and after that the data-binding. The next section then shows, how the data-binding is used to generate the visualization.

All important data is stored in the following classes: "Node" with the sub-classes "InstanceNode" and "SchemaNode", "Attribute" and "Edge".

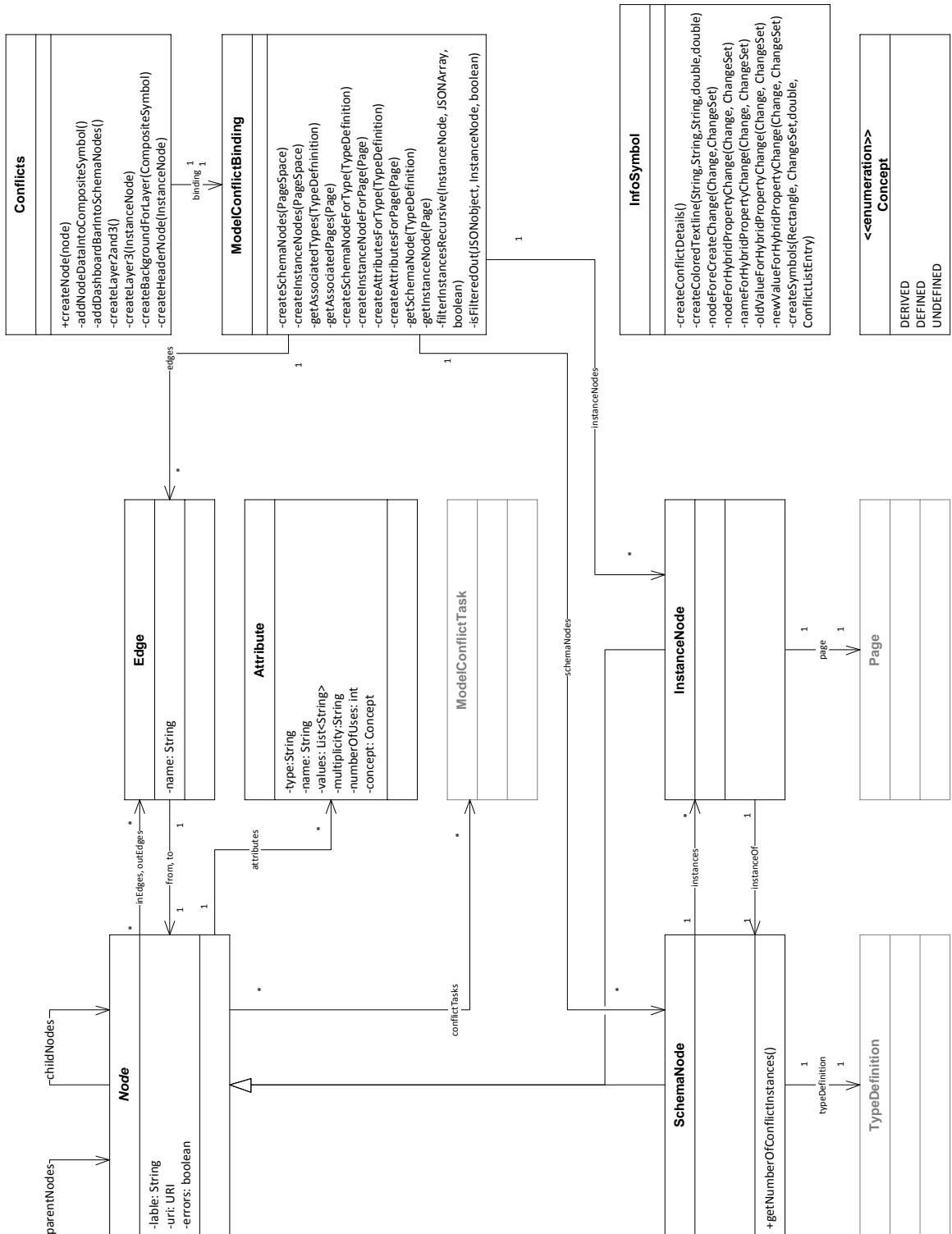


Figure 6.15: Data-structure of the Visualization Implementation

**Node, SchemaNode, InstanceNode**

Node is the abstract super-class of Schema- and InstanceNode. The label is the name of the node, the uri is the unique link to the corresponding TypeDefinition or Page and used for example for the update-function introduced in section 6.3.1. Errors is a boolean, which states whether the Node has unsolved ModelConflictTasks or not. A Node also has a list of Attributes. The class Attribute will be described later. Last but not least a Node can also have in- and outEdges and a list of children Nodes (childNodes) and parent Nodes (parentNodes) to represent associations.

A SchemaNode represents the used data of a TypeDefinition and is used to create the dashboard (cf. section 5.1). Besides a link to the TypeDefinition a Schema Node also has a list of InstanceNodes, which represent a Page of the SchemaNodes TypeDefinition. This list is called instances. The method `getNumberOfConflictInstances()` returns the number, how many of the InstanceNodes in the instances list have an unresolved conflict linked to them.

In an Object of the class InstanceNode all important data from a page is saved. Besides the information already specified in the super-class this is the Page, the InstanceNode is representing, and the SchemaNode representing the TypeDefinition of the Page.

**Edge**

An Edge is a line between two Nodes, therefore it has an attribute from, and an attribute to, for the two Nodes linked together by this particular Edge. An Edge also has a name, which is the name of the association. The name is shown for all Edges between two SchemaNodes.

**Attribute**

An Object of the type Attribute can have up to 6 attributes. Objects of the class Attribute are used to represent both attributes of TypeDefinitions and Pages. The name is the name of the attribute given in the TypeDefinition or the Page. The type of an attribute is set, when the attribute has a type-constraint, as for example date, text or number. When there is no type-constraint is set there are two options, either it is an Attribute of a TypeDefinition, then the type will be set to "No Type Constraint" or it is an Attribute of a Page, then the type will be set to the type of the first value of the attribute. The multiplicity is set as a specific multiplicity like "[1,\*]" when it is defined or set to "[\*,\*]", when there is no definition of the multiplicity. Number of uses is the amount of Pages, which use this attribute and have a value set to it. The concept of an attribute is an enumeration and either DERIVED, DEFINED or UNDEFINED. An Attribute is DEFINED when the attribute is defined in the TypeDefinition. When it is not, it is set to DERIVED when more that 50% of the instances of a single type have this attribute set and UNDEFINED when not. In the list values all values of an Attribute are saved as Strings. This is only used for Attributes of an InstanceNode.

**6.4.2 Implementation Data-Binding**

The Data-binding collects all data from the database and saves it in the data-structure explained above. It first creates the SchemaNodes, after that the InstanceNodes and in the end filters all Nodes according to the passed filters. More information about the implementation of the filter can be found in section 6.4.4. After selecting the correct workspace a SchemaNode is created for every defined type in the workspace. They are all saved in a HashMap, named SchemaNode2TypeDefinition, with TypeDefinition as key and the SchemaNode as value. This is needed for the lookup of nodes associated to the recently generated one and to avoid creating the same node twice. The algorithm is looks like the pseudo code in Figure 6.16. In the createSchemaNodes method the first loop iterates through all types of the given workspace. For every type the method checks, if it is included in the type filter. If it's not, the type gets skipped. After that check, the method gets the SchemaNode for the actual TypeDefinition. For all associated types the

```
1 //Generate all SchemaNodes with Eges:
2 private void createSchemaNodes(Workspace) {
3     for (TypeDefinition td in typeDefinitionsOfWorkspace){
4         if (filter contains td){
5             currentNode = getSchemaNode(td);
6             for (TypeDefinition tdAssociated in AssociatedTypeDefinitionTotd){
7                 if (filter contains tdAssociated){
8                     associatedNode = getSchemaNode(tdAssociated);
9                     new Edge (currentNode, associatedNode, assoziationName)
10                    add Edge to allSchemaEdges;
11                    associatedNode parentNodes add currentNode;
12                    currentNode childNodes add associatedNode;
13                }
14            }
15        }
16    }
17 }
18 }
19
20 //GetSchemaNode:
21 private SchemaNode getSchemaNode(TypeDefinition type) {
22     if (schemaNode2TypeDefinition does not contain key type) {
23         schemaNode = createSchemaNodeForType(type);
24         schemaNode2TypeDefinition put type+ schemaNode;
25         return schemaNode;
26     }
27     return schemaNode2TypeDefinition get type;
28 }
```

Figure 6.16: Create SchemaNodes Pseudo Code

method now looks for the corresponding Nodes and generates the edges between them. Finally the associated Node gets added to the currentNode as child and the currentNode as parent to the associatedNode. The getSchemaNode method works as follows: If the schemaNode2TypeDefinition HashMap contains the TypeDefinition as key, the node is already created and will be returned by just getting the value of the type-key. If the key is not in the HashMap, the Node is created with the method createSchemaNodeForType, which not only creates a SchemaNode but also all Attributes and added to the Type as key and the Node as value are added to the HashMap. After that the method returns the Node. This makes sure that every key is unique and therefore every SchemaNode is created only once.

After that the InstanceNodes are initialized similarly, using Pages instead of TypeDefinitions. There is only one difference, after creating an InstanceNode the corresponding SchemaNode is linked to the InstanceNode as instanceOf and the InstanceNode is added to the instances list of the SchemaNode. For the InstanceNodes the filter is applied after generating all Nodes, to simplify the filter process.

### 6.4.3 Viewpoint Generation

In the following section we will explain the important classes and some of their methods for the viewpoint generation. Namely these are "InfoSymbol" and "Conflicts". Whereas InfoSymbol is only a class to support the Conflict class, where the main part of the algorithm is implemented.

## InfoSymbol

The purpose of the class `InfoSymbol`, is to create the info boxes described in section 5.4. The `InfoSymbol` has a "Symbol" as `VisualObject` and a `ConflictTask` linked to it. With the help of the information provided by both, the class is able to provide a method to create the task specific info-box for the according red exclamation mark symbol. The class has the public method `createConflictDetails(Show)`, which can be called for every `InfoSymbol` and only needs a show-interaction, which will show the box for this particular symbol. There are four different cases of changes, which need to be distinguished. Every change, can be either a `HybridPropertyChange`, `SimpleValueChange`, `CreateChange` or a `DeleteChange`. A `HybridPropertyChange` can either be a change of an association or of one or more `HybridProperty` values. In the first case the `InfoSymbol` will use the `createNode()` method provided by the `Conflicts` class described below to draw a node-symbol, to which the new association points. In the second case the visualization will look the same as for a `SimpleValueChange`. The name of the attribute with a colored text, which marks the actual value as green and the old value as stroke-through, as value will be shown. This text element is generated by a method also implemented in the `InfoSymbol` class. The method needs the name of the attribute, the old and the new value and the actual position on the visualization in `x` and `y` and returns the whole text line as `CompositeSymbol`. For the differencing between old and new value the `StringUtils.difference` is used, which is provided by the Apache commons package<sup>1</sup>. A `CreateChange` is processed equal to a `HybridPropertyChange`, when there is an association change. It will also visualize the new created Object as `Node`, to provide as many information as possible. For a `DeleteChange` the printout will just be a text stating, that the page has been deleted. These were all the important facts about the class `InfoSymbol`, in the now following sections, the class `Conflicts` will be described.

## Conflicts

The class `Conflicts` generates the main part of the visualization, it gets the data input from the data-binding (cf. section 6.4.2) and uses the class `InfoSymbol` as support. First it creates the dashboard containing all `SchemaNodes` and `Edges`, their conflicts and the status bars, with the help of a graph lay-outer and a special method for the bars. The graph lay-outer needs nodes and edges as input and will draw a well structured graph. As it is not our implementation, we won't describe the lay-outer itself in detail in this thesis. When the dashboard is finished the second and third layers are created for every `InstanceNode`. To layout the second layer, a table lay-outer is used. After specifying the number of rows and columns and the elements the cells are containing, in our case the cells contain the visualization of an `InstanceNode`, it arranges all the elements in a table with equal sized cells. After a cell is set, the third layer will be generated for every cell. For the third layer we use the graph layouter again, but this time with `InstanceNodes` and their edges instead of `SchemaNodes`. There is a third layer for every `InstanceNode`, with the `InstanceNode` and its direct children. One method, which is called quite often within the class is the `createNode(Node)` method, as it is used in every layer. Due to this fact, it will be described in more detail in the following.

The method `createNode(Node)` returns a `CompositeSymbol` with all data included in the given parameter `Node`. These symbols are the base objects of the visualization as they represent a `Type-Definition` or a `Page` on the visualization server. It does not matter, whether the parameter is a `SchemaNode` or an `InstanceNode`, the method will check this by itself and visualize the data accordingly. When it's an `SchemaNode` the label will be the name of the label of the node plus the number of instances shown in the visualization and the attributes will consist of the name, the number of uses, multiplicities and type constraints. In contrast, the attributes will be shown as name and value, when the handed `Node` is an `InstanceNode`.

<sup>1</sup><http://commons.apache.org/proper/commons-lang/javadocs/api-2.6/org/apache/commons/lang/StringUtils.htm>, Last Accessed 09.12.2013

#### 6.4.4 Instance- and Type-Filter

In the following section we will describe the implementation of the type and instance filter, as shown in section 5.5. First we will have a look at the type- and afterwards at the more complex instance-filter.

##### Type-Filter

The implementation of the type-filter is as follows: The configuration of the visualization has an array called "allTypes", containing all different types of pages included in the visualization and another one called "typeFilter" only containing the types, which should be displayed in the actual visualization. These arrays are generated and filled in the first call of the data-binding. On the basis of this data the visualization is able to generate the filter dialog shown in Figure 5.8. The type-filter is a white-list filter, therefore every type, which is included in the filter will be shown in the visualization. Whenever a filter is applied the visualization will execute an update. When the typeFilter is modified, the data-binding will then only generate nodes for the types, which are still included in the filter, this is implemented with a simple if statement before a node is created. Without the nodes, the edges wont be generated as-well and the visualization will not show the filtered types.

##### Instance-Filter

For every type it is also possible to filter its instances with the help of a recursive algorithm. All instance filters consist of the following parts: the link operation to other filters, the filtered attribute, the type of comparison, the string or number which is compared with the attribute value and they can have further sub-filters with the same attributes again. The filters are always linked to exactly one type. The instances of this type will be filtered. On code level this filter is represented by JSON objects and arrays. Figure 6.17 shows the structure in an UML class diagram. The tyeOfComparison enumeration is adjusted to the type of the attribute. For instance if the filtered attribute has the type date, than only "=", before, after and NotNull are available options for the comparison, and the comparisonString has to be in the format of DD.MM.YYYY.

This filter structure is fully recursive, so it is always possible to add another sub-filter to filter a already filtered amount of instances again. When applying the filter, the data-binding will filter out all instances with the method, shown in Figure 6.18. The filter method always gets a single instance, the filterArray according to the type of the instance and a boolean, if the instance is filtered out or not, as parameters. On the first call of the method this boolean is set to false, so the instance would not be filtered out. In the loop, the method "isFilteredOut" is called for every filterObject within the filterArray, it retrurns a boolean whether the instance is filtered out for the actual filter-rule or not. The if-statement in line 6 checks if there are sub-filters stored in the actual filterObject. If the attribute attrFilter contains further filters it will call the whole method recursively and set the isFilteredOut boolean accordingly. After checking every rule, the filterInstanceRecursive method will also return a boolean which is set to true if the instance is filtered out and therefore not shown in the visualization.

In the following part we will have a closer look at the isFilteredOut method with the help of the simplified flow diagram shown in Figure 6.19. Simplified means, that not all loops and assignations of variables are shown, because the diagram would become too complex and to understand the idea behind the method they are not needed. The method needs the following parameters as input: the instance for which the filter-rules are checked, the filter-rule in form of a JSON object and the old isFilteredOut value, which determines whether the instance would be filtered out be the rules applied before. First the method checks, whether the instance as a value set to the filtered attribute. If not, it will directly set the isFilteredOutForThis rule to false, because there is no rule, which could be fulfilled. The next step is to check, if the typeOfComparison equals "NotNull". If it does

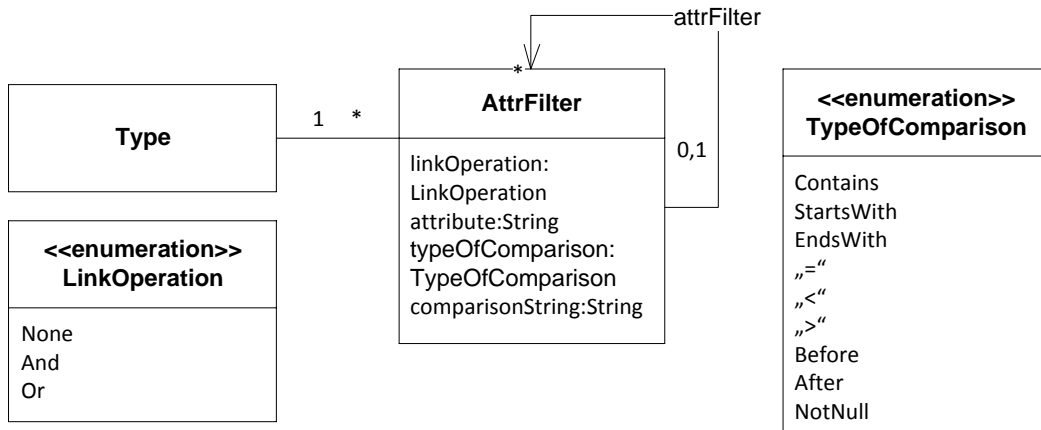


Figure 6.17: Instance Filter Class Diagram

```

1 | private boolean filterInstanceRecursive (InstanceNode instance, JSONArray filterArray,
2 | boolean isFilteredOut){
3 |     try {
4 |         for (int i = 0; i < filterArray.length(); i++){
5 |             JSONObject filterObject = (JSONObject) filterArray.get(i);
6 |             isFilteredOut = isFilteredOut(filterObject, instance, isFilteredOut);
7 |             if (((JSONArray) filterObject.get("attrFilter")).length() > 0){
8 |                 isFilteredOut =
9 |                     filterInstanceRecursive(instance, (JSONArray) filterObject.get("attrFilter"),
10 | isFilteredOut);
11 |             }
12 |         } catch (JSONException e) {
13 |             e.printStackTrace();
14 |         }
15 |         return isFilteredOut;
16 |     }
17 | }

```

Figure 6.18: Instance Filter Java Implementation

the instance will not be filtered out for this rule. The next step is checking if the instance fulfills the filter rule. This is the main part of the method, due to the fact that the rules are different for every typeOfComparison. For the types "contains", "startswith", "endswith" and "=" there is a simple string comparison of the attributes value and the comparisonString including in the filter rule. "<" and ">" are only selectable, when the attributes values are all numbers. Because of this, the comparisonString has to be a number as-well and gets converted into one. Now the method is able to use the simple comparisons >and <. For attributes of the type date, the string has to be in the format described above and a date-formatter is used to compare the values and the given comparisonString. After evaluating the filter rule, the instance is either filtered out for this method or not. Now the method checks the linkOperation and depending on this, the return will be false, true or the unchanged value which was passed as parameter. The link operations "and" and "none" are logically the same in this method, this is why "none" is not shown in the diagram.



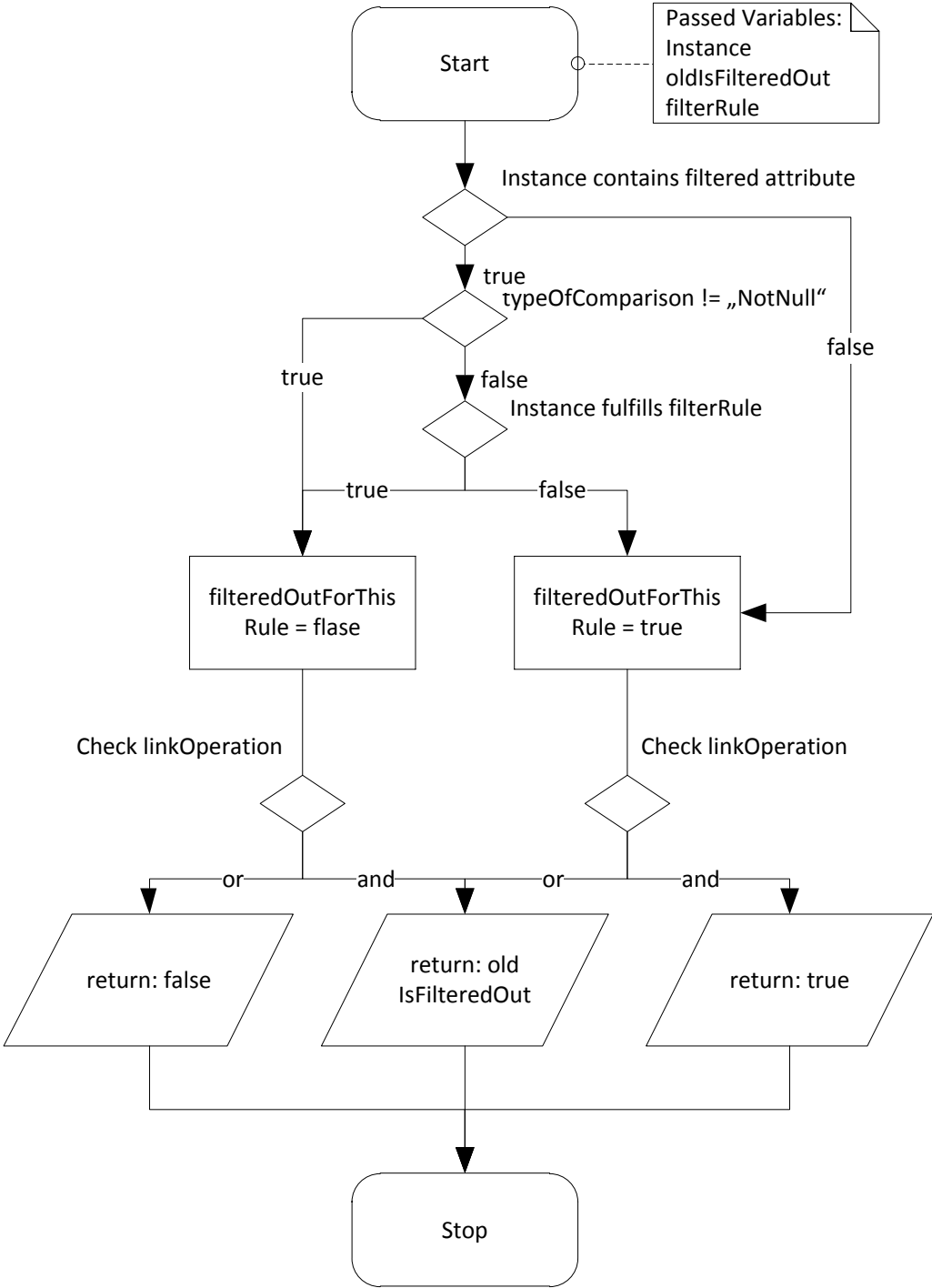


Figure 6.19: Instance Filter Flow Diagram



# 7 Conclusion and Outlook

To conclude the work, the results and the questions that came up during the implementation and research are presented in this chapter. In Section 7.1 the whole work will be summarized and in Section 7.2 the result of this work is critically reflected and further research options are pointed out.

## 7.1 Summary of the Work

All the previous chapters of this thesis contributed to the following research question:

How to provide (interactive) visual means to communicate and resolve model conflicts?

The following section will give a short summary of every single chapter and how it helped to find an answer for the question.

Chapter 1 just gave a short introduction and motivated this thesis, introducing the research questions and explaining possible conflicts, which could arise during model merges. The chapter also points out the weaknesses of the actual state of the art and explains the outline of this work.

Chapter 2 presents the foundations for our own visualization. In section 2.1 the reader was introduced to interactive visualizations and a framework behind them. Furthermore NodeJS and TogetherJS, two technologies used for the visualization prototype were explained.

Chapter 3 presented the results of a literature study. The study was about model merge and text merge programs and how they visualize the conflicts occurring during the merges. The main outcome of the research is summarized in two tables at the end of the chapter.

The fourth chapter includes all possible use-cases of an interactive visualization to solve model conflicts. Furthermore it describes how our implementation provides the needed functionality to enable the use-cases. Section 4.7 includes a matrix of all use-cases and two which degree they can be handled by the visualization and which section deals with which use-case.

In chapter 5 the design of our visualization was presented. The chapter includes several mock-ups of the implemented functions. Besides the three main layers "Dashboard" (Section 5.1), "Instance Overview Layer" (Section 5.2) and "Instance Detail Layer" (Section 5.3) the layout of the filter and of different info-boxes were presented.

The last chapter, chapter 6, explained many details of our prototype implementation. First the interface to the conflict tasks, which are created by the merge algorithm, was described. Section 6.2 described the visualization framework, which was the basis of our implementation. As the functionality provided by the framework did not meet all our needs, section 6.3 explains the different framework extensions we implemented. Namely these extensions were a better update function (Subsection 6.3.1), the introduction of compensable interaction in subsection 6.3.2, the possibility to broadcast visualizations to other users (6.3.3), the important functionality to solve a conflict (Subsection 6.3.4) and at the end the algorithm for learning and batch solving in subsection 6.3.5. The other section of this chapter explains the implementation of the visualization. Starting with the data-structure in subsection 6.4.1, which is the basis for the data-binding explained in 6.4.2, at the end the generation of the viewpoint (Subsection 6.4.3) and how exactly the filter works (Subsection 6.4.4) is described.

Table 7.1: Use Case Analysis Matrix

Use-Case	Functionality provided	Further Explanation in Chapter
Resolve Multiple Conflicts	●	6.3.5
Resovle One Conflict	●	6.3.4
Resolve Instance Conflict	●	5.2, 5.3
Resolve Schema Conflict	●	5.1
Resolve Schema/Instance Conflict	●	5.1, 5.2, 5.3
View Conflict Summary	●	5.1
Drill Down Conflict	●	5.3
Solve Conflict Collaboratively	●	6.3.3, 2.3
Discuss Conflict	●	2.3
Comment Conflict	○	-
Forward Conflict	◐	4.4
Postpone Conflict	◐	4.5
Create Conflictt	○	-

## 7.2 Critical Reflection and Outlook

There are different aspects about the implementation, which could be improved and there are also possibilities for further research, which will be pointed out in the following section. The most important thing, what could be done in the future is an evaluation of the visualization. To bridge the gap between research and industry the whole implementation should be tested and commented by an industry partner, to see whether the offered functionalities are used and function as intended. Based on this evaluation further improvement of the implementation could be done.

Another aspect, which could be improved is the implementation itself. There are functions described in the use case diagram in Figure 7.1, which are not yet supported by the implementation. These could be implemented and evaluated as-well. Another thing, that could be improved is the batch solving. At the moment it is only possible to use the batch-solving after a certain amount of conflicts is already solved. It should be possible to use it from the beginning, designating the solution strategy as user and not accepting the one from the learning algorithm. Furthermore the NodeJS server broadcast functionality could be implemented in the visualization server as-well to reduce traffic and to shorten latencies between the different clients. This could lead to a better usability as the visualizations will be synchronized faster. Another aspect, which could be added in the visualization are some page specific properties, as for example the last editor or the time of the last change. At the moment it only shows the hybrid properties directly set on the page but no meta information. Adding this would improve the rate of detail and might be helpful to solve conflicts. Last but not least the performance of the implementation itself could be improved by implementing more efficient algorithms, as this was not possible due to the time constraints we had.

# Bibliography

- [BSW<sup>+</sup>09] Petra Brosch, Martina Seidl, Konrad Wieland, Manuel Wimmer, and Philip Langer. We can work it out: Collaborative conflict resolution in model versioning. In *ECSCW 2009*, pages 207–214. Springer, 2009.
- [BSWK12] Petra Brosch, Martina Seidl, Manuel Wimmer, and Gerti Kappel. Conflict visualization for evolving uml models. *Journal of Object Technology*, 11(3):2–1, 2012.
- [Eib] Joachim Eibl. Kdiff3. <http://kdiff3.sourceforge.net/>. Last Accessed:10.07.2013.
- [FAB<sup>+</sup>11] M. Farwick, B. Agreiter, R. Breu, S. Ryll, K. Voges, and I. Hanschke. Requirements for automated enterprise architecture model maintenance. In *13th International Conference on Enterprise Information Systems (ICEIS), Beijing, China, 2011*.
- [Fou] The Eclipse Foundation. Emf compare. <http://www.eclipse.org/emf/compare/>. Last Accessed: 05.07.2013.
- [Inca] Dell Software Inc. Toad data modeler. <http://www.toadworld.com/products/toad-data-modeler/default.aspx>. Last Accessed: 08.07.2013.
- [Incb] Google Inc. Google trends nodejs. <http://www.google.de/trends/explore#q=nodejs&cmpt=q>. Last Accessed: 22.11.2013.
- [Incc] Joyent Inc. Nodejs. <http://www.nodejs.org>. Last Accessed: 22.11.2013.
- [Kir] Björn Kirschner. Master’s thesis. Unpublished.
- [KKL<sup>+</sup>10] Petra Kaufmann, Horst Kargl, Philip Langer, Martina Seidl, Konrad Wieland, Manuel Wimmer, and Gerti Kappel. Representation and visualization of merge conflicts with uml profiles. In *Proceedings of the International Workshop on Models and Evolution (ME 2010) @ MoDELS 2010*, pages 53–62. Online Publication, 2010.
- [KKOS12] Timo Kehrer, Udo Kelter, Manuel Ohrndorf, and Tim Sollbach. Understanding model evolution through semantically lifting model differences with silift. In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, pages 638–641. IEEE, 2012.
- [Lab] Mozilla Labs. Togetherjs. <https://togetherjs.com/>. Last Accessed: 10.12.2013.
- [Mic] Microsoft. Outlook 2013. <http://office.microsoft.com/de-de/outlook/>. Last Accessed: 01.11.2013.
- [Par] Visual Paradigm. Visual paradigm for uml. <http://www.visual-paradigm.com/product/vpum1/>. Last Accessed: 07.07.2013.
- [Per] Perforce. P4merge. <http://www.perforce.com/product/components/perforce-visual-merge-and-diff-tools>. Last Accessed:07.07.2013.
- [Rau] Guillermo Rauch. Socket.io. <http://www.socket.io>. Last Accessed: 10.11.2013.

- [RHM<sup>+</sup>13] Sascha Roth, Matheus Hauder, Felix Michel, Dominik Münch, and Florian Matthes. Facilitating conflict resolution of models for automated enterprise architecture documentation. 2013.
- [SMR12] Michael Schaub, Florian Matthes, and Sascha Roth. Towards a conceptual framework for interactive enterprise architecture management visualizations. In *Modellierung*, pages 75–90, 2012.
- [Sou] SourceGear. Diffmerge. <http://www.sourcegear.com/diffmerge/>. Last Accessed: 12.07.2013.
- [UL] Uni-Leipzig. Eclipse model repository. <http://modelrepository.sourceforge.net/compare-merge-models.html>. Last Accessed: 06.07.2013.