Endbericht der Projektgruppe 436
VITEOS

Andreas Lepper
Christian Grimme
Dennis Schweizer
Hasan Akbulut
Louis-Marie Nguenkam Ngamo
Matthias Schley
Narssis Romain Watal-Fonkeu
Nicolas Nwabueze
Robert Petermeier
Stefan Schwedt
Wolfgang Paul

17. Dezember 2004

Inhaltsverzeichnis

Abbildungsverzeichnis

Teil I

Entwicklung des Projektes *VITEOS*

Kapitel 1

Vorbereitende Ideen und Konzepte

Modelländerungen auf höheren Abstraktionsebenen innerhalb von Softwareentwicklungsprozessen, die einem "Wasserfall-Modell"-artigen Vorgehen nachempfunden sind, ziehen zwangsläufig Veränderungen all jener Modelle mit sich, die auf einer niedrigeren Abstraktionsebene liegen. Ein einfaches Beispiel wäre die Erweiterung eines Analysemodells. Eine solche Erweiterung erzwingt eine Erweiterung und gegebenenfalls auch eine Umstrukturierung aller Modelle (Entwurf,. . . , Implementierung, . . .), die auf diesem Analysemodell aufbauen.
Ziel der Projektgruppe war ein Editor zu entwickeln, der eine von der herkömmlichen UML-Syntax losgelöste, objektorientierte Modellierung ermöglichen soll. Dieser Editor soll in der Lage sein, auf eine neue Art die Beziehungen zwischen Klassen zu modellieren, da die herkömmlichen Techniken insbesondere bei komplexen Modellen schnell unübersichtlich werden. Folgende Aspekte spielen bei der Entwicklung eine Rolle:

- Der Editor soll Versionierung unterstützen, d.h. dass es analog zu Versionsmanagement-Werkzeugen wie CVS möglich sein soll, Änderungen am Modell rückgängig zu machen und die Entwicklung an älteren Versionen fortzusetzen. Dazu wurde ein Weg gefunden, Elemente des Analyse-Modells auf Elemente bzw. Mengen von Elementen des Entwurfsmodells abzubilden.

- Die Semantik von UML sollte (weitgehend) erhalten bleiben. Das ist sinnvoll, da der Editor zur Entwicklung von Software nach objekt-orientiertem Paradigma dienen soll und sich UML hierbei bewährt hat.

- Die Syntax von UML hat sich jedoch nicht bewährt, wenn Änderungen an einem Modell automatisch auf andere Modelle übernommen werden soll. Der Grund hierfür ist, dass in der zweidimensionalen grafischen Darstellung insbesondere bei komplexen Modellen die Übersicht verlorengeht. Aus diesem Grund haben wir eine neue Form der Darstellung geschaffen, die diesen Aspekt berücksichtigt.

- Die Software-Technologie unterteilt den klassischen Entwicklungsprozess grob in die Phasen *Analyse, Entwurf* und *Implementierung.* Da der von uns entwickelte Editor v.a. in der Analyse- und Entwurfsphase zum Einsatz kommen soll, soll er ermöglichen, dass sich Änderungen am Analyse-Modell *automatisch* im Entwurfs-Modell wiederfinden und umgekehrt.

Um uns über die Anforderungen an diese Abbildung der Änderung von Modellen auf andere Modelle klar zu werden, haben wir im ersten Semester unserer Projektgruppe in zwei Gruppen das Referenz-Projekt *OpenXMotD* entwickelt. Dabei sollte sichergestellt werden, das möglichst viele Änderungswünsche eines imaginären Kunden während der Entwicklung auftreten, die Änderungen an den Modellen erfordern. Das Referenzprojekt *OpenXMotD* diente desweiteren den Projektgruppenteilnehmern in erster Linie zur Vergegenwärtigung einiger Probleme, die entstehen können,

wenn Änderungen von Anforderungen an bestehende Software eine solche Umgestaltung und Erweiterung von Analyse-, Entwurfs- und Implementierungsmodellen erfordern. Die im Referenzprojekt entstandenen "Modelländerungsdurchläufe" wurden analysiert und Strategien entwickelt, die eine Automatisierung solcher "Durchläufe" ermöglichen und die Bewältigung anfallender Probleme unterstützen sollen. Dieser Referenz Entwicklungsprozess wurde dabei dokumentiert und fließt in die Entwicklung des Editors ein.
Da nicht nur die *OpenXMotD*-Entwicklungphasen in zwei verschiedenen Gruppen (Blau und Rot) stattfanden, sondern auch die anschließende Analysephase, entstanden wie zuvor gesehen auch zwei Konzepte. Die beiden genannte Konzepte sahen einen gezielten Einsatz von Versionierungstechniken vor. Sie wurden nach den Präsentationen diskutiert, weiter verfeinert und anschließend miteinander vereinigt. Das daraus entstandene Konzept legte den Grundstein zum eigentlichen Entwicklungsbeginn von *VITEOS* .
Der weitere Verlauf der PG wurde durch die Realisierung von 3 Releases bestimmt, wobei für jedes Release ein Auslieferungstermin festgesetzt wurde. Als Release wird hier eine konsistente Menge von Elementen, sogenannten Viteos-Komponenten bezeichnet, die als Ganzes die spezifierten Anforderungen erfüllt. Das erste Release wurde für den 28. Mai 2004 geplant, das zweite für den 30. Juni 2004 und das dritte und letzte für den 31. Juli 2004.

1.1 Versionierung und Datenhaltung

1.1.1 Versionierungsidee

Der Versionierungsmechanismus für das Tool *VITEOS* basiert auf einer tabellarischen Beschreibung der zu versionierenden Objekte. Die Gruppe entschied sich für eine tabellarische Darstellung, da sie davon überzeugt ist, dass dies die Datenhaltung per SQL erheblich vereinfachen würde. Zudem erlaubt diese Darstellung eine möglichst overhead-arme Datenhaltung und bietet effizienten Zugriffsoperationen für die Versionierungsmechanismen. Die für das Projekt versionierte Objekten sind:

- Konfigurationen
- Klassen
- Eigenschaften von Klassen
- Assoziationen
- Methoden
- Attribute
- Releases
- Sinngruppen

In dem folgenden Abschnitt wird das von uns konzipierte Objekt - Sinngruppe - näher erläutert. Ansonsten werden die versionierte Objekten (Tabellen mit ihren aktuellen Anzahl und Belegung der Spalten) in den verschiedenen Releases beschrieben.

1.2 Modellierungskonzepte und Benutzervorgehen

Der Begriff, *Sinngruppe*, ist ein Bezeichner für eine Menge von sinngemäß zusammenhängend markierten Klassen.

1.2.1 Erstellen von Sinngruppen/Klassen

Bei der Herstellung von Beziehungen zwischen Klassen in verschiedenen Sinngruppen führt das Werkzeug beim Überschreiten der Gewichtungsschwelle (Automatisierungshorizont) eine Vereinigung der Sinngruppen durch. Die Bildung einer Unter-Sinngruppe wird ggf. vorgeschlagen (Klassen mit starkem Zusammenhang in einer Gruppe könnten eine Untersinngruppe bilden). Sinngruppenzugehörigkeit kann frei eingefügt werden, mit der Einschränkung bei der Gewichtsschwelle der Assoziationen, die nicht umgangen werden kann. Ebenfalls kann eine Klasse wieder aus einer Sinngruppe entfernt werden. Überlagerungen von Sinngruppen können bei diesem Vorgehen automatisch entstehen (siehe auch Abb. 1.5). Freie Assoziationen werden von Sinngruppen nicht beachtet. Sie ergeben erst durch angehängte(inzidente) Klassen Sinngruppen.

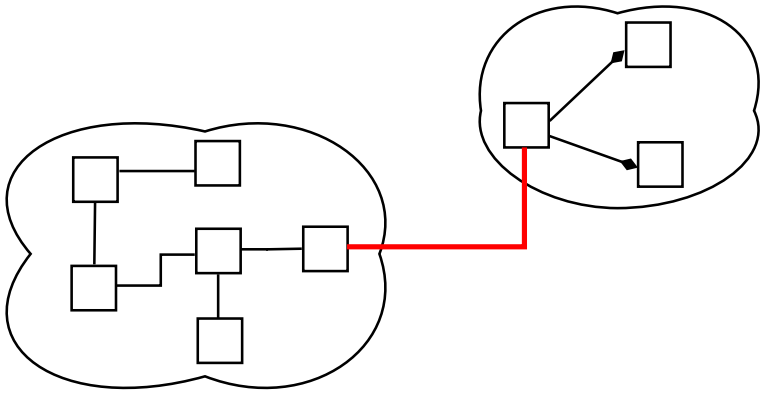

Abbildung 1.1: Die Sinngruppen werden ab einer bestimmten Assoziationsstärke (rot) vereinigt

Betrachtung des Hinzufügens von Beziehungen (Restriktionen)

Beim Hinzufügen von Kanten, die die Gewichtsschranken überschreiten, wird folgendermaßen vorgegangen:

I. Atomares Hinzufügen einer Klasse

- Klasse wird unabhängig von der Richtung der Assoziation in die Sinngruppe eingefügt.
- Der Name der Sinngruppe bleibt erhalten.
- Die Klasse kann auch in den Durchschnitt mehrerer Sinngruppen gezogen werden, wie es in Abbildung 1.2 zu sehen ist.

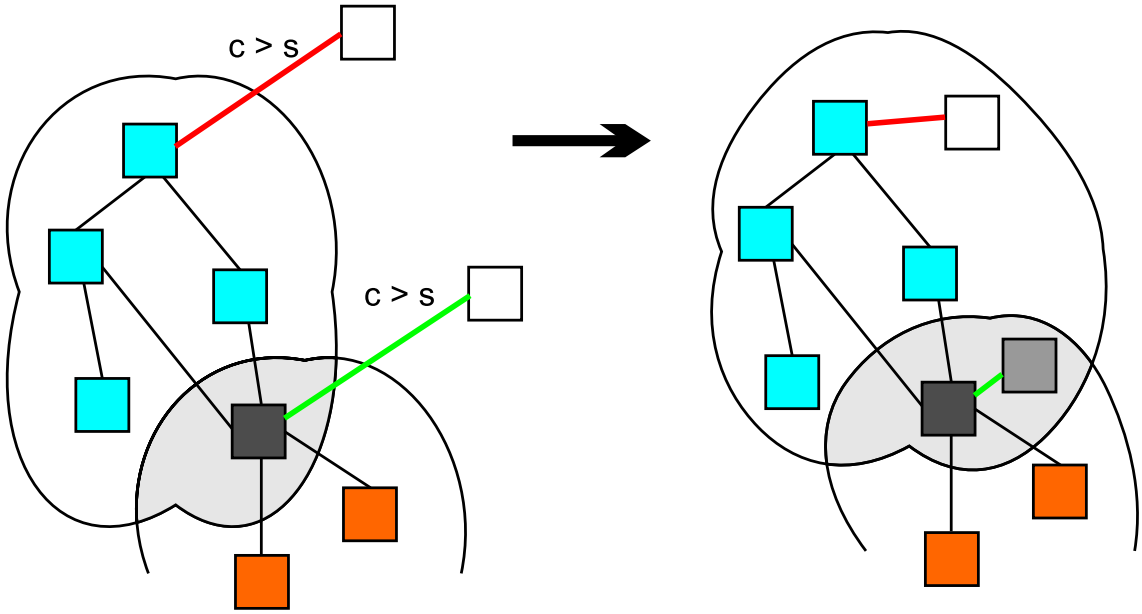

Abbildung 1.2: Die Klasse wird in die Sinngruppen einbezogen (analog im Schnitt)

II. Herstellen einer Beziehung zwischen zwei Sinngruppen (disjunkt oder nicht disjunkt) Es gibt zwei Möglichkeiten, auf das Einfügen einer Assoziation (Meta) mit $c > s$, wobei s Gewichtungsschranke, c Gewicht der Assoziation, zu reagieren:

1. Vereinigung der zwei Sinngruppen
   - Die Vereinigung der Sinngruppen wird auch auf den Namen der Sinngruppen durchgeführt. Siehe Abbildung 1.3
2. Eine Sinngruppe wird Unter-Sinngruppe der anderen wie es in Abbildung 1.4 geschildert ist:
   - Es wird dabei die Richtung der Assoziation berücksichtigt.
   - Die Namen der Sinngruppen bleiben erhalten.
   - Assoziationen mit $c > s$ sind zwischen Sinngruppe und Unter-Sinngruppe erlaubt.

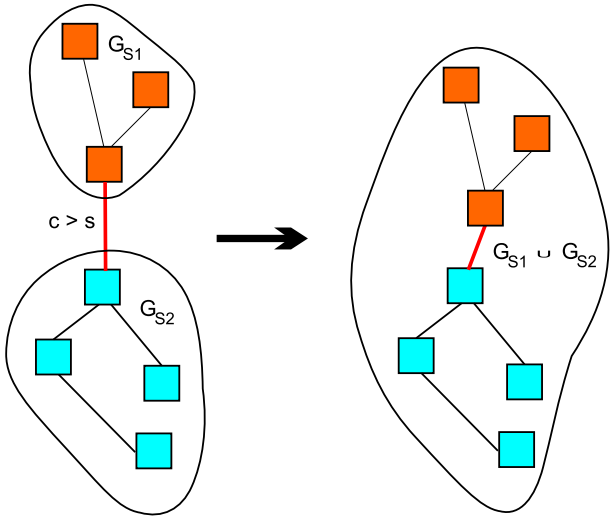

Abbildung 1.3: Die Sinngruppen G_{S1} und G_{S2} werden vereinigt

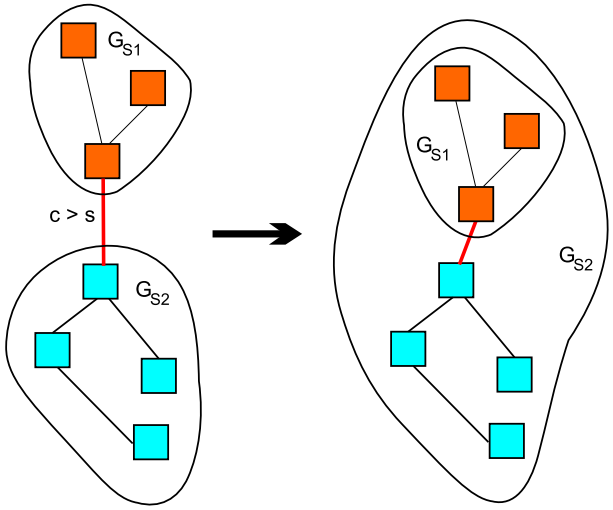

Abbildung 1.4: Die Sinngruppe G_{S1} wird Unter-Sinngruppe von G_{S2}

1.2.2 Löschen von Sinngruppen/Klassen

Wird die letzte Klasse einer Sinngruppe gelöscht, wird gefragt, ob die Sinngruppe ebenfalls gelöscht werden soll. Andernfalls wird die Klasse und die mit ihr inzidenten Assoziationen gelöscht. Beim Löschen von Sinngruppen ist (siehe Abb. 1.5) zu beachten, dass Klassen im Schnitt mit anderen Sinngruppen generell nicht entfernt werden, außer nach expliziter Bestätigung.

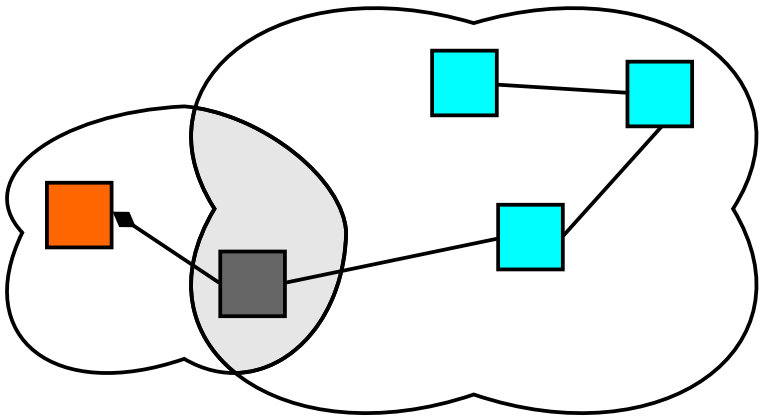

Abbildung 1.5: Die Klasse im Schnitt der Sinngruppen wird nur auf Benutzerwunsch entfernt

1.3 Das Grafische Benutzer Interface (GUI)

1.3.1 UML-Perspektive

Die UML-Perspektive soll den Zusammenhang zwischen Klassen darstellen. Dazu wird jedoch nicht auf die gewöhnliche, graphische Darstellung zurückgegriffen, sondern es wird wie im Folgenden beschrieben vorgegangen.

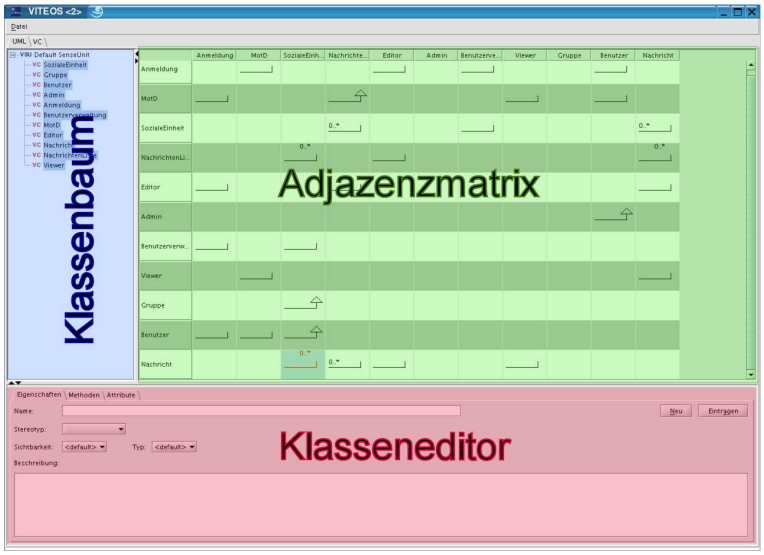

Abbildung 1.6: Idee für die Darstellung der UML-Perspektive

In Abb. 1.6 wird ein Anhaltspunkt für die Darstellung dieser Perspektive gegeben. Ihre Aufteilung soll hier kurz beschrieben werden.

- **Oben links** befindet sich eine Darstellung des Modells als Baumstruktur. Der Klassenbaum siehe Abb. (1.6) beginnt mit einer künstlichen Sinngruppe (*genannt DefaultSenceUnit*), welches als Startknoten fungiert. Die Struktur des Modells ist im Grunde ein Graf, wo die Elemente in verschiedene Sinngruppen unterteilt sind. Überträgt man diesen Graf in eine Baumstruktur lässt es sich nicht vermeiden, das einzelne Elemente in mehreren Sinn- und

Untersinngruppen mehrfach auftauchen. Das heisst, wenn ein Element in mehreren Sinngruppen des Modells vorkommt, so existieren im Baum mehrere Knoten die dieses Element repräsentieren.

- **Oben rechts** wird das Modell als Matrix dargestellt. Wir bezeichnen diese Darstellungsform im weiteren immer mit dem Begriff der Adjazenzmatrix. Die Adjazenzmatrix soll individuell anpassbar sein (mehr dazu im entsprechenden Abschnitt)
- **Unten über die ganze Breite** befindet sich ein Bearbeitungsfenster, in dem das gerade aktivierte Element bearbeitet werden kann. Da wir uns in der UML-Perspektive befinden, sollten hier nur Klassen bearbeitet werden.
- Über den vier Fenstern befindet sich eine **Menüleiste** (sieh Abb. 1.6) für verschiedene Funktionen wie zum Beispiel das Einfügen von Klassen und Assoziationen.

1.3.1.1 Adjazenzmatrix

Die in Abb. 1.7 gezeigte Adjazenzmatrix stellt den Zusammenhang der Klassen des Diagramms dar. Sie kann folgendermaßen beschrieben werden:

Abbildung 1.7: Beispiel einer Adjazenzmatrix

- **Tabelleneinträge sind Icons**, die die Art der Assoziation zwischen den Klassen angeben.
- Die **Leserichtung der Assoziationen** ist festgelegt. Die erste Spalte der Matrix gibt diejenigen Klasse an, die die Quelle der Assoziation ist. Die Klassen der ersten Zeile sind die Senken aller Assoziationen.
- Die **Reihenfolge**, in der die Klassen angeordnet werden, sollte vom Benutzer zu beeinflussen sein. Dies sollte sogar soweit gehen, dass der Benutzer mit Hilfe einer Art von Zoom die Senken (Klassen, bei denen eine Assoziation oder Vererbung endet) Auswählen kann. Damit kann er einen kleineren Ausschnitt betrachten. Weitere Erläuterungen zu dem Vorgehen finden sich im Abschnitt der Baumstruktur (Kapitel 1.3.1.2).
- **Regeln:** eine Adjazenzmatrix darf nur quadratisch sein ($n \times n$). Die einzige Ausnahme bildet der Anzeigemodus, der zulässt, dass einige ausgewählte Senken und alle Quellklassen angezeigt werden. Betrachten wir mal die Abb. 1.7, angenommen wir wollen im Anzeigemodus die Klasse23 und die dazugehörigen Relationen näher betrachten. Als Bild wird dann die einzige Zeile Klasse23 und die Spalten Klasse1, Klasse2 und Klasse5 visualisiert.

1.3.1.2 Baumkomponente

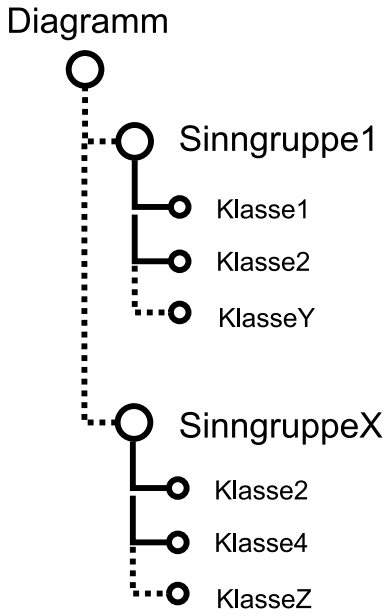

Abbildung 1.8: Allgemeine Darstellung des Gruppen- und Klassenbaums

- **Aufbau:** Die Wurzel des Baumes, siehe Abbildung 1.8, repräsentiert das gesamte Modell. Wird dieser Knoten geöffnet, erscheinen alle Gruppen auf höchster Ebene sowie alle Klassen, die in keiner Gruppe sind. Dabei werden Gruppen durch Knoten und Klassen durch Blätter dargestellt. Wird eine Gruppe geöffnet, werden wiederum alle Untergruppen dieser Gruppe und alle Klassen, die in keiner Untergruppe sind, angezeigt. Klassen können im Baum demnach mehrfach auftreten, wenn sie in mehreren Gruppen sind.

- Das **Auswählen** eines Elementes in der Baumstruktur sollte dazu führen, dass in der Tabelle automatisch zum ausgewählten Element gesprungen wird.

- Zusätzlich soll eine **Mehrfachauswahl** von Elementen möglich sein. Dies führt zu einer Neustrukturierung der Adjazenzmatrix. Durch das Auswählen von Elementen des Baumes wird eine Selektion bezüglich der UML-Anzeige durchgeführt. Es werden nur die Elemente in der Adjazenzmatrix angezeigt, die ausgewählt sind:

  1. Auswahl des obersten Knoten (Diagramm): Es wird das gesamte Diagramm angezeigt.
  2. Auswahl einer Sinngruppe: Die Anzeige innerhalb der Adjazenzmatrix wird auf die Klassen innerhalb der Sinngruppe beschränkt.
  3. Auswahl einzelner Klassen: Es werden nur einzelne Klassen in der Adjazenzmatrix dargestellt.

  Die beschriebenen Aktionen zur Auswahl können natürlich beliebig kombiniert werden, so dass genau der Ausschnitt angezeigt wird, auf dem der Benutzer arbeiten möchte (z.B. Auswahl mehrerer Sinngruppen oder das Hinzufügen einzelner Klassen zur Ansicht). Das Anzeigen der ausgewählten Elemente geschieht

  1. als die ausgewählten Elemente selbst, oder

2. als alle Elemente des Diagramms.

Die gewünschte Art kann evtl. über einen Button in der entsprechenden Klassenbaum gewählt werden. Die Klassen, gegen die aufgetragen wird, werden in der Adjazenzmatrix in der ersten Spalte dargestellt.

- Der Baum soll in dieser Perspektive die **Operationen** Hinzufügen und Entfernen von Klassen unterstützen. Die Operationen finden im Baum statt.
  - Eine Klasse kann über den Diagramm-Knoten ohne Gruppenbindung in das Diagramm eingefügt werden.
  - Eine Klasse kann direkt in eine Sinngruppe eingefügt werden.
  - Wird eine Klasse gelöscht, so muss sie automatisch aus allen Gruppen entfernt werden.

1.3.2 Sinngruppen-Perspektive

Die Sinngruppen Perspektive legt eine besondere Betonung auf die Sinngruppen-Struktur im Diagramm. Hier soll diese Struktur angezeigt, aber auch verändert werden können. Wir schlagen wieder einen Aufbau wie in Abb. 1.6 vor, gehen also auch von einer analogen Baumstruktur, wie in Abb. 1.8 aus. Trotzdem beherbergt der Baum in dieser Perspektive andere Funktionalitäten:

- Sinngruppe erstellen / löschen / aufheben / einbetten (Unter-Sinngruppe) / verschieben
- Klasse erstellen / löschen / in Sinngruppe einfügen / aus Sinngruppe herausnehmen
- Sinngruppe vereinigen

1.3.2.1 Listendarstellung

Für die Darstellung der Sinnguppen wird eine Listendarstellung gewählt.

- Hier werden alle Klassen einer ausgewählten Sinngruppe und deren Unter-Sinngruppe dargestellt.
- Die Liste kann zur Darstellung des Schnittes mehrerer Gruppen benutzt werden, indem einfach mehrer Gruppen ausgewählt und so nur die gemeinsamen Elemente angezeigt werden.
- Klassen können in die Listendarstellung eingefügt werden. Damit ergibt sich die Möglichkeit Klassen entweder in die oberste Sinngruppe einzufügen, oder sie sogar in den Schnitt, also mehrere Gruppen einzubetten.

1.4 Versionierungsperspektive

Diese Perspektive stellt Funktionen bereit, die die Versionierung betreffen. Hier kann der Benutzer zu älteren Versionen des gesamten Modells oder eines ausgewählten Bereichs springen oder einen neuen Versionszweig eröffnen. Unser Ziel ist es, die Versionierung, analog zu Versionsmamagement-Werkzeugen wie CVS u.a., zu unterstützen.

1.4.1 Aufbau und Funktionalität

Der Aufbau ähnelt dem der anderen Perspektiven.

- **Oben rechts** befindet sich das Hauptfenster, in dem ein horizontaler Versionierungsbaum angezeigt wird. Knoten dieses Baumes sind die vom Benutzer angeregten Releases, die atomaren Zwischenschritte werden hier ausgeblendet. In diesem Baum kann entweder direkt ein Release ausgewählt werden, zu dem zurück gesprungen werden soll (also ein Baumknoten) oder ein Bereich zwischen zwei Releases (also eine Kante), in dem zu einem bestimmten Zwischenschritt gesprungen werden soll.

  Realisiert ist der Baum mit Hilfe einer Tabelle, so dass für jeden Knoten (Release)und jede Kante (Zwischenschritt-Bereich) eine Tabellenzelle benutzt wird.

S	——	R_1	——	A_1	——	R_2	...
				|			
				R_3	——	A_2	...

Abbildung 1.9: Versionierungsbaum, realisiert als Tabelle

- **Unten** kann ein bestimmter Zwischenschritt eines Bereiches gewählt werden, zu dem gesprungen werden soll. Dies kann mit Hilfe einer Liste realisiert werden, in der jeder einzelne Schritt aufgeführt ist. Eine weitere Möglichkeit der Realisierung besteht in einem horizontalen Balken, wie man ihn von Videoschnittsoftware kennt. Für jeden Zwischenschritt hat der Balken eine Markierung, die vom Benutzer ausgewählt werden kann. Informationen über die einzelnen Schritte müssten per Label oder in einem kleinen Extrafenster (unten links) bereitgestellt werden.

Teil II

Umsetzung der Ideen und Konzepte

Kapitel 2

Release 1

Ziel hierbei ist die Realisierung eines ersten Prototyps, der mit einem Modell arbeitet und über eine History verfügt. Das 1. Prototyp sollte folgendes unterstützen:

- Funktionalität:
  - Klassen, Attribute, Methoden hinzufügen
  - Klassen, Attribute, Methoden ändern und löschen
  - Relationen hinzufügen, ändern und löschen.
- Zu realisierende Perspektive
  - UML

Die auszuführenden Operationen in diesem Release berücksichtigten nur einen Versionsstrang. Die Unterstützung von Sinngruppen sowie die Realisierung der Sinngruppen-Perspektive erfolgen in späteren Releases.

2.1 Das Konzept der Datenbank

Für das 1. Release hat sich gegenüber dem Zwischenbericht einiges geändert. Es wurde nun beschlossen nur in einer Richtung zu versionieren. Das heisst zunächst ohne Branch zu versionieren. So ist Branch-Spalte aus sämtlichen Tabellen ausgeschieden.

2.2 Versionierte Objekte

Um die Versionierung realiseren zu können, haben wir am Anfang zahlreiche Tabellen in der Datenbank angelegt. Die folgenden Tabellen und Attribute wurden konzipiert:

- Konfigurationen: Durch Definition einer Konfiguration wird also ein komplettes Diagramm gespeichert. Eine Konfiguration besteht aus allen Klassen (`KlasseID`) des Diagramms in der aktuellen Version (`KlasseVersion`) und einem `ÄnderungsText`, der die vorgenommene Änderung beschreibt . Die Konfigurationen selbst werden durch einen eindeutigen Bezeichner (`ID`) identifiziert.

Konfigurationen			
ID	KlasseID	AenderungsText	KlasseVersion
...	...	...	...

- KlasseEigenschaften: Diese Tabelle enthält die Klassen-ID (`KlasseID`), den Namen (`Name`) der Klassen, die Art des zugeordneten Stereotyps (`Stereotyp`) und eine Angabe über die Sichtbarkeit der Klasse (`Visibility`). Desweiteren macht die Tabelle Angaben darüber, ob es sich um ein Interface (`isInterface`), eine abstrakte (`isAbstract`) oder eine vererbte Klasse(`isFinal` handelt. Ferner wird mitprotokolliert von welcher Versionsnummer die Klasse existierte (`KlasseVersionVon`), so dass die Entwicklungshistorie nachvollziehbar ist.

KlasseEigenschaften							
Visibility	isInterface	Klassen-ID	KlasseVersionVon	Name	istAbstrakt	isFinal	Stereotyp
...	...	...	...	...	...	...	...

- Methoden: Sie enthält die Methode-ID (`ID`) und die ID der Klasse (`KlasseID`). Weitere Attribute sind die aktuelle Version der Methode (`MethVer`), der (`Name`), die `Sichtbarkeit`, der `Rückgabewert` und `isFinal`, das angibt, ob die Methode vererbt werden darf. Das Attribut `isStatic` gibt an, ob die Methode static ist. Die Speicherung der Parameter wird über eine weitere Tabelle erledigt. Zudem enthält die Tabelle eine Spalte, die angibt von welcher Version der Klasse, die Methode zu der Klasse gehörte (`KlVerVon`).

Methoden								
ID	MethVer	KlasseID	KlVerVon	Name	Sichtbarkeit	Rückgabewert	isStatic	isFinal

- Attribute: Die Tabelle entspricht im Wesentliche der Tabelle für die Methoden. Der einzige Unterschied liegt in dem Fehlen der Spalte Rückgabewert, die in der Methoden-Tablle fehlt.

Attribute							
ID	Version	KlassenID	KlasseVersionVon	Name	Sichtbarkeit	isStatic	isFinal
...	...	...	...	...	...	...	...

- Assoziationen: Sie enthält die `ID` der Assoziation, die `Version` der Assoziation, die Ausrichtung (`Richt`), das `Label`, den `Typ` (Assoziation, Vererbung, Aggregation oder Komposition). Zudem gibt es zwei Spalten, die die Klassen-IDs der Klassen zwischen denen die Assoziationen verlaufen angeben (`Kl1ID` und(`Kl2ID`) . Ferner werden die Multiziplitäten fur beide Enden der Assoziation mitprotokolliert (`Multi1` und `Multi2`). Um nachvollziehen zu können, von welcher Versionen die Assoziation zwischen den Klassen besteht, werden fur jede der beiden beteiligten Klassen entsprechende Einträge verwaltet (`Kl1VonVer` und `Kl2VonVer`).

Assoziationen										
ID	Version	Label	Kl1ID	Kl1VonVer	Multi1	Kl2ID	Kl2VonVer	Multi2	Typ	Richt
...	...	...	...	...	...	...	...	...	...	...

- MethodenParameter: Die Tabelle enthält die wichtigsten Parameter (`MethodeID, Name` und `MethodenVersion`) der Methoden. Ferner wird der `Typ` der Methoden mitprotokolliert.

MethodenParameter			
MethodenID	MethodenVersion	Name	Typ
...	...	...	...

- Release: Releases können vom Entwickler selbst festgelegt werden und stellen benannte Konfigurationen dar. Über Konfiguration und Fixpunkt werden die einzelne Phasen des Entwicklungsprozesses transparent. Die Tabelle enthält den Namen (`Name`) des Releases und die aktuelle ID der Konfiguration-Tabelle (`KonfigID`).

Release	
Name	KonfigID
...	...

- Sinnngruppe: Jede Sinngruppe erhält eine `ID`, einen eindeutigen Bezeichner (`Name`), eine `Version` und eine Klassen-ID (`KlassenID`) für die Klassen, die in die Sinngruppe gehören. Ferner wird ein Verweis auf eine Konfiguration (`KonfigIDVon`) gespeichert.

Sinngruppen				
ID	Name	Version	KonfigIDVon	KlassenID
...	...	...	...	...

2.3 Die Funktionalitäten der Datenbank

Wird an der Oberfläche des Tools eine Aktion getätigt, werden automatisch eine Reihe von `MySQL`-Befehlen in der Datenbank gestartet. Diese Befehle werden hier in Form von Aktivitätsdiagrammen näher erläutert.

2.3.1 Einfügen einer neuen Klasse

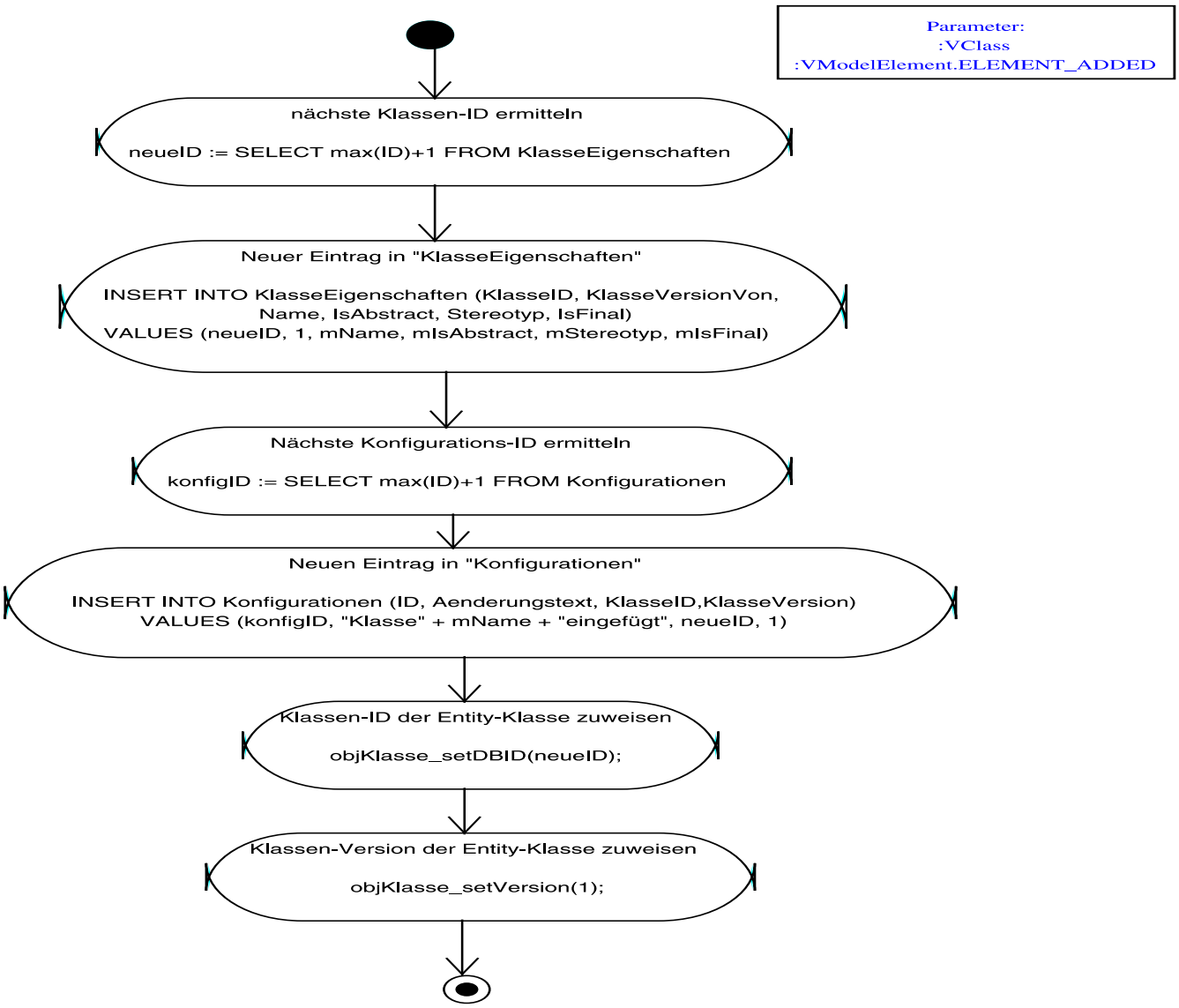

Abbildung 2.1: Klasse einfügen

Das Aktivitätsdiagramm in Abb. 2.1 stellt das Einfügen einer neuen Klasse dar. Es wird zunächst eine neue Klassen-ID für die neue Klasse ermittelt. Die höchste ID der Tabelle `KlasseEigenschaften` wird gelesen und um 1 erhöht. Diese wird als neueID gespeichert. Die neue Klasse wird dann in der Tabelle `KlassenEigenschaft` eingetragen. Dabei wird die neue Klassen-ID, neueID, in der Spalte `KlasseID` eingetragen. Die anderen Spalten der Tabelle `KlassenEigenschaften` werden mit den entsprechenden Werten gefüllt. Danach wird ein neuer Eintrag in der Tabelle

`Konfigurationen` gemacht. Das geschieht in zwei . Auch hier wird zunächst eine neue Konfigurations-ID ermittelt indem die höchste ID der Tabelle `Konfiguration` um 1 erhöht wird. Der Wert wird dann als konfigID gespeichert. Mit dem INSERT Befehl wird ein neuer Eintrag - mit der neuen ID - in der Tabelle `Konfiguration` gemacht. Dazu wird ein Änderungstext eingetragen, der aus einem String "Klasse"*mName*"eingefügt" besteht, wobei *mName* der Name der neuen Klasse ist. neueID wird in der Spalte `KlasseID` eingetragen und in der Spalte `KlasseVersion` wird 1 eingetragen. Dann wird die neue KlasseID mit dem Befehl *objKlasse.setDbID(neueID)* der Entity-Klasse zugewiesen. Als Letztes wird dann die neue Klassen-Version, 1, mit dem Befehl *objKlasse.setVersion(1)* der Entity-Klasse zugewiesen.

2.3.2 Löschen einer Klasse

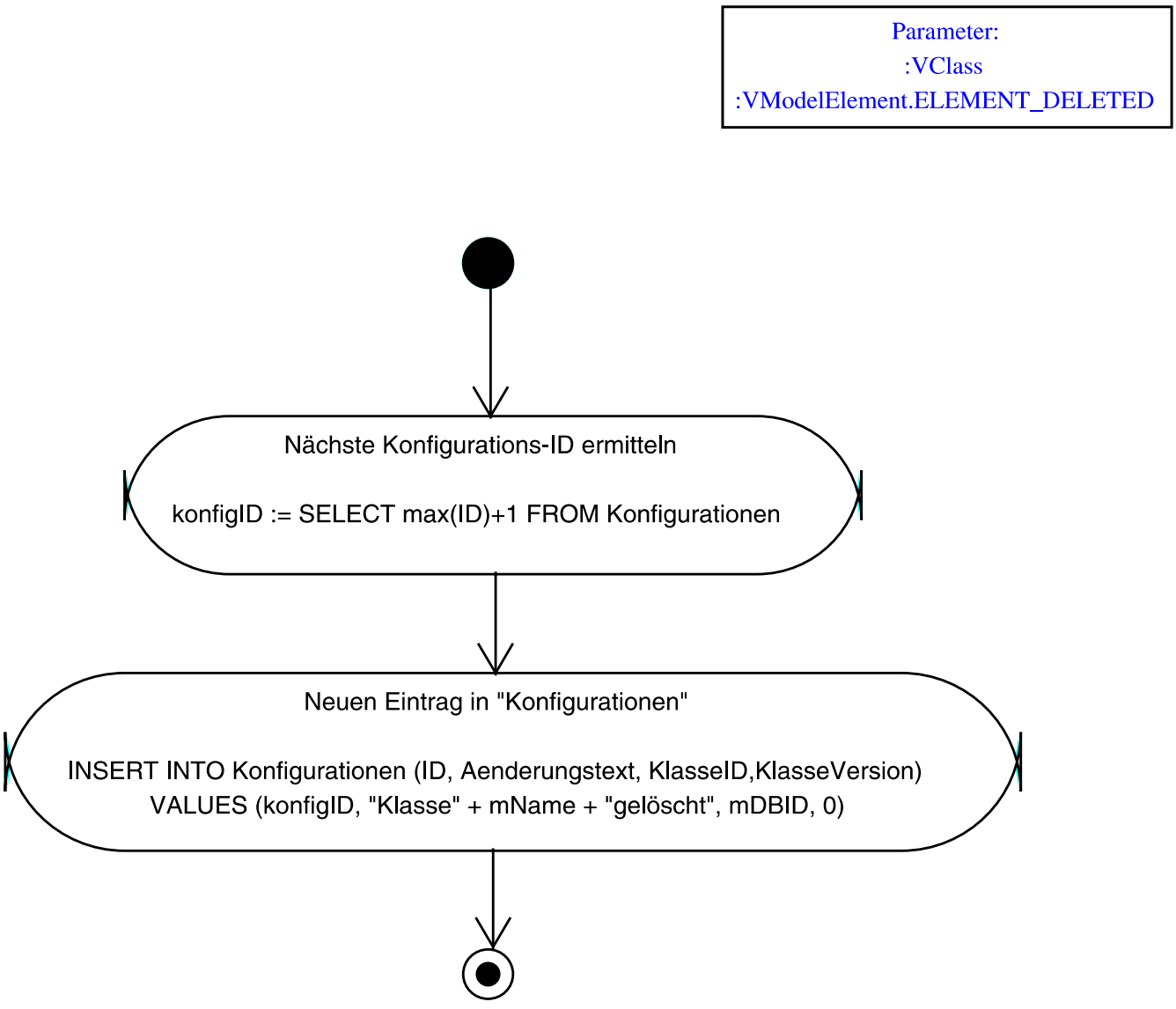

Abbildung 2.2: Klasse löschen

Abb. 2.2 stellt das Vorgehen beim Löschen einer Klasse dar. Wird eine Klasse gelöscht, muss ein neuer Eintrag in der Tabelle `Konfiguration` gemacht werden. Dazu wird zunächst eine neue Konfigurations-ID ermittelt, indem die höchste ID um 1 erhöht wird. Diese wird als konfigID

gespeichert. Mit dem INSERT Befehl wird in die Datenbank geschrieben: In der Spalte `ID` wird der Wert von konfigID eingetragen und als Änderungstext wird ein String "Klasse ABC gelöscht" eingetragen. Der Wert KlasseID wird beibehalten und Version wird auf 0 gesetzt, da die Klasse ABC gelöscht worden ist.

2.3.3 Ändern der Klasseneigenschaften

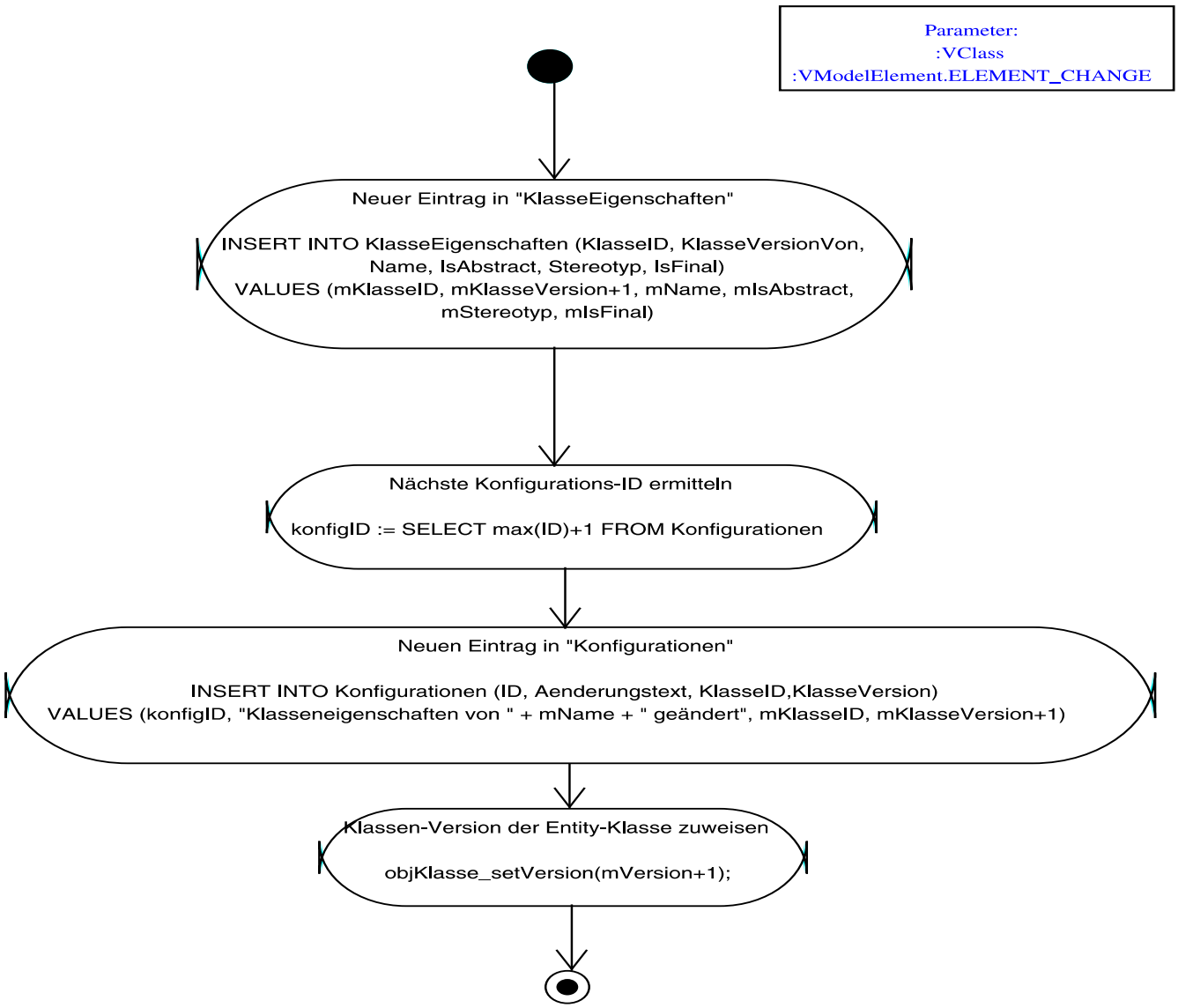

Abbildung 2.3: Klasseneigenschaften ändern

Wird eine Klasse geändert, wird eine Reihe von Aktionen gestartet. Es wird zunächst ein neuer Eintrag in der Tabelle `KlassenEigenschaften` gemacht, dabei wird der Wert von `KlasseVersionVon` um 1 erhöht. Danach wird ein neuer Eintrag in der`Konfigurationen` Tabelle gemacht. Für den neuen Eintrag wird die nächste ID ermittelt, indem die höchste ID der Tabelle `Konfigurationen` um 1 erhöht wird. Mit dem INSERT Befehl wird konfigID in der `ID`-Spalte eingetragen, als `Änderungstext` wird einen String gespeichert, der die Änderung beschreibt. Die Klasse-Version wird danach um 1 erhöht. Am Ende wird die Klasse-Version mit dem Befehl *objKlasse.setVersion(version+1)* der Entity-Klasse zugewiesen. Siehe Abb. 2.3

2.3.4 Hinzufügen einer Methode, Ändern einer Methode

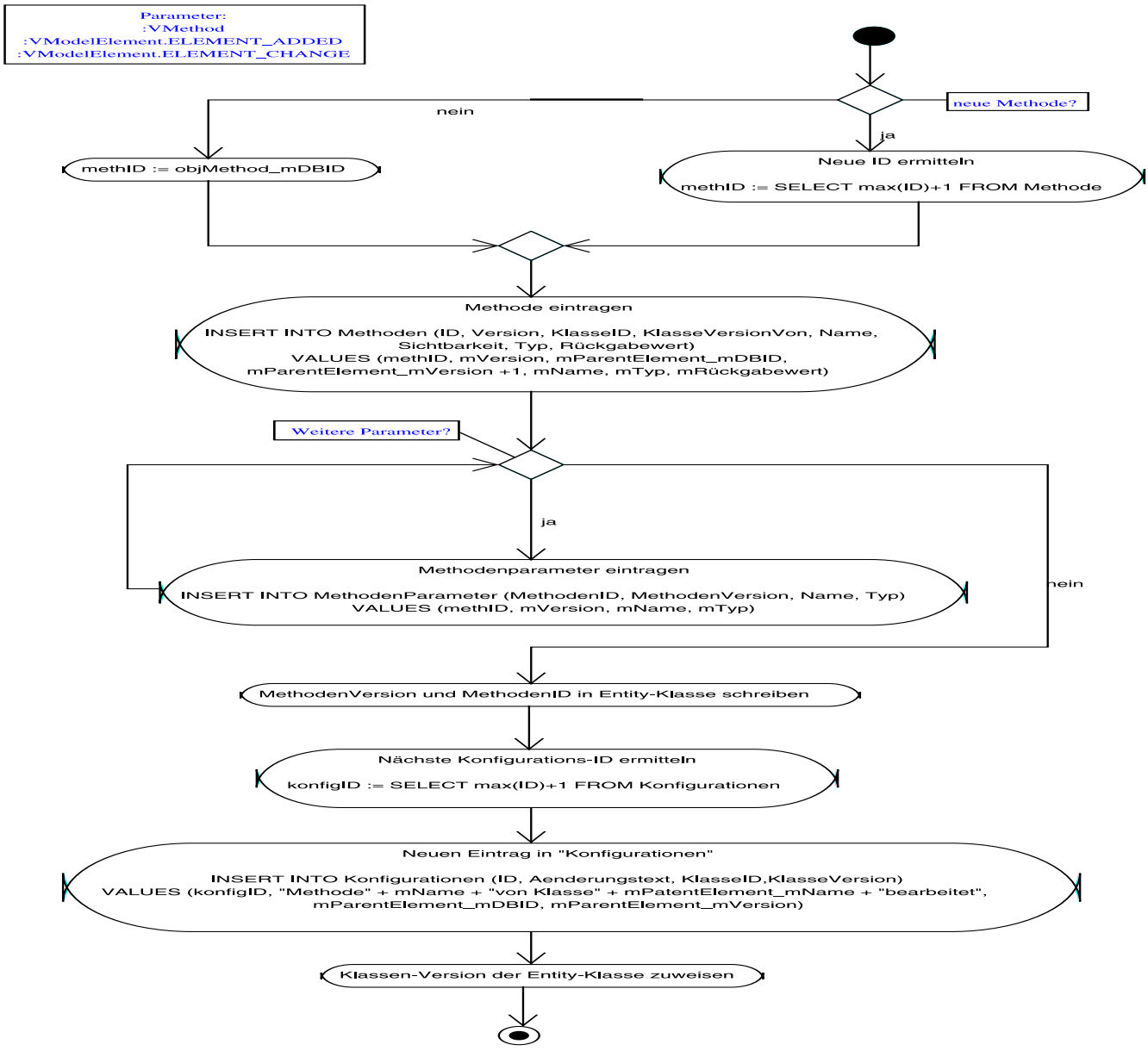

Abbildung 2.4: Methode einfügen/ändern

Das Aktivitätsdiagramm in Abb. 2.4 zeigt das Hinzufügen und Ändern einer Methode. Je nachdem, ob es sich um das Hinzufügen einer neuen Methode handelt oder das Ändern einer Existierenden, wird die Variable *methID* bestimmt. Wird eine neue Methode hinzugefügt, hat *methID* den Wert max(ID)+1. Handelt es sich um eine bereits vorhandene Methode, dann wird die Methoden-ID aus der Datenbank gelesen und methID zugewiesen. Dann werden in der Tabelle `Methoden` die Werte eingetragen, wobei ID = *methID* und `KlasseVersionVon` um 1 erhöht wird. Sollten weitere Parameter hinzugefügt werden, so wird dies in der Tabelle `MethodenParameter` getan. Falls nicht, werden MethodenVersion und MethodenID in Entity-Klasse geschrieben. Als Nächstes wird ein neuer Eintrag in der Tabelle `Konfigurationen` gemacht. Der Änderungstext lautet "Methode abc von Klasse ABC bearbeitet". Die Klasse-Version wird anschließend der Entity-Klasse zugewiesen.

2.3.5 Löschen einer Methode

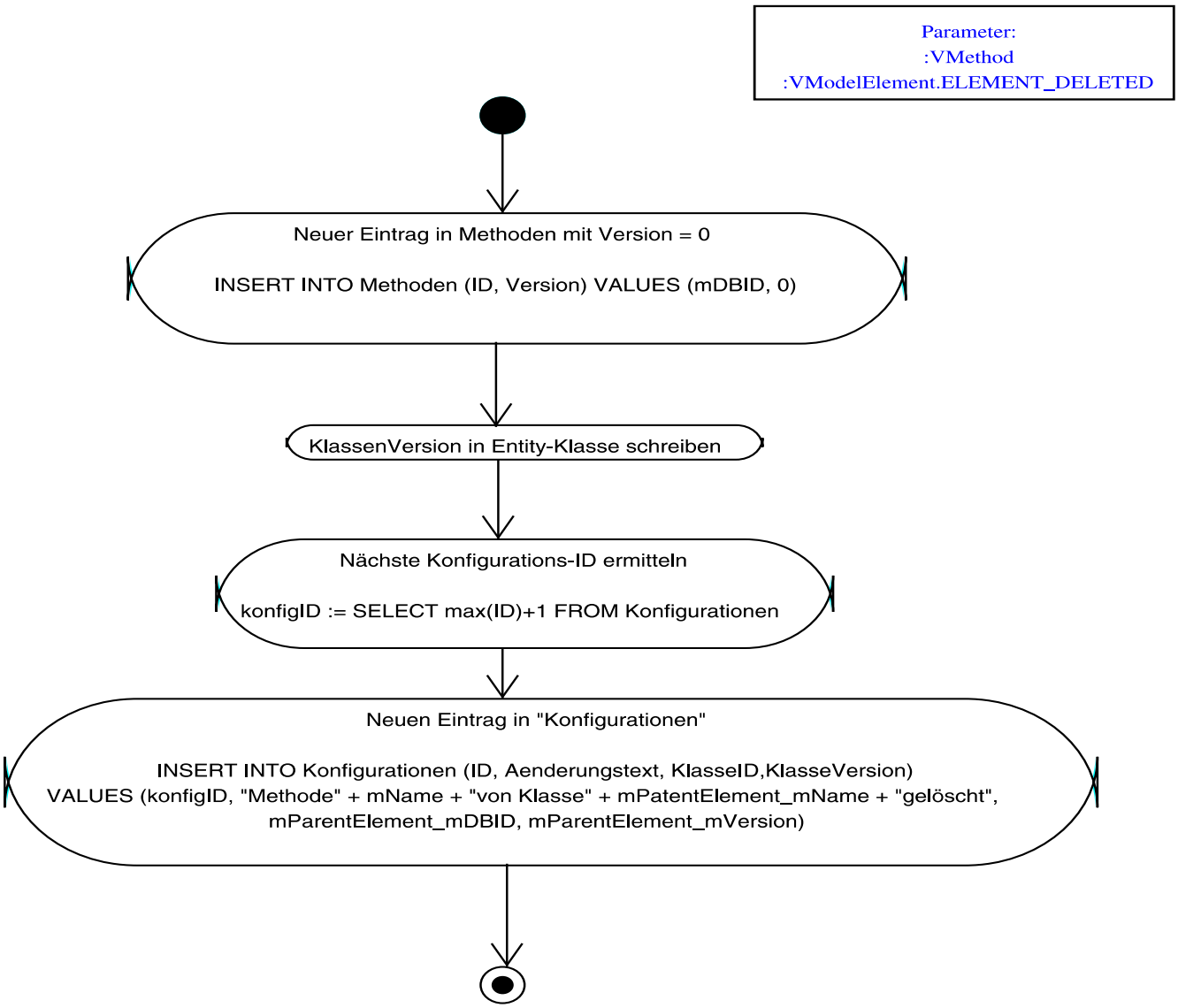

Abbildung 2.5: Methode löschen

Wird eine Methode gelöscht, wird ein neuer Eintrag in der Tabelle `Methoden` mit Version = 0 gemacht. Die neue Klasse-Version mit Version = 0 wird in der Entity-Klasse geschrieben. Eine neue Konfigurations-ID wird ermittelt. Anschließend wird einen neuen Eintrag in der Tabelle `Konfiguration` gemacht. Als `Änderungstext` wird der String "Methode *mName* von Klasse *mParentElement.mName* gelöscht" gespeichert. Siehe Abb. 2.5

2.3.6 Attribute einfügen und ändern

Der Ablauf dieser Funktion gleicht jenem der Methoden-Behandlung. Ein Parameterteil muss hier allerdings nicht gespeichert werden.

2.3.7 Attribut löschen

Diese Funktion verhält sich ebenfalls analog zu dem Vorgehen bei der Löschung einer Methode.

2.3.8 Hinzufügen und Ändern einer Assoziation

Das Aktivitätsdiagramm in Abb. 2.6 stellt das Hinzufügen und Ändern einer Assoziation dar. Je nachdem, ob eine neue Assoziation hinzugefügt oder eine bestehende bearbeitet wird, wird eine neue Assoziations-ID, *assID*, ermittelt. Beim Hinzufügen einer neuen Assoziation wird die ID aus dem Entity-Objekt gelesen. Wird lediglich eine Änderung vorgenommen, wird *assID* gelesen und um 1 erhöht. Dann werden die Werte in der Tabelle `Assoziationen` eingetragen. Dabei werden die Versionsnummer, Klasse1VonVersion und Klasse2VonVersion jeweils um 1 erhöht. AlsNächstes werden die Assoziations-Version und Assoziations-ID in der Entity-Klasse gespeichert. Die Klassen-Versionen werden auch anschließend in der Entity-Klassen eingetragen. Zum Schluß wird die nächste Konfigurations-ID ermittelt indem die höchste ID der Tabelle `Konfiguration` um 1 erhöht wird. Die beiden bei der Assoziation betroffenen Klassen werden mit der neu ermittelten ID in der Tabelle eingetragen. Dabei lautet der Änderungstext: "Assoziation zwischen *mParentElement1.mVersion* und *mParentElement2.mVersion* bearbeitet". Als `KlasseID` und `KlasseVersion` wird dann die entsprechende ID und Klasse-Version eingetragen.

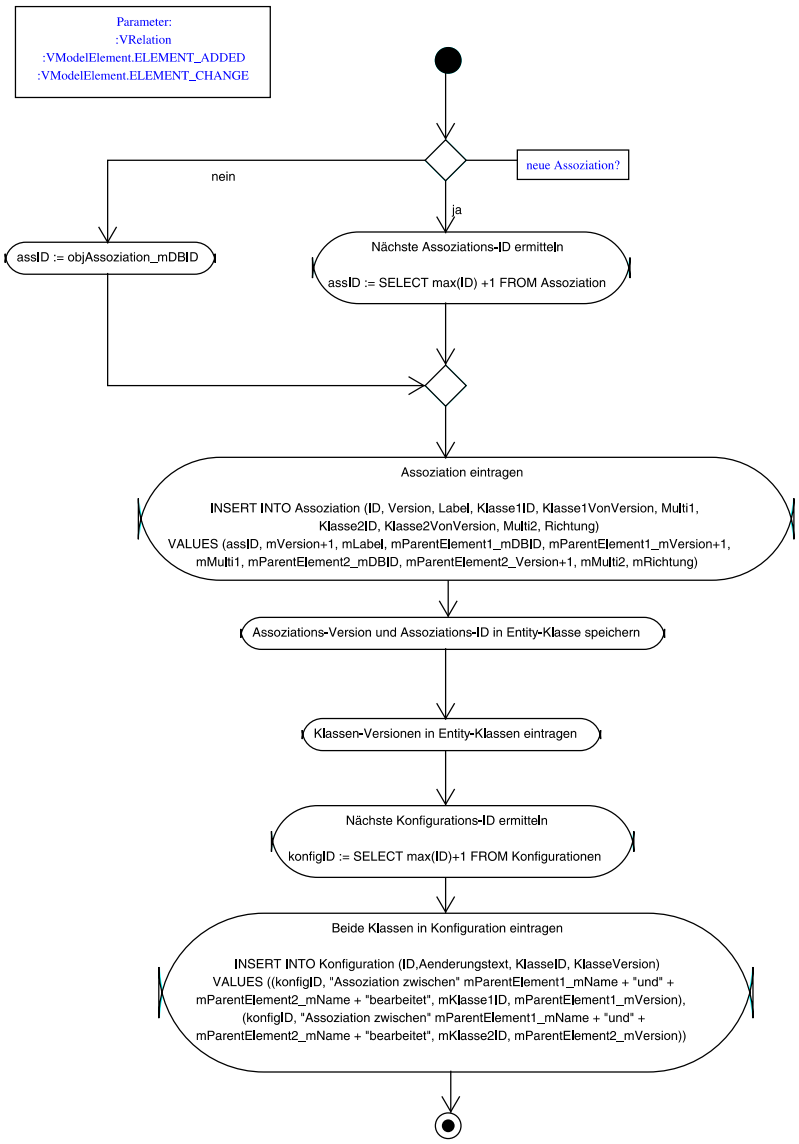

Abbildung 2.6: Assoziation einfügen

2.3.9 Assoziation löschen

Auch diese Funktion verhält sich wie die Löschung einer Methode. Es wird eine neuer Eintrag in der Tabelle Assoziationen mit der Version 0 angelegt.

2.4 Das Konzept der Benutzeroberfläche

2.5 Der Perspektiven-Mechanismus auf Basis von XML-Spezifikation und JInternalFrame

Die GUI-Gruppe hat sich dazu entschlossen, den bisher verfolgten Ansatz zur Realisierung des Perspektiven-Konzepts (siehe Zwischenbericht Kapitel 14.3) aufzugeben und stattdessen eine alternative Implementierung umzusetzen. Der bisherige Ansatz sah vor, dass eine Perspektive eine Menge von internen Fenstern, d.h. Sub-Klassen von javax.swing.JInternalFrame darstellt, wobei jedes dieser internen Fenster seine Größe und Position (jeweils relativ zum Hauptfenster) kennt. Die Spezifikation der einzelnen Perspektiven wurde zur Laufzeit aus einer XML-Datei dynamisch geladen. Ein Vorteil dieses Ansatzes ist seine hohe Flexibilität. Ein wesentlicher Nachteil jedoch ist, dass zur Ausnutzung dieser Flexibilität eine durchaus umständliche Interaktion des Benutzers gefragt ist. Da das in *VITEOS* eingesetzte Perstpektiven-Konzept jedoch implementiert wurde und sich als in höchstem Maße flexibel erwiesen hat, wurde sie erst nach Abwägen ihrer Vor- und Nachteile sowie der Vor- und Nachteile des später zu implementierenden Ansatzes verworfen. Daher soll diese Idee im folgenden näher beschrieben und diskutiert werden.

2.5.1 Idee

Bei ihrem Ansatz hat sich die GUI-Gruppe dazu entschlossen, dem Benutzer unterschiedliche Funktionalitäten in unterschiedlichen Fenstern zur Verfügung zu stellen. Diese Fenster sollten als Sub-Klassen von JInternalFrame realisiert werden, die in einem von Swing erzeugten Fenster beliebig angeordnet werden können. Siehe dazu Abb. 2.7. Eine Perspektive soll dabei eine definierte Menge solcher Fenster sein, wobei zu jedem Fenster seine Position und Größe innerhalb des Hauptfensters spezifiziert ist. Die Position und Größe sollten dabei nicht als absolute Werte, sondern relativ zur tatsächlichen Breite und Höhe des Hauptfensters verwaltet werden. Da diese Werte erst zur Laufzeit bekannt sind, sollten die konkreten Werte für Breite und Höhe jedes einzelnen inneren Fensters ebenfalls erst zur Laufzeit berechnet werden. Aus diesem Grund wurden diese Werte, ebenso wie die genaue Sub-Klasse von JInternalFrame, in einer XML-Datei perspectives.xml spezifiziert, die beim Programmstart eingelesen wird. Diese XML-Datei sollte beliebig viele Perspektiven spezifizieren, die von der Klasse PerspectiveReader eingelesen und als Liste von Objekten vom Typ Perspective repräsentiert wurden. Aus dem Hauptfenster dann eine Perspektive über ihren Namen abfragbar sein. Die entsprechenden Sub-Klassen wurden über das Java-API Reflection zur Laufzeit geladen und instanziiert.

2.5.2 Die Datei perspectives.xml

Im folgenden soll die von der GUI-Gruppe erstellte Datei erläutert werden. Die zur Entwicklung benutzte Datei sieht dabei so aus:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<perspectives>
```

```
<perspective name="UML">
  <iframe name="Modell-Baum"
  class="edu.unido.pg436.viteos.perspectives.uml.TreeStructure">
    <x>0</x>
    <y>0</y>
    <width>25</width>
    <height>65</height>
    </iframe>
  <iframe name="Roll-Over" class="javax.swing.JInternalFrame">
    <x>0</x>
    <y>65</y>
    <width>30</width>
    <height>35</height>
  </iframe>
  <iframe name="Adjazenz-Matrix"
  class="edu.unido.pg436.viteos.perspectives.uml.AdjacencyMatrix">
    <x>25</x>
    <y>0</y>
    <width>75</width>
    <height>65</height> <
  </iframe>
  <iframe name="Bearbeiten"
  class="edu.unido.pg436.viteos.perspectives.uml.ElementEditor">
    <x>30</x>
    <y>65</y>
    <width>70</width>
    <height>35</height>
  </iframe>
 </perspective>
</perspectives>
```

Das Root-Element <perspectives> enthält dabei die Menge der Perspektiven, die dem Programm zur Verfügung stehen. Eine Perspektive besteht dabei aus dem Element <perspective>. Es besitzt ein Attribut Name, über das im Programm auf die Perspektive zugegriffen werden konnte. Außerdem kann sie beliebig viele Elemente vom Typ <iframe> enthalten. Jedes dieser Elemente spezifiziert eine Sub-Klasse von JInternalFrame, sowie dessen relative Größe und Position im Hauptfenster. Das Attribut class gibt dabei den Namen der entsprechenden Sub-Klasse an, die zur Laufzeit zu laden und instanziieren ist. Das Attribut name wird in der Titelleiste des jeweiligen inneren Fensters angezeigt. Die enthaltenen Elemente <x>, <y>, <width> und <height> spezifizieren Position und Größe des inneren Fensters innerhalb des Hauptfensters. Die Werte sind relativ zur Größe des Hauptfensters, die ja erst zur Laufzeit bekannt ist. Somit gilt für die Adjazenz-Matrix des Programms, dass sie vom Typ edu.unido.pg436.viteos.perspectives.uml.AdjacencyMatrix ist sowie dass ihre linke obere Ecke auf 25% der Breite und 0% der Höhe des Hauptfensters beginnt und dass ihre Breite 75% der Breite und ihre Höhe 65% der Höhe des Hauptfensters beträgt.

2.5.3 Die DTD für die Datei perspectives.xml

Im folgenden wird eine Document Type Definition (DTD) für die Datei perspectives.xml angegeben. Sie wurde bei der Entwicklung nicht benutzt und wird hier nur der Vollständigkeitshalber

erwähnt.

```
<?xml version='1.0' encoding='utf-8'?>
<!ELEMENT perspectives(perspective*)>
<!ELEMENT perspective(iframe*)>
<!ATTLIST perspective
            name    CDATA    #REQUIRED
          >
<!ELEMENT iframe(x, y, width, height)>
<!ATTLIST iframe
            name    CDATA    #IMPLIED
            class   CDATA    #REQUIRED
          >
<!ELEMENT x (#PCDATA)>
<!ELEMENT y (#PCDATA)>
<!ELEMENT width (#PCDATA)>
<!ELEMENT height (#PCDATA)
```

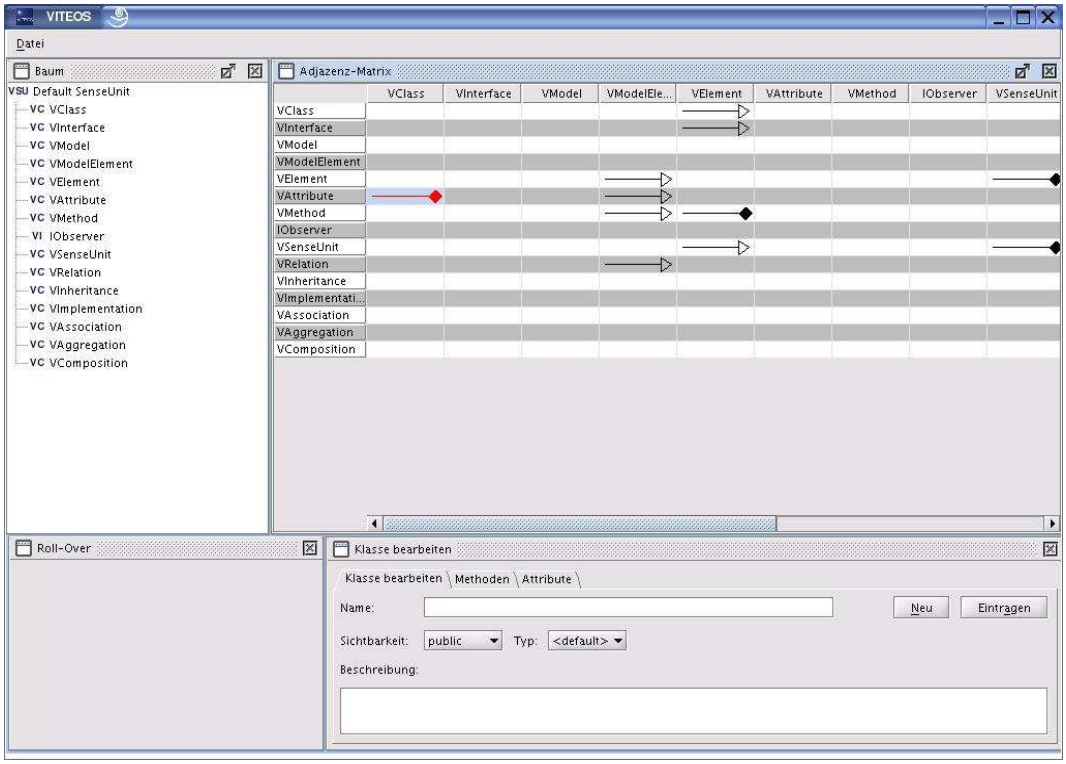

Abbildung 2.7: Hauptfenster mit vier inneren Fenstern

2.5.4 Vor- und Nachteile

Im Folgenden sollen die Vor- und Nachteile des beschriebenen Ansatzes diskutiert werden.

2.5.4.1 Vorteile

Der Vorteil des Ansatzes liegt eindeutig in seiner hohen Flexibilität. Durch Bearbeiten der Datei perspectives.xml ist es sehr einfach möglich eine neue Perspektive zu erzeugen oder eine vorhandene

zu modifizieren. Durch den Einsatz von JInternalFrames kann auch der Benutzer des Programms zur Laufzeit das Programm sehr flexibel nutzen. Es ist jede beliebige Anordnung der vorhandenen Fenster möglich. Prinzipiell ist es sogar möglich, das Programm um eigene, von JInternalFrame abgeleitete Klassen zu erweitern und diese in vorhandene oder neue Perspektiven einzubinden. Abbildung 2.7 zeigt das Hauptfenster mit vier inneren Fenstern (wie in perspectives.xml spezi - ziert). In Abbildung 2.8 hat der Benutzer eines der Fenster entfernt und Größe und Position eines anderen angepasst.

2.5.4.2 Nachteile

Die Nachteile dieses Ansatzes liegen v.a. darin, dass die hohe Flexibilität ein hohes Maß an Aufwand erfordert. Siehe Abb. 2.8. Der Nutzer kann die inneren Fenster von Hand vergrößern oder

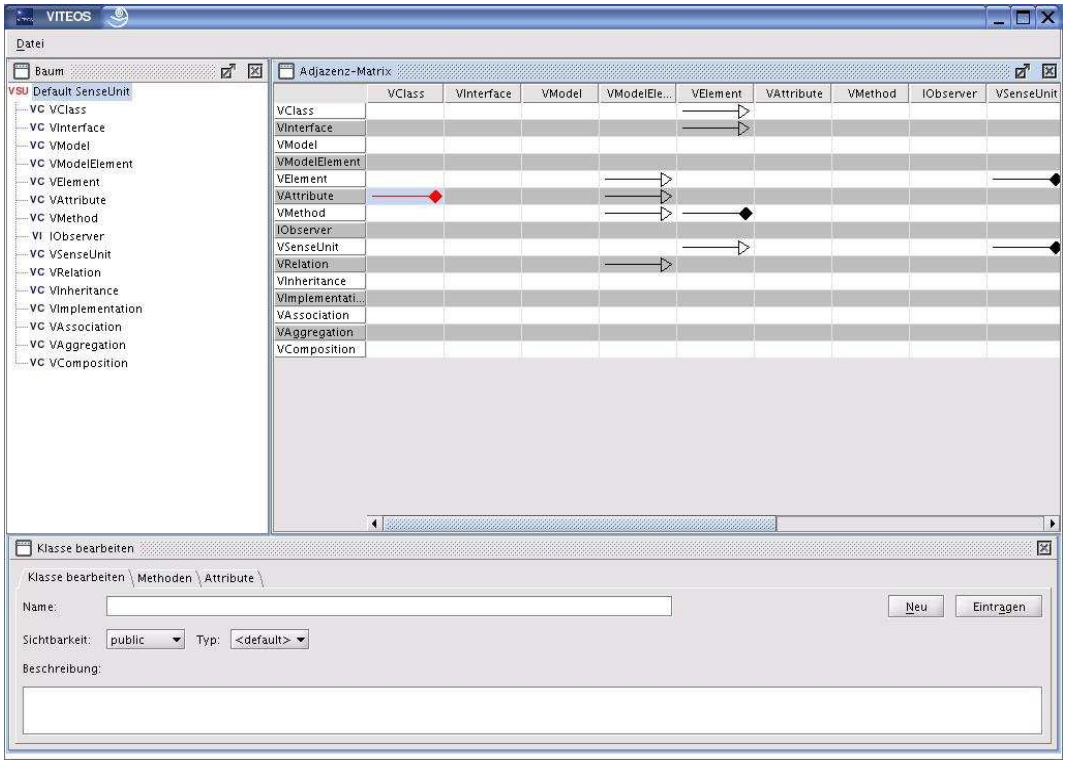

Abbildung 2.8: Vom Benutzer verändertes Hauptfenster mit drei inneren Fenstern

verkleinern, muss aber dann selbst dafür sorgen, dass angrenzende innere Fenster ebenfalls angepasst werden. Darüber hinaus werden die inneren Fenster beim Vergrößern oder Verkleinern des Hauptfensters nicht automatisch angepasst. Dies ist auch schwierig, da nicht klar ist, welche relativen Dimensionen bei einer Größenanpassung zu Grunde gelegt werden sollen: Die in perspectives.xml definierten oder die aktuellen vor der Vergrößerung des Hauptfenster, die eventuell bereits vom Benutzer von Hand modifiziert wurden.

2.5.5 Konsequenz

Als Konsequenz hat sich die GUI-Gruppe entschlossen, den auf einer XML-Spezifikation (vgl. 2.3.1) basierenden Ansatz zur Realisierung des Perspektiven-Konzepts aufzugeben und stattdessen zu versuchen, mit anderen Mitteln einen möglichst guten Kompromiss zwischen Flexibilität, Benutzerfreundlichkeit und Programmieraufwand zu finden. Nach dem nun von der GUI-Gruppe

verfolgten neuen Ansatz sind die bisherigen inneren Fenster von JPanel abgeleitet. Jede Perspektive wird wiederum durch eine von JPanel abgeleitete Klasse repräsentiert, die die anderen Panels lädt und mittels JSplitPane-Objekten dynamisch vergrößerbar und verkleinerbar macht. Ein Minimieren oder Maximieren eines einzelnen Panels ist damit ebenfalls machbar, wodurch man beispielsweise ausschließlich die Adjazenz-Matrix angezeigt bekommt. Zwischen den einzelnen Perspektiven kann man über Karteireiter (Tabs) umschalten. Bei dieser Realisierung des Perspektiven-Konzepts ist es also immer noch möglich die Größe einzelner Elemente beliebig zu verändern. Nicht mehr möglich ist es, sich beliebige Kombinationen von Elementen anzeigen zu lassen. So muss man jetzt die UML-Perspektive aufrufen, um mit der Adjazenz-Matrix zu arbeiten oder die Methoden und Attribute einer Klasse zu verändern. Siehe Abb. 2.9. Diesen Nachteil halten sie aber für akzepta-

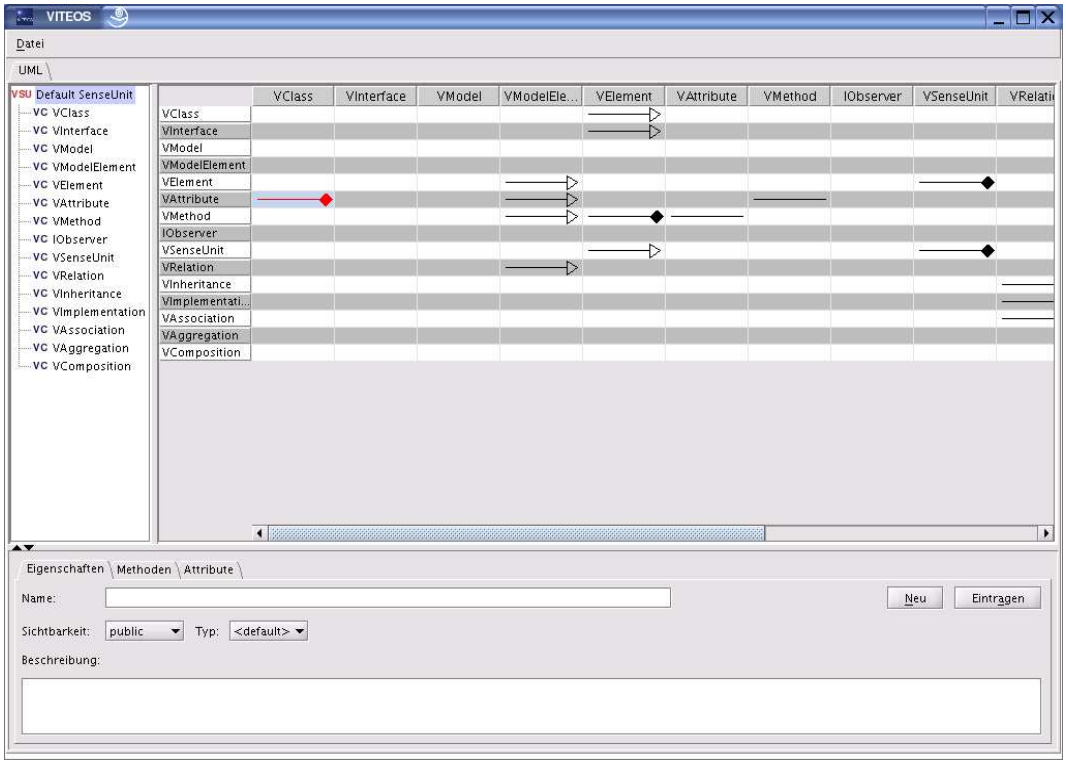

Abbildung 2.9: Das neue Hauptfenster, das seine Elemente als JPanels mit Hilfe von JSplitPanes anordnet

bel, da die Oberfläche durch die Änderung insgesamt komfortabler zu benutzen ist. Abbildung 3 zeigt, wie das Hauptfenster des Programms nach der Änderung aussieht. Das Roll-Over-Fenster (siehe Abb. 2.7), welches sich links neben dem Editor befindet und dem man keine Funtionalität zuordnen konnte, wurde inzwischen entfernt.

2.5.6 Der neue Ansatz zur Realisierung der Perspektiven

In diesem Kapitel wird der neue Ansatz detaillierter erläutert. Die einzelnen GUI-Elemente wie etwa die Adjazenz-Matrix, der Sinngruppen-Baum und der Editor für Klassen und Interfaces sind von JPanel abgeleitet. Sie werden innerhalb eines weiteren Panels, das die Perspektive repräsentiert, angezeigt. Mit Hilfe mehrerer JSplitPane-Objekte werden diese einzelnen Panels von einander abgegrenzt, wobei eine dynamische Anpassung der Größen möglich ist. Die vorgesehenen drei Perspektiven sind in einer JTabbedPane enthalten, wodurch man schnell zwischen den Perspektiven wechseln kann.

2.5.6.1 Vor- und Nachteile des neuen Ansatzes

Der beschriebene Ansatz bietet nicht die Flexibilität des bisher verfolgten Ansatzes. Es läßt sich innerhalb einer Perspektive kein weiteres GUI-Element anzeigen, dass nicht zu dieser Perspektive gehört. So ist es jetzt z.B. nicht mehr möglich, innerhalb der Versionierungsperspektive die Adjazenz-Matrix zu sehen. Dafür muss die Perspektive gewechselt werden. Dieser Nachteil wiegt jedoch nicht sehr schwer, da die beschriebene Flexibilität angesichts der vorgesehenen Funktionalität des Programms wenig bis gar keinen Nutzen erwarten lässt. Der Vorteil des neuen Ansatzes ist v.a., dass sich beim Vergrößern oder Verkleinern von GUI-Komponenten die angrenzenden Komponenten automatisch verkleinern oder vergrößern. Dies erscheint der GUI-Gruppe als benutzerfreundlicher.

2.5.6.2 Das Interface IModelObserver

Da Swing nach dem MVC-Prinzip entworfen wurde besitzen Komponenten ein Datenmodell, das bei Änderungen seinen View updatet. Sämtlichen in der GUI von VITEOS benutzten Datenmodellen liegt ein Objekt vom Typ VModel zugrunde. Die Datenmodelle gemäß Swing-API sollen nun das Interface IModelObserver implementieren bzw. dessen abstrakte Sub-Klasse ModelAdapter erweitern. Sie können sich dann bei einem Objekt der Klasse VModel registrieren und werden von diesem über Änderungen am Modell informiert. VModel ist ebenfalls dafür zuständig, die Datenbankschicht über die Änderung zu informieren.

2.5.7 Realisierung der Versionierungsperspektive

2.5.7.1 Hauptkomponente

Wie geplant besteht die Versionierungsperspektive aus zwei mit Hilfe eines Splitpanes realisierten Bereichen. Der obere Pane soll einen als Tabelle implementierten Baum für die Version-Releases enthalten. Die Zellen der Tabelle enthalten entweder einzelne Releasepunkte oder stellen die Zwischenschritte zwischen zwei auf einander folgenden Releases dar. Beim Selektieren eines solchen Zwischenbereichs können also alle Versionpunkte bearbeitet werden, die zwischen den entsprechenden Releases angelegt worden sind. Die Tabelle besteht nur aus Releases (ein Versionstrang) in eine Richtung, da das Branchen noch nicht unterstützt wird. Die selektierten Versionspunkte werden im unteren Pane angezeigt, wo sie dann bearbeitet werden können. Dieser untere Bereich wird zunächst mit Hilfe einer einspaltigen Tabelle realisiert, da diese sich im Gegensatz zu einer Liste bei Notwendigkeit (Darstellung zusätzlicher Parameter) sehr einfach erweitern lässt. Zusätzlich bietet eine Toolbar Funktionalitäten für die beiden Bereiche. Diese wird direkt im Hauptfenster integriert.

2.5.7.2 Unterstützte Funktionalität

Die in der Versionierunsperspektive unterstützten Funktionalitäten werden über Kontextmenüs und die oben erwähnte Toolbar realisiert. Folgende Funktionalitäten sind dafür vorgesehen:

1. Liste der VersionPoints :
   - VersionPoint(VP) benennen (d.h. VP als Release festsetzen)
   - Sprung (Zum ausgewählten VP springen)
2. Tabelle der VersionReleases:
   - Release entfernen

- Navigieren
- (Branchen: im 3. Release)

2.5.7.3 Datenstruktur

Zur Darstellung der Versionreleases wird eine Liste solcher Elemente erwartet. Wird ein Bereich zwischen zwei Releases in der Tabelle selektiert, wird eine Liste aller Versionpunkte zwischen diesen Releases aus der Datenbank geholt und im unteren Pane dargestellt.

2.6 Zwei APIs für VITEOS: Apache log4j und JGoodies Looks

In der Zeit zwischen dem Wintersemester 2003/2004 und dem Sommersemester 2004 hat sich die GUI-Gruppe für den Einsatz zweier APIs entschieden. Bei *Apache log4j* handelt es sich um eine Bibliothek zum Protokollieren interner Systemzustände (Logging). *JGoodies Looks* dagegen stellt ein sogenanntes *Pluggable Look-and-Feel* dar, also eine dynamisch ladbare Implementierung des Teils der Swing-API, der für das Rendern von GUI-Komponenten zuständig ist. Im Folgenden sollen diese beiden APIs genauer vorgestellt werden.

2.6.1 Apache log4j

Die Aufgabe von log4j ist es primär, die Suche nach Fehlern im Code zu unterstützen.[1] Durch Auswertung der Systemzustände, d.h. der Werte von Variablen, sowie deren Veränderung im Laufzeitverhalten, ist es möglich, die Stelle, an der ein Fehler verursacht wird, einzugrenzen.

Um dies zu erreichen gibt es prinzipiell zwei Methoden: Debugger sowie Log-Statements.

Debugger bieten die Möglichkeit, zur Laufzeit den Status jeder Variablen abzufragen, sowie schrittweise den Kontrollfluss des Programms zu durchlaufen. Außerdem bieten sie über Breakpoints die Möglichkeit, dieses schrittweise Durchlaufen ab einem bestimmten Punkt im Code zu beginnen. Hier zeigt sich ein Problem, vor dem man als Programmierer steht: Man muss entweder schon eine Ahnung haben, wo die Ursache eines Fehlers liegen könnte um an sinnvollen Stellen einen Breakpoint zu setzen. Oder man „quält“ sich durch sehr viele Einzelschritte, was viel Zeit in Anspruch nimmt.

Log-Statements in ihrer einfachsten Form sind lediglich Aufrufe einer Funktion, die den Wert einer Variablen auf das Standard-Ausgabe-Gerät bzw. Standard-Fehlerausgabe-Gerät des Systems schreibt (in Java also `System.out.println(...)`). Diese Ausgabe lässt sich z.B. in eine Text-Datei umleiten und dann mit Hilfe von Werkzeugen wie `grep` auswerten.

Apache log4j erweitert diese sehr simple Form des Loggings um eine Reihe von Funktionen, indem es die Konzepte *Logger*, *Appender* und *Layout* einführt.

2.6.1.1 Logger

Objekte vom Typ `org.apache.log4j.Logger` sind die Objekte, die die Log-Ausgabe erzeugen. Sie stehen generell in einer Eins-zu-Eins-Beziehung zu Klassen des Programms, d.h. jede Klasse hat ihren eigenen Logger. Erzeugt (und dabei auch konfiguriert) werden sie vom sogenannten Root-Logger, der beim Start des Systems konfiguriert werden muss. Dies geschieht in der Regel durch das Einlesen einer Konfigurationsdatei (mehr dazu im folgenden Abschnitt). Daraus folgt, dass

[1] log4j ist nicht als Konkurrenz zu JUnit zu sehen, eher als Ergänzung: Mit der Ausgabe von log4j kann man nach einem Fehler suchen, dessen Existenz man vorher mit JUnit nachgewiesen hat.

das Logging dynamisch konfigurierbar ist, d.h. es kann zur Laufzeit ein- und ausgeschaltet bzw. angepasst werden.

Jede Klasse initialisiert ihren Logger also in ihrem Konstruktor, bzw. in einer Methode `initLogger()`, die vom Konstruktor aufgerufen wird. Hierbei wird das `Class`-Objekt dieser Klasse dem Logger übergeben:

```
logger = Logger.getLogger(this.getClass());
```

Log-Statements werden erzeugt über die Methoden

```
public void debug(Object message);
public void info(Object message);
public void warn(Object message);
public void error(Object message);
public void fatal(Object message);
```

bzw. über die generische Methode

```
public void log(org.apache.log4j.Level l, Object message);
```

Um ein möglichst flexibles und zur Laufzeit konfigurierbares Logging-Verhalten zu ermöglichen unterstützt log4j verschiedene Level. Dabei muss unterschieden werden zwischen dem Level des Systems (der beim Root-Logger gespeichert ist) und dem Level einer Nachricht. Es existieren die fünf Level DEBUG, INFO, WARN, ERROR und FATAL. Auf diesen ist eine totale Ordnung definiert und es gilt:

$$\text{DEBUG} < \text{INFO} < \text{WARN} < \text{ERROR} < \text{FATAL}$$

Ist der Level einer Nachricht kleiner als der des Systems, so wird diese Nachricht nicht ausgegeben! Ist also der Level des Systems z.B. ERROR, so wird eine Nachricht mit Level DEBUG, INFO oder WARN nicht ausgegeben, eine mit ERROR oder FATAL wird ausgegeben.[2]

2.6.1.2 Appender und Layouts

log4j kann seine Ausgabe einfach auf der Text-Konsole ausgeben. Es sind jedoch noch mehr Möglichkeiten zur Ausgabe vorhanden. So kann in Dateien, GUI-Komponenten, über TCP-Sockets an entfernte Server, an Java Messaging Services (JMS), NT-Event-Logger-Dienste und Unix-Syslog-Server geschrieben werden. Bei Ausgaben in Dateien sind auch Roll-Over-Mechanismen vorhanden, d.h. bei Überschreiten einer bestimmten Dateigröße oder eines Datums wird die Datei komprimiert und archiviert und die weiteren Ausgaben landen in einer neuen Datei.

Implementierungen des Interface `org.apache.log4j.Appender` definieren dabei, *wohin* geloggt wird. So schickt z.B. die Klasse `FileAppender` die Ausgaben in eine Datei.

Sub-Klassen der Klasse `org.apache.log4j.Layout` dagegen bestimmen *das Format* der Ausgabe. So erzeugt die Klasse `SimpleLayout` ein simples Textformat, bei dem der Name der Klasse gefolgt von der eigentlichen Nachricht ausgegeben wird. `HTMLLayout` dagegen erzeugt eine HTML-Datei mit Tabellen, und `XMLLayout` erzeugt eine XML-Ausgabe gemäß einer ebenfalls von log4j spezifizierten Document Type Definition.

Der Appender bekommt das Layout beim Konstruktoraufruf mit übergeben. Der Logger wiederum erhält bei seiner Erzeugung den oder die Appender des Root-Loggers.[3]

[2] Hinweis: Es sind komplexere Konfigurationen von Levels möglich. Logger können baum-artige Hierarchien bilden (deren Wurzel immer der Root-Logger ist), innerhalb derer dann unterschiedliche Levels gültig sind. Außerdem werden Levels innerhalb der Hierarchie vererbt. Da wir aber nicht planen diese Möglichkeiten zu nutzen wird an dieser Stelle nicht näher darauf eingegangen.

[3] Mehrere Appender sind möglich.

2.6.1.3 Konfiguration zur Laufzeit über Dateien

Wie oben erwähnt kann die Konfiguration des Root-Loggers zur Laufzeit durch das Laden einer Konfigurationsdatei angepasst werden. Diese sollte dem Root-Logger möglichst früh nach dem Programmstart mitgeteilt werden (auf jeden Fall bevor andere Logger initialisiert werden). Dazu reicht ein einziger Aufruf einer Methode aus der Klasse PropertyConfigurator:

```
PropertyConfigurator.configure("config/logging.cfg");
```

Die Konfigurationsdatei liegt dabei im Unterverzeichnis „config“ und heißt „logging.cfg“.

Im Folgenden sei nur die von uns während der Entwicklung verwendete Konfigurationsdatei vorgestellt und erläutert.

```
log4j.rootLogger=DEBUG, A1
```

Hiermit wird dem Root-Logger der Level DEBUG zugewiesen sowie der Appender A1, der in den folgenden drei Zeilen spezifiziert wird.

```
log4j.appender.A1=org.apache.log4j.ConsoleAppender
```

Hier wird spezifiziert, dass A1 vom Typ `ConsoleAppender` sein soll.

```
log4j.appender.A1.layout=org.apache.log4j.PatternLayout
```

Hiermit wird A1 ein Layout vom Typ `PatternLayout` zugewiesen.

```
log4j.appender.A1.layout.ConversionPattern=%r [%t] %-5p %C{1} - %m%n
```

In dieser Zeile wird über ein Pattern spezifiziert wie die Ausgabe von A1 aussehen soll. In diesem Falle wird die seit dem Programmstart vergangene Zeit in Millisekunden, gefolgt von dem momentanen Thread in eckigen Klammern, gefolgt vom Level der Nachricht und dem Klassennamen, gefolgt von „ - “ und der eigentlichen Nachricht ausgegeben.

Der folgende Aufruf

```
logger.debug("Logging konfiguriert.");
```

erzeugt gemäß obigem Pattern die folgende Ausgabe:

```
0 [main] DEBUG MainWindow - Logging konfiguriert.
```

2.6.2 JGoodies Looks

Bei JGoodies Looks handelt es sich um ein Pluggable Look-and-Feel (LnF) für die GUI-Bibliothek Swing. Die von Java bzw. dem Java 2 SDK mitgebrachten Look-and-Feels scheinen uns für VITEOS aus mehreren Gründen nicht geeignet zu sein. Zum einen sind diese Gründe ergonomischer Natur, zum anderen das Empfinden, dass diese LnFs „einfach häßlich“ sind.

JGoodies Looks bietet mehrere ausgereifte LnFs, die nicht nur besser aussehen als die von Sun mitgelieferten, sondern unserer Ansicht nach auch die Ergonomie eines GUI-Programms steigern. Das Projekt hat in der Java-Community einen sehr guten Ruf und wird auch von Sun unterstützt.

2.6.2.1 Benutzung

Wir haben uns für ein spezielles LnF entschieden, das `Plastic3DLookAndFeel`. Die Einbindung in ein Java-Programm ist sehr einfach. Die zwei Aufrufe

```
UIManager.setLookAndFeel(new Plastic3DLookAndFeel());
SwingUtilities.updateComponentTreeUI(this);
```

setzen das Plastic3DLookAndFeel für das Hauptfenster des Programms und für alle Child-Komponenten.

Kapitel 3

Release 2

Als Kern-Feature des zweiten Prototypen wurden die Versionierungsperspektive und die dazugehörigen Funktionalität bestimmt. Die Umsetzung der Sinngruppen und die automatisierte Verschmelzung von zwei Versionierungssträngen sollten auch ermöglicht werden. Zu den in 1. Release genannten Funtionalitäten sollte das 2. Release zusätzlich folgenden Features unterstützen:

- Umsetzung des Versionierungsmechanismus (zunächst ohne Sinngruppen)
- Modellierung mit verzweigbaren Versionierungssträngen
- Rücksprung zu Vorgängerversionen eines Modells
- Weiterentwicklung des Modells ab dieser früheren Version
- Unterstützung von mehreren Projekten (Laden und Erstellen von Projekten)

Um die oben angegebenen Funktionalitäten realisieren zu können, sollte eine Versionierungspespektive auf der graphische Oberfläche entwickelt werden, die den Entwurf der GUI-Komponente für die Darstellung der Versionierungsstränge unterstützt. Darüber hinaus sollte die Darstellung der Liste der einzelnen Schritte innerhalb eines Versionierungsstranges möglich sein.
Die Datenbank sollte das Branching unterstüzen und die Erzeugung eines Branches, Rücksprunge und Festlegung von Releases ermöglichen. Eine Undo-Funktion sollte entworfen und intergriert werden, damit Schritte aus der versionierten Entwicklung des Modells entfernt werden können.

3.1 Das Konzept der Datenbank

Für das 2. Release hat sich datenbankmäßig nicht viel geändert. Die versionierte Objekten sind die gleiche geblieben. Um das Branchen realisieren zu können, wurden die Tabellen um die Branch-Spalte erweitert. So sieht für das 2. Release bespielsweise die Konfigurationen-Tabelle nun aus:

Konfiguration				
ID	KlasseID	AenderungsText	KlasseVersion	Branch
...	...	...	...	...

3.1.1 Die geänderten Aktivitätsdiagramme

Beim 2. Release verlaufen alle Funktionen genauso wie es oben beim 1. Release beschrieben wurde. Der einzige Unterschied besteht darin,wie es schon oben beschrieben wurde, dass die Tabellen um die Spalte `Branch` erweitert werden. Siehe Abbildungen 3.1 und 3.2. Weil es in 2. Release gebrancht wird, werden die Releasepunkte festgehalten und in der Datenbank gespeichert. Um einen neuen Releasepunkt zu erstellen, wird zunächst festgestellt, ob es sich um einen Branch handelt. Ist es kein Branch, wird die Variable maxBranch gleich dem aktuell höchsten Branch gesetzt. Wird gebrancht, so wird die nächste Branch-ID ermittelt, indem die höchste Branch-ID um 1 erhöht wird. Der Wert wird maxBranch zugewiesen. Der neue Branch wird danach in das Modell (Entity-Ebene) eingetragen. Anschließend werden der Name, die Konfig-ID und der Branch in der Tabelle `Release` eingegeben. Siehe Abbildung 3.3.

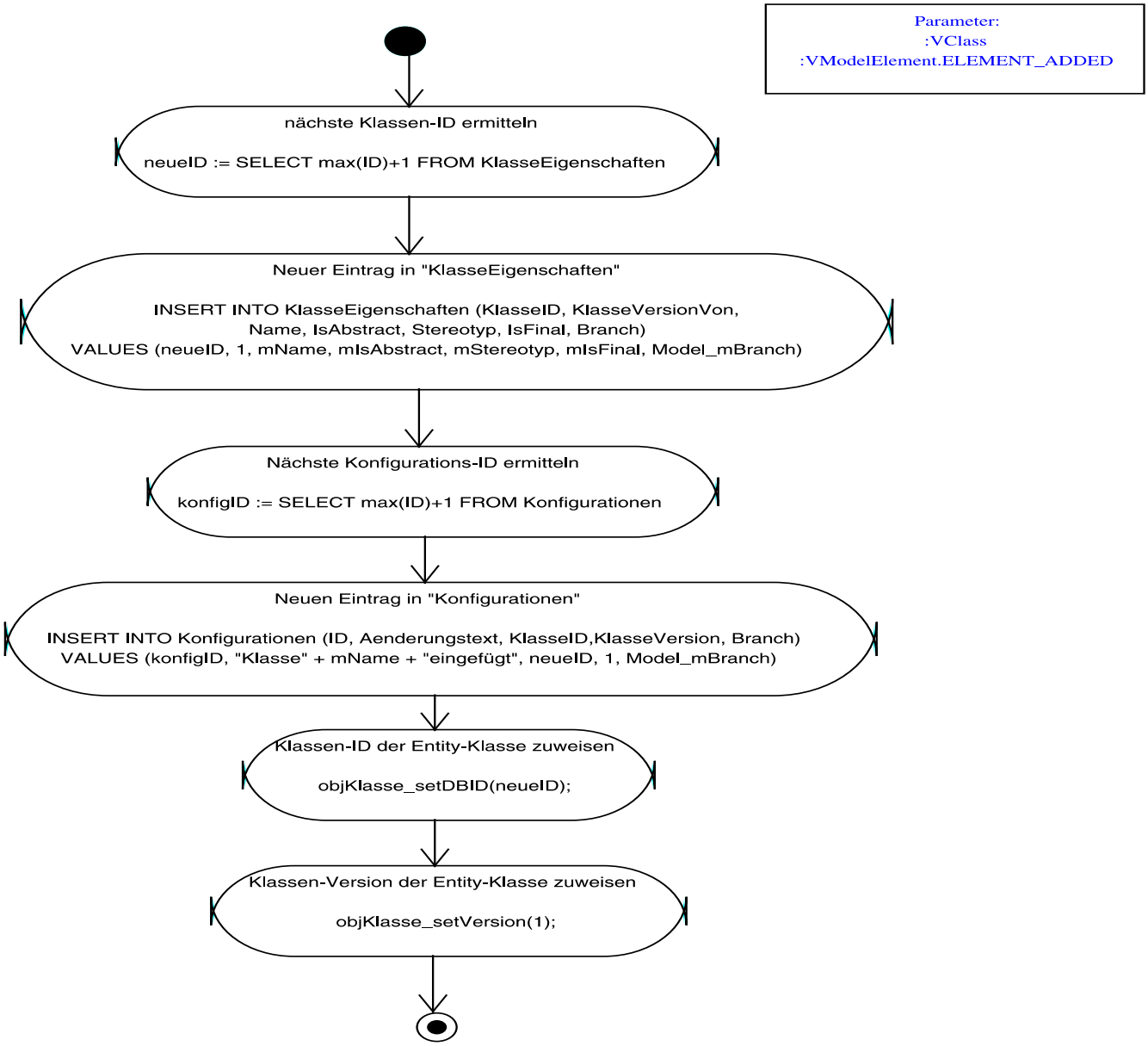

Abbildung 3.1: Klasse Einfügen

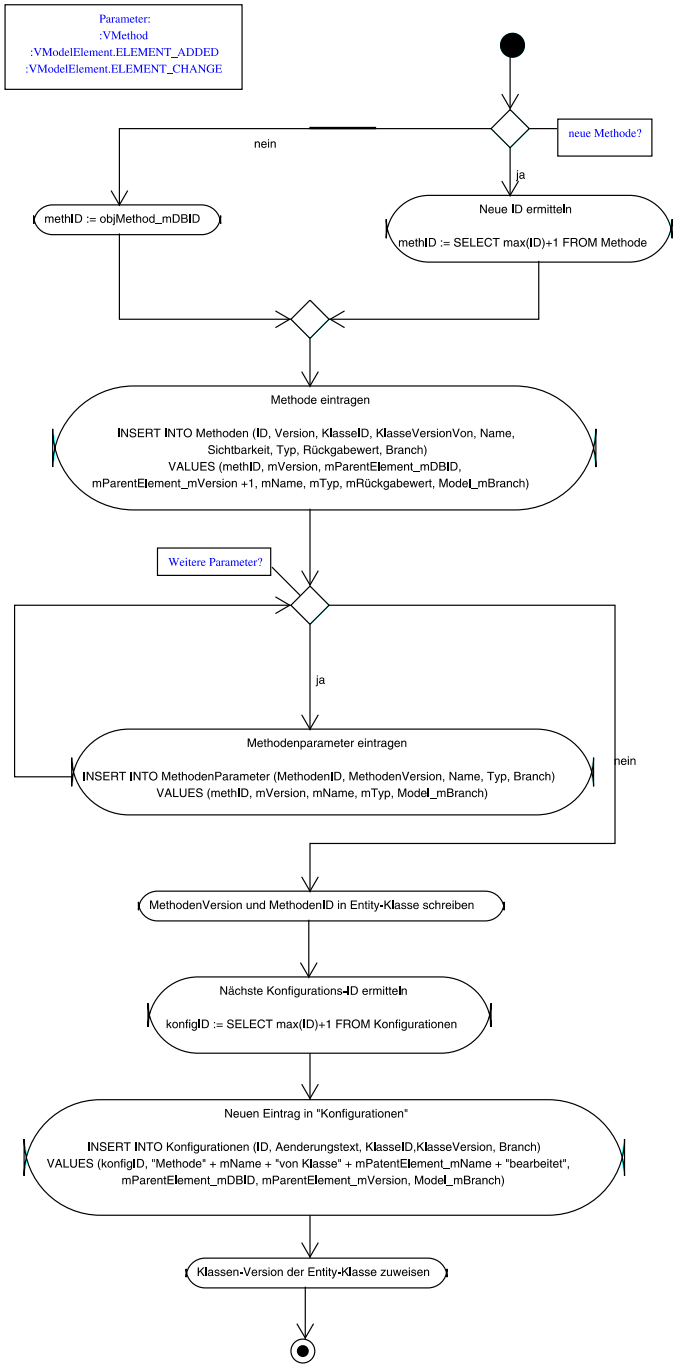

Abbildung 3.2: Methode einfügen/ändern

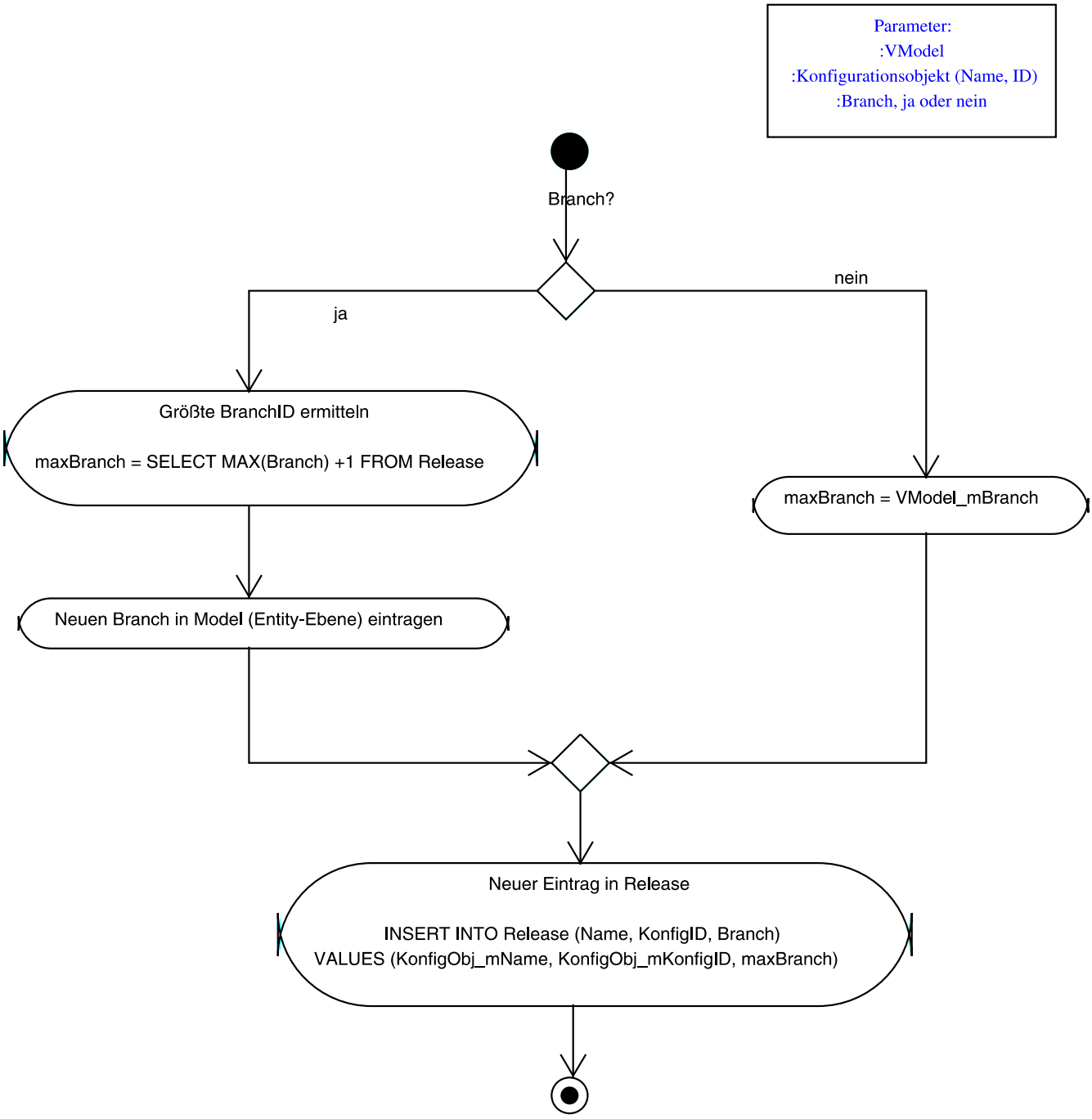

Abbildung 3.3: Releasepunkt erstellen

3.2 Das Konzept der Benutzeroberfläche

Im zweiten Release bietet unser Tool dem Benutzer die Möglichkeit, vom Hauptzweig der Entwicklung einen Zweig *(Branch)* abzuspalten um daran arbeiten zu können. Dabei ist es weiterhin Möglich, die Änderungen die man betätigt hat in den Hauptzweig zu integrieren oder aktuelle Änderungen aus dem Hauptzweig (oder anderen Zweigen) in den eigenen Zweig einfließen zu lassen *(Merge)*.

3.2.1 Branchen und Mergen

Das Modell beginnt immer mit einer generischen Struktur, die folgendermaßen aussieht. Release – Strang – Release genauer, <*Start*>–Strang–<*Ende von Branch 1*>. Ein generischer Release ist ein spezieller Versionspunkt. Der Start Release des Modells ist immer generisch und leer, das zweite Release <*Ende von Branch 1*> was beim Erzeugen des Modells generiert wird ist auch generisch, kann jedoch in einen echten Release umgewandelt werden in dem der Benutzer diesem Release einen festen Namen zuordnet. Im Verlauf der Entwicklung kann der Programmierer benannte Releases beliebig oft umbenennen. An dem Strang, der die beiden Releases verbindet, wird die Folge von den einzelnen Versionspunkten dargestellt. Bei der nächsten Änderung des Modells wird immer ein generischer Release erzeugt, welcher wiederum in einen echten umgewandelt werden kann. Das Verzweigen ist nur in echten Releases erlaubt, weiterhin kann an einem Release nur einmal verzweigt werden (siehe Abbildung 3.4). Die untereinander Verzweigten Releases repräsentieren

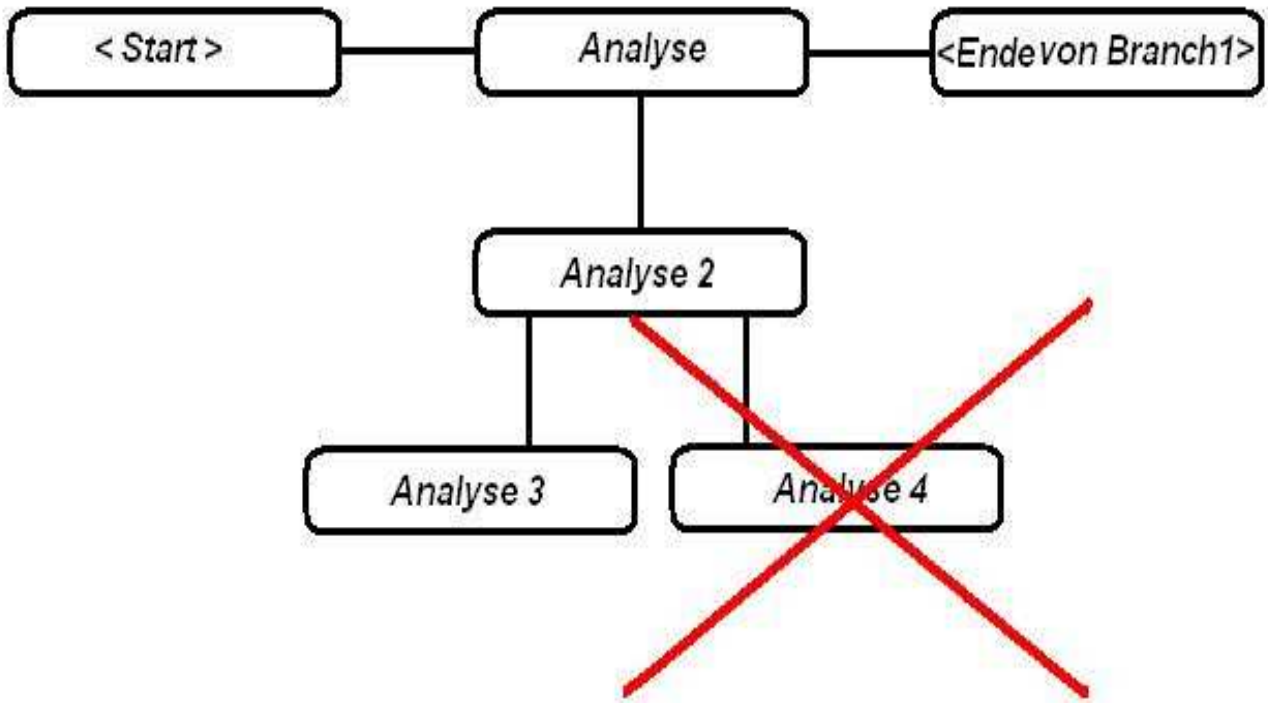

Abbildung 3.4: Modell Verzweigen

das Gleiche Modell. Im Gegensatz zum Verzweigen ist das Integrieren (*mergen*) von Releases nicht nur in echte sondern auch in generische Releases möglich. Tritt beim Mergen ein Konflikt auf, so wird an dieser Stelle ein Dialogfenster eingeblendet, wodrauf der Entwickler das entstandene

Problem manuell lösen kann (siehe Kapitel 9.4). In der unten aufgeführten Abbildung 3.5 sehen wir wie man einen neuen Release erzeugt.

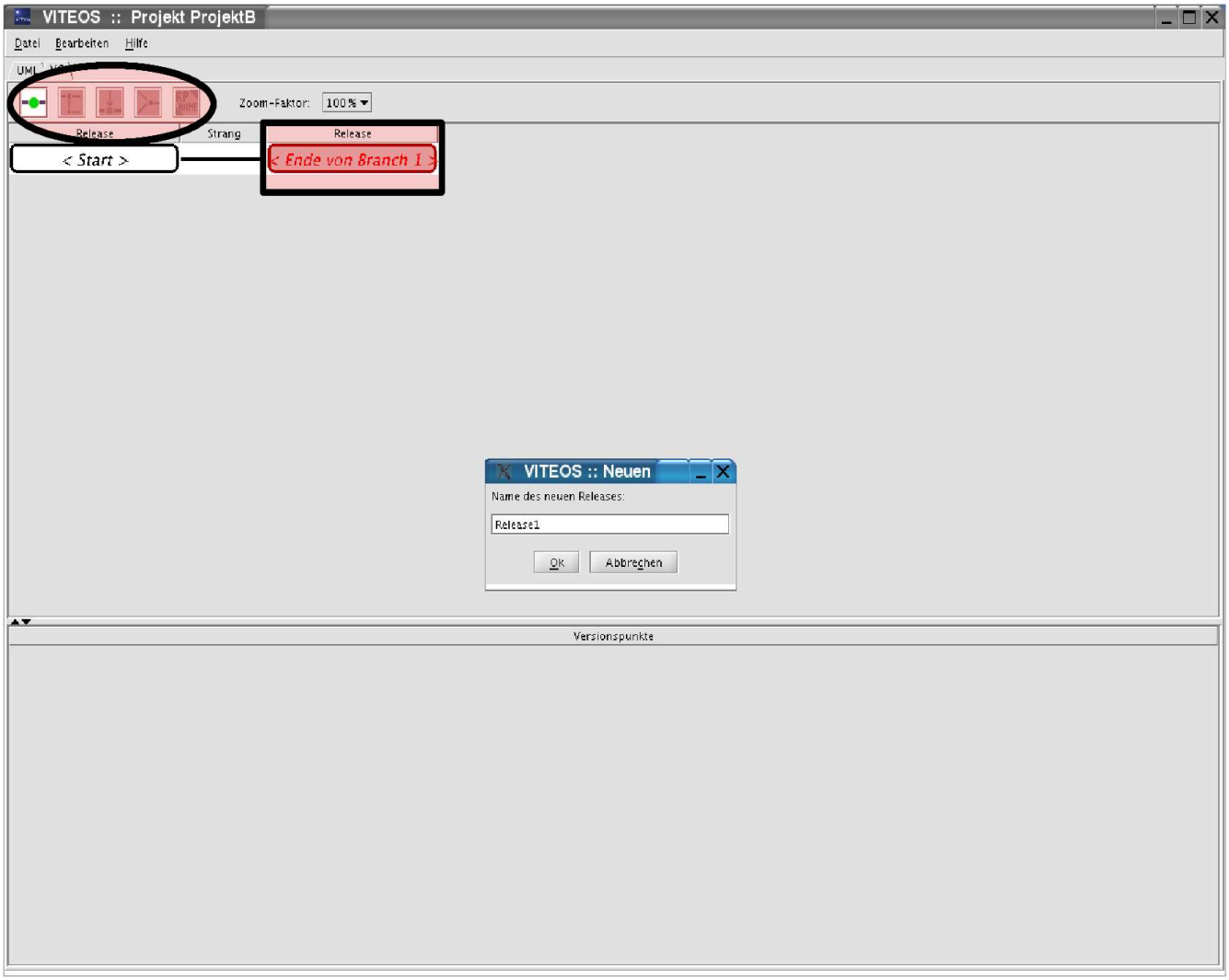

Abbildung 3.5: Neuen Release erzeugen

Auf der Toolbar beim VersionControl haben wir folgende Buttons mit folgenden Funtionen:

Echten Release erzeugen: Dieser Button dient dazu einen generischen Release in einen echten umzufunktionieren.

Verzweigen: Mit diesem Button kann man an echten Releases verzweigen.

Release laden: Dieser Button dient dazu die im Modell existieren Releases nach belieben zu laden.

Mergen: Hiermit kann man Zweige miteinander verschmelzen.

Echten Release umbennenen: Mit diesem Button kann man benannte Releases umbennen.

Desweiteren ist auf der ToolBar eine Zoomfunktion integriert, welche dazu beiträgt bei großen Modellen mit n-Releases den Überblick zu behalten. Durch das heranzoomen und horizontal wie vertikal Scrollen des Modells erhält man einen eindeutig besseren Übersicht über die aufgebaute Struktur des zugrundeliegenden Modells.

Kapitel 4

Release 3

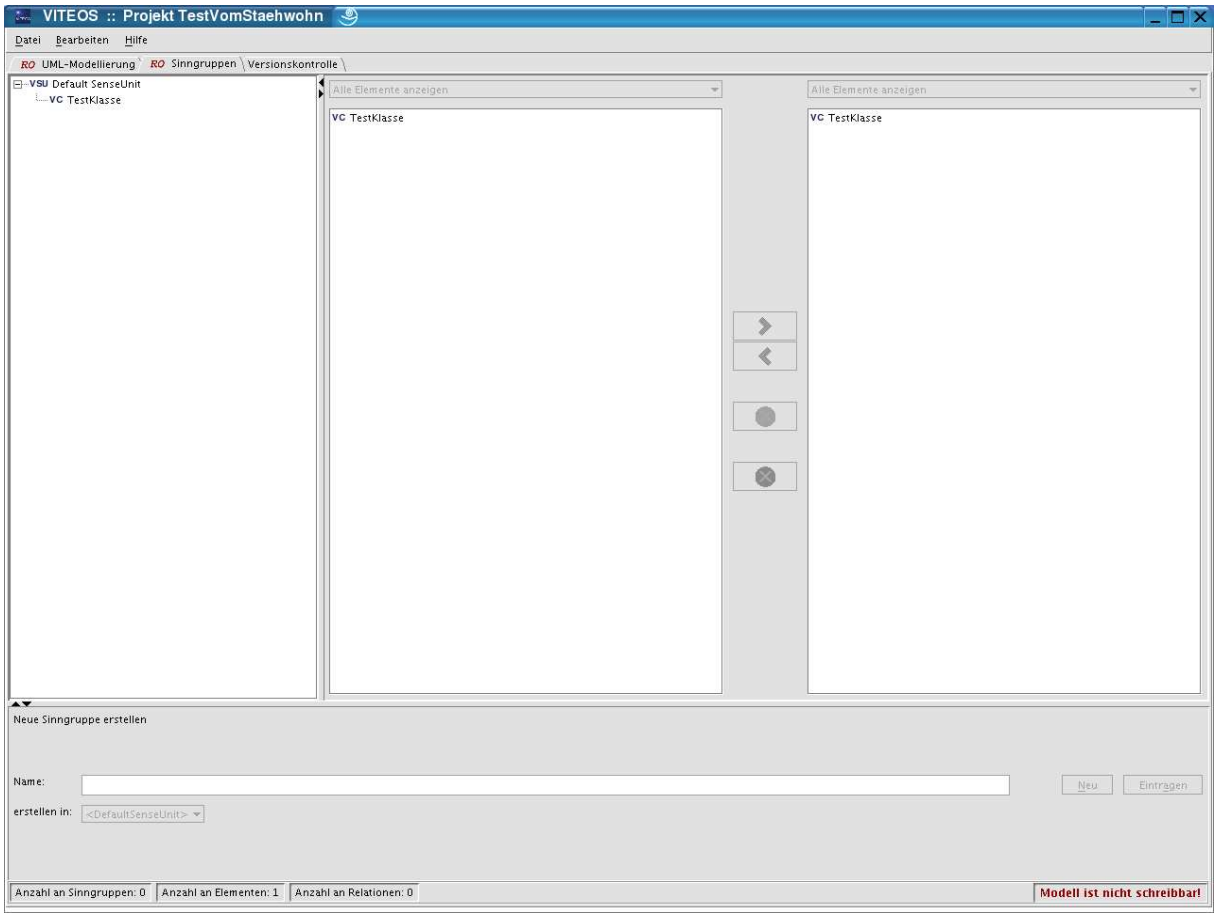

Abbildung 4.1: GUI : Sinn-Gruppe

Mit dem dritten Release sollte die Umsetzung der Sinngruppen ermöglicht werden. Die Sinngruppen-Perspektive wurde zum gegebenen Zeitpunkt realisiert. Der linke Baum auf dieser Perspektive ist das aktuelle Modell, welches wir auch aus der UML-Perspektive kennen. Hier hat das Ganze nur ein anderes PopupMenu mit den folgenden Auswahlkriterien um auf dem Modell arbeiten zukönnen:

- verschieben nach
- kopieren nach

- aus Sinngruppe entfernen
- umbennenen
- löschen

Auf der rechten Seite haben wir dann zwei Fenster wo man sich alle Elemente oder die gewünschte Sinngruppe anzeigen lassen kann. Auf diesen beiden Fenstern kann man dann Elemente von einer Sinngruppe in die andere verschieben (siehe Kapitel 9.5). Unten ist ein Editor eingefügt, welches das erstellen von neuen Sinngruppen ermöglicht. Bei der Weiterentwicklung der Oberfläche wurde noch eine zusätzliche Funktion eingefügt. Lädt man beispielsweise ein Release an dem man nicht mehr Branchen kann, dann erlaubt uns die GUI nur noch das Lesen des Modells. Dies wird einerseits durch *RO* (read only), welche auf den Karteireitern eingeblendet wird und anderseits auf der Statuszeile rechts durch das einblenden von, *Modell ist nicht schreibbar*, deutlich gemacht.

Teil III

Technische Struktur des Projektes *VITEOS*

Kapitel 5

Motivation

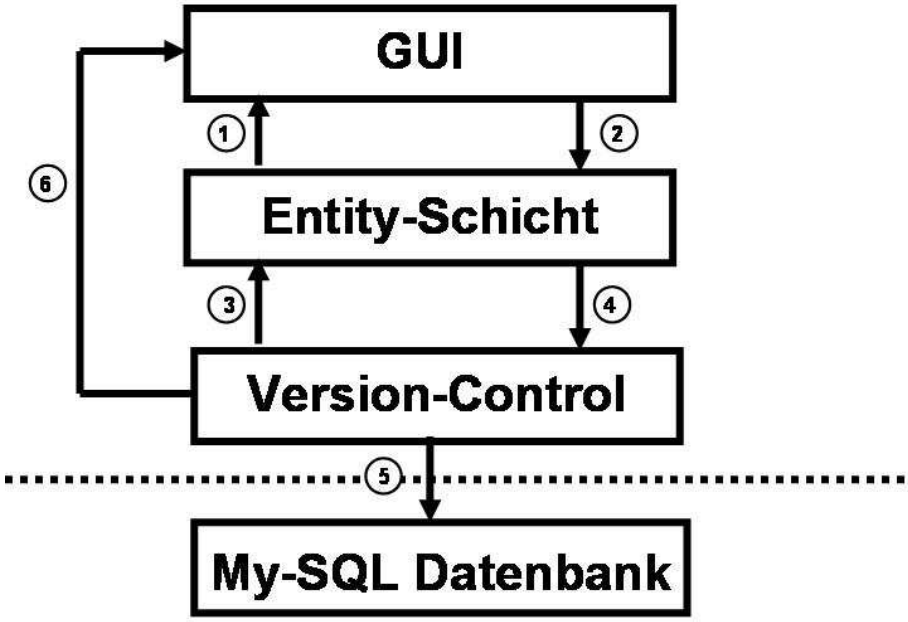

Abbildung 5.1: Die 4 *ViTEoS* -Ebenen

Für die Realisierung des Editors, wurde beschlossen, eine Trennung von Datenhaltung und Applikation vorzunehmen. Das *ViTEoS*-Projekt wurde in 4 Ebenen abstrahiert. Abbildung 5.1 stellt eine Art Metamodell der Ebenen dar. Es wurde eine graphische Oberfläche (`GUI`), mit den verschiedenen Funktionalitäten des Editors, entwickelt. Für die Datenhaltung wurde eine *MySQL*-`Datenbank` aufgesetzt. In `VERSION-CONTROL` wird versioniert. Diese Ebene ist auch für die Speicherung und das Auslesen von Daten zuständig. Sie hat aber, vor allem, den Zweck die darunterliegende Datenbank zu kapseln. Die gleiche Version-Control-Schnittstelle sollte idealerweise benutzt werden, wenn eine andere Datenbank (z.B. CVS) zugrundegelegt wird. Zwischen `VERSION-CONTROL` und `GUI` wurde eine Abstraktionsebene (`ENTITY-SCHICHT`) realisiert. Hier befindet sich alle UML-Objekte, die auch im Modell sind. Die Pfeile zwischen den Schichten stellen eine Kommunikationsbeziehung dar. `GUI` zeigt den Inhalt der `ENTITY-SCHICHT` an. Die `ENTITY-SCHICHT` repräsentiert, als Datenstruktur oder Container für Information, das Modell, das der Benutzer erstellt hat. Wird im Editor eine Änderung vorgenommen, so wird in `ENTITY-SCHICHT` eine entsprechende Änderung gemacht (Pfeil 2). Die geänderte Entity-Objekte werden an `VERSION-CONTROL` weitergeleitet (Pfeil

4). In `VERSION-CONTROL` wird eine neue Version erzeugt und in der Datenbank gespeichert (Pfeil 5). In der anderen Richtung, holt `VERSION-CONTROL` beispielsweise ein Modell, auf Anfrage, aus der Datenbank (Pfeil 5) und greift auf die `ENTITY-SCHICHT` zu (Pfeil 3), wo die Daten gehalten werden. Die `ENTITY-SCHICHT` wiederum greift auf `GUI` zu (Pfeil 1), wo das Modell angezeigt wird. Die Version-Control-Schicht kann auch direkt auf die Graphische Oberfläche zugreifen um dort Änderungen vorzunehmen (Pfeil 6). In diesem Teil wird die technische Struktur des Projektes und die verschiedene Schichte detailliert beschrieben.

5.1 Die Klasse-Übersicht des *VITEOS* -Tools

Um die Zusammenarbeit der verschiedenen Schichte verständlicher zu machen, wollen wir in diesem Abschnitt die einzelne Schichte und die in ihnen enthaltenen Klassen näher erlätern. Hier sind die Unterteilungen der Datenbank-Klassen in Schichten:

5.1.1 Die Entity-Schicht

Die Entity-Schicht umfasst folgenden Klassen:

- IModelObserver
- ModelMergingControl
- VAssociation
- VAggregation
- VClass
- VersionPoint
- VersionRelease
- VMethod
- VModel
- VModelElement
- VModelElementWithParent
- VComposition
- VAttribute
- VRelation
- VElement
- VSenseUnit
- VImplementation
- VInheritance

5.1.2 Die Version-Control-Schicht

Die Version-Control-Schicht enthält folgenden Klassen:

- VersionBase
- VersionAttribute
- VersionSenseUnit
- VersionModelTools
- VersionClass
- VersionControl
- VersionMethod
- VersionModelLoader
- VersionRelation

Das Klassendiagramm in Abbildung 5.2 veranschaulicht die Beziehungen der verschiedenen Klassen der Entity- und Datenbank-Schichten. In den nächsten Kapitel werden die einzelne Klasse detalliert beschrieben.

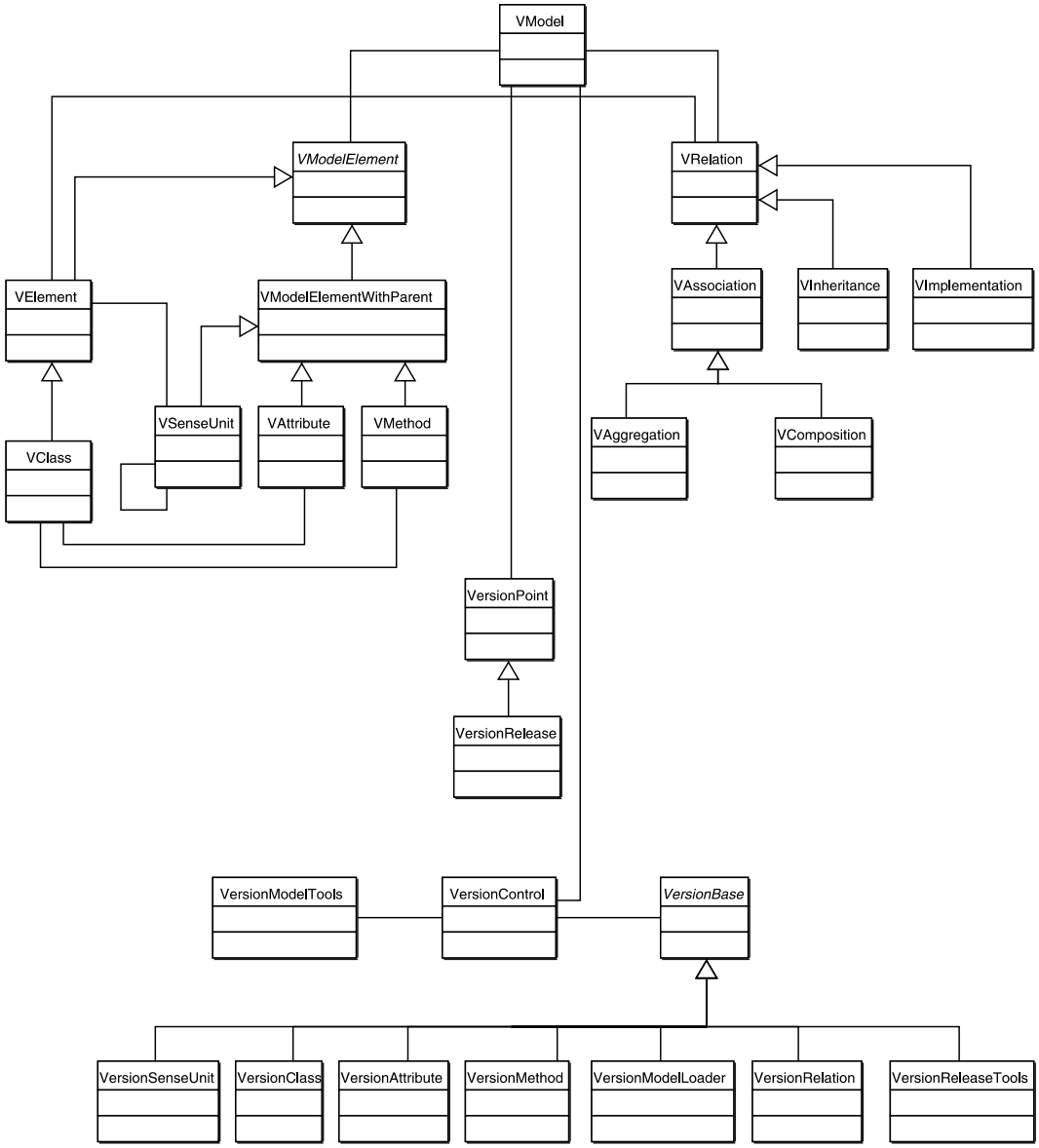

Abbildung 5.2: Übersicht der Klassen

Kapitel 6

Die Entity-Klassen

6.1 Die Klasse-Übersicht der Entity-Schicht

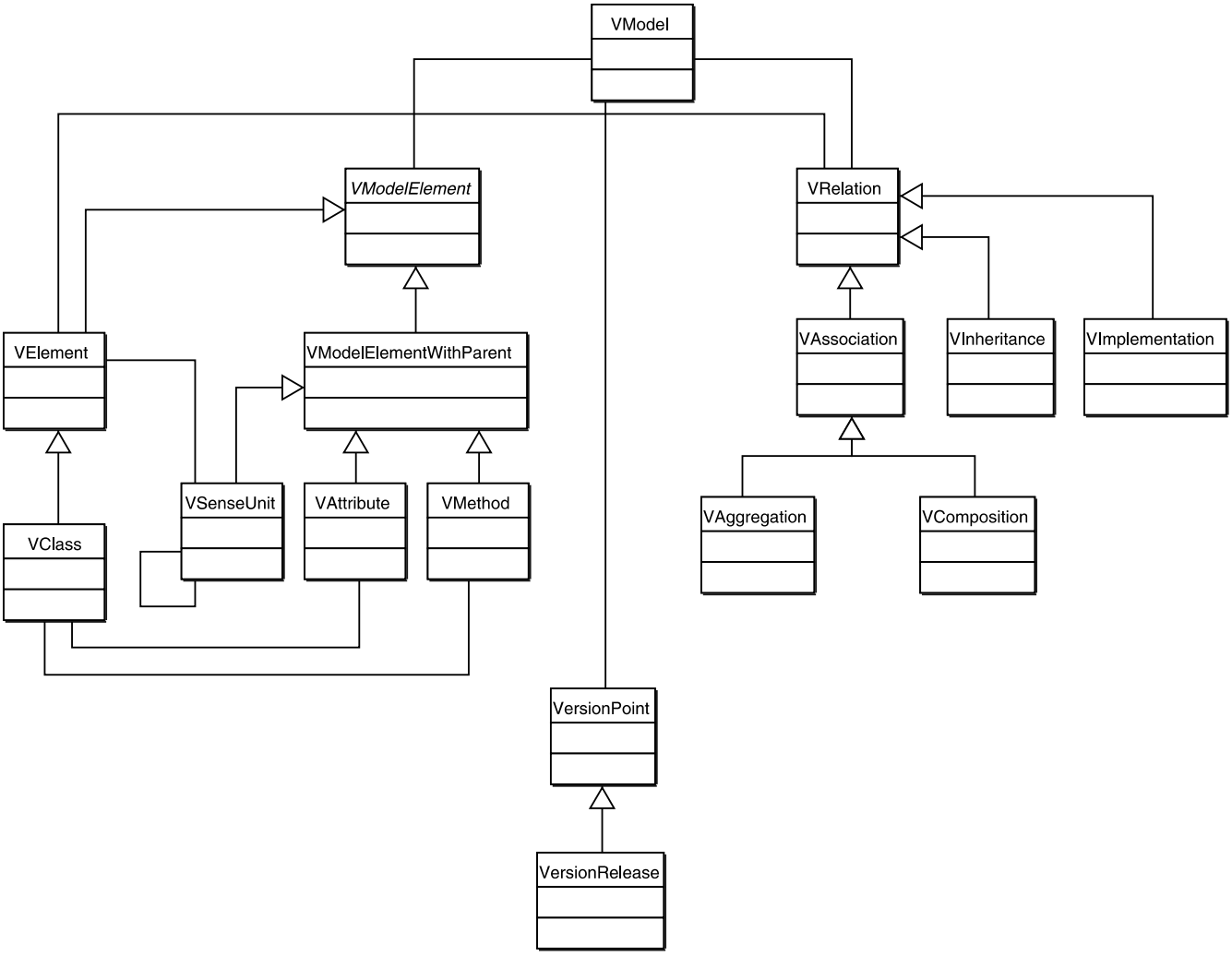

Abbildung 6.1: Die Entity-Klassen

6.2 *IModelObserver.java*

Das Interface *IModelObserver.java* implementiert Swing-Models wie das AdjacencyMatrixTabelModel um sich bei VModel als Observer registrieren zu können. Bei Änderungen am VModel, werden die Swing-Models dann benachrichtigt. Hier wurden die folgende Methoden deklariert.

- elementAdded(VElement elm): Mit dieser Methode wird einem Model ein Element hinzugefügt.
- elementAdded(VSenseUnit parent, VElement elem): Ein neues Element wird dem Modell hinzugefügt, wobei direkt eine Sinngruppe spezifiziert wird in die es eingefügt werden soll.
- elementInsertedIntoSenseUnit(VSenseUnit parent, VElement elem): Ein existierendes Element elem wird in eine Sinngruppe eingefügt.
- elementChanged(VElement elem): Ein Element wird geändert (z.B. neuer Name, abstract, final oder Interface).
- elementRemoved(VElement elem): Ein Element wird aus dem Modell entfernt.
- elementRemovedFromSenseUnit(VSenseUnit parent, VElement elem): Ein Element wird aus einer Sinngruppe entfernt.
- relationChanged(VRelation rel): Eine Relation wird geändert: Label, Start- oder End-Multiplizität wird geändert. Die Methode deckt jedoch nicht den Fall ab, dass Start- und End-Element geändert werden.
- relationChanged(VRelation oldRel, VRelation newRel): Eine Relation wird geändert, indem sie durch eine andere von einem anderen Typ ersetzt wird.
- relationAdded(VRelation rel): Eine neue Relation wird dem Modell hinzugefügt.
- relationRemoved(VRelation rel): Eine bestehende Relation wird aus dem Modell entfernt.
- senseUnitAdded(VSenseUnit su): Eine neue Sinngruppe wird dem Modell hinzugefügt.
- senseUnitAdded(VSenseUnit parent, VSenseUnit su): Ein neues Element wird dem Modell hinzugefuegt, wobei direkt eine Sinngruppe spezifiziert wird in die es eingefuegt werden soll.
- senseUnitInsertedIntoSenseUnit(VSenseUnit parent, VSenseUnit su): Eine existierende Sinngruppe wird in eine andere Sinngruppe eingefügt.
- senseUnitRemoved(VSenseUnit su): Die Methode entfernt eine Sinngruppe aus dem Modell.
- senseUnitRemovedFromSenseUnit(VSenseUnit parent, VSenseUnit su): Eine Sinngruppe wird aus einer anderen Sinngruppe entfernt, jedoch nicht aus dem Modell.

6.3 *ModelMergingControl.java*

Die Klasse dient dazu, Konflikte beim Einfügen einer Liste von Objekten aus einem anderen Branch in den aktuellen Branch zu ermitteln. Sie werden als ArrayList von Diff-Objekten zurückgegeben. Hier werden die folgenden Methoden implementiert:

- getMergeConflicts(ArrayList mergeList): Sie liefert alle Konflikte, die beim Mergen auftreten.

- addIntoModel(VClass diffClass, VClass modelClass): Die Methode fügt dem Modell eine Klasse, Methode oder ein Attribut hinzu.
- changeModelClass(VClass diffClass): Überschreibt eine ModelKlasse, indem die Klassen-Eigenschaften neu gesetzt werden. Anschließend werden die Attribute und Methoden überschrieben. Sollte die Klasse in einem Interface geändert werden, müssen die bestehende Attribute entfernt werden.
- changeModelRelations(): Sie ändert die Relation des Merge-Modells sodass es 2 gerichtete Relationen zwischen Start- und End-Element gibt. Ist im aktuellen Modell keine entsprechende Relation vorhanden, so wird sie hinzugefuegt.
- existingInheritanceCircle(VClass startClass, VClass endClass): Sie liefert einen boolesche Wert, ob die Relation des Modells eine Vererbung ist.
- rekCircleSearch(VClass startClass, VClass endClass): Sie liefert einen booleschen Wert, ob die Start- und End-Elemente Klassen sind.
- classTypMergeChange(VClass changeClass): Sie ändert den Klassentyp durch das Mergen.
- classTypChange(VClass changeClass): Die Methode ruft die update-Methode, wenn den Klassen-Typ durch den Benutzer geändert wird.
- deleteFalseRelations(VClass changedClass): Sie buffert der zu ändernden Relationen. Anschließend werden die Relationen gelöscht und die Menge der gebufferte Relationen wird zurückgegeben.
- classRelationCheck(TreeMap relTree): Sie fügt eine neue Relation hinzu.
- loadMergingModel(VersionPoint vp): Sie liefert das Merge-Modell der Versionspunkt vp.

6.4 *VAssociation.java*

Diese Klasse erbt von der Klasse VRelation und stellt eine Assoziation zwischen zwei Klassen dar. Sie implementiert die Methode getMirroredAssociation. Bei ungerichteten Assoziationen gibt die Methode die invertierte Assoziation der betroffenen Assoziation zurück. Das ist in dem Fall ein Objekt, das identisch ist bis auf den Unterschied, dass das Start- und End-Element vertauscht sind. Da die Methode nur für ungerichtete Assoziationen vorgesehen ist, wird eine Exception ausgelöst, wenn die Assoziation gerichtet ist. Damit die Methode von den Subklassen nicht überschrieben wird, wird sie als final deklariert.

6.5 *VAggregation.java*

Diese Klasse ist eine Unterklasse der Klasse *VAssociation* und stellt eine Aggregation bzgl. UML dar.

6.6 *VClass.java*

Sie erbt von VElement und stellt die Klassen gemäß UML mit der folgenden Attributen dar:

- description: Ein String, der die Klasse beschreibt.
- stereotype: Ein String, der den Stereotyp der Klasse speichert.
- listOfAttributes: Ein Array, das eine Liste der Attribute darstellt.
- isAbstract: Ein boolescher Wert, der angibt, ob die Klasse abstrakt ist.
- isFinal: Ein boolescher Wert, der angibt, ob die Klasse final ist.
- isInterface: Ein boolescher Wert, der angibt, ob es sich um ein Interface handelt.

6.7 *VersionPoint.java*

Sie stellt einen Versionspunkt mit einer ID und einer Beschreibung dar.

6.8 *VersionRelease.java*

Diese Klasse ist eine Unterklasse der Klasse VersionPoint und stellt ein Release dar.

6.9 *VMethod.java*

Sie erbt von der Klasse VModelElementWithParent. Sie implementiert Methoden die die Signatur der Methoden setzt und zurückgibt. Die Klasse hat die folgenden Attribute:

- name: Ein String der den Name der Methode speichert
- visibility: Ein String der die Sichtbarkeit der Methode angibt.
- parameter: Ein Vektor der die Parameterliste darstellt.
- returnValue: Ein String der den Rückgabewert der Methode festhält.
- isStatic: Ein boolescher Wert, der angibt, ob die Methode static ist.
- isAbstract: Ein boolescher Wert, der angibt, ob die Methode abstrakt ist.
- isFinal: Ein boolescher Wert, der angibt, ob die Methode final ist.

6.10 *VModel.java*

Die Klasse implementiert die folgenden Methoden:

- addElement(VElement element): Die Methode fügt ein neues Element in der Liste hinzu und setzt eine neue Liste zur Verwaltung der Schlüssel von VRelationen, an denen das Element beteiligt ist. Anschließend werden alle ModelObserver benachrichtigt.

- removeElement(VElement element): Die Methode holt die Schlüssel aller VRelationen, an denen das Element beteiligt ist. Die VRelationen werden anhand der Schlüssel aus der Liste listOfRelation entfernt. Ferner wird das Element aus allen Sinngruppen entfernt. Anschließend werden alle ModelObserver benachrichtigt.

- addRelation(VRelation rel): Sie fügt eine neue Relation hinzu. Als Schlüssel für jede Relation wird der Name des Start- und EndElements verwendet. Dieser Schlüssel wird dann in relationKeys aufgenommen und zwar in die SchlüsselListen von Start- und EndElement. Ist die Relation eine Assoziation und ist sie ungerichtet, so wird die "invertierte" oder "gespiegelte" Assoziation ebenfalls eingefügt. Ist die Assoziation ungerichtet und stimmen Start- und End-Elemente nicht überein, so wird eine gespiegelte Assoziation erzeugt. Anschließend werden alle ModelObserver benachrichtigt.

- addMirroredAssociation(VAssociation as): Sie fügt eine "gespiegelte" Assoziation hinzu und benachrichtigt alle ModelObserver.

- removeRelation(VRelation rel): Die Methode entfernt die Relation mit dem Schlüssel rel.startElement.name + rel.endElement.name sowie die Teilschlüssel in relationKeys aus der Liste listOfRelation. Handelt es sich um eine gerichtete Assoziation, so wird ebenfalls die "invertierte" oder "gespiegelte" Assoziation gelöscht. Anschließend werden alle ModelObserver benachrichtigt.

- removeMirroredAssociation(VAssociation as): Die Methode löscht die gespiegelte Assoziation von dem Objekt as und gibt das gelöschte Objekt zurück.

- changeRelation(VRelation oldRel, VRelation newRel): Sie wird aufgerufen, wenn eine Relation durch eine andere von einem anderen Typ ersetzt wird (z.B. Assoziation durch Komposition). Wichtig dabei ist, dass die neue und alte Relationen diesselbe ID haben. Anschließend werden alle ModelObserver benachrichtigt.

- changeRelation(VRelation rel): Die Methode wird beim Ändern einer Relation aufgerufen, z.B. beim Ändern eines Labels oder einer Multiplizität.

- directedAssociationDirectionRemoved(VAssociation as): Die Methode entfernt eine Assoziation und fügt eine gespiegelte Assoziation ein.

- undirectedAssociationDirectionAdded(VAssociation as): Hier wird einer ungerichteten Assoziation eine Richtung gegeben und die gespiegelte Assozaition entfernt, dabei wird ermittelt, welche der beiden die spezielle ID für gespiegelte Assoziationen - VModelElement.ID_MIRRORED_ASSOCIATION - besitzt.

- getMirroredAssociation(VAssociation as): Sie gibt die zu einer Assoziation gespiegelte Assoziation zurück.

- addSenseUnit(VSenseUnit su): Sie fügt eine neue Sinngruppe hinzu und benachrichtigt alle ModelObserver.

- removeSenseUnit(VSenseUnit su): Sie löscht eine vorhandene Sinngruppe und benachrichtigt alle ModelObserver.

- joinSenseUnit(VSenseUnit su1, VSenseUnit su2): Sie vereinigt 2 Sinngruppen

- getName(): Die Methode gibt den Name zurück.

- getListOfRelations(): Sie gibt die Liste der Relationen zurück.
- getListOfElements(): Sie gibt die Liste der Elemente zurück.
- getListOfSenseUnits(): Sie gibt die Liste der Sinngruppe zurück.
- setListOfElements(ArrayList listOfElements): Sie setzt die Liste der Relationen.
- setName(String string): Sie setzt den Name auf string.
- setListOfSenseUnits(ArrayList senseUnits): Sie setzt die Liste der Sinngruppen.
- getVersionPoint(): Sie liefert einen Versionspunkt zurück.
- setListOfRelations(TreeMap listOfRelations): Sie setzt die Liste der Relationen auf listOfRelations
- setVersionPoint(VersionPoint point): Sie setzt den Versionspunkt auf point.
- getRelationKeys(): Sie liefert ein TreeMap mit den Schlüsseln zurück.
- getRelation(VElement startElement, VElement endElement): Sie liefert zu zwei Elementen das VRelation-Objekt zurück, das sie verbindet, dabei wird die VRelation direkt aus listOfRelation geholt.
- mirrorUndirectedAssociations(): Sie wird nach dem Auslesen des kompletten Modells aus der Datenbank einmal aufgerufen. Alle ungerichteten Assoziationen werden dann gespiegelt. Dies geschieht durch Aufruf von VAssociation.getMirroredAssociation().
- createRelationKey(VElement start, VElement end): Sie erzeugt die Schlüssel, die in der TreeMap listOfRelations auf Relationen gemappt werden.
- createRelationKey(VRelation rel): Sie erzeugt die Schlüssel, die in der TreeMap listOfRelations auf Relationen gemappt werden.
- registerModelObserver(IModelObserver observer): Sie fügt den ModelObserver, observer, hinzu.
- unregisterModelObserver(IModelObserver observer): Sie entfernt den ModelObserver observer.
- elementChanged(VElement elem): Sie wird von Elementen aufgerufen, wenn sich ihr Zustand geändert hat: Name, abstract, final, Interface.
- elementMethodAdded(VMethod m): Sie wird von Elementen aufgerufen, wenn eine neue Methode hinzugefügt wird.
- elementMethodDeleted(VMethod m): Sie wird von Elementen aufgerufen, wenn eine Methode gelöscht wird.
- methodChanged(VMethod m): Sie wird von Elementen aufgerufen, wenn eine Methode geändert wird.
- elementAttributeAdded(VAttribute a): Sie wird von Elementen aufgerufen, wenn ein Attribut hinzugefügt wird.

- elementAttributeDeleted(VAttribute a): Sie wird von Elementen aufgerufen, wenn ein Attribut gelöscht wird.
- attributeChanged(VAttribute a): Sie wird von Elementen aufgerufen, wenn ein Attribut geändert wird.
- initLogger(): Hier wird Logging initialisiert.

6.11 *VModelElement.java*

Sie ist eine abstrakte Klasse die das Modell-Element verwaltet und implementiert die folgenden Methoden, die für die Versionierung benötigt werden:

- setDbID(int i): Sie setzt die Datenbank-ID, dbID auf i.
- getDbID(): Sie liefert die Datenbank-ID zurück.
- getVModel(): Sie liefert das Model, vModel zurück.
- setVModel(VModel model): Sie setzt das Modell auf model.
- setVersion(int v): Sie setzt die Version auf v.
- getVersion(): Sie liefert die Version zurück.

Mit dem Attribut ID_NEW_ELEMENT wird die ID, für ein neues Element, das noch nicht in der Datenbank abgelegt ist, auf -1 gesetzt. Damit ungerichtete Assoziationen in der Adjazenz-Matrix in der gespiegelten Zeile auftauchen, wird das Attribut ID_MIRRORED_ASSOCIATION benötigt. Der Wert wird auf -10 gesetzt, weil die gespielte Assoziation niemals in der Datenbank abgelegt werden sollte.

6.12 *VModelElementWithParent.java*

Diese Klasse erbt von der Klasse VModelElement und ermöglicht dass der Vater des Elements zurückgegeben werden kann. Z.B. Attribut kennt Klasse.

6.13 *VComposition.java*

Sie ist eine Unterklasse der Klasse VAssociation und stellt eine Composition bzgl. UML dar.

6.14 *VAttribute.java*

Sie erbt von der Klasse VModelElementWithParent und repräsentiert ein Attribut. Mit der Klasse werden die Eigenschaften(name, type, visibility, usw.) eines Attributs gesetzt und zurückgegeben.

6.15 *VRelation.java*

Diese Klasse erbt von der Klasse VModelElement. Die Klasse stellt ein Objekt mit 2 Klassen(startElement und endElement) zur Verfügung und gibt an, ob die Relation zwischen den Elementen gerichtet oder ungerichtet ist.

6.16 *VElement.java*

Sie ist eine Unterklasse der Klasse VModelElement und implementiert die folgenden Methoden:

- getListOfMethods(): Sie liefert als ArrayList eine Liste aller Methoden.
- getName(): Sie liefert den Name des Elements zurück.
- setListOfMethods(ArrayList listOfMethods): Sie setzt die Liste der Methoden.
- setName(String string): Sie setzt den Name des Elements.
- addMethod(VMethod m): Sie fügt die Methode (m) in der Liste der Methoden hinzu.
- removeMethod(VMethod meth): Sie entfernt die Methode (meth) aus der Liste der Methoden
- getVisibility(): Sie liefert die Sichtbarkeit des Elements als String zurück.
- setVisibility(String string): Sie setzt die Sichtbarkeit des Elements.

6.17 *VSenseUnit.java*

Sie erbt von der Klasse VModelElement. Die Klasse verwaltet das Hinzufügen oder Löschen eines Elements in eine oder aus einer Sinngruppe. Ferner ermöglicht sie das Hinzufügen einer neuen, oder das Löschen einer bereits existierenden Sinngruppe.

6.18 *VImpementation.java*

Sie erbt von der Klasse VRelation. Die Klasse beschreibt eine Implementation und stellt eine gerichtete Beziehung zwischen 2 Objekten dar.

6.19 *VInheritance.java*

Sie erbt gemäß UML von der Klasse VRelation. Die Klasse stellt ein Objekt mit 2 Klassen(startElement und endElement) zur Verfügung und gibt an, ob die Relation zwischen den Elementen gerichtet oder ungerichtet ist.

Kapitel 7

Die Datenbank-Klassen

7.1 Die Klasse-Übersicht der Datenbank-Schicht

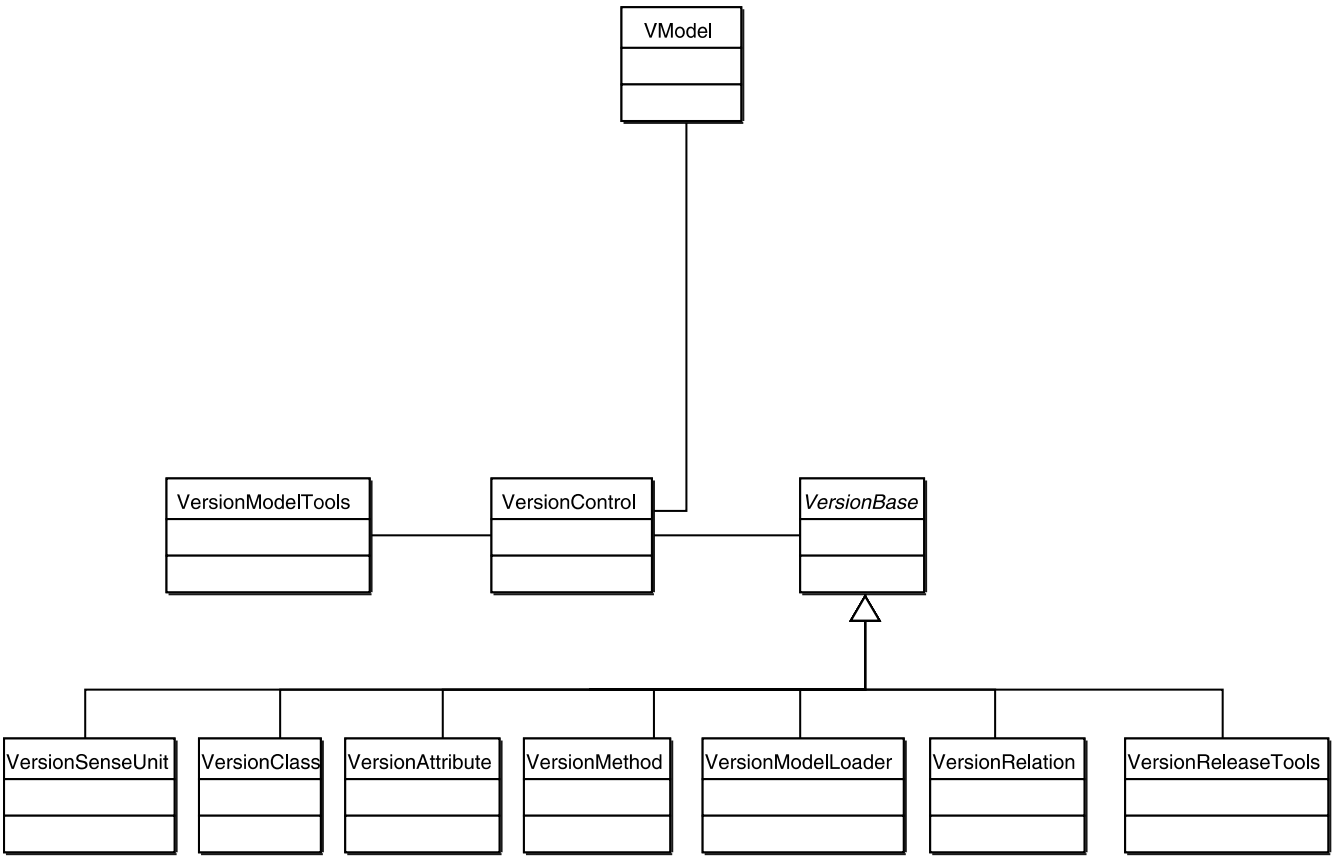

Abbildung 7.1: Die Datenbank-Klassen

7.2 *PreparedStatementLogable.java*

Sie implementiert die Klasse PreparedStatement und gibt in die Konsole aus, ob die SQL Statements erfolgreich ausgeführt wurden oder nicht.

7.3 *VersionBase.java*

Sie ist eine abstrakte Klasse, die die nächste ID ermittelt und sie in der Tabelle Konfigurationen schreibt. Sie schreibt außerdem vor, dass alle von ihr erbenden Klassen die Methode update() implementieren müssen.

7.4 *VersionAttribute.java*

Sie erweitert die Klasse VersionBase und implementiert die folgenden Methoden:

- deleteAttribute(VAttribute attrib): Diese Methode setzt den Parameter "KlasserVersionBis" des vorgegebenen VAttribute-Objekts auf die aktuelle Versionsnummer, dadurch gilt das Attribut als "gelöscht". Anschließend wird die Versionnummer der Klasse inkrementiert und eine neue Konfiguration erstellt.
- addOrChangeAttribute(VAttribute attrib, int action): Sie fügt ein neues Attribut ein oder ändert die Versionsnummer des Attributes, falls es bereits vorhanden war und nur bzgl. seiner Einträge verändert wurde.
- update(VModelElement elem, int action): Sie ruft, basierend auf dem Wert des Interger-Parameters "action", die der kodierten Funktion entsprechende Methode auf.

7.5 *VersionSenseUnit.java*

Die Klasse erweitert die Klasse VersionBase und implementiert die folgenden Methoden:

- addNewSenseUnit(VSenseUnit elem): Sie fügt eine neue Sinngruppe hinzu.
- deleteSenseUnit(VSenseUnit elem): Sie löscht eine vorhandene Sinngruppe.
- changeSenseUnit(VSenseUnit elem): Sie ändert eine bereit existierende Sinngruppe.
- update(Object elem, int action): Je nachdem welche Aktion die Methode als Parameter bekommt, wird eine der 3 Methoden(addNewSenseUnit, deleteSenseUnit oder changeSenseUnit) aufgrufen. Daraufhin wird entweder eine Sinngruppe geändert, gelöscht oder hinzugefügt.

7.6 *VersionModelTools.java*

Die Klasse ließt die Datei jdbc.properties und beim erfolgreichen Anmelden, wird die Datenbank erzeugt. Anschließend werden alle `MySQL`-Statements ausgeführt. Sie hat außerdem die Methode getModels(), die beim Starten des Programms aufgerufen wird, um die in der Datenbank vorhandene Modelle anzuzeigen.

7.7 *VersionClass.java*

Die Klasse erweitert die Klasse VersionBase und implemetiert die folgenden Methoden:

- addNewClass(VClass elem): Methode für das Einfuegen einer neuen Klasse, der neuen Klasseneigemschaften und das Eintragen der neuen Klasse in eine neue Konfiguration.

- deleteClass(VClass elem): Methode zum löschen einer Klasse; Bei den zur Klasse gehoerenden Azzoziationen wird der Wert für "KlasseiBisVersion" gesetzt, dadurch gilt sie als gelöscht. Sie erzeugt auch eine neue Konfiguration.
- changeClass(VClass elem): Mit dieser Methode wird die Klassen-Eigenschaften geändert, dabei wird die Klasse-Version um eins erhöht. Anschließend wird eine neue Konfiguration erzeugt.

7.8 *VersionControl.java*

Die Klasse ist ein Singleton und initialisiert der Logging API zur Ausgabe von Infos über die Versionierung. Ferner wird ein Verweis auf den aktuellen Versionierungs-Logger zurückgegeben. Außerdem leitet sie die update-Aufrüfe an die entsprechenden Klassen weiter und stellt alle Methoden zur Verfügung, die für die Versionierung benötigt werden.

7.9 *VersionMethod.java*

Die Klasse erweitert VersionBase und definiert einen neuen Versionpunkt, in dem eine Methode gelöscht wurde.

7.10 *VersionModelLoader.java*

Die Klasse erbt von der Klasse VersionBase und erzeugt alle Klassen-Objekte und hängt dies in das Modell.

7.11 *VersionRelation.java*

Die Klasse VersionRelation erweitert die Klasse VersionBase und implementiert die folgenden Methoden.

- changeRelation(VRelation rel, String op): Die Methode unterscheidet zwischen Hinzufügen einer neuen und Ändern einer vorhanden Relation. Beim Hinzufügen wird in das Objekt die nächste ID, assID, geholt und die Version auf 1 gesetzt. Beim Ändern wird die Version im Objekt um 1 erhöht. Falls die Assoziation von einer Klasse auf sich selbst gerichtet ist, wird nur eine neue Konfiguration erzeugt.
- deleteRelation(VRelation elem): Mit dieser Methode wird eine bestehende Assoziation gelöscht. Die beiden Klassen der gelöschten Assoziation werden aktualisiert. Es werden zwei neue Versionen der Klassen erzeugt und zwei neue Konfigurationen der betroffenen Klasse werden geschrieben.
- createAssConfiguration(int konfigID, String text, int ClassID, int ClassVersion): Diese ist eine private Methode die das Erstellen einer neuen Konfiguration mit übergebenen Werten dient. Im Unterschied zu der gleichnamigen Methode in VersionBase, ist es möglich, zweimal die selbe Konfigurations-ID zu schreiben.
- update(VModelElement elem, int action): Je nachdem, um welche Aktion es sich handelt, wird eine der Methoden - addRelation(), changeRelation() oder deleteRelation - aufgerufen. Somit wird eine Relation hinzugefügt, geändert oder gelöscht.

Kapitel 8

Die GUI-Klassen

In diesem Kapitel werden die Klassen, die für die Realisieung der Benutzeroberfläche und für die Interaktion mit der Datenbank zuständig sind, aufgeführt und erläutert. Die Klassen sind in Packages aufgeteilt, so werden sie auch dargestellt. In der unten dargestellten Abbildung 8.1 sieht man

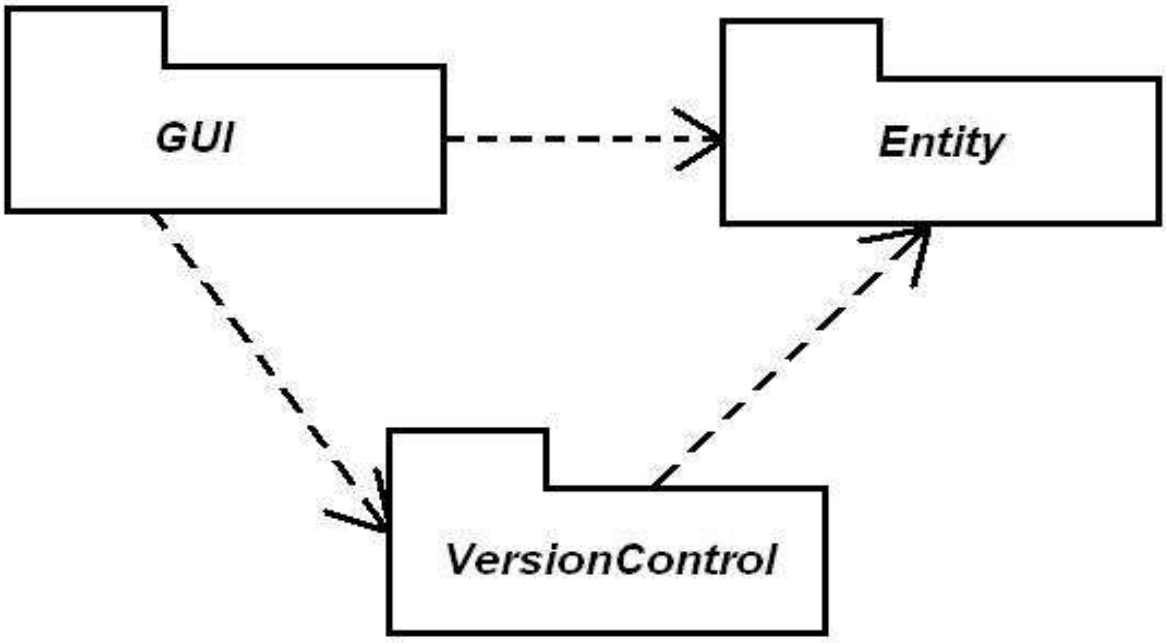

Abbildung 8.1: GUI : Interaktionsmodell

kurzerhand, wie die einzelnen Schichten, die in Kapitel 5 vorgestellt worden sind, untereinander interagieren. Über die GUI-Schicht kann man auf der Entity-Schicht arbeiten und diese anzeigen. Die Entity-Schicht stellt immer das aktuelle Modell dar. Über die GUI ist es möglich, das aktuelle Modell zur verändern. Änderungen am Modell werden von der GUI an die VersionControl weitergeleitet, welche dann die Änderungen in die Datenbank übernimmt. Wird das Modell durch die VersionControl auf die Entity abgelegt, so holt sich die GUI dieses Modell von der Entity und stellt das Modell visuell dar.
In dem Package *framework* gibt es eine Klasse *MainWindow*, die die anderen Packages (welche für die einzelnen Pespektiven zuständig sind) kennt. Siehe dazu Abbildung 8.2. Wie man klar er-

kennen kann, kennen sich die anderen Packages untereinander nicht. Ausser den beiden Packages *VersionControl* und *VersionControlMerge*, *VersionControlMerge* ist ein Unterpackage von *VersionControl*. Die Klasse *MainWindow* implementiert den *ChangeListener*, welcher das Hauptfenster bei

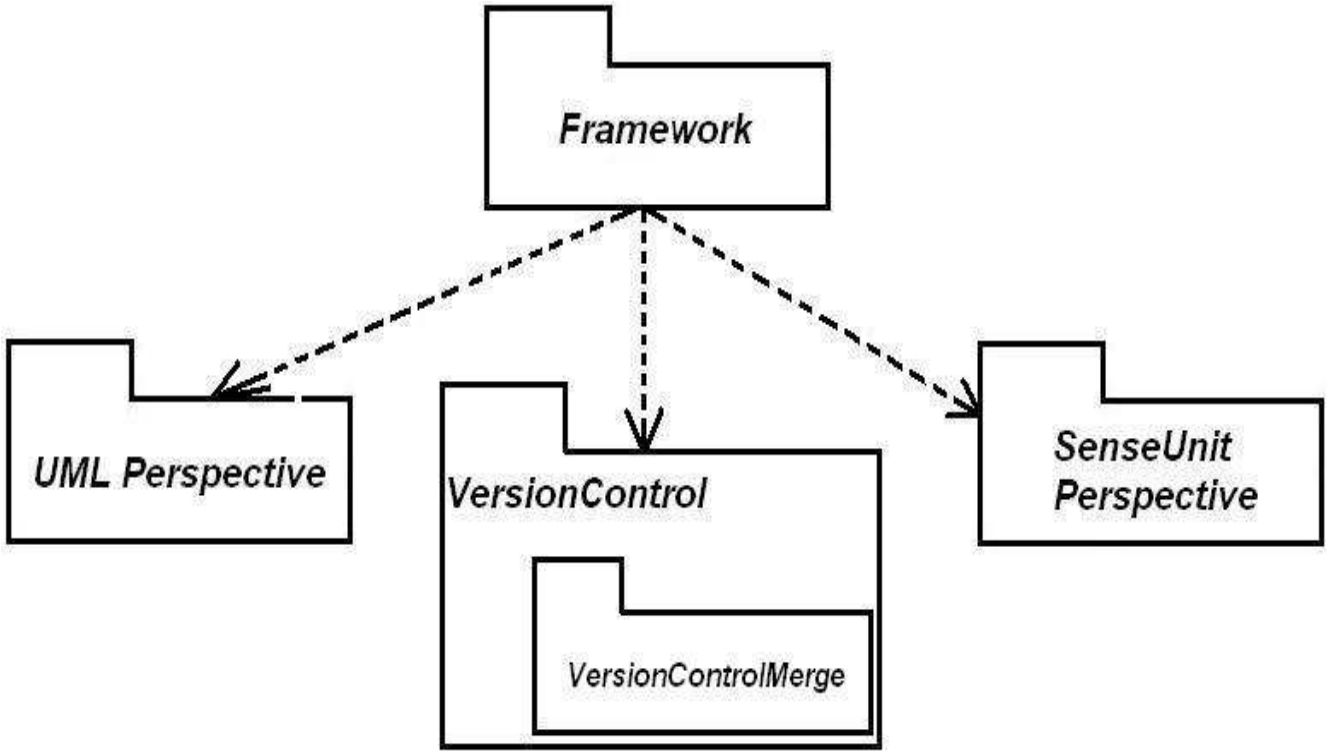

Abbildung 8.2: GUI : Packages im Überblick

der *JTabbedPane* als Überwacher anmeldet. Die *JTabbedPane* ist eine Klasse, die dem Entwickler erlaubt, zwischen den Verschiedenen Perspektiven, welche in einer Art von Karteireitern dargestellt sind, beliebig umzuspringen. Ist der Entwickler auf der VersionControl Perspektive und klickt beispielsweise die UML Perspektive an so wird diese angezeigt. Desweiteren geschieht dabei noch folgendes: dieser Perspektivenwechsel wird von dem *ChangeListener* erkannt und an die Klasse *VersionControlPerspektivePanel* weitergeleitet, welche die Methode *refreshVersionTree()* aufruft, die dann den Versionierungsbaum an die Klasse *VersionControlTreePanel* weiterleitet wo die Methode *refresh()* dafür sorgt das die Perpektive neu aufgebaut wird. In den folgenden Kapitel werden nun die einzelnen Packages detaillierter erläutert.

8.1 *Config.java*

Diese Klasse ist für das Lesen der Config-Dateien zuständig. Die allgemeine Konfiguration (z.B. der Oberfläche) soll von der Konfiguration für den Datenbank-Zugriff getrennt sein. Daher werden die 2 Dateien - config/viteos.properties und config/jdbc.properties gespeichert. Die Klasse ist als Singleton implementiert, um möglichst globalen Zugriff von jeder Klasse des Programms auf die Konfiguration zu ermöglichen.

8.2 *edu.unido.pg436.viteos.perspectives.framework*

In diesem Package werden die GUI-Klassen implementiert die nicht zu den drei Perspektiven gehören. Und natürlich auch die Klasse *MainWindow* die den sogenannten Rahmen des Tools *VITEOS* bildet, wo die drei Perspektiven drin enthalten sind.

8.2.1 *MainWindow.java*

Wie Anfangs erwähnt stellt diese Klasse das Hauptfenster des Programms dar und ist ein Singleton. Das Interface *ChangeListener* wird implementiert, um auf *ChangeEvents* der *JTabbedPane*, die die drei Perspektiven enthaelt, reagieren zu können. Wenn die Versionierungs-Perspektive nämlich angezeigt wird, so soll sie über die Versionskontrolle ein komplettes Update durchführen und sich neu zeichnen.

8.2.2 *StartDialog.java*

Ist das Dialogfenster, welches beim Starten des Tool eingeblendet wird. Durch diesen Fenster kann man entweder ein bestehendes Modell laden oder sich dafür entscheiden ein neues Modell anzulegen. Hat man sich für ein bestehendes Modell entschieden so kommt die Methode *getSelectedModel()* zum einsatz. Diese Methode lädt das selektierte Modell.

8.2.3 *NewProjectDialog.java*

Hat man sich im StartDialog dafür entschieden ein neues Projekt anzulegen, so wird man auf diesem Dialogfenster aufgefordert dem Projekt einen Namen zu geben. Nach dem man den Namen festgelegt hat, wird dieser noch mit dem OK-Button bestätigt. Das neue Projekt ist nun angelegt.

8.2.4 *AboutDialog.java*

Dieser Dialogfenster beinhaltet eine Liste über die Teilnehmer und den Namen des Veranstalters der Projektgruppe, einige Information über die Lizenz, unter dem das Projekt veröffentlicht wird, und eine Reihe von Dankaussagungen an verschiedenen Toolherstellern, deren Tools bei der Entwicklung des Projekts *VITEOS* eingesetzt worden sind. Siehe Abbildung 8.3. Die Liste der Autoren, die Lizenz sowie die Dankaussagungen holt sich die Klasse aus einer *HTML-Datei*.

Abbildung 8.3: GUI : Der AboutDialog

8.2.5 *ConfigDialog.java*

Diese Klasse erzeugt ein Dialogfenster, welches dem Benutzer die Möglichkeit bietet, die Datenbankparameter zu konfigurieren (einzustellen). Siehe Abbildung 8.4.

Abbildung 8.4: GUI : Der ConfigDialog

8.2.6 *ViteosSplashScreen.java*

Der Viteos-splash-screen ist ein Fenster, das angezeigt wird, während Daten im Hintergrund geladen werden. Siehe Abbildung 8.5

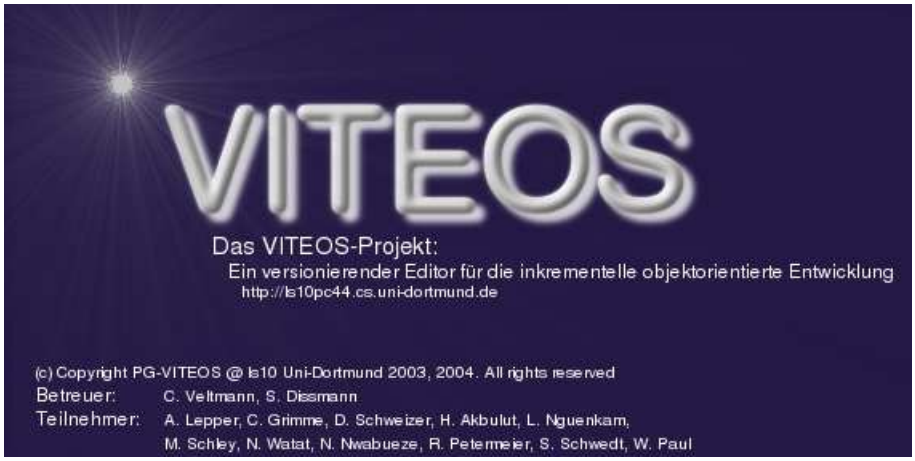

Abbildung 8.5: GUI : Der Splash Screen

8.3 *edu.unido.pg436.perspectives.uml*

In der unten aufgeführten Abbildung 8.6 sieht man die UML Perspektive. Ein Hauptfenster, unterteilt in drei Unterfenster :

- Baumstruktur *(links)*
- Adjazenz-Matrix *(rechts)*
- Editor *(unten)*

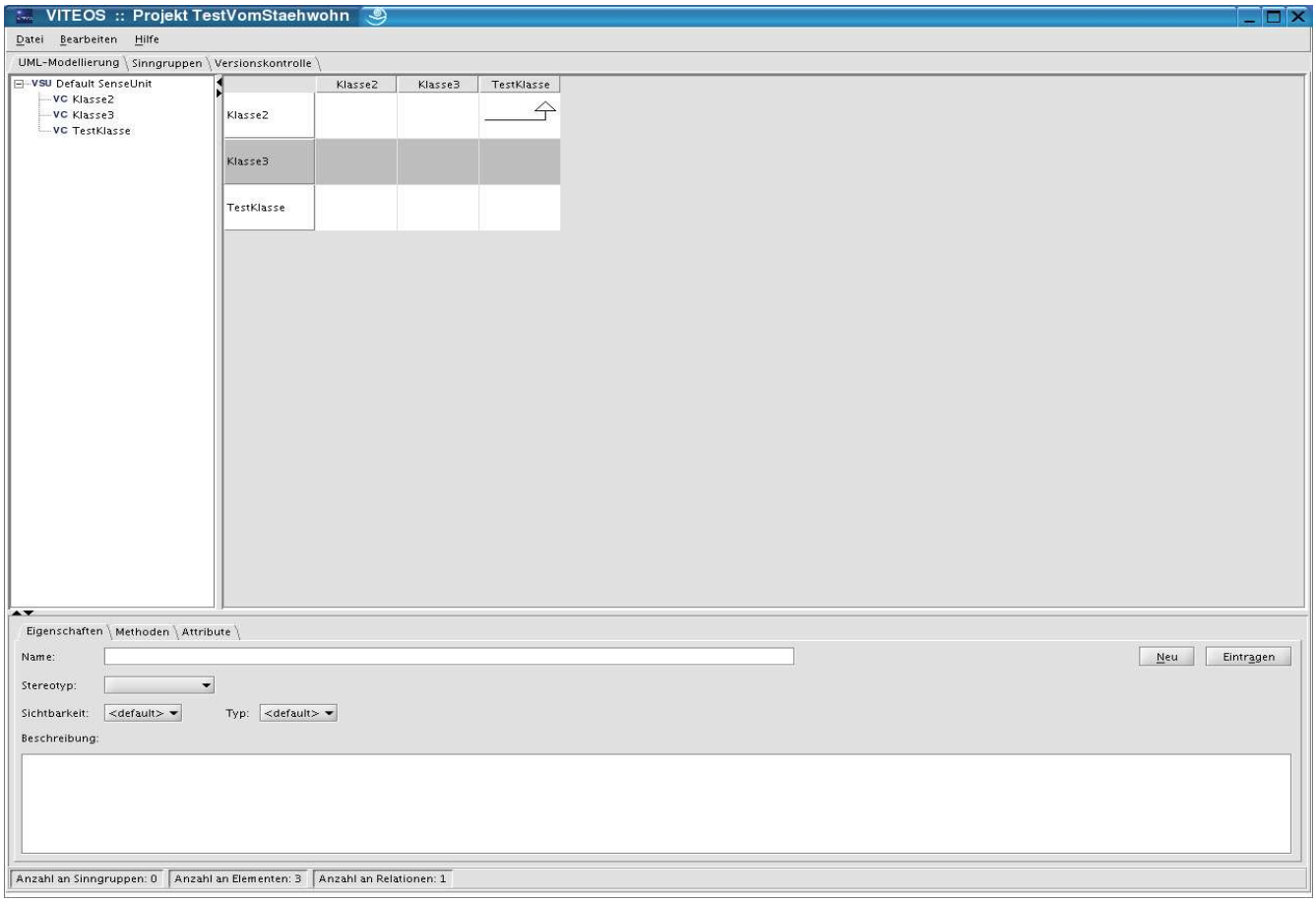

Abbildung 8.6: UML Perspective

Desweiteren ist auf dem Hauptfenster *(unten links)* eine Statuszeile integriert, welche den Benutzer informiert, wieviele Sinngruppen, Elemente und Relationen das akzuelle Modell beinhaltet. Im Folgenden werden die Klassen die für die Darstellung und Funktionen der Unterfenster zuständig sind, vorgestellt.

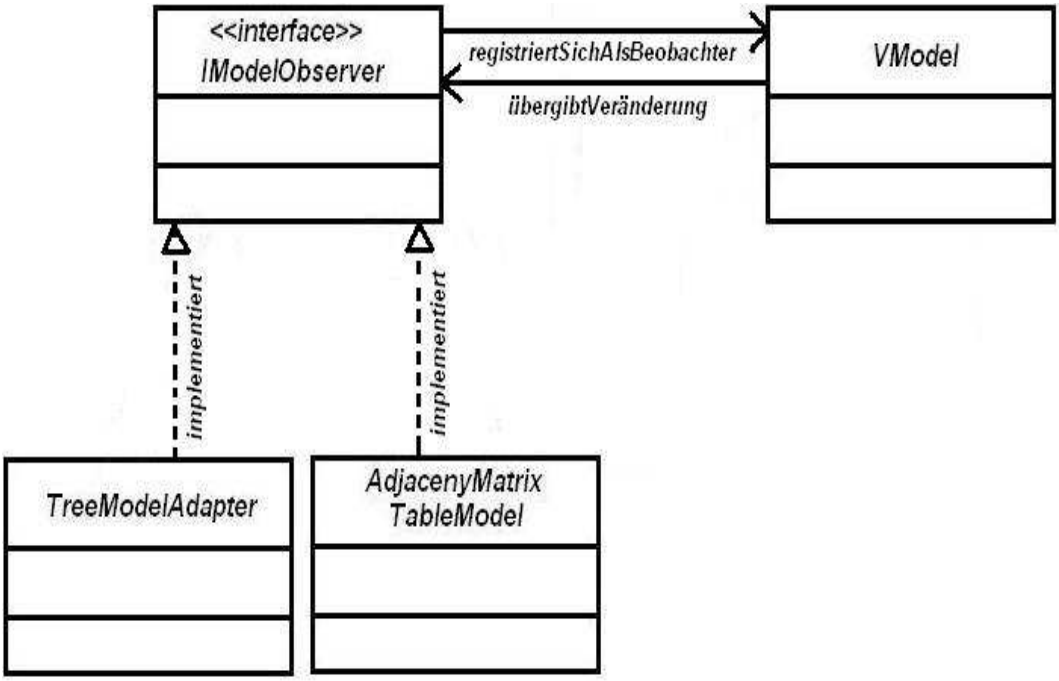

Abbildung 8.7: GUI : Klassendiagramm der Interaktionen

Auf der oben aufgeführten Abbildung 8.7 sieht man eine Klassenstruktur, die uns die Idee näherbringen soll, die diesem Package zugrunde liegt. Die Klasse *TreeModelAdapter* ist zuständig für die Darstellung der Baumstruktur und die Klasse *AdjacencyMatrixTable* für die Darstellung der Matrix. Beide Klassen implementieren die Schnittstelle *IModelObserver*. Diese Schnittstelle registriert sich als Beobachter in der Klasse *VModel*, in der sich immer das aktuelle Modell befindet. Ändert sich an dem aktuellen Modell etwas, beispielsweise wird eine neue Klasse eingefügt oder eine Relation wird verändert, so wird dies durch den *VModel* an den *IModelObserver* mitgeteilt. Somit kriegen die Klassen die den *IModelObserver* implementiert die Veränderung auch mit.

8.3.1 *Hauptfenster*

8.3.1.1 *UMLPerspectivePanel.java*

Dies ist die Hauptklasse in diesem Package, sie stellt das Hauptfenster mit den drei Unterfenstern dar. Desweiteren legt die Klasse fest oder besser gesagt bestimmt diese Klasse, ob das Modell editierbar ist oder nicht.

8.3.2 *Baumstruktur*

8.3.2.1 *TreeModelAdapter.java*

Implementiet den *IModelObserver* und fügt alle Sinngruppen der ersten Ebene und deren Inhalte und alle Elemente, die in keiner Sinngruppe vorkommen, in die Wurzel des Baumes ein. Die Wurzel des Baumes ist der *default SenseUnit*.

8.3.2.2 *TreePopupMenu.java*

Klasse für das PopupMenu der Baumstruktur. Sie definiert verschiedene MenuItems für den Anzeigemodus der Matrix, das Editieren oder Löschen von Elementen.

- show(Component invoker, int x, int y, int selectionCount): Überlädt die show() Methode aus dem *JPopupMenu*. Ist das Modell nicht editierbar, werden die Menüs zum Editieren und Löschen deaktiviert. Mit der zusätzlichen Parameter *selectionCount* wird bestimmt, welches Menü beim editierbaren Modell aktiv sein soll.

8.3.2.3 *TreeStructure.java*

Erzeugt ein neues TreeStructureFenster mit dem PerspectivePanel als Parameter, der Panel liefert bei Bedarf eine Instanz der Matrix und des Editors zurück.

- editSelectedElement(): Wird vom PopupMenu aufgerufen, um das zu editierende Element zu setzen, welches zuerst ermitelt werden muss. Ist kein Element selektiert oder das selektierte Element ist kein VElement, passiert nichts.
- selectEditableElement(VElement element): Übergibt das selektierte Element zur Bearbeitung an den Editor.
- ubdateMatrixView(int updateType): Ermöglicht eine retriktive Sicht der Matrix. Zuerst werden die selektierten Baumknoten in eine ArrayList eingefügt und diese zum update der Matrix weitergeleitet. UpdateType bestimmt den Typ der Restriktion:
  a-) für eine Restriktion der Spalten
  b-) für eine Restriktion der Zeilen
  c-) für eine NxN Restriktion
- getSelections(): Berechnet aufgrund der selektierten Knoten die Menge der Elemente, die in der Adjazenzmatrix angezeigt werden sollen. Enthält der gerade selektierte Knoten ein VSenseUnit-Objekt, werden alle Elemente dieses Objekts und die dessen Unter-VSenseUnits in die oben genannte Menge aufgenommen. Da die Elemente zuerst in einem HashSet eingefügt werden, kann für die Ordnung nicht garantiert werden.
- getSenseUnitContain(VSenseUnit selectedSU, HashSet selection): Fügt die VElemente der VSenseUnit und die ihrer Unter-VSenseUnits in die Liste der selektierten Elemente ein. HashSet stellt sicher, das jedes VElement genau einmal in die Liste aufgenommen wird.
- deleteSelection(): Löscht die selektierten Elemente. Gehört eins der zu löschenden Elemente zu mehreren VSenseUnits, wird es auch aus diesen entfernt.

8.3.3 *Adjazenz-Matrix*

8.3.3.1 *AdjacencyMatrix.java*

Hauptklasse der Adjazenz-Matrix, stellt die Matrix in Form einer Tabelle dar.

- setDisplayedRows(ArrayList selectedElements): Wenn im Sinngruppenbaum eine Menge von Elementen selktiert wird und diese in der Adjazenz-Matrix in den Zeilen dargestellt werden soll, so wird von dort aus diese Methode aufgerufen.
- setDisplayedColumns(ArrayList selectedElements): Wenn im Sinngruppenbaum eine Menge von Elementen selktiert wird und diese in der Adjazenz-Matrix in den Spalten dargestellt werden soll, so wird von dort aus diese Methode aufgerufen.

- setDisplayRowsAndColumns(ArrayList selected Elements): Wenn im Sinngruppenbaum eine Menge von Elementen selktiert wird und diese in der Adjazenz-Matrix in den Zeilen und Spalten dargestellt werden soll, so wird von dort aus diese Methode aufgerufen.

8.3.3.2 *AdjacencyMatrixTableCellEditor.java*

Dient dazu, um die einzelnen Zellen der Adjazenz-Matrix zu bearbeiten. Das heißt, das ein Popup Menü aufgerufen wird mit der man die Relationen der in der Adjazenz-Matrix dargestellten Klassen bearbeiten oder besser geasagt bestimmen kann.

8.3.3.3 *AdjacencyMatrixTableCellRenderer.java*

Wird zum zeichnen aller Spalten und Zeilen der Adjazenz-Matrix benutzt. Bekommt ein VRelation-Objekt übergeben, das dann gezeichnet wird.

8.3.3.4 *AdjacencyMatrixTableModel.java*

Wrapper-Klasse um ein VModel, um dieses in der Adjazenz-Matrix darzustellen. Überwacht alle Aktionen die auf die Adjazenz-Matrix ausgeübt werden, wie das Eifügen oder Löschen einer Relation.

8.3.3.5 *RelationEditor.java*

Das zuvor erwähnte Popup Menü für die Bearbeitung der Relationen.

8.3.4 *Editor*

8.3.4.1 *ElementEditor.java*

Mit dem Element Editor kann man neue Klassen erstellen. Er ist aufgebaut in einer Karteireiterform. In der ersten Sicht kann man die Klasse erstellen, in der zweiten Sicht die dazugehörigen Attribute und in der dritten Sicht die Methoden der Klasse.

8.4 *edu.unido.pg436.perspectives.versioncontrol*

Auf der folgenden Abbildung 8.8 ist die Oberfläche der Versionskonrolle zu sehen. Sie ist aufgeteilt in zwei Fenster

- Fenster zur Darstellung des Modells *(oben)*
- Fenster zur Darstellung der Versionspunkte *(unten)*.

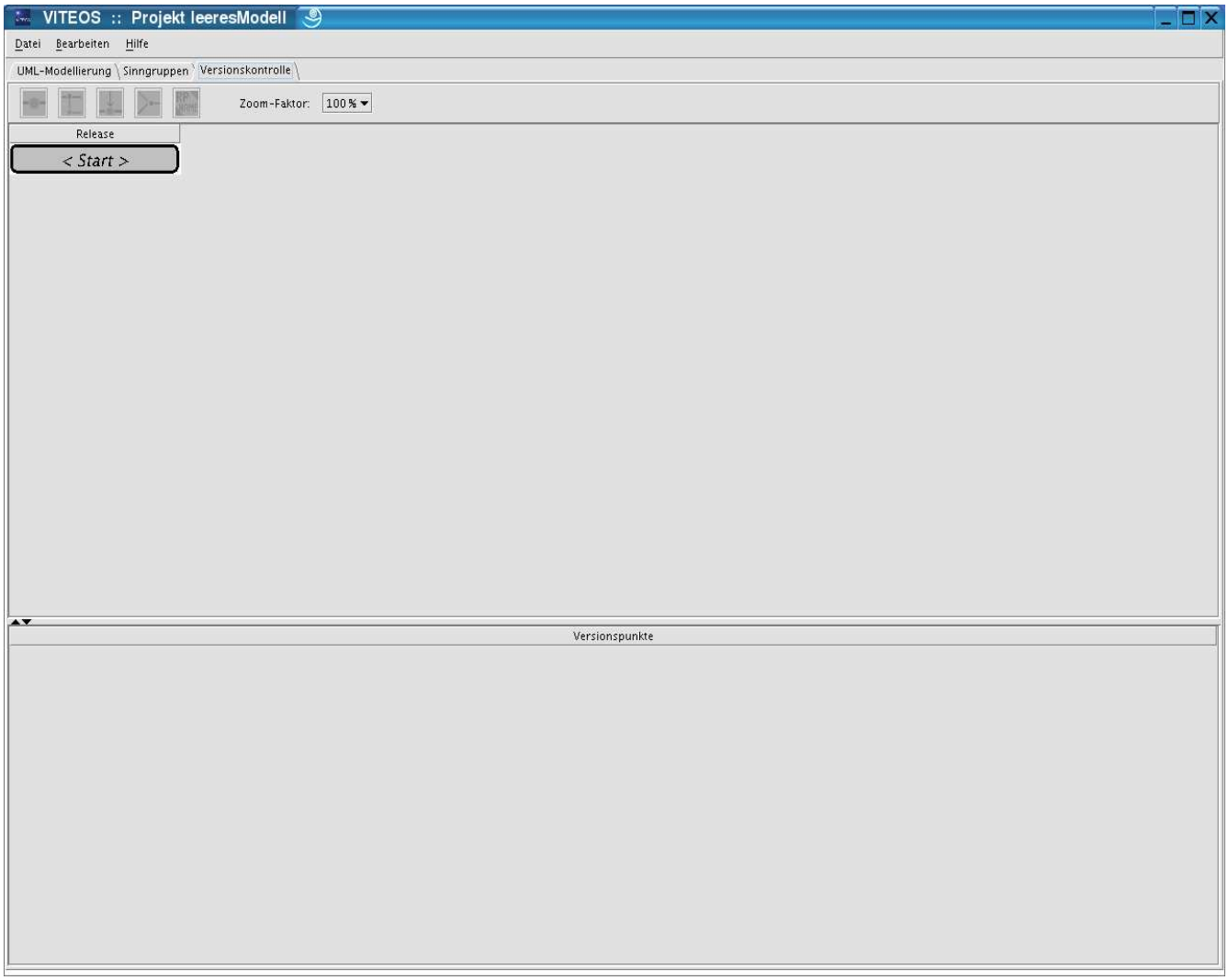

Abbildung 8.8: Versionskontrolle : Leeres Modell

Die Versionskontrolle wird durch die folgenden Klassen realisiert :

8.4.1 *VersionControlPerspectivePanel.java*

Dies ist das Hauptfenster des Packages. Die Interfaces *ListSelectionListener* und *TableColumnModelListener* werden implementiert, um sich bei der Versionskontrolle als Listener anmelden zu können. Bei Änderungen an der Selektion von Zeile oder Spalte wird dann über die Klasse *VersionControllTreePanel* die die *VersionControlToolBar* benachrichtigt, je nach Selektion in der Tabelle ihre Buttons auf *enabled* oder *disabled* gesetzt. Klickt ma beispielsweise in der Versionskontrolle

auf ein Strang, so werden alle Buttons auf disabled gesetzt (d.h. sie werden deaktiviert), weil man an einem Strang weder branchen noch mergen kann. Klickt man einen echten Release an, an dem schon mal verzweigt worden ist, so werden die Buttons EchteReleaseErzeugen und Verzweigen ausgeleuchtet, an dieser Stelle kann man nur noch den Release laden, mergen oder umbenennen.

- setDisplayedVersionPoints(ArrayList versionPoints): Wird von der Baumstruktur aufgerufen, um in der Versionspunkt-Liste die selektierten Versionspunkte anzuzeigen.
- refreshVersionTree(): Leitet die Baumstruktur an die *(VersionControlTreePanel)* weiter, damit diese sich updated.
- refreshVersionPointList(): Leitet die Versionspunkt-Liste an die *(VersionPaintPanel)* weiter, damit diese sich updated.
- createNonGenericRelease(String newReleaseName): Wird vom Versionierungsbaum aufgerufen, um einen generischen Release zu einem echten (benannten) Release zu machen.
- mergeReleaseFromToolbar(): Wird vom Versionierungsbaum aufgerufen, um einen Release mit dem aktuellen Modell zu mergen.
- renameRelease(VersionRelease rel): Wird vom Versionierungsbaum aufgerufen, um einen Release umzubennenen.
- renameSelectedRelease(): Wird aus der Toolbar der Versionierungsperspektive aufgerufen, um den im Baum selektierten release umzunbennenen.
- merge(Versionpoint vp): Allgemeine Methode zum Mergen eines Versionpunktes mit dem aktuellen Modell. Parameter vp ist der zu mergende Versionspunkt. Desweiteren lädt diese Methode das temporäre Modell und extrahiert daraus die Klassen, die er dem Benutzer präsentiert. Lässt den Benutzer eine Liste von VClass selektieren und gibt diese iterativ und einzeln der ModelMergeControl weiter. Für den Fall das es Konflikte gibt wird ein Dialog eingeblendet, der Benutzer die Möglichkeit bietet den Konflikt manuell zu lösen. Alle weiteren Operationen finden auf dem VModel-Objekt statt, das die GUI-Klassen bereits kennen. Sie werden über *IModelObserver*-Methoden benachrichtigt. Nur der Versionierungsbaum muss noch explizit upgedatet werden.
- branch(): Wird vom Versionierungsbaum aufgerufen, um zu branchen. Holt sich den aktuell selektierten Release vom Versionierungsbaum. Die Überprüfung des dieses Releases findet hier nicht mehr statt, da die Buttons über diese Methode aufgerufen werden sind entsprechend die selektierten Releases aktiv oder inaktiv.
- loadVersionPointFromTree(): Wird vom Versionierungsbaum aufgerufen, um einen selektierten Release zu laden.
- loadVersionPointFromList(VersionPoint vp): Wird von der Versionspunktliste aufgerufen, um einen selktierten Versionspunkt oder Release zu laden.

8.4.2 *Fenster zur Darstellung des Modells*

8.4.2.1 *VersionControlTreePanel.java*

Klasse zur Darstellung des Versionierungsbaums.

- refresh() : Wird aufgerufen, um den Versionierungsbaum neu aufzubauen. Die Methode holt sich von der Versionskontrolle die Datenstruktur und baut das *TableModel* komplett neu auf.
- showReleasePopupMenu(VersionRelease rel, int x, int y): Zeigt das PopupMenu über einem Release.
- updateToolbarButtonsStatus(): Wird aufgerufen, um den Status des Buttons in Abhängigkeit von der in der Baum-Tabelle selektierten Zeile neu zu setzen. Leitet weiter an *updateButtonsStatus(elem)* der Toolbar.
- zoom(String factor): Zoomt die Tabelle hinein oder hinaus.

8.4.2.2 *VersionControlTableModel.java*

Klasse die konstruiert ist, um den *TableModel* darzustellen.

- constructTableRow(ArrayList curBranch, int curRowIndex,int curColumnIndex): Konstruiert rekursiv die Zeilen der Tabelle. Setzt voraus, das bereits alle Zeilen *(tableRows)* mit leeren Elementen VC_TYPE_NULL vorinitialisiert sind.
  - a-) : @param curBranch ArrayList mit den Releases für die aktuelle Zeile
  - b-) : @param curRowIndex Index der aktuellen Zeile (in der Tabelle)
  - c-) : @param curColumnIndex Index der aktuellen Spalte (in der Tabelle)
- initializeTable(): Berechnet die Anzahl der Zeilen und Spalten (die Attribute numCols und numRows werden mit Werten gefüllt) und initialisiert die ArrayList *tableRows* entsprechend mit *numRows* ArrayLists der Länge *numCols*.
- calculateNumberOfRowsAndColumns(ArrayList curBranch, int curRowIndex,int curColumnIndex): Rekursive Methode, die die Anzahl der Zeilen und Spalten berechnet. Diese werden dann noch mit 2 multipliziert.
- *class* VersionControlTableCellElement : Klasse der die Tbellenzellen der einzelnen Releases repräsentiert. Dieser muss nicht existieren, da die Zelle auch leer sein kann, einen Strang oder Branch repräsentieren kann. Daher dafault = null.

8.4.2.3 *VersionControlTableCellRenderer.java*

Wird zum Zeichnen aller Spalten der Adjazenz-Matrix außer der ganz Linken benutzt. Bekommt ein *VersionControlTableModel.VersionControlTableCellElement*-Objekt übergeben, das dann gezeichnet wird.

8.4.2.4 *VersionControlToolBar.java*

Steuert die Funtionen des Toolbars. Branchen, mergen, ReleasePoint erstellen ...

8.4.2.5 *ReleaseNameDialog.java*

Dialogfenster der die Aktionen auf den Releases kontrolliert oder besser gesagt steuert. Beispielsweise kann man über diesen Dialogfenster einen neuen Release erstellen, oder einen bestehenden Release umbennenen.

8.4.3 *Fenster zur Darstellung der Versionspunkte*

8.4.3.1 *VersionPointPanel.java*

Hauptklasse der Versionspunktfensters.

- jumpToVersionPoint(): Der selektierte Versionspunkt wird zunächst in einen neuen Release umgewandelt und anschließend als aktuelles Release gesetzt.
- showMergeModelDialog(): Zeigt den Dialog für das Mergen der Versionspunkte an.
- showVersionpoints(string modelName, ArrayList vpList): Zeigt die in dem selektierten Strang enthaltenen Versionspunkte an. Der Parameter *modelName* repräsentiert den Namen des Aktuellen Modells dar und der Parameter *vpList* ist die Liste der zur zeigenden Versionspunkte.
- versionPointToRelease(): Ermöglicht es, einen Versionspunkt als Release festzulegen. Ist der selektierte Versionspunkt bereits ein Release, passiert nichts. Der Versionierungsbaum und die Liste der Versionierungspunkte müssen dann aktualisiert werden, da die vorherige Darstellung nicht mehr gültig ist. Das neue Release erzwingt eine Teilung des bis dahin selektierten Strangs. Als Rückgabewert wird entweder das neue Release oder null (falls der eingegebene Name nicht zulässig ist) geliefert.
- refresh(): Löscht die Tabelle mit den Versionspunkten.
- showPopup(mouseEvent e): Zeichnet das PopupMenu. Ist kein Knoten selektiert, wird die Selektion auf dem Knoten gestzt, der anstelle des Mausklicks liegt.

8.4.3.2 *VersionPoinTableModel.java*

Klasse die konstruiert ist, um den *TableModel* der Versionierungspunkte darzustellen.

8.4.3.3 *VersionPointtableCellRenderer.java*

Wird zum Zeichnen der einzelnen Versionspunkte benötigt.

8.4.3.4 *VersionPoinPopupMenu.java*

Klasse füe das PopupMenu der Versionpunkttabelle. Sie definiert verschiedene MenuItems für den Anzeigemodus der Matrix, das Editieren oder Löschen von Elementen.

- show(Component invoker, int x, int y, int selectionCount): Überläd die show() Methode aus JPopupMenu. Mit Hilfe des zusätzlichen Parameters selectionCount wird bestimmt, welches Menü aktiv ist.

8.5 *edu.unido.pg436.perspectives.versioncontrol.merging*

8.5.1 *ConflictResolutionTableModel.java*

Diese TableModel hilft dem Benutzer beim manuellen Auflösen von Konflikten (siehe Abbildung 8.9), die beim Mergen zweier Releases enstehen. Es stellt den aktuellen Stand des Modells dem Stand des einzufügenden Modells gegenüber und bietet dem Benutzer die Möglichkeit zu entscheiden, welcher verwendet werden soll und gibt die Selektion an die Klasse *VersionControlManualMergingDialog* weiter.

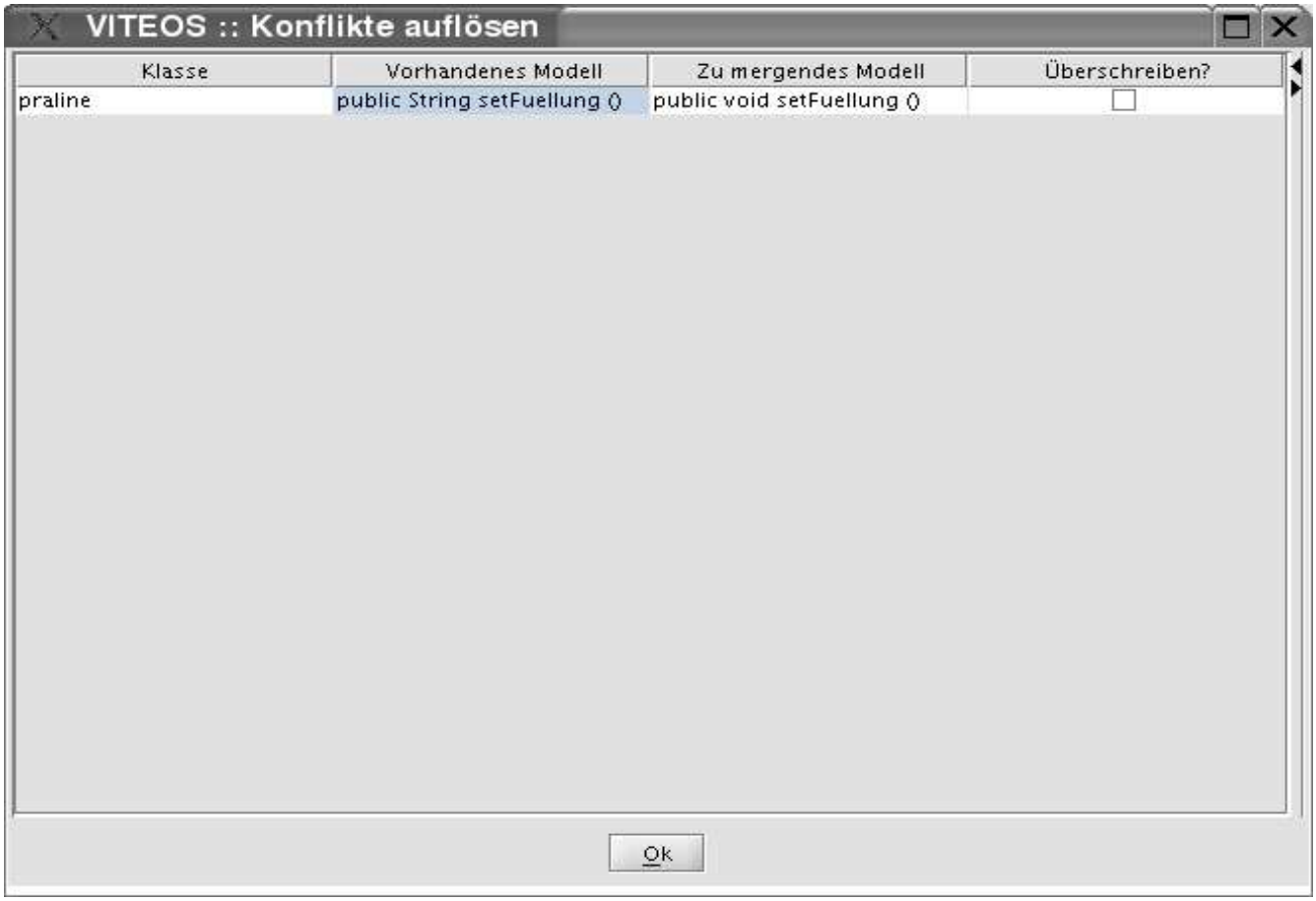

Abbildung 8.9: GUI : Merge Konflikt Dialog

8.5.2 *VersionControlManualMergingDialog.java*

Diese Klasse generiert die oben aufgeführte Abbildung 8.9 zum manuellen Lösen der beim Mergen enstandenen Konflikte.

- mergeAndClose(): Iteriert über das TableModel. Hat der Benutzer *Überschrieben* selektiert, so wird der Eintrag in der Diff-Klasse gelassen. Hat er das nicht, so wird der Eintrag aus der Diff-Klasse entfernt. Anschließend werden die Diff-Klassen iterativ der ModelMergeControl gefüttert.

8.5.3 *VersionPointMergeDialog.java*

Dialog, der zur Feststellung der zumergenden Versionspunkte, konstruiert worden ist.

- setModel(): Initialisiert den Baum mit Elementen aus dem übergebenen Modell.
- getSelections(): Durchläuft den Mergingbaum und filtert alle nicht selektierten knoten aus. Das daraus resultierende Modell kann nun mit dem aktuellen Modell gemergt werden.
- getSelections(VersionPointTreeNode node): Untersucht den Teilbaum dessen Wurzel der übergebene knoten ist und entfernt alle Elemente aus dem temporären modell, deren Knotendarstellung nicht selektiert sind. Der Parameter *node* ist die Wurzel des aktuell zu betrachtenden Teilbaums.

- getMergingMode(): Wird von VersionControlPerspectivePanel aufgerufen, und liefert ein VModel-Objekt zurück, das genau die für das Mergen selektierten VElement und VSenseUnit Objekte enthält. Als Rückgabewert liefert das VModel-Objekt, das zum aktuellen Modell gemergt werden soll, null zurück (falls nichts selektiert oder der Vorgang abgebrochen wurde).

8.5.4 *VersionPointTreeModel.java*

Klasse zur Darstellung und Berechnung der Baumstruktur des zuentwerfenden Diagramms.

- loadTreeNodes(TreeMap suList, TreeMap elementList): Fügt alle Sinngruppen der ersten Ebene und deren Inhalte und Elemente, die in keiner Sinngruppe vorkommen, in die Wurzel des Baumes ein. Der Parameter *sUnitList* ist die Liste aller Sinngruppen und *elementList* die Liste aller Elemente.
- loadTreeNodes(...): Lädt den Inhalt eines Elements in das TreeModel.
- updateSUAffiliation(VElement elem, int value): Aktualisiert die Sinngruppenstruktur.

8.5.5 *VersionPointTreeNode.java*

Klasse, die zu Bearbeitung der einzelnen Knoten der Baumstruktur benötigt wird.

- StringToString(): Methode die dazu beiträgt, dass der Name des Knotens korrekt angezeigt wird.
- setSelected(boolean isSelected): Der Zustand dieses Knotens und dessen Kinder wird auf dem übergebenen Wert gesetzt. Das heißt: Wird ein Baum de/selektiert, werden auch seine Kinder als selektiert oder nicht selektiert angezeigt.

8.5.6 *VersionPointTreeRenderer.java*

Wird zum Zeichnen der Aktionen, die im Merge-Package anfallen, benötigt.

8.6 *edu.unido.pg436.perspectives.senseunit*

8.6.1 *SenseUnitPerspectivePanel.java*

Diese Klasse ist die Hauptklasse in diesem Package. Sie stellt das Haupfenster für die Sinngruppen dar (siehe Abbildung 8.10). Dieses Fenster ist wiederum aufgeteilt in drei Unterfenster:

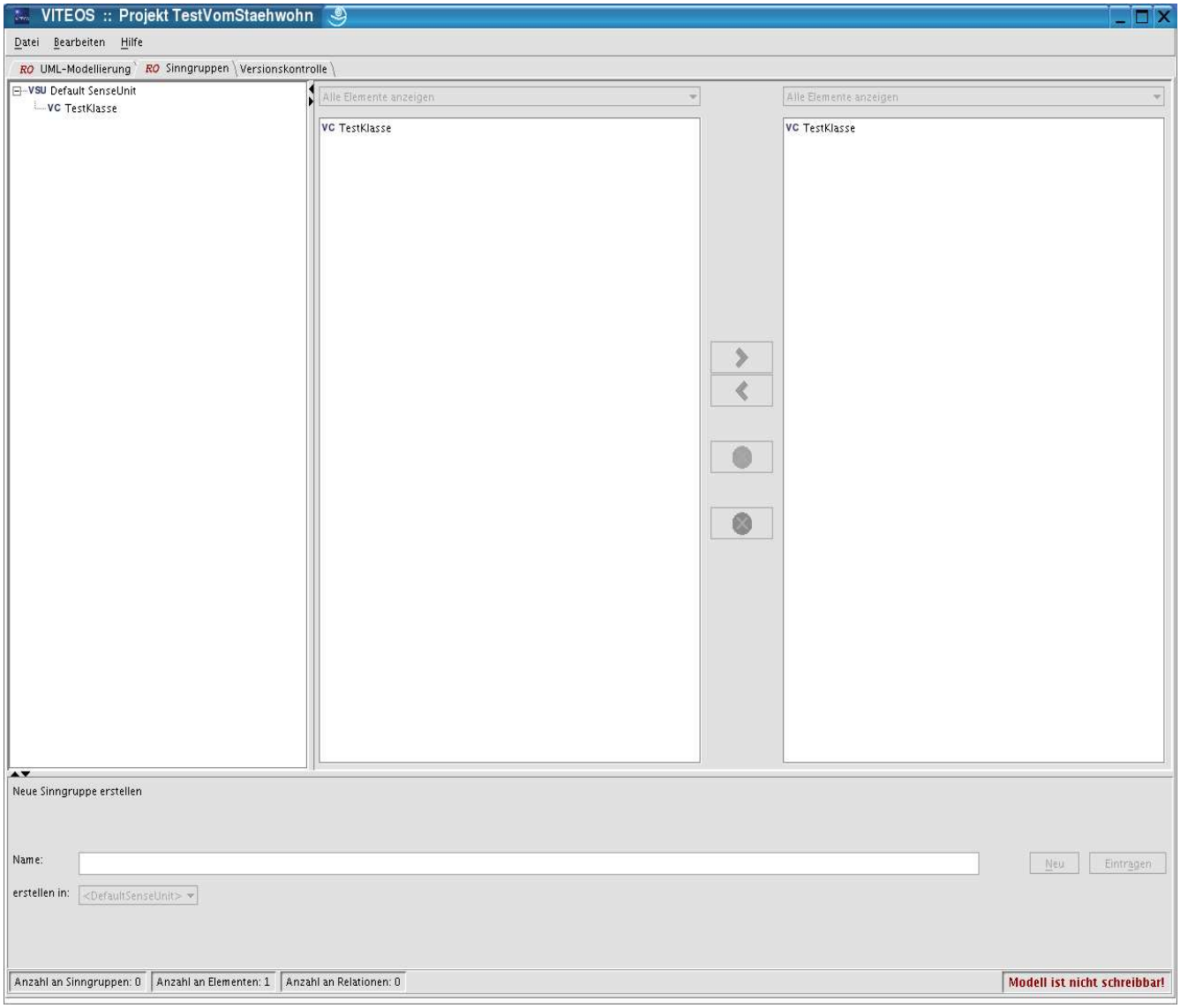

Abbildung 8.10: GUI : Sinn-Gruppe

- Baumstruktur *(links)*
- Sinngruppen Manager *(rechts)*
- Editor *(unten)*

8.6.2 *Baumstruktur*

8.6.2.1 *SenseUnitTreeStructure.java*

Erzeugt ein neues TreeStructure Fenster mit dem PerspectivPanel als Parameter. Das Parameter Panel liefert bei Bedarf eine Instanz der Matrix und des Editors zurueck. Das aktuelle Modell wird über VModel geliefert und durch die Klasse *ViteosTreeModel* als Baumstruktur auf der linken Seite des Hauptfensters dargestellt.

8.6.2.2 *SenseUnitTreePopupMenu.java*

PopupMenu über die Linke Baumstruktur, sie definiert verschiedene MenuItems für die Elemente des SinngruppenBaums.

- verschieben nach
- kopieren nach
- aus Sinngruppe entfernen
- umbenennen
- löschen

8.6.2.3 *ViteosTreeModel.java*

Implementiert den *IModelObserver* der bei *VModel* als Überwacher registriert ist. Über den *IModelObserver* holt sich die Klasse den aktuellen Modell. Mit Hilfe der Klasse *ViteosTreeNode*, wo auch der generische Knoten *default SenseUnit* erzeugt wird, werden die Knoten als Objekte angelegt.

8.6.2.4 *ViteosTreeModelListener.java*

Überwacht die Änderungen in der Baumstruktur, beispielsweise ob ein Knoten eingefügt worden ist und stellt diese in der Baumstruktur dar. Die Methode treeNodeInserted berechnet den Punkt, wo der Knoten eingefügt wird.

8.6.2.5 *ViteosTreeRenderer.java*

Stellt die vom Entwickler erstellte Baumstruktur des Modells visuell dar.

8.6.3 *Sinngruppen Manager*

8.6.3.1 *SenseUnitManager.java*

Stellt das rechte Fenster dar. Auf diesem Fenster kann man sich auf zwei Seiten Sinngruppenlisten anzeigen lassen. Durch die Buttons in der Mitte kann man Elemente aus einer Sinngruppe in die Andere verschieben oder Elemente aus den einzelnen Sinngruppen entfernen oder ganze Sinngruppen löschen.

8.6.3.2 *SenseUnitCellRenderer.java*

Diese Klasse stellt die Einträge, die in den beiden Listen auf diesem Fenster zu sehen sind, visuell dar.

8.6.4 *Editor*

8.6.4.1 *SenseUnitCreator.java*

Diese Klasse ist zuständig für den Editor der Sinngruppen Perspektive. Mit dem Editor kann man neue Sinngruppen erstellen.

Teil IV

Abschlussbetrachtung

Nach der Abschluss–Präsentation des Tools *VITEOS* am 29.07.2004, haben sich die Projektgruppen Teilnehmer noch einmal zusammengesetzt um über die Projektgruppe und das Tool zu diskutieren. Die Schwerpunkte dieser Diskussionsrunde waren in erster Linie:

- die Adjazenzmarix: Diese Perspektive bietet uns den Vorteil, das der Aufwand, den man bei der Erstellung einer z.B. Analysephase mit Hilfe eines auf Diagramme basierendes Tools (z.B. Together) betreibt, erheblich reduziert wird. Desweiteren hat man auf dieser Perspektive die Möglichkeit, die Abhängigkeiten bzgl. einzelner Klassen vorteilhaft zu ermitteln und getrennt zu betrachten. Als Nachteil hat sich erwiesen, dass die Adjazenzmatrix in ihrer Gesamtheit ziemlich unübersichtlich ist und die Kommunikation über Aspekte des Diagramms durch die Matrix sehr erschwert wird.

- Versionierungsmechanismus: Dieser Mechanismus scheint vorteilhaft zu sein, weil er 1. die Entwicklung durch die Übernahme früherer Entwicklungschritte beschleunigt, weil er 2. eine flexibele Aufgabenbearbeitung durch Verzweigung der Entwicklungsstränge ermöglicht und weil er 3. eine anschauliche Darstellung des Entwicklungsverlaufes bietet.

- die Sinngruppe: Die Vewendung des Sinngruppen–Konzepts ist eine große Hilfe bei der semantischen Einteilung der Klassen. Es fördert die Benutzbarkeit der Adjazenzmatrix und des Sinngruppenbaums. Weiterhin gleicht es in der momentanen Realisierung, den Paketdiagrammen in der UML. Jedoch lässt das Konzept Wünsche zur Verbesserung offen.

- Handhabbarkeit der Benutzerschnittstellen: Sie ist übersichtlich in die genannten drei Perspektiven eingeteilt, was das Arbeiten mit diesem Tool ziemlich angenehm macht. Jedoch scheint die Benutzung der Klassen- und Sinngruppeneditors für den Neueinsteiger ziemlich schwierig zu sein.

Im Großen und Ganzen können wir sagen, das *VITEOS* seine Stärken bei der Versionierung hat. Sie umfasst neue Konzepte zum inkrementellen Entwurf von Software. Die Weiterentwicklung des Adjazenzmatrix ist zwingend. *VITEOS* wird nur als ergänzende Sicht auf den Entwurf, die in Zusammenhang mit der Nutzung der Standart UML-Tools realisiert werden sollte, empfohlen.
Desweiteren haben wir über die vergangen 2 Smester diskutiert, was hat man falsch gemacht, was hätte man besser machen können etc. An erster Stelle Stand die Arbeitsaufteilung. Dies hätte man besser gestalten können, einige PG-Teilnehmer haben durch die nicht übersichtliche Einteilung mehr leisten müssen als andere. Das Testen und das Refactoring wurden mehr in den Hintergrund gestellt, dies hätte man in einem größeren Spektrum einordnen können eventuell sogar müssen.
Was die Planungsphase im ersten Semester angeht, war die Gruppe geteilter Meinung. Einige haben die Meinung verteten, dass das nicht sinnig gewesen ist die Planungsphase so in die Länge zu ziehen und einige der Teilnehmer fanden diesen eingeschlagenen Weg jedoch vorteilhaft für die Entwicklung des Tools. Weil jeder der einzelnen Teilnehmer einen eigenen Programmierstil hat, war es trotz der Einhaltung der Code–Konventionen, den Source des anderen zu verstehen und darauf zu arbeiten nicht gerade einfach. Die GUI-Gruppe war jedoch der Meinung, dass sie sich untereinander, was den Source angeht, gut verstanden haben und auch jeweils an den Code von den anderen programmiert haben. Den folgenden letzten Satz wollen wir an die, nach uns kommenden Projektgruppen, weitergeben. *Nur ein gesunder Mix aus XP, Chaos und konservativem Programmieren scheint in unserer Gruppe zum Erfolg geführt zu haben.*

Teil V

Anhang

Kapitel 9

Benutzerhandbuch

Um das Programm fehlerfrei benutzen zu können, wird das Benutzerhanbuch zur Verfügung gestellt.

9.1 Einleitung

Durch *VITEOS* kann man die Konzeption und Entwicklung einer inkrementellen, objektorientierten Modellierung mittels einer grafischen Oberfläche realisieren. Die Grundidee ist, daß in allen Phasen der Entwicklung ein durchgängiges Entwicklungsparadigma basierend auf der Zerlegung in Objekte zugrunde gelegt wird. Die Idee des einheitlichen Paradigmas führte zum Einsatz von UML. Gerade

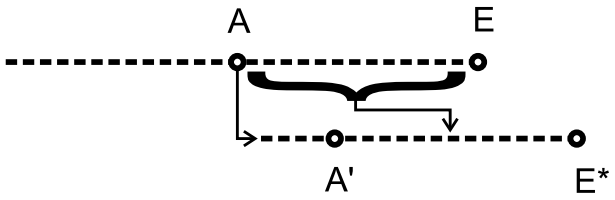

Abbildung 9.1: Versionierungsvorgehenweise

im Bereich der Werkzeugunterstützung hat es dies dazu geführt, daß für ein in der Entwicklung befindliches Softwareprodukt nur noch genau ein Modell geschaffen wird, welches im Verlauf der Entwicklung modifiziert und manipuliert wird. Besonders deutlich wird diese Situation beispielweise an dem Entwicklungswerkzeug Together, bei dem darüber hinaus eine ständige Konsistenz zwischen UML-Modell und Programmiersprachencode sichergestellt wird. Das Programmmodell entspricht immer dem graphischen UML-Modell und somit liegt ein einziges Modell in unterschiedlicher Notation vor. Der Ein-Modell-Ansatz widerspricht sowohl früheren und als neueren Erkenntnissen der Softwaretechnik und allen Erfahrungen in der praktischen Softwareentwicklung, da er verhindert, daß im Rahmen des Entwicklungsprozesses gezielt auf das Ergebnis eines früheren Entwicklungsabschnitts, also auf ein bestimmtes Abstrationsniveau, zurückgegangen und auf diesem und von diesem aus weitergearbeitet werden kann. Werden bei der Entwicklung eines Softwareproduktes Mängel oder Fehler erkannt, so erfordert ein systemtisches softwaretechnisches Vorgehen ein Wiederaufsetzen auf dem Zwischenergebnis, in dem das Problem geschaffen worden ist. Ein solches Vorgehen ist nur schwer oder gar nicht durchführbar, da das Zwischenergebnis in einem günstigen Fall nur erweitert worden ist, in ungünstigen Fällen aber durch Veränderungen völlig umgestaltet wurde.

9.2 UML-Perspektive

Die UML-Perspektive stellt den Zusammenhang zwischen Klassen dar. Dazu wird nicht auf die gewöhnliche, graphische Darstellung zurückgegriffen, sondern es wird wie im Folgenden beschrieben vorgegangen. In Abb 9.2 wird ein Anhaltspunkt für die Darstellung dieser Perspektive gegeben:

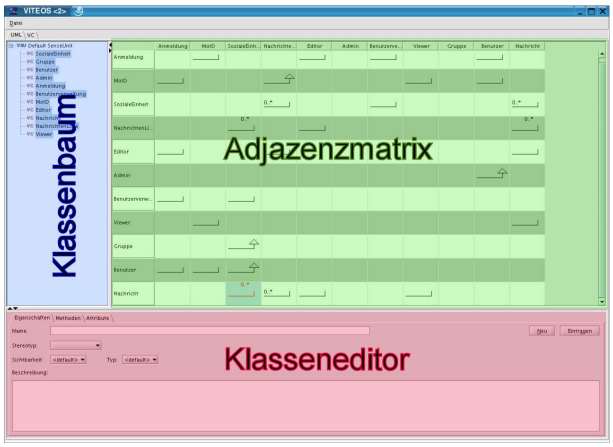

Abbildung 9.2: UML Darstellung

- Oben links befindet sich eine Darstellung des Modells als Baumstruktur, der Klassenbaum.
- Oben rechts finden Sie das Modell in einer tabellarischen Darstellung. Diese Darstellungsform bezeichnen wir mit dem Begriff Adjazenzmatrix.
- Unter befindet sich einen Klasseneditor, in dem Sie einzelne Klassen bearbeiten können.

9.2.1 Adjazenzmatrix

Die in Abbildung 9.2 gezeigt Adjazenzmatrix stellt den Zusammenhang der Klassen des Diagramms dar. Sie kann folgendermaßen beschrieben werden:

- Die Tabelleneinträge stellen die Art der Assoziation zwischen den Klassen dar oder gegeben falls ein Vererbungsbeziehung zwischen diesen dar.
- Die Leserichtung der Assoziationen ist festgelegt. Die erste Spalte der Zeile gibt diejenigen Klassen an, die die Quellen einer jeden Assoziation sind.
- Eine Adjazenzmatrix ist quadratisch (n*n).

9.2.2 Baumkomponente

Oben links in der Adjazenzmatrix befindet sich der Klassenbaum, der die Operationen Editieren, Anzeigen und Entfernen von Klassen unterstützt. Wenn Sie die Klasse mit der linken Maustaste wählen, können Sie mit der rechten eine dieser Operationen wählen.

- Aufbau: Die Wurzel, in Abbildung 9.3 des Baumes repräsentiert das gesamte Modell. Wird dieser Knoten geöffnet, erscheinen alle Gruppen auf höchster Ebene sowie alle Klassen, die in keiner Sinngruppe enthalten sind.

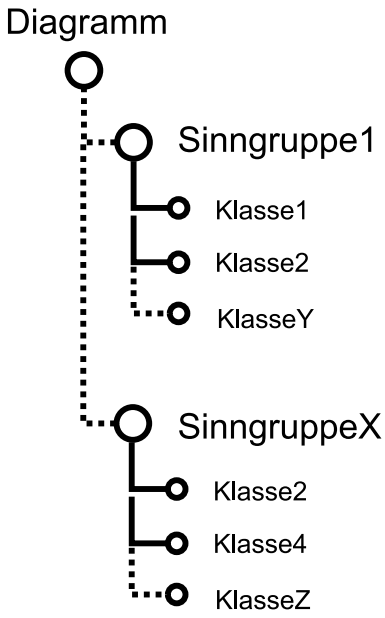

Abbildung 9.3: Baum Darstellung

- Editieren: Mit Editieren erscheint die Klasse im Klasseneditor.
- Entfernen: Mit Entfernen wird die Klasse gelöscht.
- Anzeigen: Mit Anzeigen können Sie das vorhandene Modell der Klassen als Spalten, Zeilen oder als Matrix in der Adjazenzmatrix anzeigen lassen.

9.2.3 Klasseneditor

Unter dem Klassenbaum und der Adjazenzmatrix befindet sich einen Klasseneditor, in dem Sie Methoden hinzufügen und bearbeiten können.

9.3 Sinngruppen-Perspektive

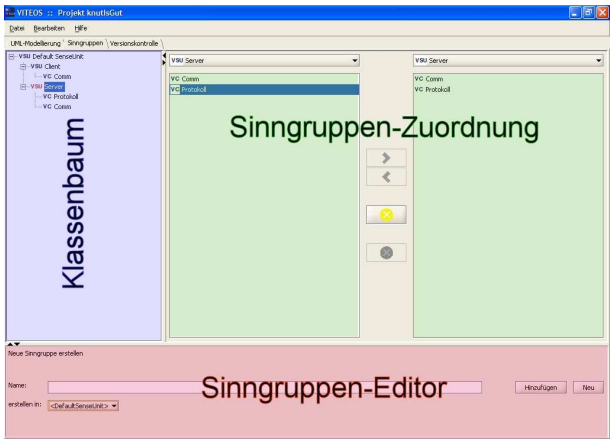

Abbildung 9.4: Sinngruppe Perspektive

Die Sinngruppen-Perspektive legt eine besondere Betonung auf die Sinngruppen-Struktur im Diagramm. Hier können Sie die Struktur anzeigen, aber auch verändern.

9.3.1 Sinngruppen erstellen

Um eine Sinngruppe zu erstellen, tragen Sie den Namen der Sinngruppe in das horizontale Feld-Name ein, und bestätigen Sie mit dem *Eintragen* Button. *Neu* löscht die Eingabe, die Sie erstellen wollen. Siehe Abbildungen 9.5 und 9.6.

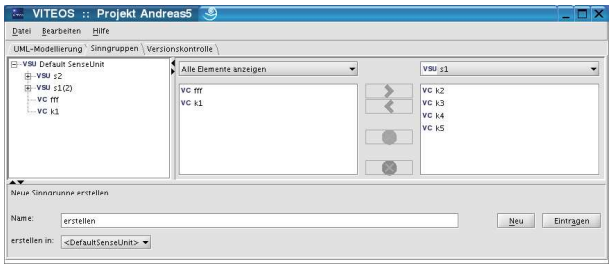

Abbildung 9.5: Sinngruppe erstellen

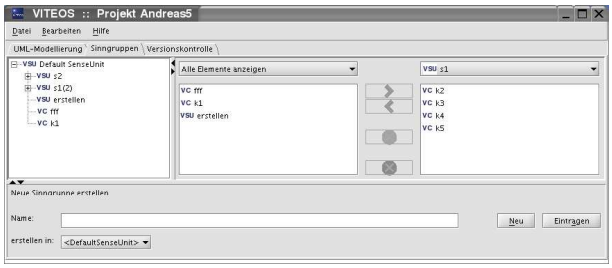

Abbildung 9.6: Sinngruppe erstellen

9.3.2 Element einer Sinngruppe verschieben

Um die Elemente der Sinngruppe zu verschieben, wählen Sie zuerst die Sinngruppe einschließlich des Elements, das sie verschieben wollen aus. Mit dem gelben Button entfernen Sie Elemente aus der Sinngruppe, andernfalls löschen Sie sie mit dem roten Button. Siehe Abbildungen 9.7 und 9.8.

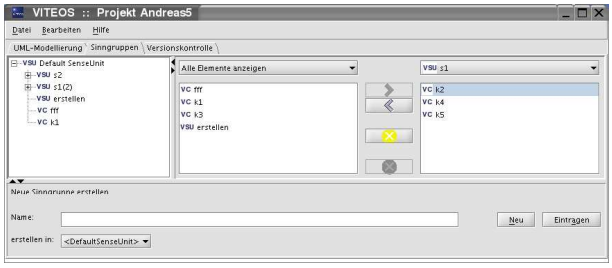

Abbildung 9.7: Element aus Sinngruppe entfernen

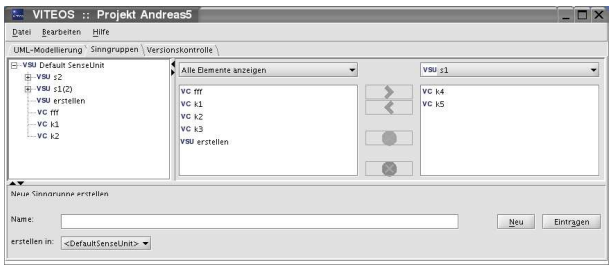

Abbildung 9.8: Element aus Sinngruppe entfernen

9.3.3 Die Elemente auf die rechte Seite verschieben

Um die Element der Sinngruppe auf die rechte Seite zu verschieben, markieren Sie mit der linken Maus-Taste das Element, das Sie verschieben wollen und bestätigen die Auswahl mit dem > Pfeil. Siehe Abbildungen 9.9 und 9.10.

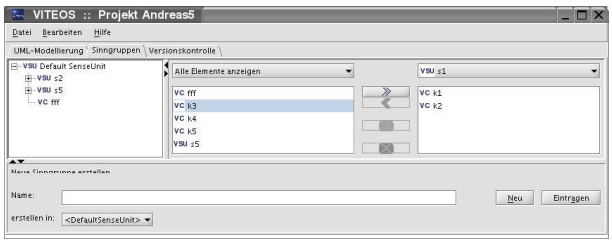

Abbildung 9.9: Verschiebung nach Rechts

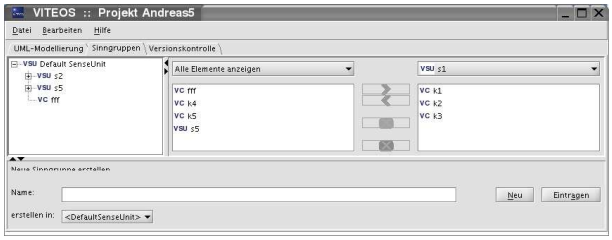

Abbildung 9.10: Verschiebung nach Rechts

9.3.4 Sinngruppe auf die linke Seite verschieben

Um die Elemente der Sinngruppe auf die linke Seite zu verschieben, markieren Sie mit der linken Maus-Taste das Element, das Sie verschieben wollen und bestätigen Sie mit dem < Pfeil. Hier gibt es auch die Möglichkeit mit einem Klick auf den gelben Button, das Element aus der Sinngruppe entfernen. Siehe Abbildungen 9.11 und 9.12.

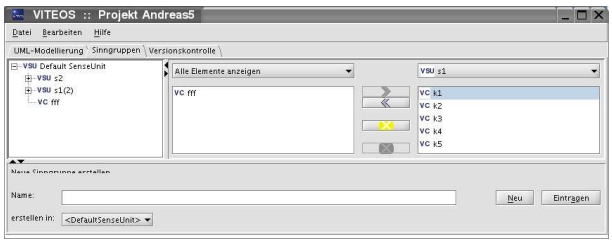

Abbildung 9.11: Verschiebung nach Links

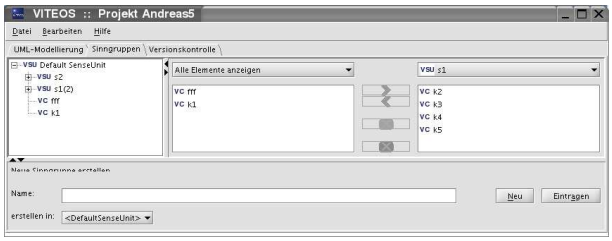

Abbildung 9.12: Verschiebung nach Links

9.3.5 Sinngruppe Löschen

Mit der roten Button löschen Sie automatisch ein markiertes Element der Sinngruppe. Siehe Abbildung 9.13 und 9.14.

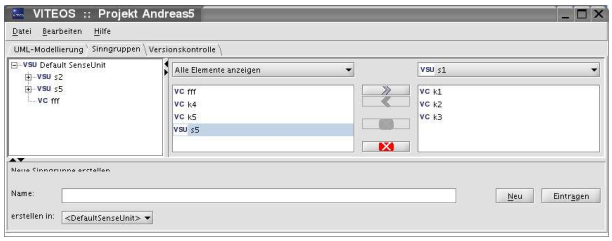

Abbildung 9.13: Sinngruppe

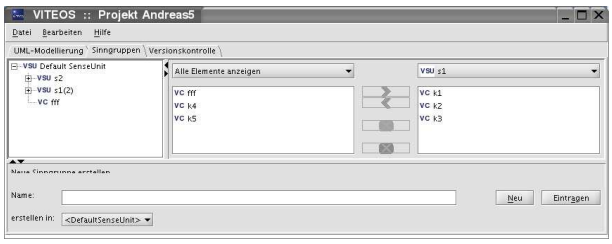

Abbildung 9.14: Sinngruppe löschen

9.4 Versionierungsperspektive

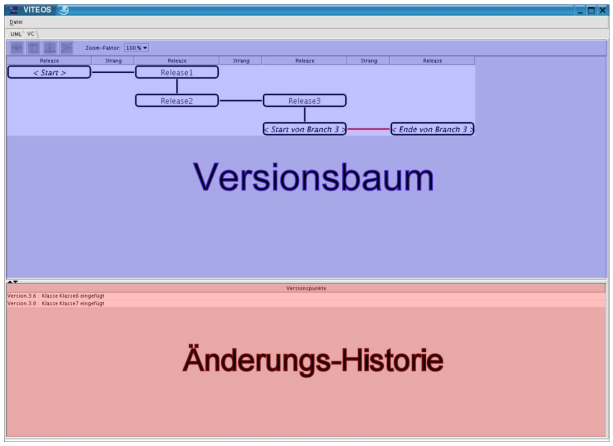

Abbildung 9.15: Versionierung Darstellung

9.4.1 Aufbau und Funktionalität

Diese Perspektive stellt Funktionen bereit, die die Versionierung betreffen. Hier kann der Benutzer zu älteren Versionen des gesamten Modells oder zu einem anderen Zweig springen oder Zweige miteinander verschmelzen (Merge). Abbildung 9.15 zeigt eien Versionierungsbaum, Knoten dieses Baumes heißen Releases. In diesem Baum können Sie entweder direkt ein Release auswählen, zu dem sie zurückspringen wollen oder Sie wählen einenBereich zwischen zwei Releases (also eine Kante), mit der Sie zu einem bestimmten Zwischenschritt springen können. Sie können weiterhin einen bestimmten Zwischenschritt eines Bereiches wählen, zu dem Sie springen wollen. Dieses geschieht mit Hilfe der Liste im unteren Bildschirmabschnitt, in der jeder einzelne Schritt aufgeführt ist. Alle Änderungen, die nach dem ausgewählten Schritt stattgefunden haben, werden rückgängig gemacht. Siehe Abbildung 9.16

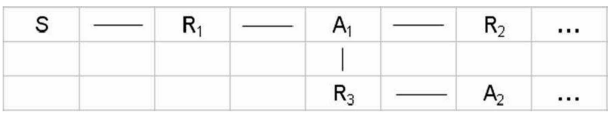

Abbildung 9.16: Versionierungsbaum Darstellung

9.5 Zoom-Faktor

Sie können verschiedene Vergrösserungsstufen wählen, um die Versionierungsansicht anzusehen. Siehe Abbildung 9.17

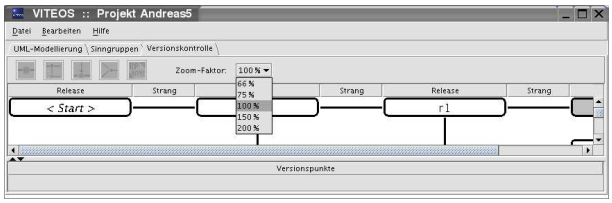

Abbildung 9.17: Versionierungsansicht

9.6 Versionspunkte als Release, Version Laden und Release Mergen

Mit einem Doppelklick auf den Versionierungstrang können Sie eine Liste von Versionspunkten auf dem Bildschirm ansehen. Hier gibt es die Möglichkeit eine Versionspunkt zu markieren und mit der rechten Maustaste den Versionspunkt als Relaese festzulegen. Weiterhin kann die aktuelle Version zur selektierten Version werden oder die selektierte Version mit der aktuellen Version verschmelzen werden. Siehe Abbildung 9.18

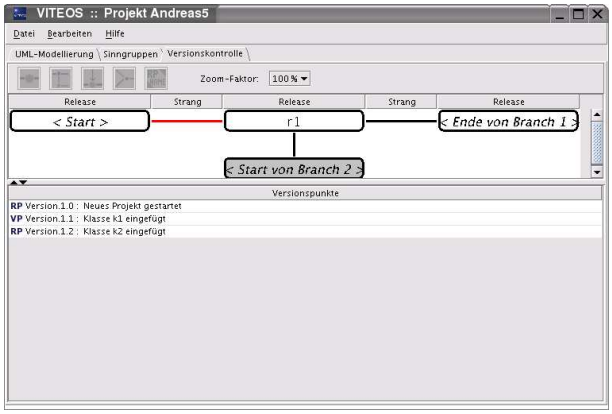

Abbildung 9.18: Liste von Versionspunkte

Hier können Sie die Versionspunkte markieren, die Versionspunkte als Release festlegen, als Version laden und als Release mergen. Siehe Abbildung 9.19

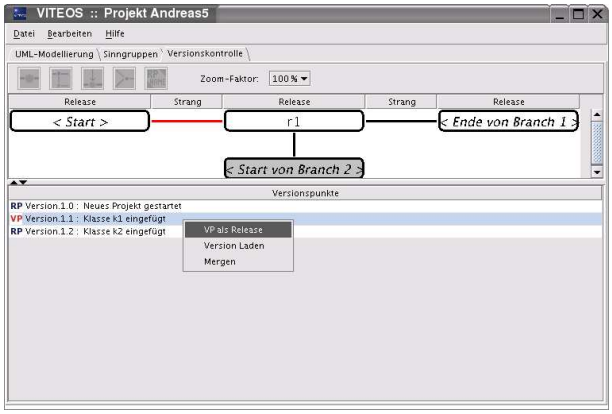

Abbildung 9.19: Realease, Laden und Mergen

Beispiel: nn ist ein festgelegtes Release. Siehe Abbildungen 9.20 und 9.21

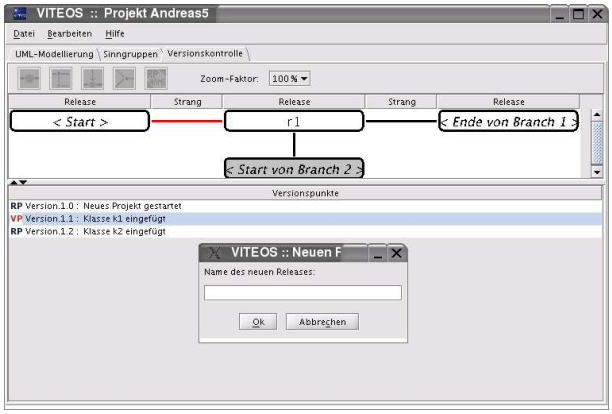

Abbildung 9.20: Version Punkte als Release festzulegen

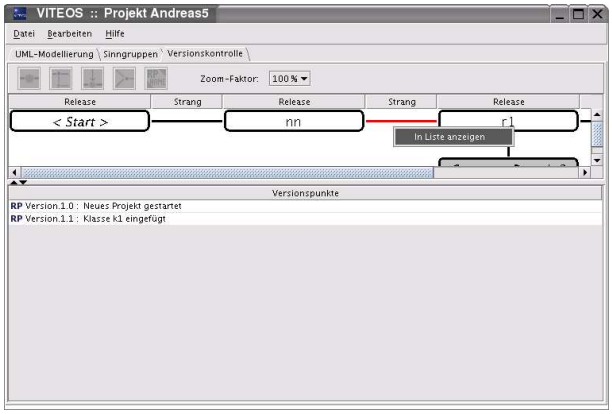

Abbildung 9.21: Version Punkte als Release festzulegen

Sie haben die Möglichkeit Version zu laden. Desweiteren können Sie hier, der Version einen neuen Namen geben. Siehe Abbildung 9.22

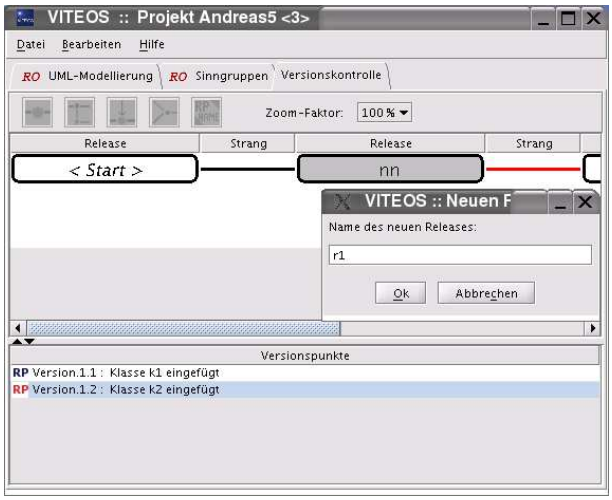

Abbildung 9.22: Version Laden

Hier können Sie Mergen, Sie haben die Möglichkeit einen ausgewählten Element in ein aktuelles Modell einzufügen. Siehe Abbildung 9.23.

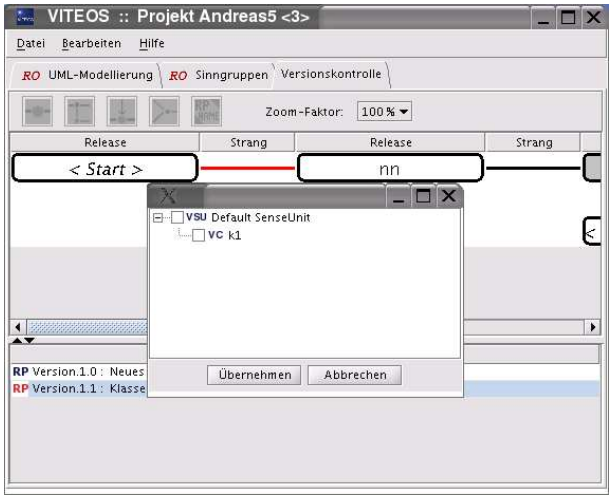

Abbildung 9.23: Mergen

Sie haben auch die Möglichkeit eine Liste anzuzeigen, indem Sie einmal mit der linken Maustaste dem Versionierungstrang aktivieren und einmal mit der rechten Maustaste auf *in Liste Anzeigen* klicken. Es erscheint sofort eine Liste von Versionspunkten unter dem Bilschirm. Siehe Abbildung 9.24

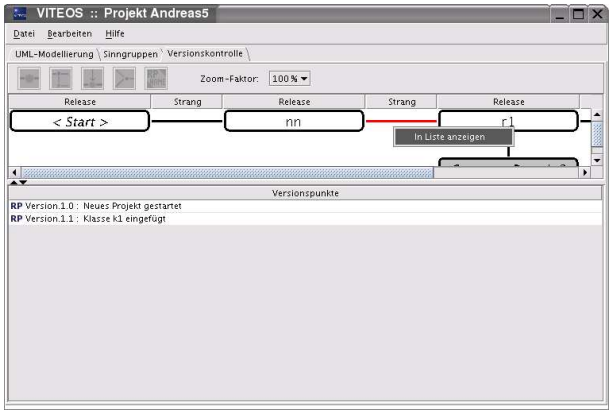

Abbildung 9.24: Liste Anzeigen

9.7 Release erzeugen

Um ein Release zu erzeugen, können Sie am Enden eines Zweiges mit der linken Maustaste auf das Release klicken und mit der rechten ein Release erzeugen anwählen. Siehe Abbildungen 9.25, 9.26 und 9.27

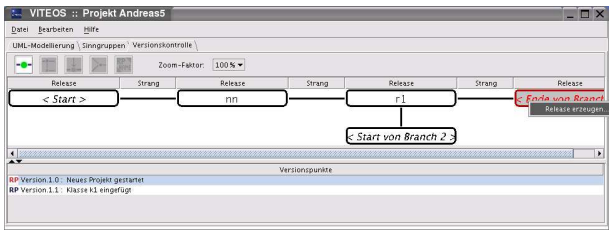

Abbildung 9.25: Release erzeugen

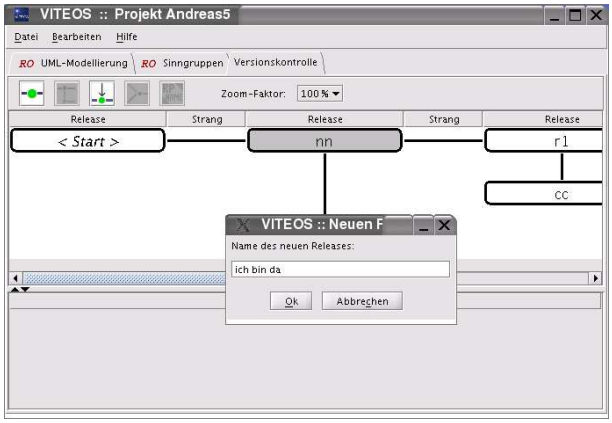

Abbildung 9.26: Release erzeugen

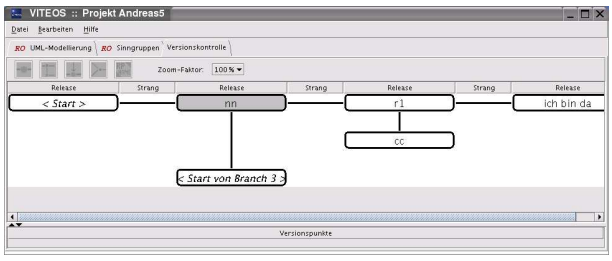

Abbildung 9.27: Release erzeugen

Sie können am Start von Branch eine Release laden, erzeugen oder mergen. Siehe Abbildung 9.28

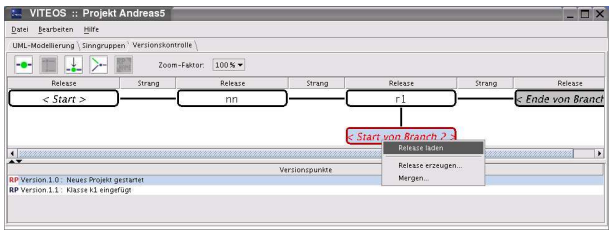

Abbildung 9.28: Release laden, erzeugen oder mergen

9.8 Neuen Zweig eröffnen

Am Enden eines Releases können Sie einen neuen Branch eröffnen, inden Sie mit der linken Maustaste auf Release aktivieren und mit der rechten Maustaste eine der entsprechenden Funktionen wählen. Siehe Abbildungen 9.29, 9.30, 9.31 und 9.32

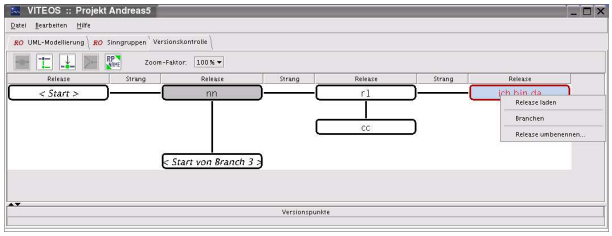

Abbildung 9.29: Release Branchen

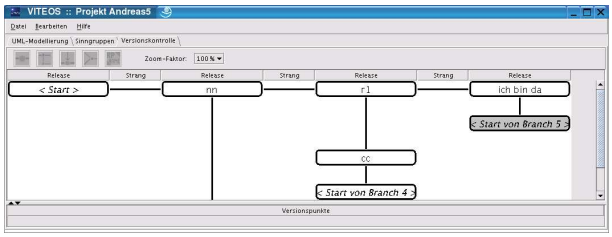

Abbildung 9.30: Release Branchen

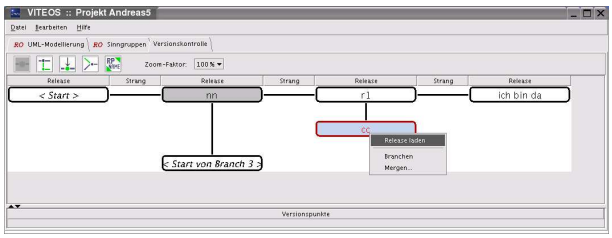

Abbildung 9.31: Release Branchen

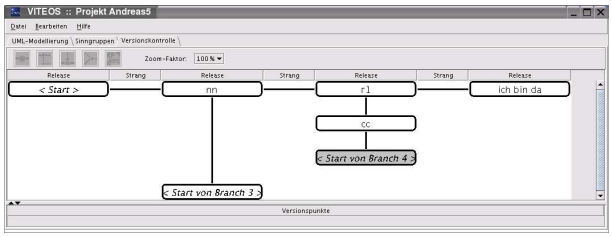

Abbildung 9.32: Release Branchen

9.9 Programm laden

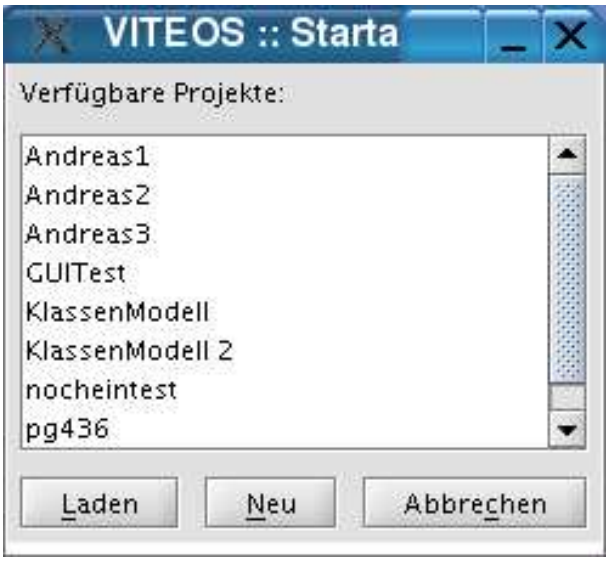

Abbildung 9.33: DB auswählen

Sobald sie das Programm starten, wird standardmäßig ein Startdialog aufgerufen, der Ihnen verschiedene Möglichkeiten bietet, in die Arbeit mit *ViTEoS* einzusteigen. Sie können das Programm *Abbrechen*, ein vorhandenes Modell *Laden* und über den Button *Neu* ein neues Modell erzeugen. Siehe Abbildung 9.33.

9.10 Programm beenden

Um das Programm zu beenden, klicken Sie auf Datei und anschliebend auf Beenden. Das Programm können Sie auch mit der Tasten-Kombination Strg-X beenden. Siehe Abbildung 9.34.

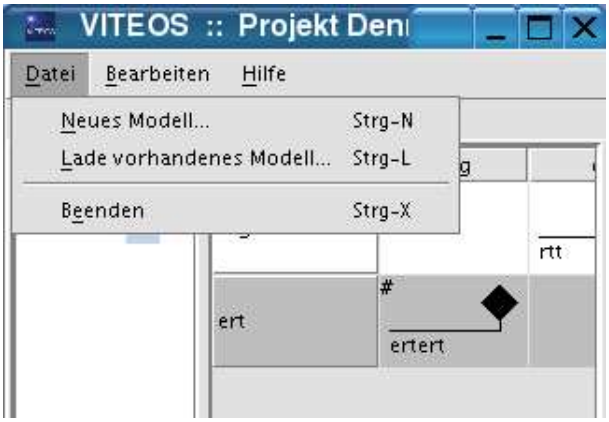

Abbildung 9.34: Progamm beenden

9.11 Eigenschaften hinzufügen

Um Eigenschaften hinzuzufügen, geben Sie zunächst den entsprechenden Namen an und wählen das Stereotyp, die Sichtbarkeit und den Typ im Feld des Klasseneditors. Die Beschreibung können Sie dem Feld einen Nanen geben. Mit einem Klick auf *Eintragen* wird die Klasse im Modell gespeichert und im Klassenbaum angezeigt.

9.12 Klasse löschen

Markieren Sie in der UML-Perspektive die Klasse, die Sie löschen möchten. Klicken Sie mit der rechten Maustaste darauf und wählen im Kontext-Menü Entfernen aus. Siehe Abbildung 9.35 Es erscheint die Bestätigungsmaske mit der Frage, ob Sie wirklich löschen wollen. Sie haben die Möglichkeit mit "Ja" das Löschen zu bestätigen, danach verschwindet das Element aus der Liste und aus der Matrix. Andernfalls können Sie mit Nein den Löschvorgang abbrechen. Siehe Abbildung 9.36

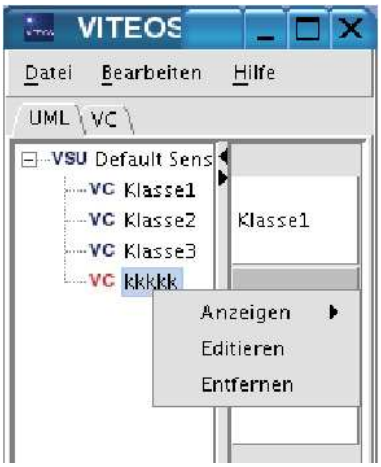

Abbildung 9.35: Klasse löschen

Abbildung 9.36: Klasse löschen

9.13 Methoden hinzufügen

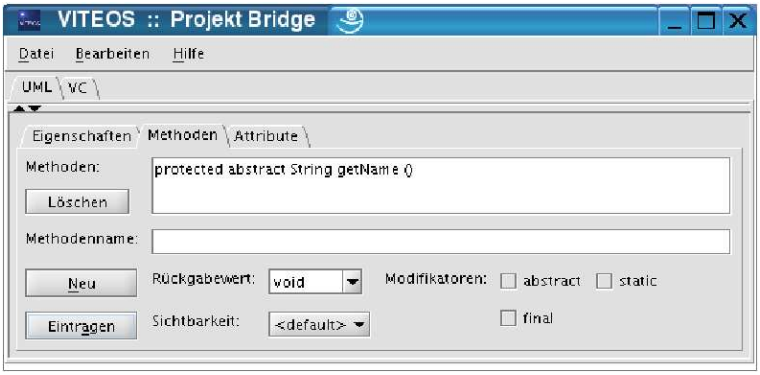

Abbildung 9.37: Methode hinzufügen

Um Methoden hinzuzufügen, tragen Sie zunächst den Methodennamen ein und wählen einen passenden Typ, eine Sichtbarkeit und die entsprechenden Modifikatoren. Mit einem Klick auf *Eintragen* wird die Methode in der Methodenliste oberhalb des Eingabefeldes gespeichert. Es gibt die Möglichkeit zusätzlich Methoden zu entfernen, indem Sie die entsprechende Methode markieren und mit dem "Button" *Löschen* entfernen. Siehe Abbildung 9.37

9.14 Attribute hinzufügen

Um die Attribute hinzuzufügen, tragen Sie zunächst den Attributnamen ein und wählen einen passenden Rückgabewert, eine Sichtbarkeit und die entsprechenden Modifikatoren. Mit einem Klick auf *Eintragen* werden die Attribute in der Attributliste oberhalb des Eingabefeldes gespeichert. Es gibt noch die Möglichkeit einzelne Attribute zu entfernen, indem Sie diese markieren und mit dem "Button" *Löschen* entfernen. Siehe Abbildung 9.38

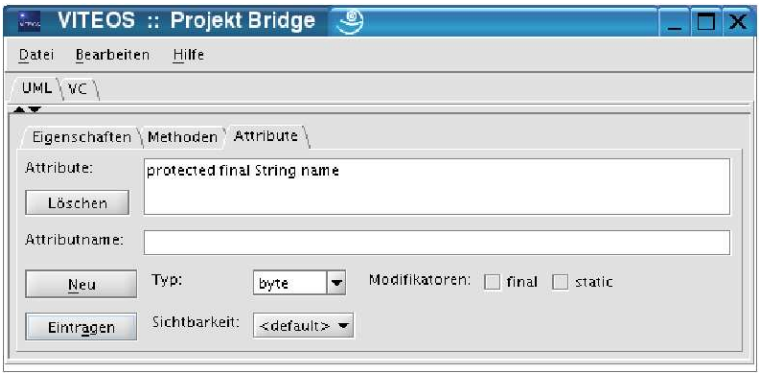

Abbildung 9.38: Attribut hinzufügen

9.15 Neue Relation hinzufügen

Um eine neue Relation zwischen die Klassen hinzuzufügen, klicken Sie einmal mit der *linken Maustaste* auf dem entsprechenden Feld in der Adjazenzmatrix und drücken auf die "E-Taste". Es erscheint ein Fenster, in dem Sie die Bezeichnung, Start- und End- Multiplizität eintragen, sowie Sie den Relationstyp wählen. Mit dem *OK-Button* können Sie die neue Relation speichern und mit dem *Abbrechen Buttom* das Einfügen abbrechen. Die Liste der Menübefehle richtet sich nach der Reihenfolge der Menüs in der Menüleiste. Zur Orientierung finden Sie zu jedem Menü jeweils eine entsprechende Abbildung. Ein Menübefehl kann über ein Symbol im Programmfenster von *ViTEoS* aufgerufen werden.

9.16 Neues Modell zu erstellen

Um ein neues Modell zu erstellen, klicken Sie auf Datei und wählen Sie *N*eues Modell aus. Alternativ können Sie das mit den Tasten-Kombination Strg-N erledigen. Siehe Abbildung 9.39. Es erscheint eine Maske, in der Sie den Namen des Projekts eingeben können. Geben Sie einen Namen ein und klicken Sie anschließen auf OK. Das neue Modell wird dann erstellt. Siehe Abbildung 9.40 Klicken Sie auf "Abbrechen", falls Sie kein neues Modell laden wollen. Das schon geladene Modell wird weiterhin angezeigt.

Abbildung 9.39: Neues Modell laden

Abbildung 9.40: Neues Modell laden

9.17 Vorhandenes Modell laden

Klicken Sie auf *Datei* und wählen Sie *Lade vorhandenes Modell* im Kontext-Menü aus. Siehe Abbildung 9.41 . Es erscheint eine neue Maske mit den verfügbaren Projekten. Alternativ können Sie

Abbildung 9.41: Modell laden

die Maske mit der *Strg-L* Tasten-Kombination aufrufen. Wählen Sie in der Maske das zu ladene Modell aus. Nach einem Klick auf *Laden* wird das ausgewählte Modell geladen. Sie können das zu ladene Modell ebenso mit einem Doppelklick laden. Wollen Sie kein vorhandenes Modell mehr laden, so klicken Sie auf *Abbrechen*. Die -Modell-Maske wird geschlossen und Sie können wieder an dem bereits geladenen Modell arbeiten. Siehe Abbildung 9.42

Abbildung 9.42: Modell auswählen

9.18 Klasse oder Interface editieren

Markieren Sie die Klasse, die Sie editieren wollen und klicken Sie darauf mit der rechten Maustaste. Wählen Sie *Editieren* in dem erscheinenden Kontext-Menü aus. Die Eigenschaften der Klasse werden im Editor zur Änderung angezeigt. Sie können anschließend im Editor Ihre Änderungen vornehmen. Wenn Sie fertig sind, klicken Sie auf *Eintragen*. Die Änderungen werden übernommen. Siehe Abbildungen 9.43 und 9.44.

Abbildung 9.43: Klasse löschen

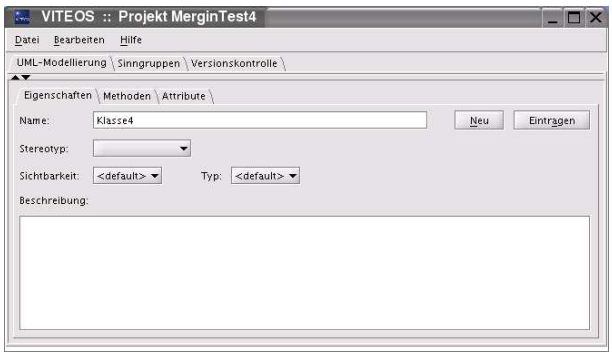

Abbildung 9.44: Löschen im Editor

9.19 Methode bearbeiten

Mit einem Doppelklick aktivieren Sie zunächst in dem Klassenbaum, die Klasse, die die zu ändernde Methode enthält. Klicken Sie anschließend im Editor auf den Methoden-Reiter. Mit einem

Doppelklick auf die zu ändernde Methode wird die Methode in einem Änderungsmodus angezeigt. Machen Sie Ihre Änderungen und klicken Sie anschließend auf *Eintragen*. Die Änderungen werden übernommen. Siehe Abbildung 9.45.

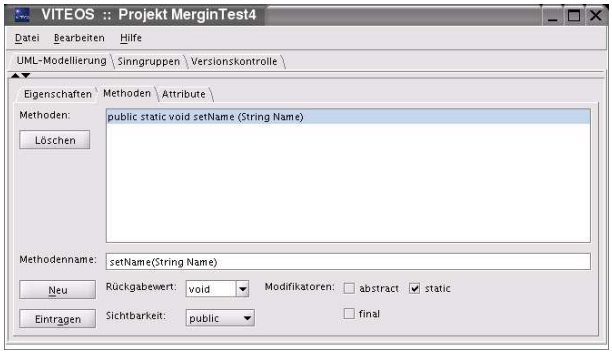

Abbildung 9.45: Methode bearbeiten

9.20 Methode löschen

Mit einem Doppelklick aktivieren Sie zunächst in dem Klassenbaum, die Klasse, die die zu löschende Methode enthält. Klicken Sie anschließend im Editor auf den Methoden-Reiter. Markieren Sie die Methode, die Sie löschen wollen und klicken Sie anschließend auf *Löschen*. Die Methode wird gelöscht und verschwindet aus der Liste der Methoden. Siehe Abbildung 9.46

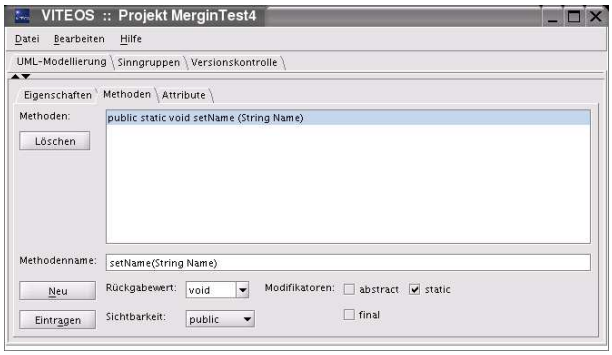

Abbildung 9.46: Methode löschen

9.21 Attribut bearbeiten

Mit einem Doppelklick aktivieren Sie zunächst in dem Klassenbaum, die Klasse, die das zu ändernde Attribut enthält. Klicken Sie anschließend im Editor auf den Attribute-Reiter. Mit einem Doppelklick auf das zu ändernde Attribut wird das Attribut in einem Änderungsmodus angezeigt. Machen Sie Ihre Änderungen und klicken anschließend auf *Eintragen*. Die Änderungen werden übernommen. Siehe Abbildung 9.47.

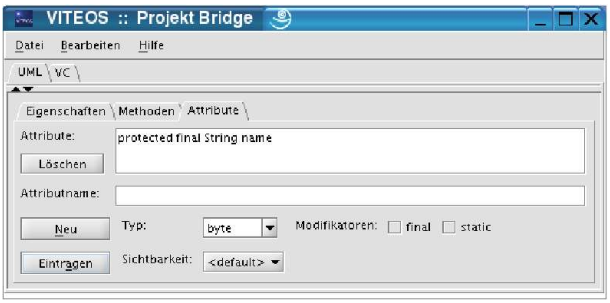

Abbildung 9.47: Attribut bearbeiten

9.22 Attribut löschen

Mit einem Doppelklick aktivieren Sie zunächst in dem Klassenbaum, die Klasse, die das zu löschende Attribut enthält. Klicken Sie anschließend im Editor auf den Attribute-Reiter. Markieren Sie das Attribut, das Sie löschen wollen und klicken anschließend auf *Löschen*. Das Attribut wird gelöscht und verschwindet aus der Liste der Attribute. Siehe Abbildung 9.48

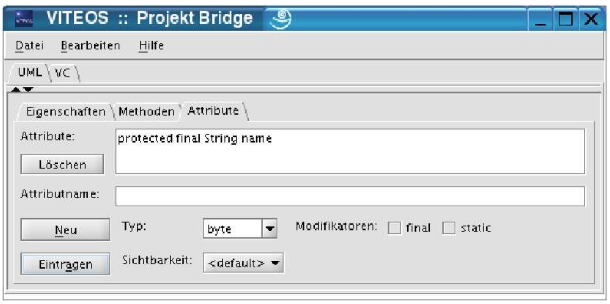

Abbildung 9.48: Attribut löschen

9.23 Konfiguration bearbeiten

Abbildung 9.49: Konfiguration bearbeiten

Über die Menüpunkte *Bearbeiten-Einstellungen* gelangen Sie in die Konfiguration-Maske. Die Konfiguration-Maske können Sie über die Strg-E Tastenkombination aufrufen. Siehe Abbildung 9.49 . In der Konfiguration-Maske teilen Sie dem Programm mit, wo es die Datenbank findet,

Abbildung 9.50: Die Einstellung der Datenbank ändern

sowie den Benutzer und sein Passwort. Siehe Abbildung 9.50. Mit einem Klick auf den *OK*- oder *Übernehmen*-Button werden die Änderungen übernommen. Wollen Sie die Änderungen verwerfen, so klicken Sie auf *Abbrechen*.

Kapitel 10

Die Evaluierung des Projektes *VITEOS*

10.1 Notwendigkeit und Vorgehen einer Evaluation mit Versuchspersonen

Da *VITEOS* ein neues Produkt ist, dessen Arbeitsweise sich bis auf die verwendete Modellierungs-Notation grundlegend von anderen bereits existierenden Tool unterscheidet, ist es notwendig die Konzepte, die *VITEOS* zugrunde liegen, unter realen Bedingungen zu überprüfen.
Dazu werden zwei Gruppen von Versuchspersonen gebildet, von denen eine Gruppe sowohl etablierte Werkzeuge wie auch *VITEOS* beherrscht. Die andere Gruppe sollte idealerweise die UML Notation jedoch weder *VITEOS* noch andere Werkzeuge gut kennen.
Die erste Gruppe mit ausreichend Kenntnissen auf beiden Gebieten wird durch unsere Projektgruppe gebildet. Dabei werden acht Teilnehmer auf vier Untergruppen aufgeteilt die aus jeweils zwei Personen bestehen.
Alle Gruppen erhalten die gleiche Aufgabe, die jeweils von zwei Gruppen mit *VITEOS* bzw. Together bearbeitet werden.
Das gleiche Vorgehen wird auf eine achtköpfige Gruppe angewandt, die aus außenstehenden Personen gebildet wird.
Um den wirklichen Nutzen der beiden Programme abschätzen zu können, dürfen generell keine Hilfsmittel (Zettel, Stifte, Tafeln, ...) außer den vorliegenden Programmen benutzt werden.

10.1.1 Ziele der Evaluation

- Aufschlüsse über die Handhabbarkeit der Benutzerschittstellen
- Beurteilung der Visualisierung in der Adjazenzmarix (Überblick über das Diagramm, Nutzung der Selektionsfunktion, ...)
- Einschätzung bezüglich des Nutzens des Versionierungsmechanismus
- Erwarteter Zeitaufwand im Vergleich zur graphischen Modellierung
- Zeitaufwand für die Einarbeitung in das jeweilige Programm

10.2 Projektbeschreibung für die PG-Gruppe

10.2.1 1. Iteration

Ein Wirt einer Szene-Gastronomie möchte seine Gäste näher kennen lernen und plant deshalb die Anschaffung eines computergestützten Fragebogens. Die Befragung soll selbstverständlich im Rahmen der aktuellen Datenschutzbestimmungen ablaufen. Dazu muss die Anonymität der Befragten gewahrt bleiben. Die Befragten dürfen auch nicht das Gefühl haben, dass ihre Angaben ihrer Person zugeordnet werden könnten.
Mit Hilfe dieses Bildschirm-Fragebogens kann der Wirt seine Gäste einzeln befragen und ihre Antworten direkt in das Programm eingeben. Zusätzlich soll eine Möglichkeit geschaffen werden, Stammgäste wiederholt zu befragen. Dazu geben die Gäste einen selbst gewählten Code ein.
Da der Wirt regelmäßige Fragebogenaktionen plant, muss er seinen Fragebogen immer wieder neu zusammenstellen können. Dazu will er auf einen Fragenkatalog von 'Standardfragen' zurückgreifen können, die mit dem Programm ausgeliefert werden, z.B. nach Alter, Geschlecht usw. Zusätzlich will er die Möglichkeit haben, eigene Fragen zu erstellen. Dabei wählt er den Typ der Frage (z.B. ja/nein-Frage, Antwortauswahl aus Liste, Bewertungen von 1-5 o. Ä.) und formuliert seine Frage. Man unterscheidet zwischen offenen und geschlossenen Fragen. Als Ergebnis einer offenen Frage erhält man eine Liste von Antworten, die bei der Auswertung des Fragebogens nach ihrem Inhalt kategorisiert werden. Bei geschlossenen Fragen interessieren in erster Linie die Häufigkeiten der Antworten.
Jederzeit muss der Wirt die Möglichkeit haben, Fragen zu editieren, also den Text der Frage oder der Antworten zu verändern, natürlich nicht während einer laufenden Befragung. Vom Wirt eingegebene Fragen werden dem internen Fragebogenkatalog hinzugefügt und gespeichert.
Der Wirt will sich die ausgefüllten Fragebögen ansehen können. Außerdem will er über eine bestimmte Menge von Fragebögen einer Befragung (z.B. über die, die in einem Monat ausgefüllt wurden) eine Auswertung machen. Ihn interessieren sowohl quantitative als auch qualitative Aspekte. Je nach Typ der Frage bietet sich dafür ein Torten-, Balken- oder Kurven-Diagramm an. Zusätzlich soll das Programm Mittelwerte und Standardabweichung ausrechnen und in den Diagrammen einzeichnen, in denen es Sinn macht. Der Mittelwert macht z.B. bei der Frage nach dem Geschlecht keinen Sinn.
Das Programm muss in der Lage sein, mit fehlenden Angaben sinnvoll umzugehen. Dazu ist es wichtig, zu wissen, wie viel Prozent der Befragten diese Frage beantwortet haben. Die Eigenschaft von elektronischen Fragebögen, dass nur dann die nächste Frage erscheint, wenn die vorherige beantwortet wurde, wird als ausgesprochen lästig empfunden und führt zu vielen nicht ausgefüllten Fragebögen. Weiterhin sollen einmal gegebenen Antworten korrigierbar sein, wenn die Befragten später merken, dass sie sich vertan haben.

10.2.2 2. Iteration

Besonders interessant bei der Auswertung ist die Kreuzung und Verknüpfung von Fragen. Solche Verknüpfungen sollen in Abhängigkeit von wählbaren Parametern wie, Geschlecht, Alter, Stammgast ja oder nein, vom Programm automatisch generiert werden. Beispielsweise interessiert den Wirt, welches Getränk die 30-40 jährigen Frauen bevorzugen. Auch hier soll das Programm die Möglichkeit geben, Fragen zu verknüpfen.
Eine einmal vorgenommene Auswertung soll natürlich später noch zugreifbar sein. Da die Befragungen wiederholt werden, interessiert ihn auch der zeitliche Verlauf. Genauso möchte der Wirt Auswertungen drucken, um sie eventuell seinen Mitarbeitern oder Zulieferern zu zeigen. Der Wirt sollte die Möglichkeit haben, Auswertungen selber zu definieren.

10.2.3 3. Iteration

Aus Gründen der Bequemlichkeit, soll es Gästen möglich seinen Fragebögen von zuhause aus ausfüllen zu können. Dazu muss es dem Benutzer ermöglicht werden sich über das Internet authentifizieren zu können. Die Details, wie die Authentifizierung genau ablaufen soll, bleiben der Fantasie der PG überlassen.

[Angelehnt an die erste Projektbeschreibung zum SoPra SS 2004]

10.2.4 Der Fragebogen

Fragebogen

Allgemeine Angaben

Dipl. Informatiker (Uni/FH): ☐ ja ☐ nein

Semesterzahl: _____

Erfahrung mit UML: ☐ Anfänger ☐ Fortgeschritten ☐ Experte ☐ vierter Amigo

Bekannte UML-Tools: _____

_____

Projektverlauf

Entwicklungs-Dauer: 1. Iteration _____

2. Iteration _____

3. Iteration _____

Anzahl Klassen: 1. Iteration _____

2. Iteration _____

3. Iteration _____

Maximale Relationszahl an einer Klasse: _____

Kurze Beschreibung des Vorgehens zu Beginn einer neuen Iteration (was hast Du gemacht?):

_____

_____

_____

Bewertung des verwendeten Werkzeugs

Werkzeug: ☐ *VITEOS* ☐ Together

Arbeitsgeschwindigkeit: - □ □ □ □ □ +

Versionsverwaltung der Diagramme: - □ □ □ □ □ +

Übersichtlichkeit des Modells: - □ □ □ □ □ +

Bitte schreibe hier auf, was Dir an dem Werkzeug gefällt oder nicht gefällt:

Würdest Du das Programm zur Modellierung nutzen? □ ja □ nein

10.2.5 Auswertung der Fragebogen

Nach dem die Versuchpersonen das Werkzeug *VITEOS* evaluiert haben, wurden sie gebeten den oben dargestellten Fragebogen auszufüllen. Die Hälfte der Befragten waren Informatiker und die andere Hälfte nicht. Sie alle waren jedoch mit UML und einigen Tools wie Together, Argo UML oder Rational Rose vertraut. Für die 1.Iteration brauchten die Nicht-Informatiker im Durchschnitt 30 min., für die 2.Iteration 20 min. und für die 3. Iteration 10 min.. Die Informatiker haben für die 1.Iteration im Durchschnitt 60 min., für die 2.Iteration zwischen 30-40 min. und für die 3.Iteration ca. 20 min. gebraucht. In der 1.Iteration hatten die Nicht-Informatiker im Durchschnitt 12 Klassen, in der 2.Iteration 10 Klassen und in der 3.Iteration auch 10 Klassen. Die Informatiker hingegen benötigten für die 1.Iteration im Durchschnitt 20 Klassen, für die 2.Iteration auch 20 Klassen und für die 3.Iteration um die 13 Klassen. Im Großen und Ganzen sind die Entwickler zufrieden mit neuen Tool *VITEOS*. Und 50% der allgemein Befragten wären bereit diesen Tool für Ihre Entwicklung einzusetzen und die anderen 50% wären das nicht.