Gruppe GR-6

Entwurfsbeschreibung

Universität Leipzig, Institut für Informatik

Version vom 27. Juni 2005

Inhaltsverzeichnis

Kapitel 1

Allgemeines

1.1 Charakterisierung

Die zu entwickelnde Software soll die Mitarbeiter der Firma SoftConsult in die Lage versetzen, interne Abläufe während der Analysephase durch ein Begriffsnetz darstellen zu können.

1.2 Systemvoraussetzung

Zum Ausführen der Rahmenapplikation und damit des *plugins* wird eine *java virtual machine standard edition* in der Version 1.4 oder später benötigt.

Kapitel 2

Produktübersicht

Das zu entwickelnde *plugin* – ein *tab widget* für Protégé – soll eine

- anpassbare,
- selektive,
- sortierbare,
- editierfähige und
- tabellarische

Sicht auf die Instanzen einer Klasse einer Wissensbasis ermöglichen.

Kapitel 3

Struktur- und Entwurfsprinzipien

Wenn im folgenden Paketnamen in der Gestalt `~.PaketA` angegeben sind, so befindet sich `PaketA` im Wurzelpaket des *plugins*, d.h. der vollständige Pfad ist

`de.unileipzig.informatik.bis.owltoitab.PaketA`

3.1 MVC-Architektur

Das *plugin* realisiert das Entwurfsmuster MVC und damit die Idee der Trennung des Programms in die drei Einheiten Datenmodell (*model*), Präsentation (*view*) und Programmsteuerung (*controller*).

Das Ziel dieser Vorgehensweise ist ein flexibles Programmdesign, um u.a. eine spätere Änderung oder Erweiterung einfach zu halten und die Wiederverwendbarkeit der einzelnen Komponenten zu ermöglichen. Beispielsweise lässt sich so auf recht einfache Weise die Swing-Benutzerschnittstelle durch eine Weboberfläche ergänzen.

Außerdem sorgt das Modell bei großen Anwendungen für eine gewisse Übersicht und Ordnung durch Reduzierung der Komplexität.

Gleichzeitig bringt die Trennung auch eine einfache Rollenverteilung mit sich. Die Projektmitglieder können so parallel und unabhängig voneinander an den unterschiedlichen Einheiten der Software arbeiten.

3.1.1 Paketstruktur

Die Struktur der *packages* des *plugins* spiegelt die Verwendung des MVC-Entwurfsmusters wieder (siehe Abbildung 3.1). Die Klassen der Steuerungsschicht befinden sich direkt oder indirekt in dem *package* `control`, die Klassen der Darstellungsschicht im *package* `view` und die Klassen des Datenmodells im Paket `model`.

3.1.2 Model

Das *model* enthält die Daten und die Kernfunktionalität der Anwendung, ist unabhängig vom *view* und *controller* und hat insbesondere noch folgende zusätzliche Aufgaben: Das *model* kennt alle *views* und *controller*, die zum Einsatz kommen und informiert diese über Änderungen in den Daten.

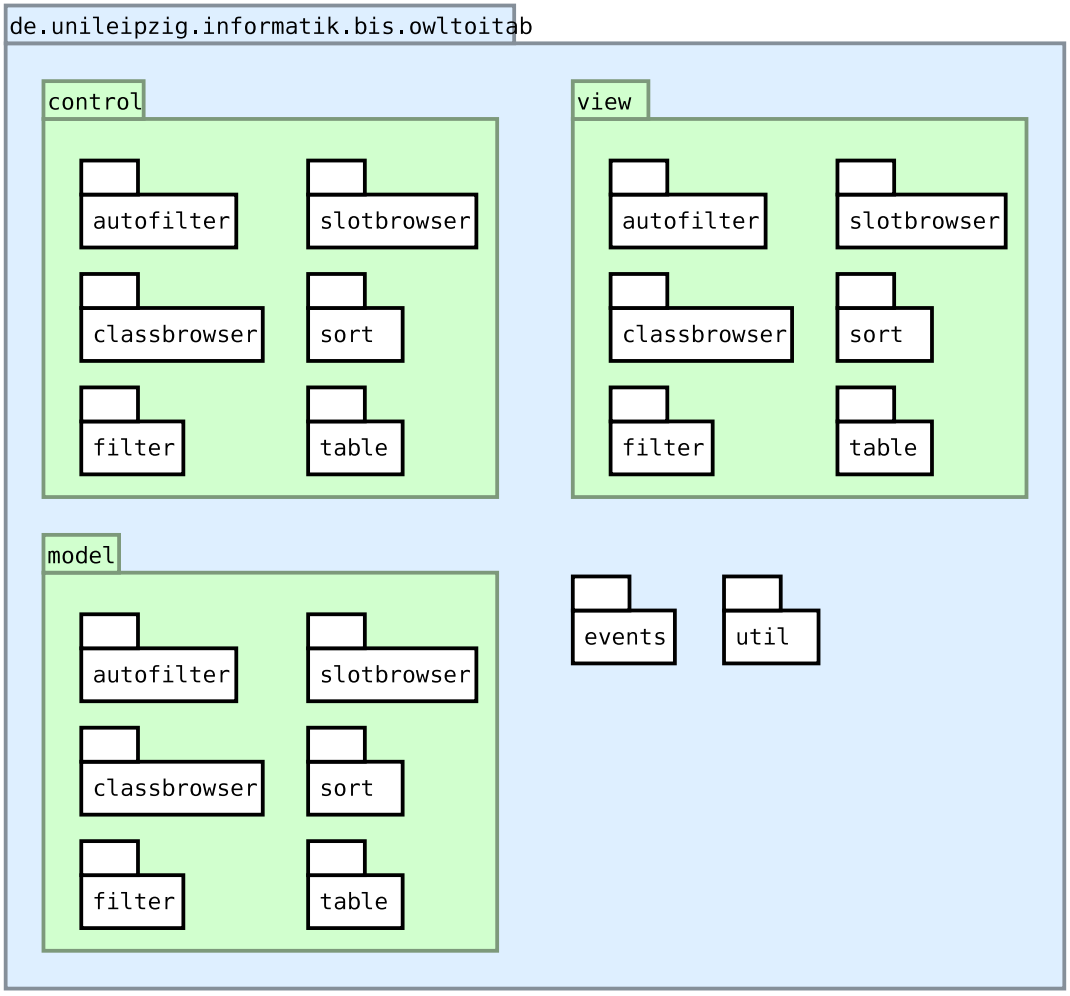

Abbildung 3.1: Paketstruktur

3.1.3 View

Die Darstellungsschicht ist lediglich für die Anzeige der Daten zuständig. Sie enthält daher keine Programmlogik. Die Ereignisse, die der Benutzer auslöst, werden an die Steuerungsschicht weitergeleitet und dort verarbeitet.

3.1.4 Control

Die *controller* sind die Eingabeschnittstelle zwischen Benutzer und *model*. Sie interpretieren die empfangenen Eingabedaten und übergeben sie dem *model*. Ein *controller* ist auf ein speziellles *view* zugeschnitten. Ist sein Verhalten vom *model* abhängig, so muss er auch auf Änderungen im *model* reagieren können.

3.2 Beobachter-Entwurfsmuster mit Asynchronität

Das *plugin* verwendet das Beobachter-Entwurfsmuster, um die Kommunikation

- zwischen *model* und *view* und
- zwischen *view* und *controller*

zu realisieren. Im Fall der Kommunikation zwischen *model* und *view* ist das *model* der Beobachtete und die *view* der Beobachter. Die Klasse `~.model.Model` erweitert die Klasse `java.util.Observable` der JAVA-Klassenbibliothek und erhält somit die nötige Funktionalität eines Beobachteten. Die Klassen des *views* implementieren das *interface* `java.util.Observer` und können sich dadurch als Beobachter bei dem *model* anmelden. Die Kommunikation zwischen den *views* und deren *controllern* geschieht auf die gleiche, asynchrone Weise.

Die Kommunikation geschieht jedoch nicht auf dem direkten (synchronen) Weg, sondern mittels einer asynchronen *event*-Zustellung mittels einer *event*-Warteschlange vom Typ `~.util.ObserverQueue`, die in einem eigenen *thread* ausgeführt wird. Die Klasse `ObserverQueue` implementiert das *interface* `Observer` und erweitert die Klasse `Observable` zu gleich. Das heißt, dass diese *event*-Zustellungsklasse sowohl Beobachter als auch Beobachteter ist. Die *observer queue* ist als einziger Beobachter am *model* angemeldet und erhält damit die *events*, die das *model* schickt. Der *thread* des *models* wird zur Abarbeitung des *events* innerhalb der *observer queue nicht* verwendet. Er platziert lediglich den *event* in der Warteschlange. Die Abarbeitung übernimmt der *thread* der Warteschlange. Der *thread* der Warteschlange ist so lange in einem *wait*-Zustand bis er durch die Platzierung eines ankommenden *events* in der Warteschlange aufgeweckt wird. Sobald dies geschieht, arbeitet er alle *events* in der Warteschlange ab. Das heißt er informiert die Beobachter des *models*, in dem er ihnen den *event* zustellt.

Durch die erreichte asynchrone Kommunikation wird das Antwortverhalten der Benutzerschnittstelle maximiert. Das bedeutet praktisch, dass die Oberfläche nicht „einfriert“, wenn der Benutzer komplizierte Berechnungen angestossen hat. Sie aktualisiert sich dagegen, sobald die Berechnungen abgeschlossen sind und die Ergebnisse bereitstehen. Auf dieser Basis ließen sich bei einer Weiterentwicklung des *plugins* einfach Fortschrittsbalken, die den Status von Berechnungen im Hintergrund andeuten, implementieren.

3.3 Angezeigte Zeichenketten und Lokalisierung

Um erstens eine Lokalisierung der Oberfläche zu erreichen und zweitens den Programmcode von anzuzeigenden Zeichenketten zu säubern werden sämtliche anzuzeigenden Zeichenketten in der Datei `~.util.resources.message.properties` eingetragen.

Bekommen kann man ein solche Zeichenkette mittels des Singleton-Objekts `~.util.Messages`. Der Zugriff erfolgt ganz konkret:

```
Messages.getMessages().get( String key );
```

Dabei ist der String `key` ein Name der dann lokalisiert wird.

Möchte man eine Lokalisierung für eine andere Sprache schreiben, so erzeugt man einfach eine weitere Datei `message_xx.properties` im *package* `~.util.resources`. Wobei `xx` für eine Sprache steht. Gültige Sprachabkürzungen sind der JAVA API Dokumentation für die Klasse `ResourceBundle` zu entnehmen. Beispielhaft exisitiert in unserem Fall zusätzlich zur Datei `message.properties` noch die Datei `message_de.properties` für eine deutsche Lokalisierung.

Hinweis `ResourceBundle` zieht eine lokalisierte Version (`message_xx.properties`) der Standard-Version (`message.properties`) vor. D.h. für Unterstützung einer neuen Sprache ist keine Anpassung von Code notwendig!

3.4 Logging-Konzept

Das Logging wird für das gesamte Pluggin von der Klasse `~.util.Logging` implementiert. Diese Klasse ist als *singleton* implementiert. Es existiert also nur eine einzige Instanz. Die Methode `getLogger()` gibt die Referenz auf die einzige Instanz zurück. Geloggt wird in einer der acht Kategorien:

- SEVERE,
- INFO,
- FINE,
- FINEST,
- WARNING,
- CONFIG,
- FINER,
- ALL

Und zwar auf folgende Art und Weise:

```
Logging.getLogging().getLogger().log( Level, String );
```

Standardmäßig wird nichts geloggt. Eine Änderung des Verhaltens soll nur über die Datei `~.util.resources.logging.properties` geändert werden!

- Derzeit kann nur ein Logger festgelegt werden. Dazu wird der Schlüssel logger.handler verwendet:

  z.B. `logger.handler = java.util.logging.ConsoleHandler`

- Das Level des Loggers wird festgelegt mit dem Schlüssel logger.level.

  z.B. `logger.level = ALL`

- Das Level des Handlers wird festgelegt mit `HANDLERNAME.level`.

  z.B. `java.util.logging.ConsoleHandler.level = ALL` (im Falle des ConsoleHandlers)

* Dieses Beispiel würde ALLE Nachrichten auf die Standardkonsole von Protégé loggen.

3.5 ANT-Automatisierung

Für die Erzeugung der JAR-Dateien, der *javadoc*-Dokumentation sowie des wöchentlichen abzugebendem *release* wurde ein Skript für Ant geschrieben, welches sich in `~/etc/build.xml` befindet. Die zu verwendeten Targets sind:

- **clean**: Dieses *target* löscht alle `.class` Dateien, die *javadoc*-Dokumentation und alle Einträge im *distribution* (dist)-Verzeichnis.

  Nur dazu gedacht, falls man das Verzeichnis an jemand anderes weitergeben möchte und keine unnötigen Daten im Projektverzeichnis existieren sollen.

- **eclipse.clean**: Für Eclipse sollen die `.class` Dateien nicht gelöscht werden, weil die meisten Entwickler die *autobuild*-Funktion verwenden. Die Wirkung ist diesselbe wie beim *target clean* außer dass die `.class` Dateien im Verzeichnis `build` nicht gelöscht werden.

- **javadoc.build**: Erzeugt die *javadoc*-API-Dokumentation im Verzeichnis `~/doc/api`. Dieses *target* führt automatisch das target *javadoc.clean* aus. Dieses *target* funktioniet derzeit noch nicht, da wir nicht an den *classpath* des Eclipseprojekts herankommen. Bis zum funktionieren bitte die in Eclipse eingebaute *javadoc*-Generierung verwenden!

- **javadoc.clean**: Löscht die aktuelle *javadoc*-API-Dokumentation.

- **eclipse.debug.dist**: Erzeugt eine *debug*-Version des Plugins im Verzeichnis `~/dist`

- **eclipse.release.dist**: Erzeugt eine *release*-Version des Plugins im Verzeichnis `~/dist`

- **eclipse.debug.install**: Erzeugt eine *debug*-Version des Plugins im Verzeichnis `~/dist` und installiert diese im `plugins`-Verzeichnis des Protégé-Installationsverzeichnises. Damit dies funktioniert muss die Umgebungsvariable `PROTEGE` richtig gesetzt sein.

- **eclipse.debug.release**: Erzeugt eine *release*-Version des *plugins* im Verzeichnis `~/dist` und installiert diese im `plugins`-Verzeichnis des Protégé-Installationsverzeichnises. Damit dies funktioniert muss die Umgebungsvariable `PROTEGE` richtig gesetzt sein.

- **swtp.build**: Erzeugt im Verzeichnis `~.dist` die Datei `release.zip`. Diese enthält vom Handbuch, der Entwurfsbeschreibung, der Aufwandserfassung und dem Story-plan alle nötigen Dateien, einschließlich *test cases*. Es wird dabei ein *release*-Version ohne in das jar integrierten Quellcode erzeugt.

- **test**: Führt die im Verzeichnis `~.test` befindlichen JUnit-Tests aus und schreibt eine Zusammenfassung in die Standardausgabe.

Andere definierte *targets* können sich ändern und deren dauerhafte Verfügbarkeit ist nicht gewährleistet.

3.6 Utilities

3.6.1 MoveModel

Da in der Werkzeugleiste vermehrt JTrees verwendet wurden, und die Funktionalität zum verschieben nicht immer wieder neu implementiert werden sollte, wurde abstrahiert und ein eine Komponente für Verschiebungsoperationen geschrieben. Dieses besteht aus

- .view.util.move.*
- .model.util.move.*

Hierbei traten jedoch massive Mehrfachvererbungsprobleme auf, die NICHT auf besonders elegante Art und Weise zu lösen sind (aufgrund der Beschränkungen der verwendeten Programmiersprache Java). Die Lösung in unserem Fall sieht so aus, dass man die Klassen als inline-Klassen aufnimmt und die als abstract deklarierten Funktionen deklariert. Dies ist eine Emulation von Mehrfachvererbung. Sie ermöglicht, dass der eigentliche Code nicht neu implementiert werden muss. Dafür ist die Anpassung immer noch ungerechtfertigt aufwendig. In zukünftigen Versionen von Java sollte dieses massive Manko aufgehoben werden. Aus Erfahrung kann man sagen, dass sich viele Probleme viel schneller mit Mehrfachvererbung hätten lösen lassen.

Die Vorgehensweise für die Verwendung ist hierbei folgende:

1. Man erstellt das Modell zu seiner Baumstruktur. Wichtig ist hierbei, dass alle Komponenten (auch die Blätter) die Klasse MoveNode als Referenz kapseln. Das reicht jedoch nicht. Die Klasse muss als inline-Klasse verwendet werden; und zwar so, dass die als abstract deklarierten Funktionen entsprechend der Baumstruktur angepasst werden. Beispiele sind im Filterwerkezug und im Sortierwerkzeug zu finden. Als zweckmäßig hat sich erwiesen, dass alle Klassen die im Baum sind von IMoveNode ableiten. So kann man immer ohne CastException auf das Interface casten.

2. Die Modellklasse, die das Wurzelelement hält muss von IMoveModel ableiten. An dieses müssen die SelectionEvents durchgereicht werden.

3. Man implementiert ein TreeModel. Hierzu ist sowohl in der Java API als auch in den oben genannten Beispielen nachzusehen. Dieser Schritt ist vergleichsweise einfach.

4. Anschließend baut man in seiner Viewkomponente die Buttons für die Verschiebeoperationen TOP, UP, DOWN, BOTTOM ein. Man baut ActionListener. Wichtig ist hierbei, dass man von AbstractActionContainer ableitet. Diese Klasse kapselt einfach nur eine Action. Dies war nötig um das Aussehen (Name und Icon) von der wirklichen Ausführung abzutrennen.

5. Als Subaction instanziert man eine MoveAction und weißt sie seinen ButtonListenern als Subaction zu.

6. Außerdem muss man der MoveAction auch das Model mitteilen. Das Model muss hierfür das Interface IModel implementieren.

7. Bei einem Rebuilt des Tree (das kann man machen wie man will) erzeugt das MoveModel ein SelectionEvent. Um die ursprüngliche Selektion wiederherzustellen dient die Klasse SelectionEventParser.

Genauere Informationen sind in den Bäumen für Filter und Sort zu finden. Die Quellcode-Dokumentation ist für das Nachvollziehen geeignet.

3.6.2 TreeNode

Dieses Werkzeugpacket in .model.util.tree liefert unsere Standardbaumknotenimplementation. Die Verwendung dieses Packets für Bäume ist aus Konsistensgründen anzuraten. Unser Filterbaum setzt darauf auf.

3.6.3 LogicTerm

Dieses Package (.model.util.logicterm) ist eine Implemenation eines logischen Terms basierend auf den TreeNodes. Es werden zwischen logischen Bedingungen und logischen Konjunktoren (AND, OR und NOT) unterschieden. Die konkrete Verwendung erfolgt wiederum im FilterModel.

3.7 TableParser

Für alle TableParser (in unserem Fall entweder Sortier oder Filtermechanismen) gibt es die Möglichkeit sich bei der Tabelle anzumelden. Fortan wird bei einer Aktualisierung der Tabelle jeder einzelne Parser aufgerufen. Die Anmeldung erfolgt über `~model.table.TableModel#addTableParser(IOWLIndividualParser)`. Hieraus ist zu entnehmen, dass jeder Parser das Interface IOWLIndividualParser implementieren muss. Näheres im Interface und javadoc.

3.7.1 FilterModel

Das Filtermodell war zunächst für wesentlich mehr Relationen (derzeit nur = und !=) vorgesehen. Der Zeitmangel hat hier auch noch Spuren dieser Zeit hinterlassen. Deshalb bitte nicht wundern wenn hier im Quelltext teilweise noch von mehr Relationen die Rede ist. Wenn einiges nicht ganz logisch erscheint nicht wundern; es nicht so, dass der Autor das ganze nicht verstanden hat, sondern dass wirklich noch Optimierungsbedarf besteht.

3.7.2 SortModel

Im SortModel sollte man besonders darauf achten, dass (derzeit noch nicht richtig zusehen) alles im voraus so entwickelt wurde, dass später weitere sogenannte SortSchemes geschrieben werden können und die Sortierkomponente weiterhin vollständig funktioniert. Ein fast funktionsfähiges Beispiel ist in .model.tools.sort.schemes.xml_gYear zu finden. Dies wird nicht benötigt weil sich Jahre schon lexikalisch korrekt sortieren lassen. Für komplexere Datumsangaben wäre aber ein weiteres Schema sinnvoll. Für welche Typen von OWLProperties das möglich sein soll kann mit der Methode isSupported festgelegt werden (im jeweiligen Sortierschema).

3.7.3 AutoFilterModel

Für das AutoFilterModel greifen wir einfach auf eine weitere Instanz das Filtermodels zurück. Somit bleibt die gesamte Funktionalität erhalten und wir haben die Garantie, dass auch alles funktioniert. Wenn ein Fehlverhalten festgestellt werden sollte, so sollte das ganze also im FilterModel korrigiert werden. Das man bei Änderungen am AutoFilter sich erst in das Filterkonzept einarbeiten muss versteht sich von selbst.

Kapitel 4

Ausstehende Verbesserungen

In der Realität würde dieser Teil nur intern zur Verfügung stehen. Eventuell würde er auch gar nicht existieren, denn es wäre mehr Zeit zur Verfügung. Die fünf Wochen Implementierungszeit waren wirklich zu kurz (für den Umfang), so dass diese Arbeiten bzw. Verbesserungen einfach nicht mehr realisiert werden konnten. Wir waren aber in der Gruppe der Meinung, diese darzustellen, dass, falls eine Weiterentwicklung stattfinden sollte, strukturiert vorgegangen werden kann. Also praktisch für eine Version release 6.

1. Der Doppelklick im Propertybrowser ist nicht abfangbar. Wir haben probiert, die Listener des JTree herauszuholen und die Ereignisse durch einen Listener der dieses Event nicht durchreicht, zu ersetzen. Die Originallistener also als eine Art Sublistener. Dies hat aber IMMER Exceptions geworfen. Nach vier Stunden suchen war keine Lösung zu finden und das Problem wurde auf die Warteliste gesetzt. Es scheint ein Problem der Java API zu sein und nicht von unserem CatchListener.

2. Das MoveModel ist später entstanden als die Verschiebeoperationen im OWLPropertyBrowser. Deswegen hat er noch die Originalmethoden in Verwendung und nicht das Utilitypackage MoveNode. Für Konformität sollte auch das noch angepasst werden.

3. Die TreeCellRenderer für Sort und Filter sind noch nicht optimal schön. Einmal sollten im Sortierwerkzeug das linke Icon in voller Größe dargestellt werden, und analog wie im OWLPropertyBrowser die Balken so breit gemacht werden wie das TreePanel breit ist. Der JTree liefert aber mit der Methode getRowBounds erst nach dem Rendern ein vernünftiges Rechteck; wir kamen in Endlosschleifen. Im OWLPropertyBrowser funktioniert das mit einem dirty hack. Deswegen wurde das am Schluss nicht mehr übernommen.

4. Die Icons im OWLPropertyTreeCellRenderer für die Darstellung der Properties sollten aus Homogenitätsgründen aus Protégé übernommen werden.

5. ToolbarButtons in der grünen Leiste der Toolbar wären schön. Und zwar für Größenänderungen des Panel.

6. Weitere Sortiertypen wären wünschenswert.

7. Es wäre sinnvoll pro Klasse die Einstellungen der einzelnen Komponenten als Kommentar in der OWL-Klasse zu speichern, so dass diese beim Klassenwechsel wiederhergestellt werden können.

8. Ein weiteres Problem ist derzeit, dass der JTree bei Aufruf der Methode getSelectionPaths() die Einträge nicht nach dem Vorkommen im JTree sortiert. Das Resultat ist, dass es bei Verschiebeoperationen bei mehreren Einträgen zu einer Änderung der Reihenfolge der markierten Einträge kommen kann. Hier müsste einfach beim speichern im SelectionEvent eine Sortierung durchgeführt werden. Dies ließe sich auch zur Verwendung in anderen Komponenten einfach (kostet aber Zeit) allgemein formulieren.

9. Im Export sollte der Exportbutton erst nach der ersten Erstellung der Tabelle funktionieren und vorher einfach deaktiviert sein. Derzeit ist er aktiv, macht ohne Tabelle aber nichts. Außerdem wäre eine Überarbeitung des Panels wünschenswert.

10. Analog zum MoveNode-Konzept sollte man das ganze auch für das Austauschen (ExchangeEvent, etc.) generalisieren.