Mitglieder: Markus Jäger, Patrick Oesterling, Lars Kolb, Sebastian Eichelbaum, Stefan Vollrath, Bei Fang, Anne Nitzsche
Datum: 26.06.2005

Entwurfsbeschreibung des Plugins GR-4

Inhaltsverzeichnis

1. Allgemeines
1.1. Kurzcharakterisierung ... 2
1.2. Systemvoraussetzungen ... 2
1.3. Produktumgebung ... 2

2. Produktübersicht ... 2

3. Grundsätzliche Struktur- und Entwurfsprinzipien für das Gesamtsystem
3.1. Allgemeines ... 4
3.2. Übersicht GUI ... 5
3.3. Übersicht Model ... 5
3.4. Auswahl der aktiven Klasse ... 7
3.5. Visualisierung der Individuals ... 8
3.6. Sortierung ... 9
3.7. Anpassen der Visualisierung (Spaltenfilter) ... 10
3.8. Editieren von Individuals ... 11
3.9. Suchen ... 12
3.10. Export/Import
3.10.1 Übersicht – Exportierfunktion ... 14
3.10.2 Übersicht – Importierfunktion ... 15
3.10.3 GUI ... 15
3.10.4. Model – Exportieren ... 16
3.10.5. Model – Importieren ... 18

4. Grundsätzliche Struktur- und Entwurfsprinzipien der einzelnen Pakete 20

1. Allgemeines

1.1. Kurzcharakterisierung

Laut Aufgabenstellung entwickeln wir ein Tab-Plugin, welches als Software-Zusatzmodul die Funktionalität von *Protégé* erweitert. Dazu muss das Plugin in den *plugins*-Ordner der Applikation *Protégé* eingefügt und in der Laufumgebung aktiviert werden.
Ziel des Plugins ist es, die Individuals einer Ontologie tabellarisch darzustellen und datenbankähnliche Such-, Filter- und Sortierfunktionen bereitzustellen um diese Darstellung nach persönlichen Wünschen anpassen zu können.

1.2. Systemvoraussetzungen

Da wir unser Plugin lediglich die Funktionalität von Protégé 3.0 erweitert, ist es ohne diese Applikation nicht ausführbar. Protégé 3.0 ist unter http://protege.stanford.edu/download/release/full/ frei verfügbar.
Ferner wird die JRE 1.4 oder höher benötigt um Protégé starten zu können.

1.3. Produktumgebung

Um das Plugin in Protégé einzubinden ist die *Plugin-jar* in den *plugins* Ordner von Protégé zu kopieren.

2. Produktübersicht

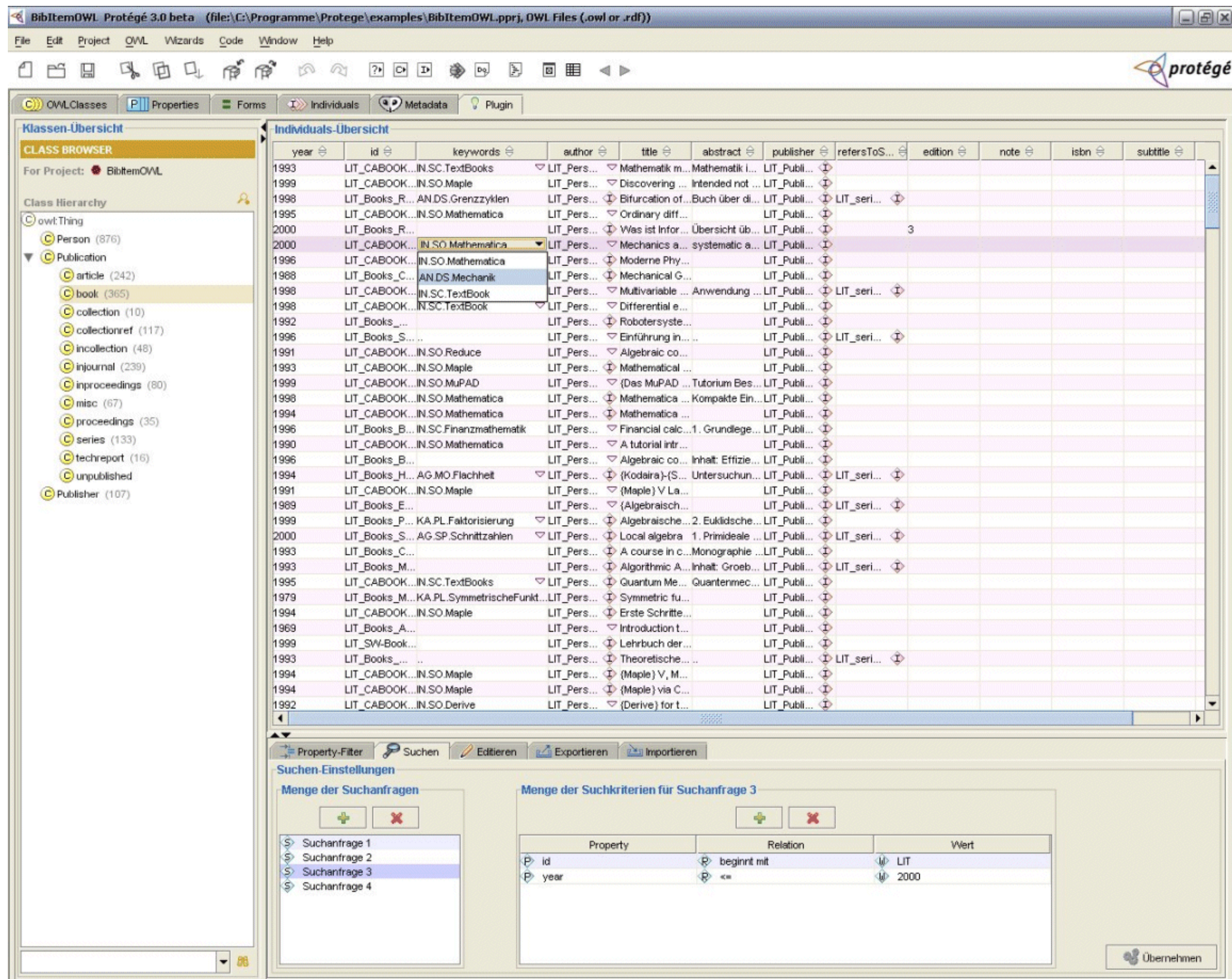
Zunächst muss unser Plugin aktiviert werden. Dies geschieht indem man ein OWL-Projekt öffnet oder erstellt und im Menüeintrag unter *Project* → *Configure* die Checkbox bei Plugin aktiviert. Anschließend öffnet sich neben *OWL Classes*, *Properties*, *Forms*, *Individuals* und *Metadata* ein neues Tab *„Plugin"*, in welchem sich die graphische Oberfläche für alle Funktionalitäten unseres Plugins befindet. Beim Speichern des Projektes wird in der zugehörigen *Protégé*-Projektdatei (*.pprj*) vermerkt, welche Tabs zuletzt geöffnet waren, so dass beim nächsten Laden des Projektes das Aktivieren entfällt.

Diese graphische Oberfläche soll an die von Protégé selbst angelehnt sein, auf der SWING-API basieren und aus drei Hauptelementen bestehen:

1)
Auf der linken Seite der Oberfläche befindet sich der Class-Browser, der alle Klassen der Ontologie sowie die Anzahl vorhandener Individuals jeder Klasse darstellt. Über diese bereits vorhandene Protégé Komponente soll die Auswahl der aktiven Klasse erfolgen, d.h. die Klasse deren Individuals dargestellt werden sollen.

2)
Den größten Platz der Oberfläche nimmt die Individuenübersicht ein. Dabei wird es sich um eine JTable handeln, in der zeilenweise die Individuals dargestellt werden. Pro Spalte der Tabelle sollen die Werte einer Property dargestellt werden. Hat eine Property mehrere Werte so wird die Einsicht in die verschieden Werte durch eine Combo-Box realisiert. Handelt es sich um eine Object-Property, also einen Verweis auf ein weiteres Individual, so soll dieses durch Anwählen eines Links in der Protégé-Individual Komponente dargestellt werden.

3)
Unter der Übersicht ist das Tab-Menu zu finden. Es besteht aus fünf Tabs, welche verschiedene Produktfunktionen repräsentieren:
Über das Property-Filter Tab sollen später einzelne Spalten der Tabelle, also Properties, über Checkboxen an- und abgewählt werden können. Die entsprechende Aktualisierung der Tabelle soll umgehend erfolgen.
Über das Suchen Tab soll der Nutzer später eine Suchanfrage an die Knowledge Base in Form einer DNF stellen welche die UND-ODER-Verknüpfung der Suchkriterien repräsentieren soll. Dabei soll die Eingabe dieser Form wird über zwei Eingabeelemente erfolgen, welche jeweils für die UND- und ODER-Verknüpfung stehen.
Im Editieren Tab soll für die aktuell markierte Zeile, der Protégé Instance Editor geöffnet werden, um die Propertyvalues des Individuals bearbeiten zu können.
Über das Exportieren Tab soll der Zielpfad für den Export der momentan dargestellten Individuals per JTextField oder per JFileChooser festgelegt werden, und der Export gestartet werden.
Analog soll über das Importieren Tab der Quellpfad der festgelegt werden um aus dieser Quelldatei Individuals in die aktuell gewählte Klasse einfügen zu können.



Screenshot Plugin v1.0

3. Grundsätzliche Struktur- und Entwurfsprinzipien für das Gesamtsystem

3.1. Allgemeines

Wie jedes Plugin für Protégé, muss auch dieses Plugin von **edu.stanford.smi. protege.widget.AbstractTabWidget** abgeleitet werden, damit es als Tab in der Protégé-Umgebung geladen werden kann. Auf diese Weise steht dem Plugin-Entwickler der Zugriff auf das Projekt und dessen *KnowlesgeBase* zur Verfügung. Er hat damit also Zugriff auf alle Klassen, Individuals und Properties um auf dieser Grundlage verschiedene Visualisierungs- oder Änderungsmöglichkeiten der Ontologie zu realisieren.

3.2. Übersicht GUI

Der Aufbau des Plugins ist so organisiert, dass möglichst stark abstrahiert wird, was bei der Programmierung der GUI bedeutet, dass diese über relativ viele Klassen verteilt wird, um einerseits eine eventuelle Wiederverwendung zu gewährleisten und um andererseits die Programmierarbeiten gut verteilen und organisieren zu können.

Die GUI des Plugins besteht aus den Klassen, die im Paket *ui* zu finden sind, welches weiter unten noch genauer erklärt wird. Der Aufbau der GUI geschieht hierarchisch, das heißt, dass die Bedienelemente, die für spezielle Funktionen da sind, in den jeweiligen Klassen zu finden sind und somit die *oberen* Klassen, die für die GUI verantwortlich sind, eher funktionslos sind, weil diese Klassen größtenteils nur das *JPanel* erweitern und damit eine *Komponenten-Träger*-Rolle haben.

Deswegen wird im Folgenden vereinfachte UML-Klassendiagramme ohne Methoden und Attribute benutzt, um den genauen Aufbau der GUI zu verdeutlichen:

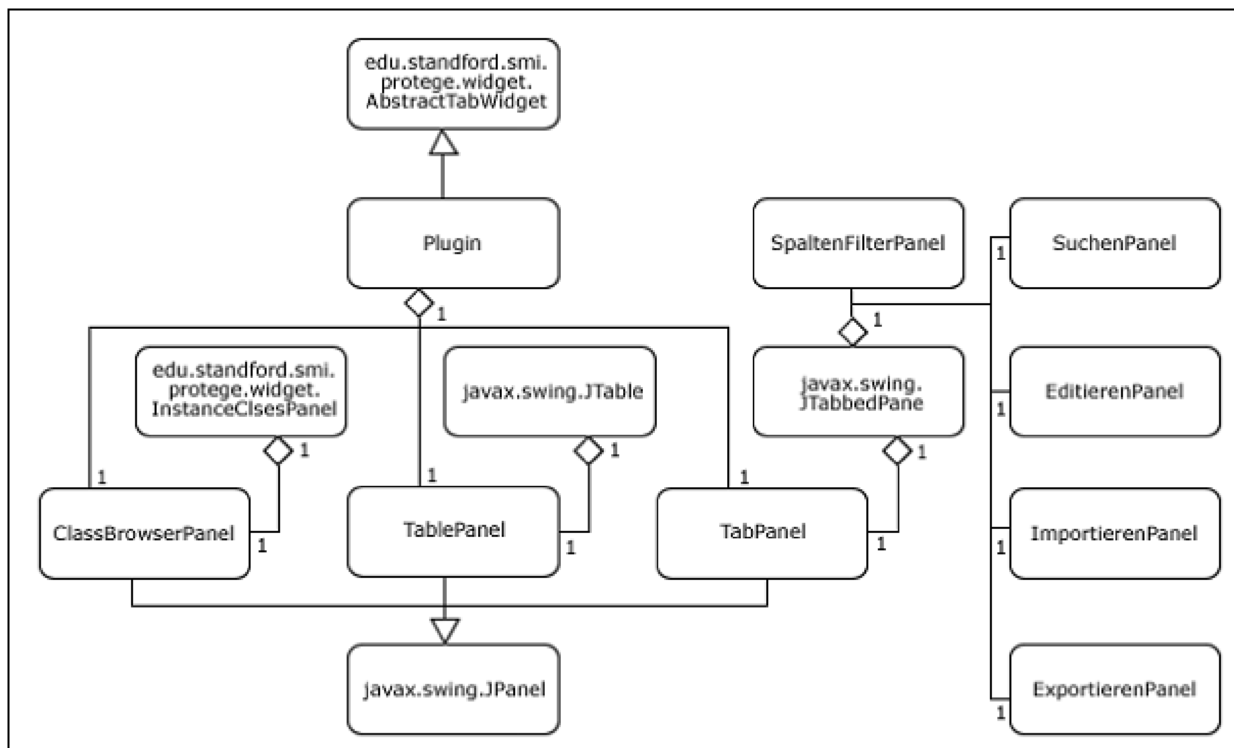


Bild1: vereinfachtes UML-Klassendiagramm der GUI

3.3 Übersicht Model

Die GUI stellt einerseits den *View* des Modells dar, welche das *Model* in irgendeiner Weise visualisiert, und auf der anderen Seite bietet die GUI Möglichkeiten, das *Model* zu manipulieren, es realisiert gleichzeitig den *Control*. Da im *Bild1* nur die GUI beschrieben ist, welche noch nicht über erwähnenswerte Funktionalität verfügt, soll nun getrennt das Wesentliche des *Models* in UML beschrieben werden:

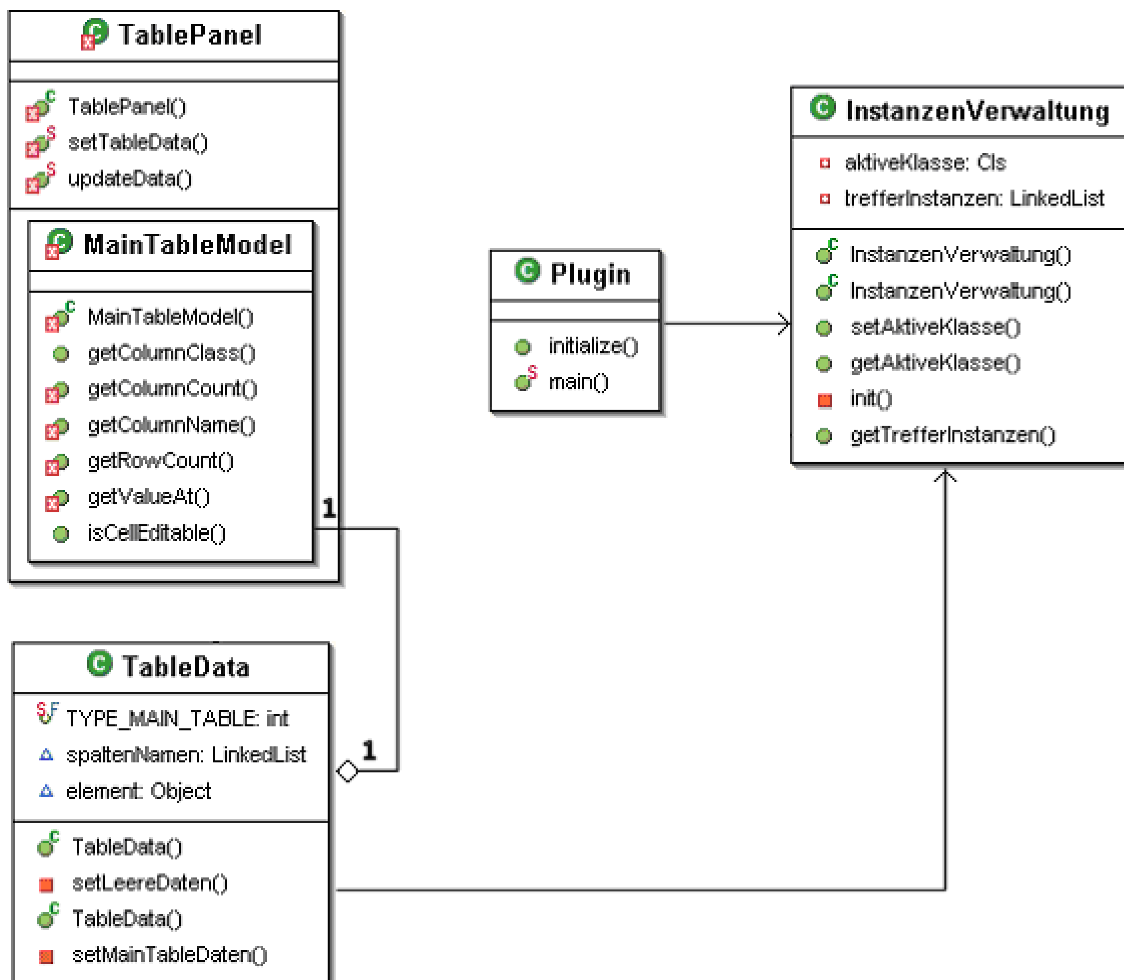


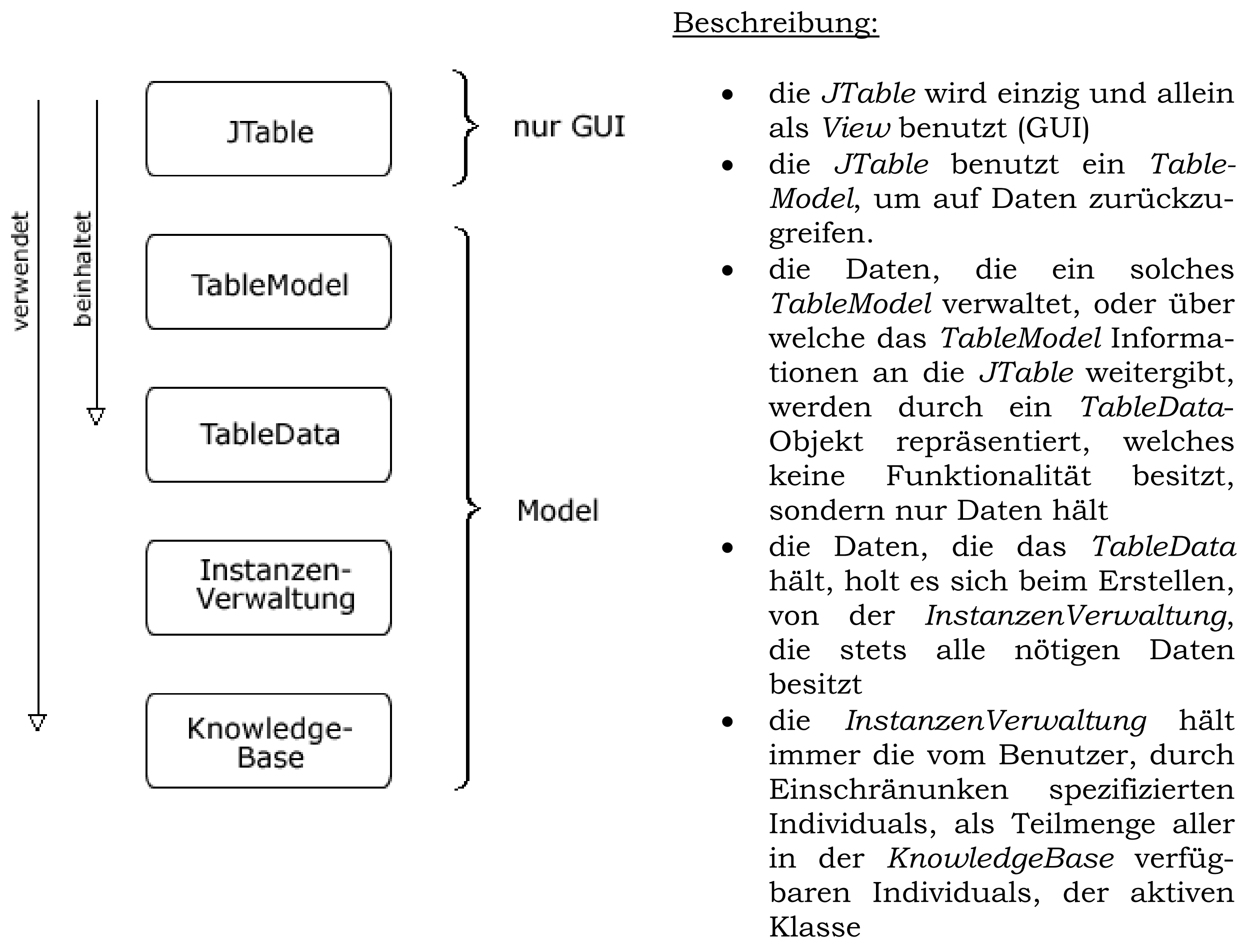
Bild2: UML-Klassendiagramm des Models

- *MainTableModel*
  → Eine *JTable* arbeitet in *JAVA* immer auf einem *TableModel*. In einer Instanz eines solchen *TableModel* werden sozusagen die Daten gespeichert, welche in der *JTable* angezeigt werden. Das *TableModel* kommt immer dann zum Einsatz, wenn sich die *JTable* neu zeichnen oder aktualisieren muss. Dann nämlich werden ganz bestimmte abstrakte Methoden, welche natürlich überschrieben werden müssen, von dem *TableModel* aufgerufen, damit die Tabelle beispielsweise weiß, wie viele Zeilen und Spalten zu zeichnen sind.

- *TableData*
  → Diese Klasse soll dazu dienen, die Daten zu halten, die eine *JTable* mit einem *TableModel* verwalten kann. Diese Daten beschränken sich auf eine Array von Spaltennamen und ein 2-dimensionales Object-Array, welches den Inhalt der Tabelle darstellt.

- *InstanzenVerwaltung*
  → Eine zentrale Klasse, die die im ClassBrowser selektierte Klasse, die *aktive Klasse,* speichert und entsprechend der Benutzereingaben eine Menge von *TrefferInstanzen*, als Teilmenge aller Individuals der *aktiven Klasse* hält, auf die dann von anderen Klassen zugegriffen werden kann.

Um den wesentlichen Zusammenhang der Modelklassen zu verstehen, soll nun darauf noch einmal näher eingegangen werden:



Beschreibung:

- die *JTable* wird einzig und allein als *View* benutzt (GUI)
- die *JTable* benutzt ein *Table-Model*, um auf Daten zurückzu-greifen.
- die Daten, die ein solches *TableModel* verwaltet, oder über welche das *TableModel* Informationen an die *JTable* weitergibt, werden durch ein *TableData*-Objekt repräsentiert, welches keine Funktionalität besitzt, sondern nur Daten hält
- die Daten, die das *TableData* hält, holt es sich beim Erstellen, von der *InstanzenVerwaltung*, die stets alle nötigen Daten besitzt
- die *InstanzenVerwaltung* hält immer die vom Benutzer, durch Einschränunken spezifizierten Individuals, als Teilmenge aller in der *KnowledgeBase* verfügbaren Individuals, der aktiven Klasse

3.4. Auswählen der aktiven Klasse

Das nun folgende Sequenzdiagramm soll den Ablauf beschreiben, welcher eintritt, wenn der Benutzer im *ClassBrowser* eine Klasse auswählt.

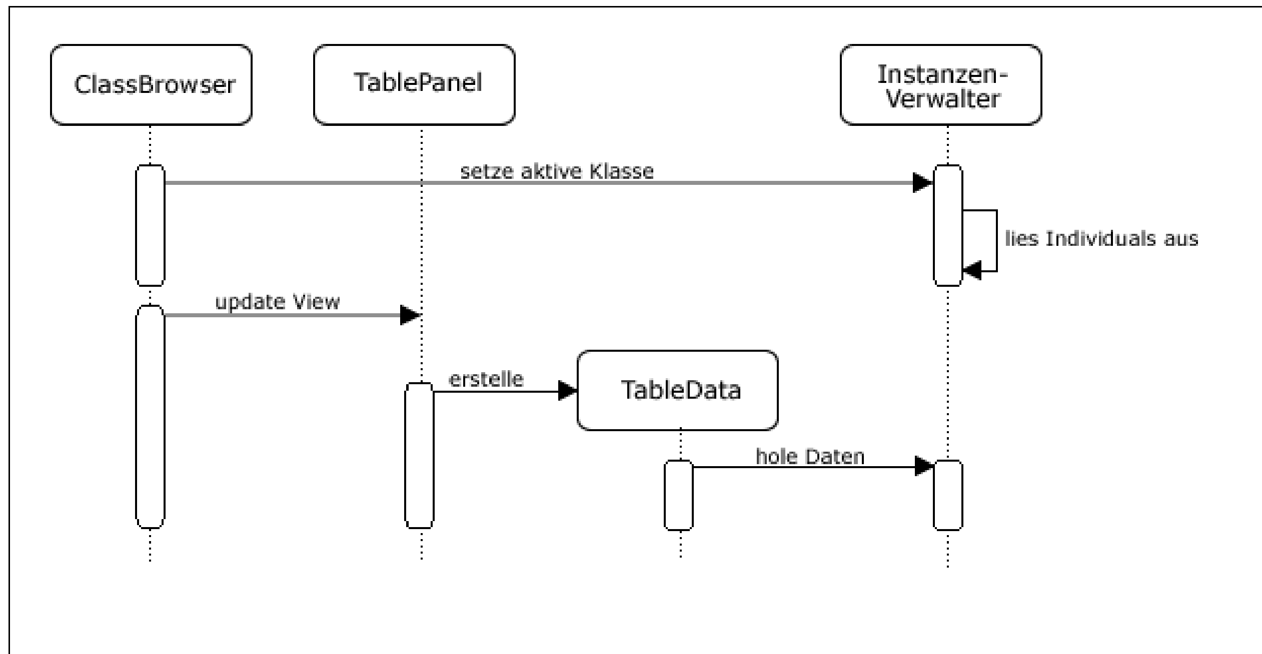


Bild3: UML-Sequenzdiagramm zur Auswahl einer OWL-Klasse und Anzeigen derer Individuals

Die *update()*-Methode mancher GUI-Elemente soll einfach nur das Neuladen der Daten sicherstellen, damit der Benutzer nach Eingabe bestimmter Parameter, in diesem Fall das Auswählen einer Klasse (und das damit verbundene Ändern der *aktiven Klasse*), immer aktuellen View des Models bekommt. Von diesen *update()*-Methoden wird auch im auch in weiteren Fällen z.B. nach Eingabe von Suchanforderungen, Gebrauch gemacht werden, um eine Änderung des Models umgehend zu visualisieren.

3.5. Visualisierung der Individuals

- Datenhaltung

  In der *Individuals-Übersicht* sollen zeilenweise die *TrefferInstanzen* der *Instanzen-Verwaltung* angezeigt werden. Dabei soll es insbesondere möglich sein, auch *multiple-Properties* anzuzeigen und direkt per Mausklick *Object-Properties* in einem neuen Fenster zu öffnen.
  Um dies zu gewährleisten und die Daten überhaupt so flexibel wie möglich zu halten, werden alle Daten im *TableData* direkt vom Typ *java.lang.Object* gespeichert.
  Im Anwendungsfall heißt das, dass beispielsweise *multiple-Properties* direkt als *java.util.LinkedList* und *Object-Properties* direkt als *Instance* aus dem Paket *edu.stanford.smi.protege.model* gespeichert werden.

- Visualisierung

  Die Visualisierung der einzelnen Elemente des 2-dimensionalen *Object*-Array soll Klassenabhängig erfolgen. Das heißt, dass für die *Individuals-Übersicht* (JTable) die *DefaultRenderer* und *DefaultEditor* überschrieben werden sollen. Diese beiden Klassen kommen immer dann zum Einsatz, wenn in einer JTable die einzelnen Elemente gezeichnet oder editiert werden sollen. Insbesondere für Objekte vom Typ *LinkedList* sollen Objekte vom Typ *JComboBox* zur Visualisierung mehrerer Elemente in einem Tabellenelement verwendet werden. M*ultiple-Properties* und *Object-Properties* sollen durch kleine Icons grafisch von den normalen Werten in der Tabelle abgesetzt werden.

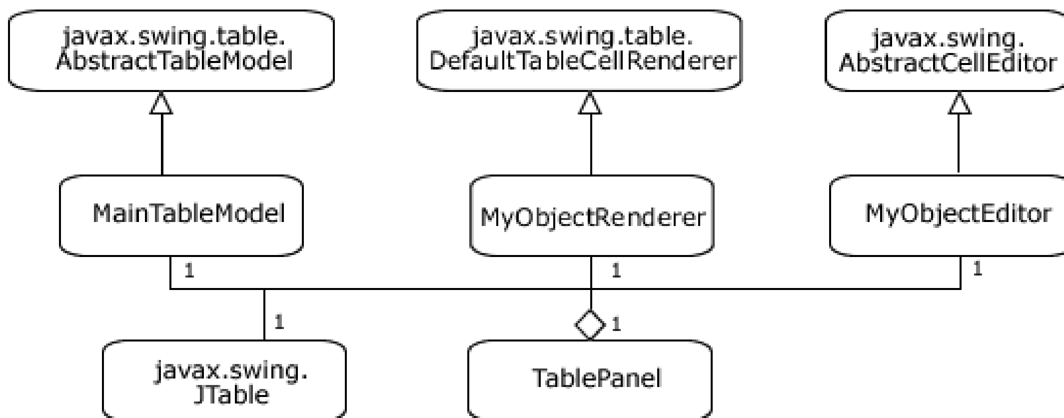


Bild5: vereinfachtes UML-Klassendiagramm für das TablePanel

3.6. Sortierung

Um später in der *Individuals-Übersicht* eine Zeile markieren zu können, welche dann im *Editieren-Panel* angezeigt werden soll, ist es nötig, das Model zu sortieren, weil die *Individuals-Übersicht* auch immer die gleiche Reihenfolge der *Individuals* anzeigt, wie sie in der *InstanzenVerwaltung* vorliegen. Das Sortieren soll in etwa so funktionieren:

Bemerkung:

Eine Sortierung ist in dem Sinne eine Funktionalität, die von der *Instanzen-Verwaltung* durchgeführt werden soll, aber natürlich von außen von jedem beliebigen Objekt aufgerufen werden kann, in dem Falle der Tabelle selbst.

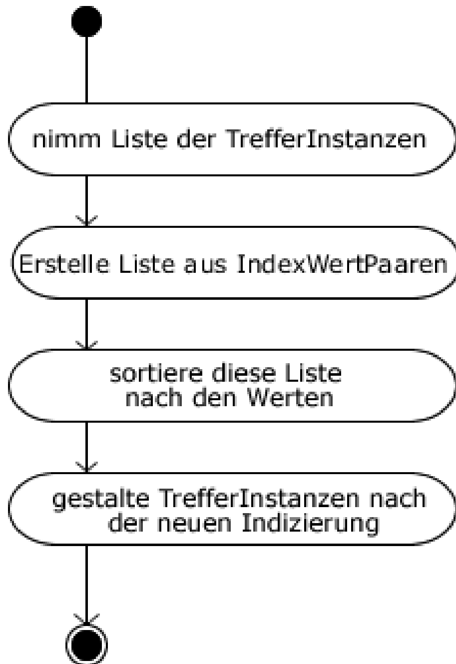


Bild4: Aktivitätendiagramm der Sortierung

Beschreibung:

- ein Objekt vom Typ *IndexWertPaar* enthält lediglich einen *int*-Wert und einen *String*-Wert
- um einfacher und schneller sortieren zu können, wird eine neue Liste solche *IndexWertPaare* erstellt
- diese Liste enthält von 0 beginnend durchnummerierte *Index*-Werte und als String-Wert die jeweiligen *Property*-Werte aller *Treffer-Individuals* der aktiven Klasse
- diese *IndexWertPaar*-Liste wird nun dem Wert nach sortiert
- am Ende werden die nun (evtl.) nicht mehr aufsteigend nummerierten Inizes benutzt, um die *TrefferInstanzen*-Liste der *InstanzenVerwaltung* neu zu ordnen

3.7. Anpassen der Visualisierung (Spaltenfilter)

zur GUI:

Eine SpaltenFilter-Funktion soll dazu dienen, Spalten bzw. die in diesen Spalten der *IndividualsÜbersicht* angezeigten Propierties ein- und auszublenden, um mehr Übersicht zu schaffen, wenn man sich nur für bestimmte Properties interessiert. Für diesen Zweck soll wieder ein *Tab* im *TabMenu* eingeführt werden, welches dem Benutzer verschiedene Einstellungen bereitstellt, um eine Anzeigeauswahl bestimmter Spalten zu ermöglichen.
Es soll eine Tabelle angezeigt werden, welche alle Properties der *aktiven Klasse* anzeigt und ihrer derzeitigen Anzeige gegenüberstellt, nämlich *sichtbar* oder *nicht sichtbar*. Der Benutzer soll dann mit Checkboxen alle die Properties markieren, welche er in der *IndividualsÜbersicht* angezeigt bekommen will.
Zusätzlich sollen kleine Hilfsbutton zur Verfügung stehen, welche dem Benutzer das evtl. aufwendige Markieren abnehmen soll, indem automatisiert ein bestimmter Bereich markiert, abgewählt oder invertiert wird.

zum Model:

In der *InstanzenVerwaltung* soll eine Liste gehalten werden, in welcher die sichtbaren *Properties* gespeichert sind. Bei der Generierung der *TableData* für die *Individuals Übersicht* sollen dann von der Menge aller Properties nur diese verarbeitet werden, die auch in der Liste der sichtbaren Slots bzw. Properties stehen.
Insofern besteht die Aufgabe des *SpaltenFilterPanels* darin, eine solche Liste anhand der Benutzereingaben zu erstellen und diese einfach an die *InstanzenVerwaltung* zu übergeben.

3.8. Editieren von Individuals

Es soll auch die Möglichkeit bestehen, einzelne Individuals der *aktiven Klasse* editieren zu können. Das soll so geschehen, dass man durch Markieren eines einzelnen Individuals mit Hilfe des **edu.stanford.smi.protege.ui.InstanceDisplay** eine umfassende Übersicht des Individuals im *EditierenTab* bekommt, welches von Haus aus eine Editierfunktion zur Verfügung stellt, welche in Echtzeit die *KnowledgeBase* verändert.
Für diese gesamte Funktionalität wird nicht mehr nötig sein, als ein Objekt vom oben genannten *InstanceDisplay* anzulegen und eine statische Methode zu implementieren, welche diesem *InstanceDisplay* das anzuzeigende Individual übergibt.

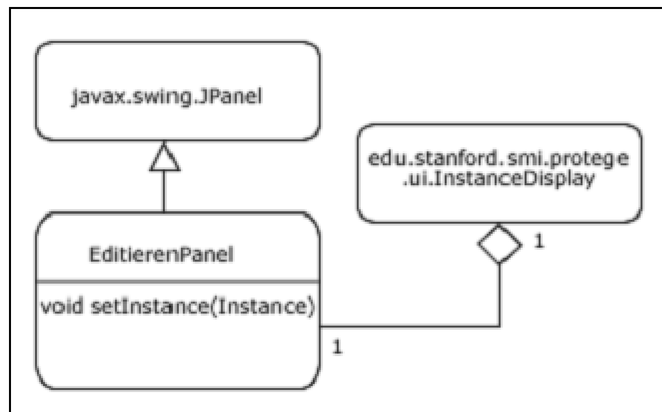


Bild6: UML-Klassendiagramm des SortierenPanels

3.9. Suchen

Die Idee ist, dass man eine Suchanforderung als *aussagenlogische Formel* interpretieren kann. Mit dieser soll eine Art Beschränkung beschrieben werden, welcher ein *Individual* standhalten muss, um der Suchanforderung zu genügen. Beispielsweise kann man bestimmte *Properties* mit bestimmten Werten fordern. Man kann dann entscheiden, ob das Ergebnis für diese Suchanforderung als *wahr* oder *falsch* einzustufen ist, wenn nämlich das *Individual* dieses Property besitzt und dieses mit dem gesuchten Wert belegt ist, oder auch nicht.
Um die gesamte Mächtigkeit der *aussagenlogische Formeln* auch nutzen zu können, soll die Suchfunktion stets Suchanforderungen in *disjunktiver Normalform (DNF)* stellen, in welche bekanntlich jeder *boolesche Formel* überführt werden kann. Dabei ist es auch nützlich, dass man diese *DNF* ohnehin intuitiv benutzt.

zum Model:

Eine *DNF* besteht immer aus UND-Blöcken, welche durch ein ODER verknüpft sind. Diese UND-Blöcke wiederum sind auch Formeln, welche durch ein UND verknüpft sind. Die kleinste Einheit bilden Formeln der Bauart: *Variable* | *Relation* | *Konstante*.

Diese Überlegungen führen zu folgendem Diagramm:

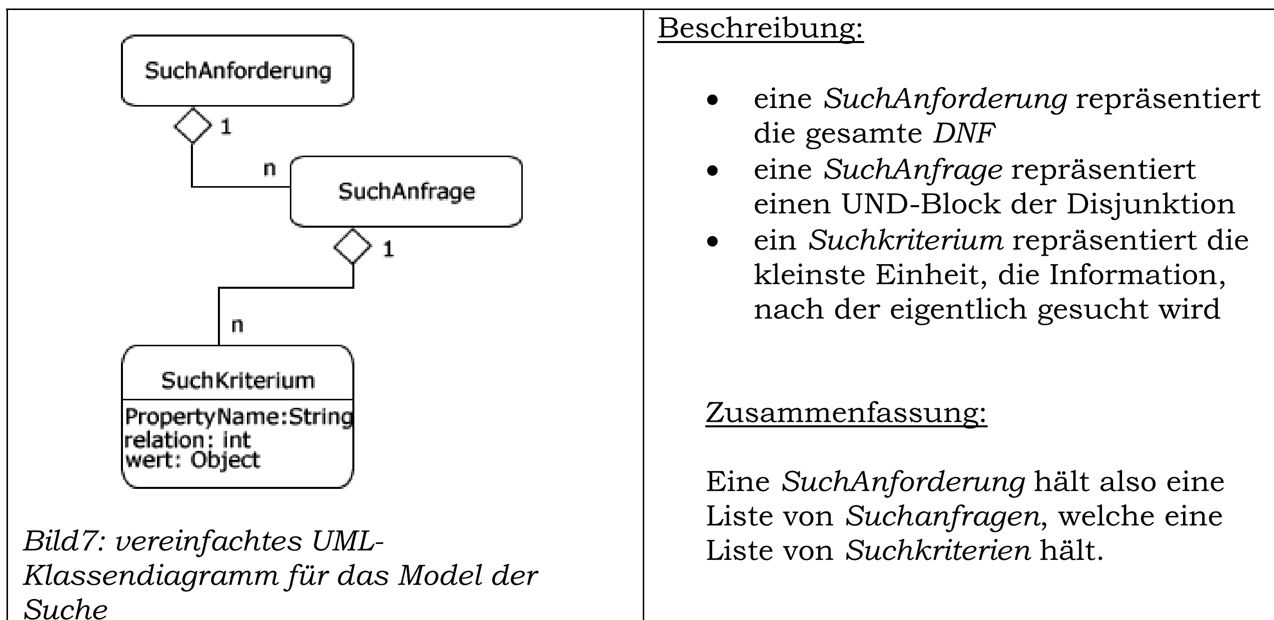


Bild7: vereinfachtes UML-Klassendiagramm für das Model der Suche

Beschreibung:

- eine *SuchAnforderung* repräsentiert die gesamte *DNF*
- eine *SuchAnfrage* repräsentiert einen UND-Block der Disjunktion
- ein *Suchkriterium* repräsentiert die kleinste Einheit, die Information, nach der eigentlich gesucht wird

Zusammenfassung:

Eine *SuchAnforderung* hält also eine Liste von *Suchanfragen*, welche eine Liste von *Suchkriterien* hält.

Das logische Zusammenspiel ergibt sich natürlich erst in der Behandlung der Daten, so dass anhand der logischen Verknüpfung der *wahr/ falsch-Ergebisse* einzelner *SuchKriterien* auf einzelne *Individuals* das Gesamtergebnis des jeweiligen *Individuals* und somit der gesamten *SuchAnforderung* ermittelt wird.

Dieses Zusammenspiel muss eine *SearchEngine* übernehmen, die mit Hilfe einer gegebenen *SuchAnforderung* aus einer Menge von *Individuals* alle diejenigen raussucht, die der Beschränkung genügen.

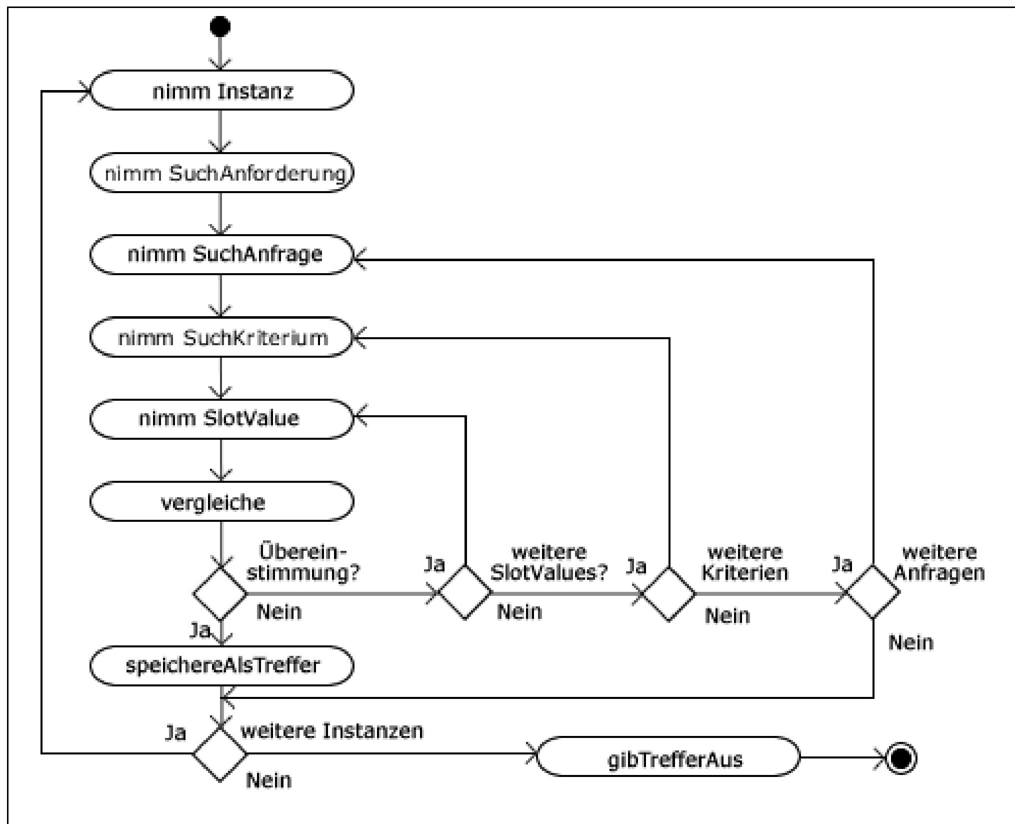


Bild8: Aktivitätendiagramm des Suchens (von Treffern)

Die Suche in einer Menge von *Individuals* wird demnach von der *InstanzenVerwaltung* direkt angestoßen, da sie auch stets die betreffende Menge an *Individuals* gespeichert hat. Dazu wird sie der *SearchEngine* die *TrefferInstanzen* (als Menge der *Individuals*) und die derzeitige *SuchAnforderung* übergeben (Bemerkung: in der *Instanzen-Verwaltung* existiert stets eine *SuchAnforderung*, diese kann auch leer sein).
Die Suche soll sowohl für Strings als auch für Zahlen funktionieren. Das bedeutet, dass für diese beiden Datentypen verschieden Relationen zur Verfügung gestellt werden und das die die Suche selbst über Parameter gesteuert werden soll, an welchen zu bestimmten Zeitpunkten erkennbar ist, ob es sich um die Suche von Zahlen oder Zeichenketten handelt, also wie der *Object*-Wert interpretiert werden muss.

zur GUI:

Auch für die Suche soll wieder ein *Tab* im *TabMenu* angelegt werden. Entsprechend des Aufbaus der *DNF* muss es nun Möglichkeiten geben, die *Minterme* und *Maxterme* zu formulieren. Dafür sollen eine Liste (**javax.swing.JList**) für die *SuchAnfragen* und eine Tabelle (**javax.swing.JTable**) für die *Suchkriterien* bereitgestellt werden.
Im Detail soll der Benutzer die einzelnen Bestandteile der *DNF* mit entsprechenden Buttons problemlos *zusammenklicken* können. Bei der genauen Spezifizierung eines *SuchKriteriums* sollen dem Benutzer durch kleine Auswahlfenster vordefinierte Werte (Property, Relation, Wert) zur Auswahl gestellt werden. Schließlich soll die fertig erstellte *SuchAnforderung* durch Klick auf den *Übernehmen*-Button an die *InstanzenVerwaltung* übergeben werden.

3.10. Export/Import

3.10.1 Übersicht - Exportierfunktion

Beim Exportieren unterscheiden wir grundsätzlich zwischen den zwei Exportformaten „Character Seperated Value“ (*.csv) und das „Ontology Web Language“ (*.owl) Format, welche über einen Save-Dialog ausgewählt werden können. Der Default-Wert ist *.owl.

- Erstes resultiert einer simplen Text Datei die tabellarisch strukturierte Daten enthält. In der ersten Zeile stehen die Spaltennamen, welche den Properties der exportierten Individuals entspricht. Die weiteren Zeilen enthalten die entsprechenden Werte der einzelnen Individuals zu den oben deklarierten Properties getrennt durch “;“.

- Der owl-Export extrahiert alle Individuals der aktiven Klasse sowie alle referenzierte Individuals einschließlich der entsprechenden Klassen- und Propertydefinitionen aus der Knowledgebase und schreibt sie in die Zieldatei.
  Dieser recht umfangreiche Export hat den Vorteil, dass im Gegensatz zum csv-Format viel mehr owl-spezifische Informationen erhalten bleiben.
  Diese extrahierte „Teilontologie“ kann nun zum einen mittels *Build New Project/Owl-Files* als eigenständige Ontologie in Protégé verwendet werden, sowie zu einer unvollständigen Ontologie hinzugefügt werden.
  Weiterhin ist anzumerken, dass die entstandene owl-Datei ein wohlgeformtes xml-Dokument ist und somit von anderen Anwendungen weiterverwendet werden kann.

3.10.2 Übersicht - Importierfunktion

Beim Importieren (aus einer Owl-Datei) werden der aktiven Klasse alle Individuals mit den entsprechenden Property Werten, sowie alle referenzierten Individuals zu deren Klassen aus der Quelldatei hinzugefügt.

Kommt es dabei zu Integritätskonflikten, wird dem Anwender die Verantwortung für die Erhaltung der Datenintegrität übertragen, da die Applikation keine Entscheidung über die „Richtigkeit“ bestimmter Werte treffen kann.

- Zu Beginn jedes Importiervorgangs muss der Anwender entscheiden, ob er überhaupt bereits existierende Individuals verändern will.
- Entscheidet er sich für die Möglichkeit des Veränderns, muss er entscheiden, ob er alle Property Werte der Individuals komplett mit den neuen Werten überschreiben möchte.
- Entscheidet er sich dagegen, also möchte der Anwender manche Property Werte beibehalten und manche übernehmen, so hat der Anwender in einem neuen Dialogfenster für jede Property des Individuals zu entscheiden, welcher Wert in die Knowledge Base übernommen werden soll.

Am Ende des Importiervorgangs muss der Anwender über entstandene Probleme (Klasse eines zu importierenden referenzieten Individuals existiert nicht, ein Individuals mit dem Namen gibt es bereits aber es gehört einer anderen Klasse an,...) und die Reaktion darauf in Kenntnis gesetzt werden.
Dies soll in einem „Log-Fenster“, geschehen in dem alle Schritte des Imports mitprotokolliert werden (analog beim Export).

3.10.3 GUI:

Für das Importieren und Exportieren werden Tab-Menü optisch identische Tabs erstellt.
Über ein *JTextfield* oder einen *JFileChooser* wird der Ziel- bzw. Quellpfad ausgewählt, und mittels eines Übernehmen Buttons wird der Import bzw. Export gestartet. Dabei ist zu beachten, dass das Standard Export Format Owl ist, möchte man also in eine csv-Datei exportieren muss im *JTextfield* explizit die Dateiendung *.csv* angegeben, oder im *JFileChooser* der Dateityp auf **.csv* gestellt werden.
Für das „Log-Fenster“, in welchem die abgearbeiteten Schritte protokolliert sowie Fehlermeldungen und Warnungen dargestellt werden, ist eine *JTextPane* vorgesehen, die bei Bedarf ein- oder ausgeblendet werden kann.

3.10.4. Model - Exportieren

Das folgende Aktivitätsdiagramm soll die prinzipielle Vorgehensweise beim OWL-Export erläutern:

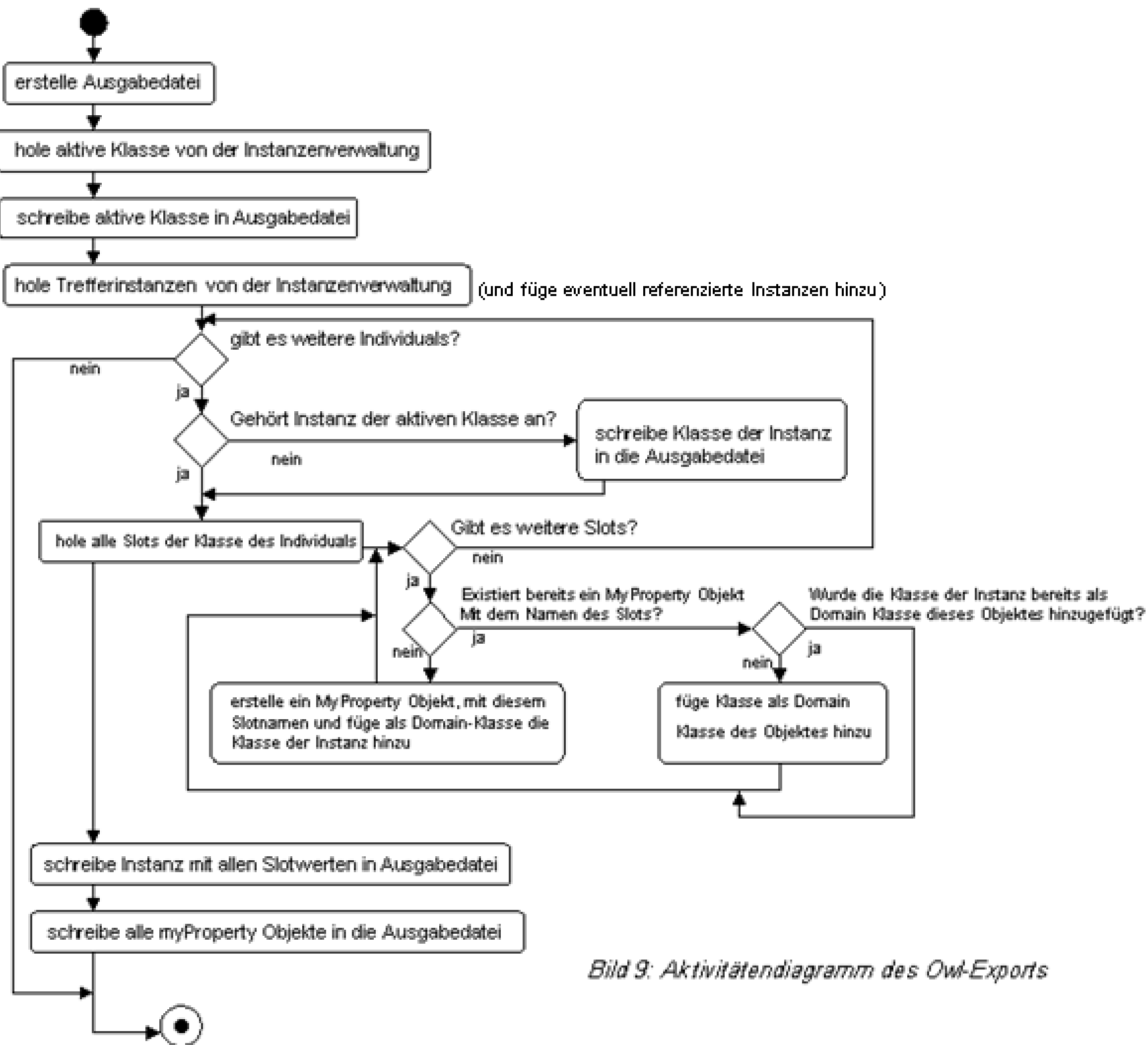


Bild 9: Aktivitätendiagramm des Owl-Exports

Die Vorgehensweise beim CSV Export ist ähnlich, aber sehr viel einfacher. Dabei wird nur die aktive Klasse sowie eine Liste der Trefferinstanzen von der Instanzen-verwaltung bezogen, und letztere tabellarisch strukturiert zeilenweise in eine Textdatei geschrieben.

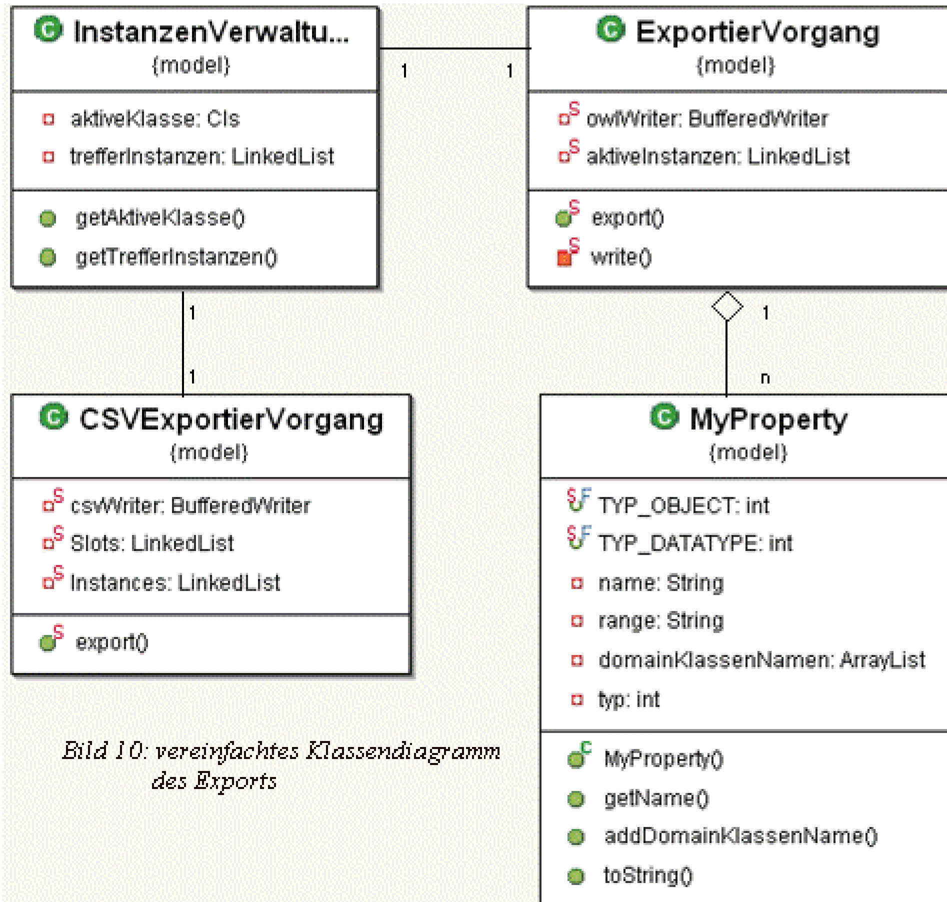


Bild 10: vereinfachtes Klassendiagramm des Exports

- *ExportierVorgang:*
  → Hierbei handelt es sich um eine Factory-Klasse. Sie hat eine statische öffentliche Methode *exportieren(…)*, welche die aktive Klasse und die Trefferinstanzen aus der Instanzenverwaltung abruft, und diese in eine Owl-Datei exportiert.
  Dazu werden mehrere Objekte der Klasse *MyProperty* erstellt.
- *MyProperty:*
  → Ein Objekt dieser Klasse repräsentiert eine Definition einer owl-Property. Es wird ausschließlich dazu benötigt, beim Schreiben der owl-Datei während eines Exportiervorgangs zu einer von der Knowledge Base erhaltenen Property alle Domain-Klassen zu sammeln, um nach dem Einlesen und Schreiben aller Individuals die Property korrekt mit allen notwendigen Informationen zu speichern.
- *CSVExportierVorgang:*
  → Diese Klasse besitzt analog zur Klasse *Exportiervorgang* ebenfalls eine öffentliche statische Methode *exportieren*(…), die auf eine ähnliche aber einfachere Art und Weise die Trefferinstanzen in eine csv-Datei exportiert.

3.10.5 Model - Importieren

Das folgende Aktivitätsdiagramm soll die prinzipielle Vorgehensweise beim Import erläutern:

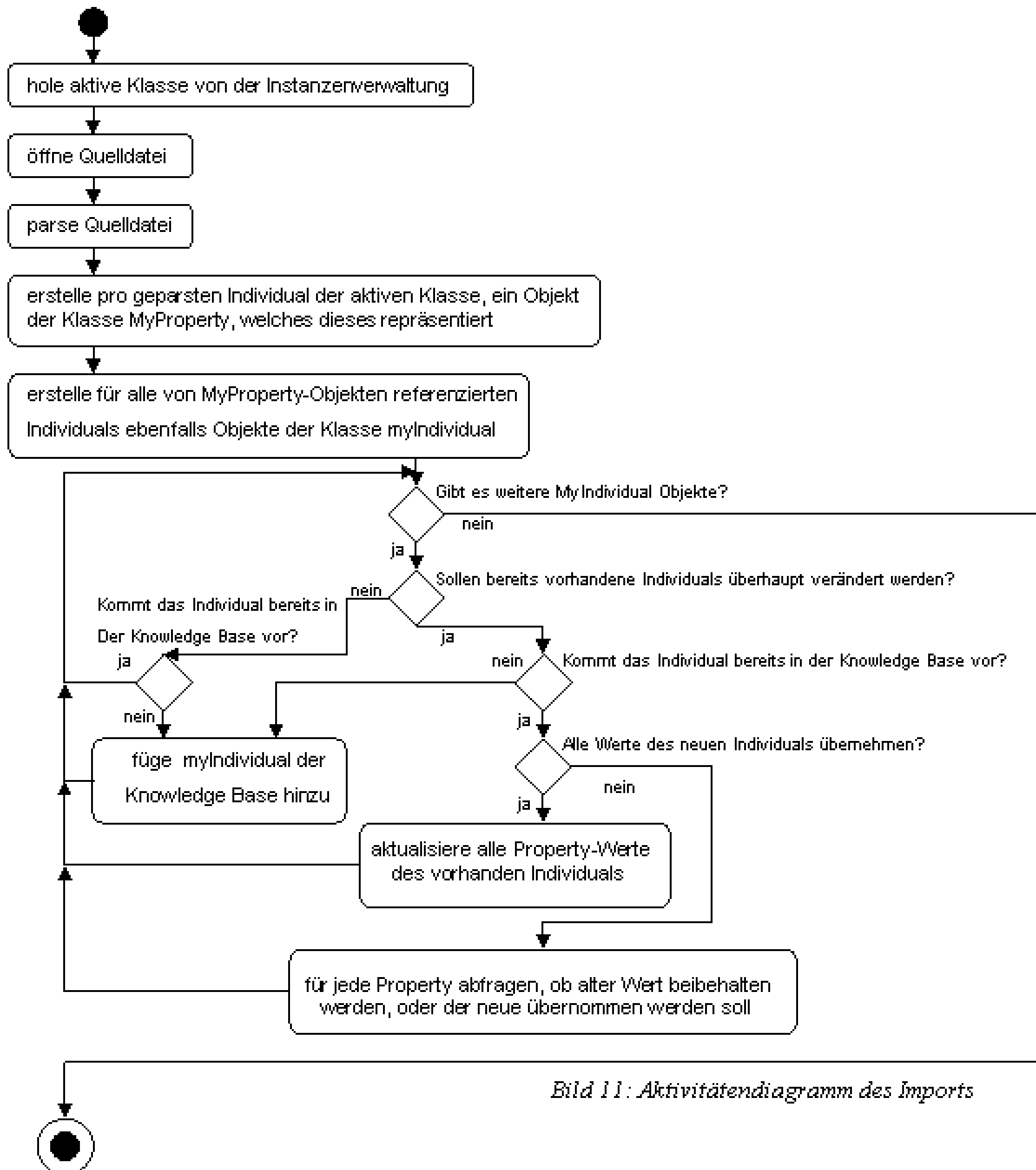


Bild 11: Aktivitätendiagramm des Imports

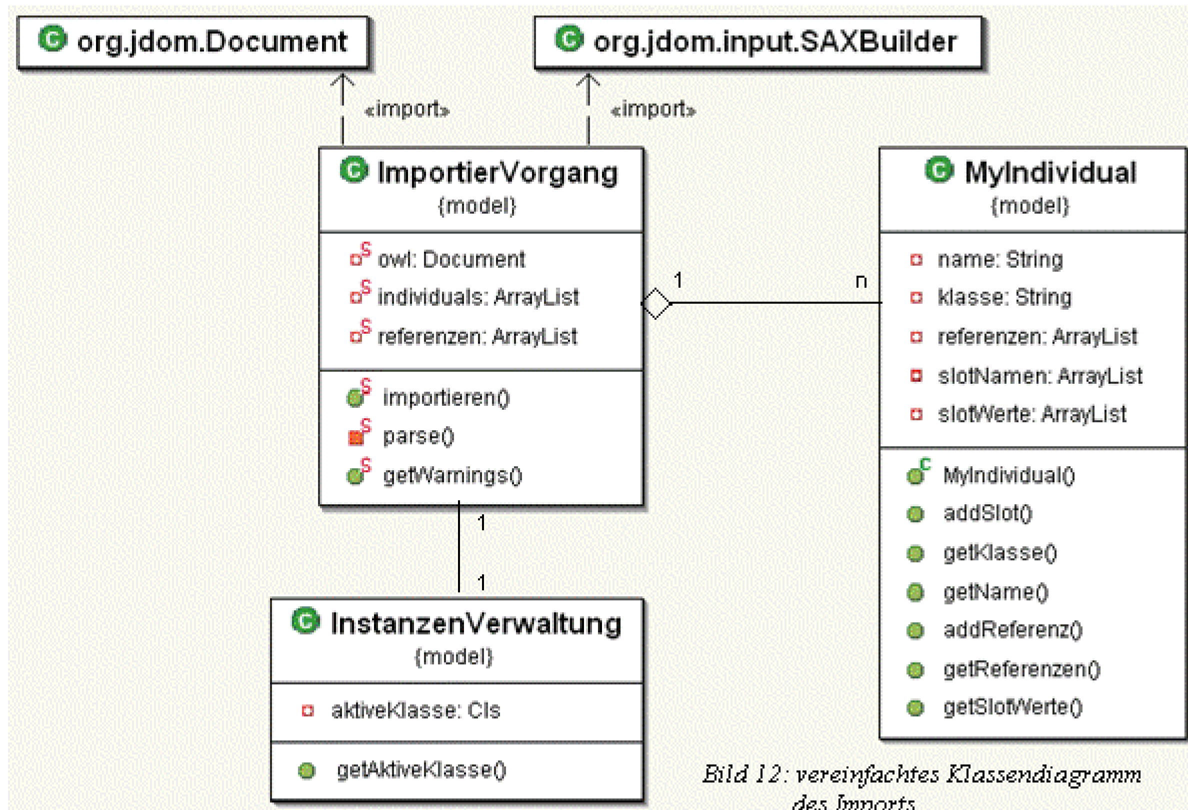


Bild 12: vereinfachtes Klassendiagramm des Imports

- *ImportierVorgang:*
  → Über die öffentliche statische Methode im*portieren(...)*, wird das Hinzufügen von Individuals zur aktiven Klasse angestoßen. Um die Quelldatei zu parsen, verwenden wir JDOM, es ist also notwendig die frei verfügbare Bibliothek *jdom.jar* zum classpath hinzuzufügen um unser Plugin ausführen zu können.
  Pro einzufügendem Individual wird dabei ein Objekt der Klasse *MyIndividual* erstellt.

- *MyIndividual:*
  → Ein Objekt dieser Klasse ist speichert die geparsten Werte eines Individuals zum tatsächlichen Import.

Anmerkung:
- die JDOM *Release Builds* sind unter http://www.jdom.org/dist/binary/ verfüg-bar
- die jdom.jar befindet sich im Ordner *jdom-1.0.zip/jdom-1.0.zip/jdom-1.0/build*

4. Grundsätzliche Struktur- und Entwurfsprinzipien der einzelnen Pakete

Das Plugin liegt in Form einer JAR-Datei vor, welche in das *plugins*-Verzeichnis des Protégé-Installationspfades in ein Extra-Verzeichnis *Plugin* (Namensänderung mögliche) kopiert werden muss. Die JAR-Datei untergliedert sich in bestimmte Unterverzeichnisse, welche die Struktur des Paketes beschreiben. Diese Pakete, in welche im Laufe der Erweiterung des Plugins noch weitere Klassen eingefügt werden, sollen nun im Einzelnen beschrieben werden:

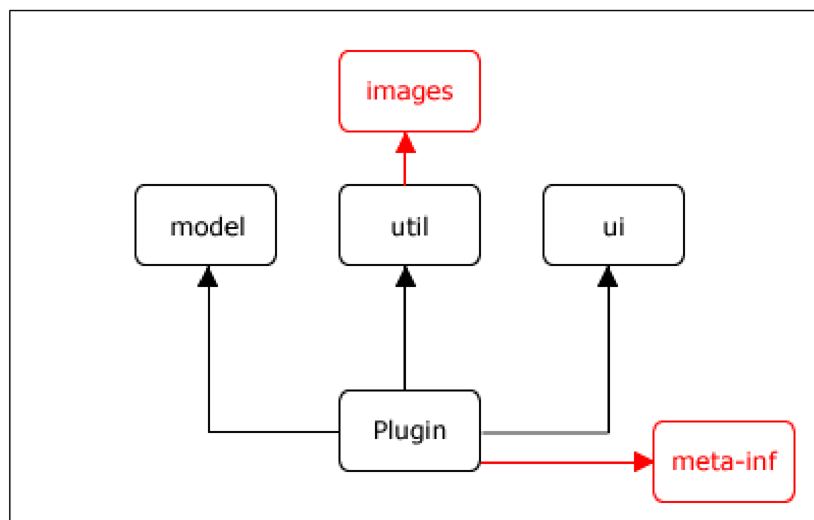


Bild9: Paket-Übersicht (Plugin.jar)

Das Bild zeigt das Paket *Plugin*, mit den Unterpaketen *model, util* und *ui*. Und ein Verzeichnis *images*, welches nicht als Paket zu verstehen ist, sondern einfach nur ein Sammelverzeichnis für alle Bilder bzw. Icons, die im Plugin Verwendung finden sollen und auch Bestandteil der JAR-Datei sein müssen.

Gemäß der Verwendung der einzelnen Klassen, sind diese entsprechend sinnvoll in die oben gezeigte *Paketstruktur* unterteilt, wobei auf größtmögliche Abstraktion geachtet werden soll, um beispielsweise *GUI (=UI… "UserInterface")* vom *Model* zu trennen. Allgemeine Hilfsklassen, welche Methoden enthalten, die an vielen anderen Stellen im Codes verwendet werden müssen und evtl. recht umfangreich sind, werden zwecks Vereinfachung und Minimierung des Codes in mindestens einer zentralen Hilfsklasse implementiert, welche sich dann Paket *util* befinden soll.

Bis zum jetzigen Zeitpunkt sind oben gezeigte Pakte mit folgenden Dateien besetzt:

Paket	Dateien
plugin	• Plugin.class
model	• InstanzenVerwaltung.class • TableData.class • SearchEngine.class • SuchAnforderung.class • SuchAnfrage.class • SuchKriterium.class • MyFileFilter.class • MyIndividual.class • MyProperty.class • ExportierVorgang.class • CSVExportierVorgang.class • ImportierVorgang.class • AbruchException.class
util	• MeineUtilities.class • ImageFactory.class
ui	• ClassBrowserPanel.class • EditierenPanel.class • ExportierenPanel.class • ImportierenPanel.class • SlotPanel.class • MyDialog.class • SpaltenFilterPanel.class • SuchenPanel.class • TablePanel.class • TabPanel.class

Für genauere Informationen, wofür die einzelnen Klassen stehen und in welchem Zusammenhang sie verwendet werden, wird auf die (bald verfügbare) *javadoc*-Dokumentation verwiesen.

In der *Plugin.jar* befindet sich außerdem noch ein Verzeichnis *meta-inf*, welches dazu dient, dem Hauptprogramm Protégé mitzuteilen, welche *class*-Datei die Hauptdatei des Plugins ist und dass es sich um ein spezielles *Tab-Widget* von Protégé handelt. Diese Informationen sind für das Laden des Plugins wichtig, und müssen deshalb auch Bestandteil der JAR-Datei sein. In der Übersicht (siehe: Bild1) ist dieses Verzeichnis rot markiert, um zu verdeutlichen, dass es sich nicht um ein Paket mit Klassendateien handelt.