Gruppe: GR-4
Mitglieder: Markus Jäger, Patrick Oesterling, Lars Kolb, Sebastian Eichelbaum, Stefan Vollrath, Bei Fang, Anne Nitzsche
Datum: 15.04.2005

1. Begriffe

- Ontologie

a) Def.:

In der Informatik versteht man unter Ontologie ein formal definiertes System von Dingen und Relationen zwischen diesen Dingen.

b) Ziel:

Das Hauptziel von Ontologien ist die Klärung von Vorstellungen um innerhalb einer Fachdomäne gemeinsam kommunizieren zu können. Da für den Wissensaustausch in größeren Gruppen die menschliche Sprache häufig zu Missverständnissen führt, ist man stets bestrebt Definitionen und Normierungen einzuführen.
Deshalb tragen Ontologien dazu bei, Prozesse der Wissensverarbeitung zu automatisieren.

c) Anwendungsbereiche für Ontologien:

Anwendungsbereiche für Ontologien sind Dokumentation, Datenbanken und Nachschlagewerke.

Dieser Begriff wurde in das Glossar aufgenommen, das die Applikation Protégé, welche unserem Projekt zugrunde liegt, ein Editor für Ontologien ist. Des Weiteren ist dieser zentrale Begriff der Semantic Web Bewegung nicht allgemein geläufig.

- Semantic Web

Die Idee des Semantic Web ist es, de Zugriff auf Informationen zu automatisieren. Da es Menschen nur möglich ist, einen kleinen Teil der im World Wide Web verfügbaren Informationen zu erlangen, ist man i.A. auf die Hilfe von Maschinen zum Extrahieren und Analysieren verwertbarer Informationen angewiesen. Derzeit liegen jedoch die meisten Informationen in einer den Maschinen unverständlichen Form vor. Das Semantic Web soll dieses Problem lösen indem, es Daten hinzufügt, um die Semantik der Inhalte formal festzulegen.

Ein Beispiel:

Die URL einer Webseite besagt, dass sie sich mit "Fußball" beschäftigt; aus der zugrunde liegenden Ontologie geht hervor, dass der Begriff "Fußball" eine "Sportart" darstellt; dementsprechend geht es auf der Website also auch um Sport (obwohl das nicht explizit direkt in den Metadaten auftaucht). Bei entsprechender Qualität der Annotation lässt sich ein hoher Grad automatischer Verarbeitung erreichen

Dieser Begriff wurde in das Glossar aufgenommen, da unser Auftragsgeber, die Firma "SoftConsult", Erfahrungen mit den Konzepten des Semantischen Webs machen möchte um die Analyseprozesse ihrer betrieblichen Tätigkeit zu erleichtern.
Um für das dazu verwendete Werkzeug, Protégé, ein Plug-In zu entwickeln ist es in der Analysephase des Entwicklungsprozesses notwendig das zu Grunde liegende Konzept zu kennen.

- Plug-In-Konzept:

Ein Plug-In ist ein Software-Zusatzmodul, welches in ein bestehendes Softwareprodukt integriert wird und dessen Funktionalität erweitert.
Softwarehersteller definieren Schnittstellen zu ihren Produkten mit deren Hilfe Erweiterungen – Plug-Ins – für diese Softwareprodukte programmiert werden können. Ein Plug-In deckt dabei eine fachliche Funktion so vollständig ab, dass keine andere Komponente des Systems etwas Spezifisches darüber wissen muss.

Ein Plug-In erweitert also nicht nur das Gesamtsystem an sich, sondern kann auch über die Schnittstellen das Verhalten oder die Funktion von anderen Plug-Ins erweitern und verändern. Damit werden Plug-Ins immer von der Komponente abhängig sein, die sie erweitern. Um eine Implementierung von Plug-Ins zu ermöglichen, müssen die Schnittstellen der entsprechenden Komponenten freigelegt und beschrieben sein.

Da wir beauftragt sind ein Plug-In zu entwickeln, ist es unbedingt notwendig zu wissen, was sich hinter diesem Begriff verbirgt. Des Weiteren ist das Plug-In Konzept ein fundamentales Prinzip von Protégé.

- JAR-Files

JAR-Dateien sind Archivdateien, die mehrere Klassen enthalten. Sie eignen sich dazu, ganze Programme zu einer Datei zusammenzufassen.
Darüber hinaus enthalten sie eine einfache Textdatei, „manifest.mf“, welche sich stets in einem Ordner „meta-inf“ befindet und weitergehende Informationen über das Archiv enthält wie z.B. Version, Main-Class oder Classpath.

Da durch einfaches Hinzufügen eines JAR files in den "plugins" Ordner ein Plug-In in Protégé eingebunden werden kann, ist es notwendig zu wissen was ein JAR-File ist und wie man ihn erstellt (dazu mehr in Plug-In Konzept von Protégé).

→ Begriffswelt von Protégé

- Class / Object / Instance:

Eine Klasse beschreibt die Eigenschaften von gleichartigen Objekten.
Ein Objekt ist eine Instanz, also eine konkrete Ausprägung einer Klasse.

- Slot / Facet / Constraint:

Slots beschreiben Eigenschaften von Objekten einer Klasse.
Ein Facet beschreibt wiederum die Eigenschaften eines Slots, er definiert Constraints (dt. Beschränkungen) bezüglich der Gültigkeit von Werten eines Slots.

- Form:

Formulare, durch welche die Instanzen optisch repräsentiert werden.

Hinter diesen sieben Begriffen verbergen sich die Hauptbestandteile eines Protégé Projektes. Da die einzelnen Mitglieder eines Teams in ihren unterschiedlichen Rollen gezwungen sind, ständig miteinander zu kommunizieren, muss Jedes mit diesen Grundlegenden Begriffen vertraut sein.

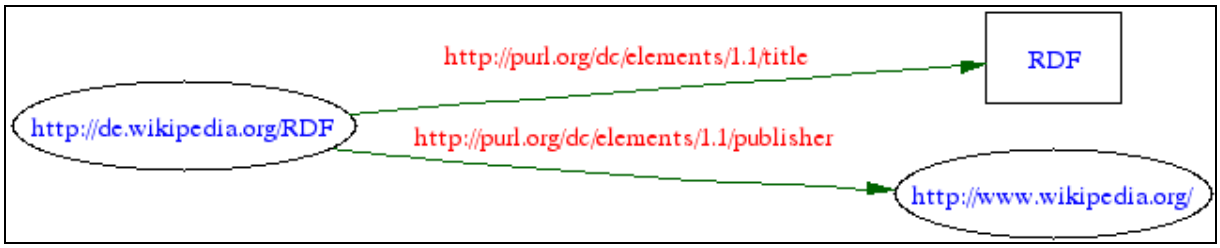
→ **Begriffswelt von RDF**

Das Resource Description Framework (RDF) ist eine Spezifikation für ein Modell zur Repräsentation von Metadaten (Informationen über Webseiten und andere Objekte. Am weitesten verbreitet ist die Serialisierung von RDF in XML. In Verbindung mit RDF-Schema und der Web Ontology Language (OWL) dient RDF als grundlegendes Format zur Repräsentation von Ontologien.

Es kann zwischen dem RDF-Modell (RDF-Graph) und der RDF-Syntax (Serialisierung) unterschieden werden. Weiter gibt es die RDF-Schema-Spezifikation, mit der eine Typisierung und Vererbung von Ressourcen und Eigenschaften definiert und die Gültigkeit von Ressourcen in Bezug auf Eigenschaften festgelegt werden kann.

1. RDF-Modell

Informationen sind in RDF in so genannten Statements abgelegt, das sind Aussagen in Form eines Tripels aus Subjekt, Prädikat und Objekt. Alle drei Bestandteile werden durch einen Uniform Resource Identifier (URI) identifiziert und damit allgemein als Ressource (im englischen Resource) bezeichnet. Als Objekt einer Aussage sind auch Zeichenketten (Literal) möglich. Die Verknüpfung mehrerer Tripel lässt sich als ein gerichteter Graph mit Knoten- und Kantenbeschriftung verstehen, wobei Subjekt und Objekt eines Tripels Knoten und die Prädikate Kanten sind.
Folgende Grafik wurde mit dem W3C RDF Validator aus dem weiter unten angegebenen Beispiel erzeugt. Dargestellt sind zwei Tripel mit dem selben Subjekt "http://de.wikipedia.org/RDF".



Die Besonderheit des RDF-Modells liegt zum einen darin, dass über die als Prädikat verwendeten Ressourcen *(*Properties) auch wiederum Aussagen getroffen werden können. Dadurch lassen sich Properties selbst mit RDF beschreiben und als Metadatenformat ablegen.
Zum anderen bilden in RDF Statements selber Ressourcen, auf die mit weiteren Statements verwiesen werden kann. Diese Technik der Aussagen über Aussagen wird als Reification bezeichnet.
Zusätzlich enthält RDF vordefinierte Datentypen für Listen und Mengen, um Gruppen von Ressourcen zusammenzufassen. Ressourcen, die keine explizite URI haben, sondern nur zur Gruppierung von anderen Objekten dienen, werden in der Regel durch so genannte "blank nodes" modelliert. Ein Beispiel dafür ist die Zuweisung eines Namens, der aus separaten Zeichenketten für Vor- und Nachnamen besteht.

2. RDF-Syntax

Das RDF-Modell (der RDF-Graph) ist unabhängig von einer speziellen Darstellungsform. Am meisten verbreitet ist die Repräsentation in XML. Für die Speicherung von RDF in Datenbanken und Datenstrukturen gibt es verschiedene Konzepte, da ein reines Ablegen der N-Tripel in einer Tabelle nicht sehr effizient ist.

Da sich die selben RDF-Aussagen in einer Syntax mitunter auf viele verschiedene Arten ausdrücken lassen, ist es sinnvoll zur Verarbeitung von RDF-Daten einen RDF-Parser zu verwenden, der auch die Validierung gegen ein RDF-Schema vornehmen kann.

Ein Beispiel:

Die Aussage "http://de.wikipedia.org/RDF hat den Titel 'RDF' und den Herausgeber http://www.wikipedia.org/" wird in RDF mit zwei Tripeln ausgedrückt.

In RDF/XML lässt sich die Aussage so ausdrücken:

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:dc="http://purl.org/dc/elements/1.1/">
  <rdf:Description rdf:about="http://de.wikipedia.org/RDF">
    <dc:title>RDF</dc:title>
    <dc:publisher rdf:resource="http://www.wikipedia.org/"/>
  </rdf:Description>
</rdf:RDF>
```

→ **Begriffswelt von OWL**

Die Web Ontology Language (kurz OWL) dient zur Erstellung, Verteilung und Publizierung von Ontologien anhand einer formalen Beschreibungssprache. Es geht darum, Terme einer Domäne und deren Beziehungen formal so zu beschreiben, dass auch Software die Bedeutung verarbeiten ("verstehen") kann. OWL ist somit ein wesentlicher Bestandteil der Semantic Web-Initiative. OWL basiert technisch auf der RDF-Syntax und historisch auf DAML+OIL, und geht dabei über die Ausdrucksmächtigkeit von RDF-Schema weit hinaus. Zusätzlich zu RDF werden weitere Sprachkonstrukte eingeführt, die es erlauben, Ausdrücke ähnlich der Prädikatenlogik zu formulieren.

1. Sprachkonstrukte:

OWL unterscheidet Klassen, Eigenschaften (properties) und Instanzen. Klassen stehen für Konzepte. Sie können Eigenschaften besitzen. Instanzen sind Individuen einer oder mehrerer Klassen.

Klassen betreffend:

<owl:class>
<owl:oneOf>
<owl:unionOf>
<owl:intersectionOf>
...

Properties betreffend:

<owl:Restriction>
<owl:allValuesFrom>
<owl:someValuesFrom>
...

2. Namensräume:

Bevor wir eine Menge von Aussagen benutzen können, benötigen wir eine genaue Beschreibung welchen speziellen Wortschatz wir benutzen wollen. Eine Standard-Anfangskomponente einer Ontologie beinhaltet eine Menge von Namensraum-Deklarationen, welche in einem öffnenden rdf:RDF Tag eingeschlossen sind. Dies bietet ein Mittel für eindeutige Interpreter-Bezeichner und macht den Rest der Ontologie-Darstellung lesbarer.

Ein Beispiel:

Das Beispiel beschreibt die Konzepte <*Person*>, <*Gender*> und <*Woman*>. Ein Frau ist definiert, als eine <*Person*> mit dem Wert <*female*> im Property <*gender*>, das der Klasse <*Gender*> angehören muss. Die Instanz <*STilgner*> ist somit als <*Person*> beschrieben eine Frau *(*<*Woman*>). Mittels Inferenz kann diese Zugehörigkeit ermittelt werden.

```
<rdf:RDF
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
    xmlns:owl="http://www.w3.org/2002/07/owl#"
    xmlns:j.0="http://localhost:8080/OWLBuergerInformation.owl#">

  <owl:Class rdf:ID="Gender"/>

  <owl:Class rdf:ID="Woman">
    <rdfs:subClassOf rdf:resource="#Person"/>
    <owl:equivalentClass>
      <owl:Restriction>
        <owl:onProperty rdf:resource="#gender"/>
        <owl:hasValue rdf:resource="#female" rdf:type="#Gender"/>
      </owl:Restriction>
    </owl:equivalentClass>
  </owl:Class>

  <owl:ObjectProperty rdf:ID="gender"
     rdf:type="http://www.w3.org/2002/07/owl#FunctionalProperty">
    <rdfs:range rdf:resource="#Gender"/>
    <rdfs:domain rdf:resource="#Person"/>
  </owl:ObjectProperty>
  <owl:DatatypeProperty rdf:ID="name"
     rdf:type="http://www.w3.org/2002/07/owl#FunctionalProperty">
    <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
    <rdfs:domain rdf:resource="#Person"/>
  </owl:DatatypeProperty>
  <owl:DatatypeProperty rdf:ID="firstname"
     rdf:type="http://www.w3.org/2002/07/owl#FunctionalProperty">
    <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
    <rdfs:domain rdf:resource="#Person"/>
  </owl:DatatypeProperty>

  <j.0:Person rdf:ID="STilgner"
     j.0:firstname="Susanne"
     j.0:name="Tilgner">
    <j.0:gender rdf:resource="#female"/>
  </j.0:Person>
</rdf:RDF>
```

2. Beschreibung der Rahmenapplikation

1. Beschreibung der Leistungsmerkmale

Protégé ist ein in frei verfügbares Software-Tool zur Entwicklung von Ontologien und Speicherung von Wissen, welches von der Stanford Medical Informatics und der Stanford University School of Medicine entwickelt wurde.
Es ist komplett in Java implementiert um Plattformunabhängigkeit zu gewährleisten und basiert auf dem SWING-GUI. Protégé erlaubt das manuelle Erstellen und Bearbeiten von Ontologien. Basierend auf einer erstellten Ontologie ist es dem Anwender möglich, durch Anlegen von Instanzen, eine Wissensbasis aufzubauen.
Die auf einzelnen Komponenten basierende Systemarchitektur ermöglicht es den Entwicklern ständig neue Funktionalitäten und Speicherformate in Form von Plug-Ins in Protégé einzubinden.
Des Weiteren kann es als Bibliothek für andere Applikationen zum Einbinden einer - in Protégé erstellten - Wissensbasis genutzt werden.
Ein weiteres Feature ist die Multiuserfähigkeit, d.h. es gibt einen Protégé-Server und mehrere Protege-Clients. Die Wissens-Basis liegt dabei auf dem Server (mehrere pro Server möglich). Ein Abgleich mit den Clients ist in Echtzeit möglich, was verteiltes Arbeiten an einer Wissens-Sammlung ermöglicht.

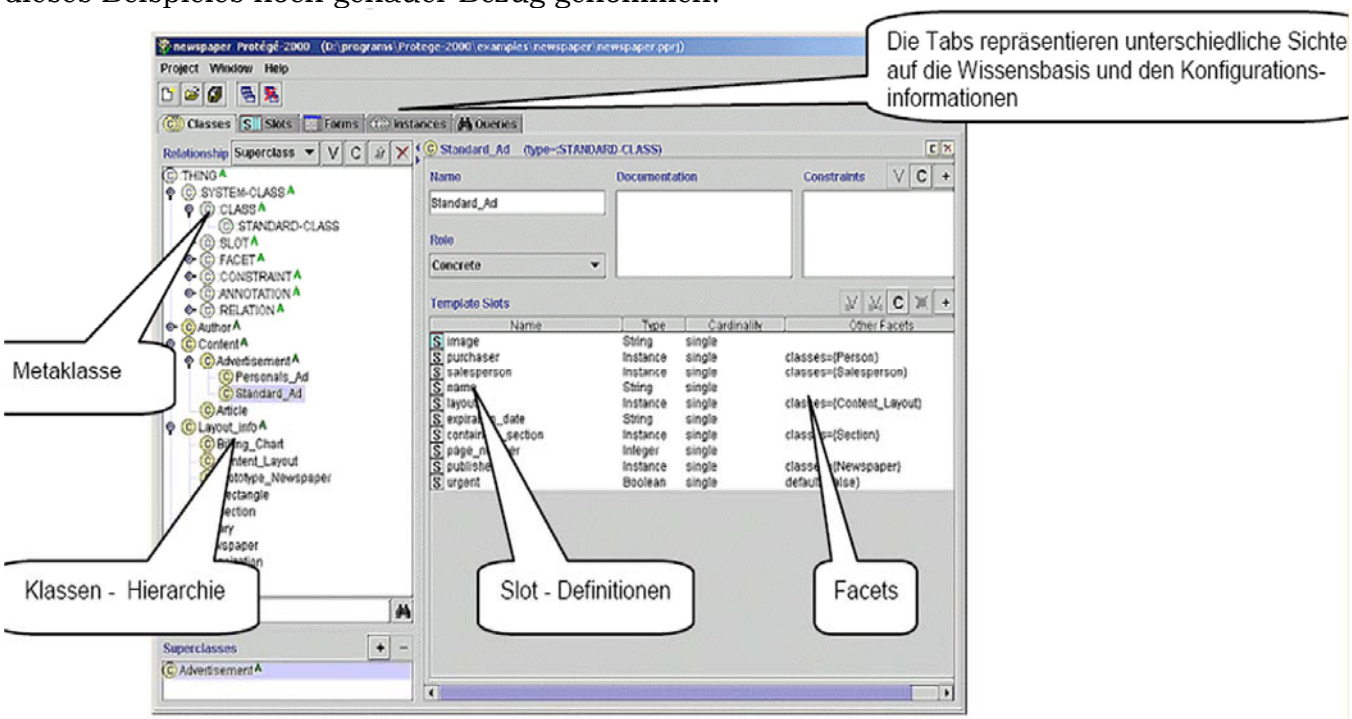
2. Darstellung äußerlich sichtbarer Aspekte der inneren Logik der Applikation

In der Benutzeroberfläche von Protégé wird die Wissensbasis durch Tabs (dt. Registrierkarten) repräsentiert. In der Basisversion von Protégé werden die Registrierkarten „Classes“, „Slots“, „Forms“, „Instances“ und „Queries“ angezeigt, wobei die drei Erstgenannten zum Aufbau eines Wissensmodells dienen. Über „Forms“ sind Änderungen im Aufbau und Erscheinungsbild der Eingabeformulare zur Wissensspeicherung möglich. Mithilfe der „Queries“ können an ein bestehendes Wissensmodell Wissensabfragen gestellt werden.

Das Protégé Wissensmodell orientiert sich am OKPC-Standard (dt. Schnittstelle für wissensbasierte Systeme). Eine Protégé-Ontologie besteht aus Klassen, „Slots“, und „Facets“.

Klassen beschreiben die in der Ontologie modellierten Begriffe. Instanzen sind konkrete Ausprägungen einer Klasse. Slots repräsentieren die Attribute einer Klasse. Facetten beschreiben ihrerseits wieder Eigenschaften der Slots, wie z.B. Einschränkungen eines Wertebereiches. In Protégé bildet eine Ontologie gemeinsam mit konkreten Instanzen ein Projekt. Die folgende Abbildung des in Protégé bereits enthaltenen Beispielprojektes soll den eben beschriebenen Zusammenhang der

einzelnen Elemente verdeutlichen. Auf die einzelnen Elemente wird später anhand dieses Beispieles noch genauer Bezug genommen.



2.1 Klassen und Instanzen

Eine Klasse beschreibt die Eigenschaften von gleichartigen Objekten. Ein Objekt ist eine Instanz einer Klasse. Aus einer Klasse können beliebig viele Instanzen erzeugt werden. Ist Die Klasse A Unterklasse einer Klasse B, so gilt für jede Instanz der Klasse A, dass sie auch eine Instanz der Klasse B ist und damit sämtliche Eigenschaften der Klasse B erbt. Bei Protégé ist Mehrfachvererbung möglich, dass bedeutet, eine Klasse kann mehrere Oberklassen haben. Die Wurzel der Klassenhierarchie ist in Protégé die Metaklasse „Thing“. Eine Metaklasse ist vergleichbar mit einer abstrakten Klasse, sie besitzt als Instanzen wiederum Klassen, sie beschreibt wie eine Klasse als Instanz der Meta-Klasse auszusehen hat. Eine Klasse kann eine Eigenschaft einer anderen Klasse sein. So können die in der Ontologie definierten Relationen realisiert werden. Eine Relation zwischen zwei Klassen kann anhand der Newspaper-Ontologie erläutert werden. Die „Article“-Klasse enthält neben den Eigenschaften „Headline“, Text“ und „Expiration Date“ auch die Eigenschaft „Author“ die zugleich eine Instanz der Klasse Autor ist.

Die Sicht auf die Klassen einer Ontologie erfolgt in Protégé über die Registrierkarte „Classes“. Hier können Klassen erstellt, bearbeitet, gelöscht und eingesehen werden. Die Klassenhierarchie wird als Baumstruktur dargestellt (siehe obige Abbildung). Der übrige Fensterinhalt rechts neben dem Baum zeigt den Namen, die Eigenschaften und die Beschreibung der markierten Klasse.
Über die Registrierkarte „Instances“ ist eine Sicht auf die Instanzen der markierten Klasse möglich.

2.2 Slots und Facets

Slots beschreiben Eigenschaften von Objekten einer Klasse. Im Gegensatz zur OOP kann ein Slot auch unabhängig von einer Klasse definiert werden bzw. können sich mehrere Klassen ein und denselben Slot teilen. Es ist auch möglich Subslots zu definieren, was dem Ableiten einer Klasse ähnlich ist.

Ein Facet beschreibt wiederum die Eigenschaften eines Slots. Er definiert Constraints (dt. Beschränkungen) bezüglich der Gültigkeit von Werten eines Slots. Mögliche Facets sind Typ (Integer, String, Instanz …), Kardinalität (ein oder mehrere Werte, Wertangabe benötigt
oder optional, …) sowie Intervallgrenzen bei Integer- und Float-Werten.

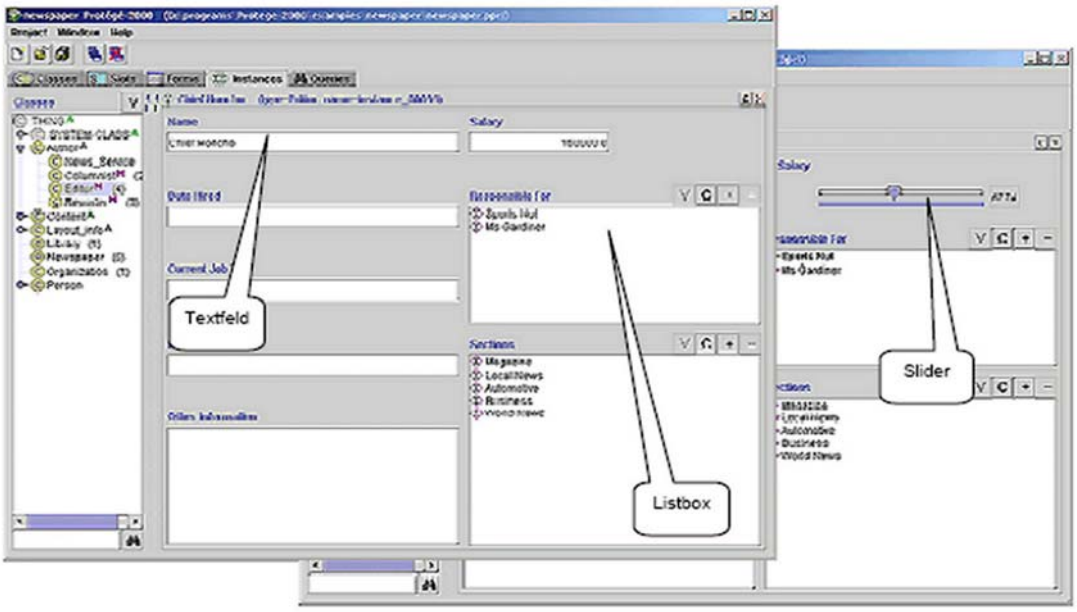
2.3 Forms

Wie bereits beschrieben erfolgt das Einsehen der Instanzen über Formulare unter der Registrierkarte „Instances".
Diese Formulare für die Instanzen werden automatisch von Protégé beim Erstellen und Bearbeiten der Klasse generiert. Für jeden Slot einer Klasse wird in diesem Formular ein „Slot-Widget" (dt. Eingabefeld) angezeigt.
Protégé unterscheidet dabei „Default-Widget-Types" und „Aditional-Widget-Types".

Erstgenannte sind die Felder, die automatisch in das Formular eingebettet werden. Dabei bestimmen Kardinalität und zulässiger Wertetyp des Slots das Aussehen und Verhalten des jeweiligen „Slot-Widgets". Zur Eingabe eines Strings genügt ein einfaches Textfeld.

„Additional-Widget-Types" sind dagegen Felder, in denen komplexere Werte dargestellt werden können, z.B. Listboxes oder Slider (siehe Abbildung 3).

Abb. 3: Protégé 2000, Instanzen - Tab[44]



Der Anwender hat die Möglichkeit über die Registrierkarte „Forms“ das von Protégé automatische erzeugte Standardformular seinen Bedürfnissen anzupassen. So können wichtige Informationen in dem Formular an oberster Stelle platziert werden und Beschriftungen von Eingabeformularen umbenannt werden. Außerdem kann durch Verwenden der „Additional-Widget-Types“ das Aussehen der Felder geändert werden. Ein Eingabefeld für das Gehalt eines Autors könnte beispielsweise, wie in Abbildung 3 dargestellt, durch einen Slider dargestellt werden.

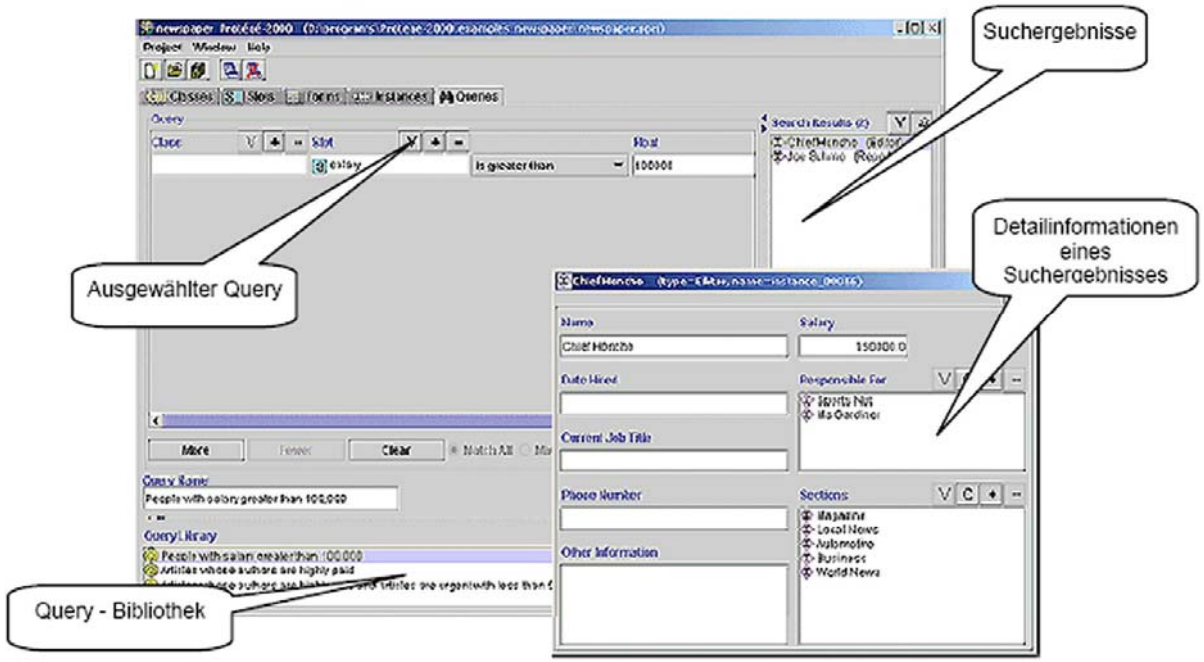
Diese Anpassungen innerhalb der „Forms“-Registrierkarte ändern also keineswegs den Inhalt der Wissensbasis, vielmehr dienen sie zur Verbesserung der Benutzerfreundlichkeit beim Einsehen und Bearbeiten der Wissensbasis.

2.4.Queries

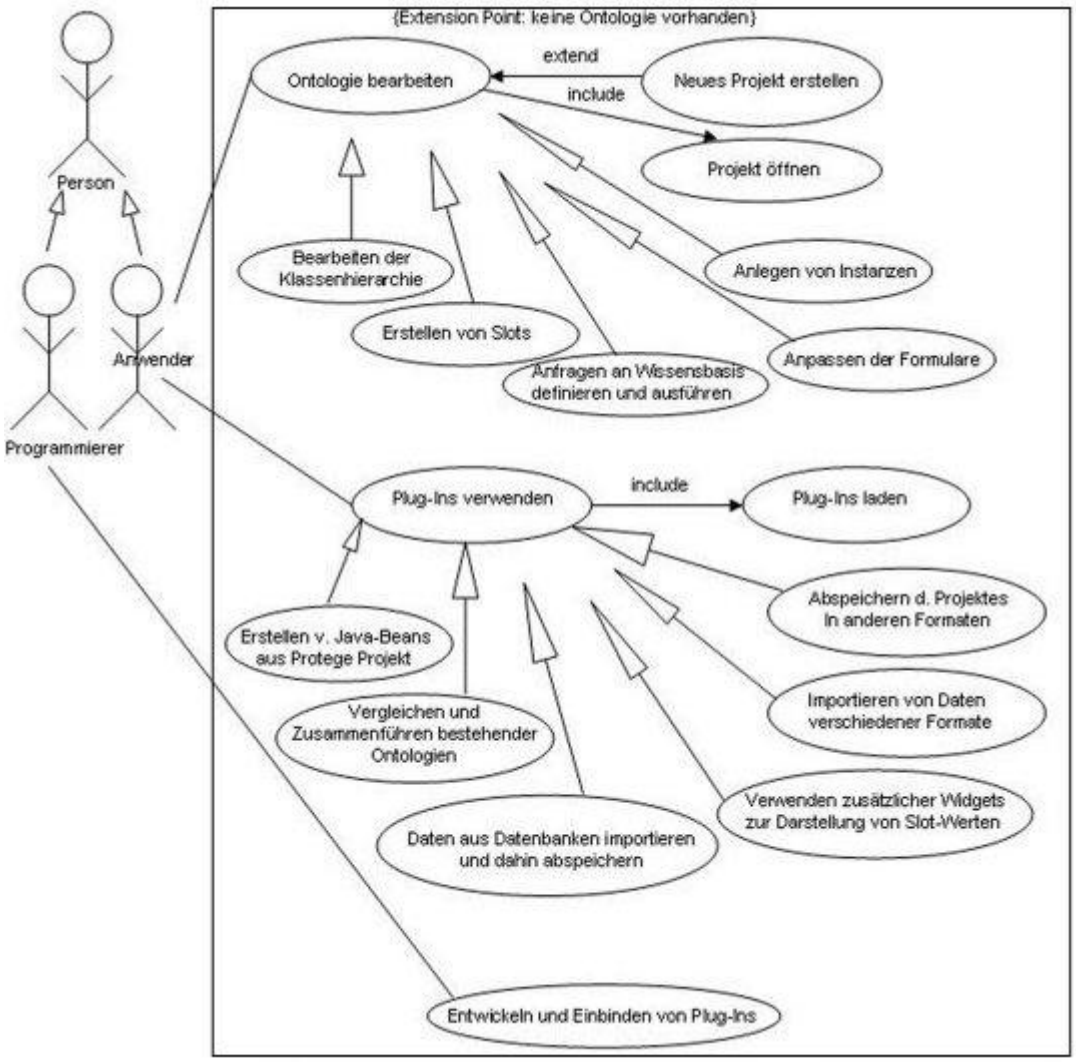
„Queries“ (dt. Abfragen) sind kein Bestandteil der Wissensbasis. Sie bieten jedoch die Möglichkeit aus der Wissensbasis Instanzen anhand von Klasseneigenschaften und Slot-Werten herauszufiltern. Der Benutzer kann „Queries“ definieren, speichern und ausführen.

Abbildung 4 zeigt die Benutzeroberfläche, welche sich hinter der Registrierkarte „Queries“ verbirgt. Gespeicherte Queries befinden sich in der „Query-Library“, von wo aus man diese auch aufrufen kann. Das Suchergebnis wird unter „Search-Results“ ausgegeben. Bei Auswahl eines Suchergebnisses werden Detailinformationen in einem weiteren Fenster ausgegeben.

Abb. 4: Protégé 2000, Queries - Tab



3. typische Anwendungsfälle



Auf die nähere Beschreibung der einzelnen Anwendungsfälle wird hier nicht weiter eingegangen, da diese bereits im Vorangegangenen erläutert wurden oder später noch erläutert werden.

3. Plug-In-Konzepte

→ Protégé

1. Übersicht

Protégé ist höchst anpassungsfähig. Es können sowohl Benutzeroberfläche als auch das Speicherformat an spezielle Bedürfnisse angepasst werden. Dies bringt verschiedene Vorteile mit sich:

- In der Benutzeroberfläche können Elemente eingebunden werden, die für bestimmte Domänen den Aufbau der Wissensbasis erleichtern.
- „Widgets" für bestimmte „Slots" können ausgeblendet werden.
- Durch die Änderbarkeit des Speicherformats kann Protégé in kürzerer Zeit an eine neue Sprache des Semantic Web angepasst werden, als ein neuer Editor für diese Sprache entwickelt werden kann.

Realisiert werden diese Anpassungen, indem zusätzliche Module an die flexiblen Systeme von Protégé „angeschlossen" werden. Diese zusätzlichen Module werden Plug-Ins genannt. Protégé ermöglicht es Entwicklern, durch Entwickeln von Plug-Ins neue Funktionalitäten einzubinden. Seine Plug-In-Bibliothek enthält Beiträge von Entwicklern aus aller Welt.

Beinahe jedes Plug-In fällte in eine der folgenden Kategorie:

- „Back-End" Plug-Ins, die Benutzen das Importieren aus und Speichern in unterschiedliche Formate (XML,…) ermöglicht
- „Slot-Widget" Plug-Ins, welche genutzt werden, um Slot-Werte beim Einsehen und Bearbeiten in einer domänen- und aufgabenspezifischen Weise anzuzeigen
- „Tab-Widget" Plug-Ins, die wissensbasierte Applikationen darstellen und mit der Wissensbasis von Protégé verbunden werden.

Ein Wissensingenieur, welcher nicht die Fähigkeit besitzt, eigene Plug-Ins zu entwickeln, kann nur bestehende Plug-Ins einbinden, da Mitarbeiter der Stanford University auf Anfrage lediglich Plug-Ins entwickeln, die einem allgemeinen Zweck dienen und nicht domänenspezifisch sind.

Des Weiteren ist zu erwähnen, dass Protégé die Möglichkeit bietet, bestehende Projekte in ein aktuelles Projekt einzubinden. Dabei ist jedoch zu beachten, dass keine Namensgleichheiten (z.B. der Klassennamen) bestehen dürfen, was in der Praxis eine erhebliche Einschränkung ist.

Im Folgenden wird auf die einzelnen Plug-In Kategorien eingegangen.

2. „Tab-Widget" Plug-Ins

Die Standard-Tabs von Protégé wurden bereits vorgestellt. Weiter Tabs können in Form von Plug-Ins aus dem Netz geladen oder selbst entwickelt werden, um die Wissensbasis wie gewünscht darstellen zu können. Einige frei verfügbare „Tab-Widget" Plug-Ins werden im Folgenden vorgestellt:

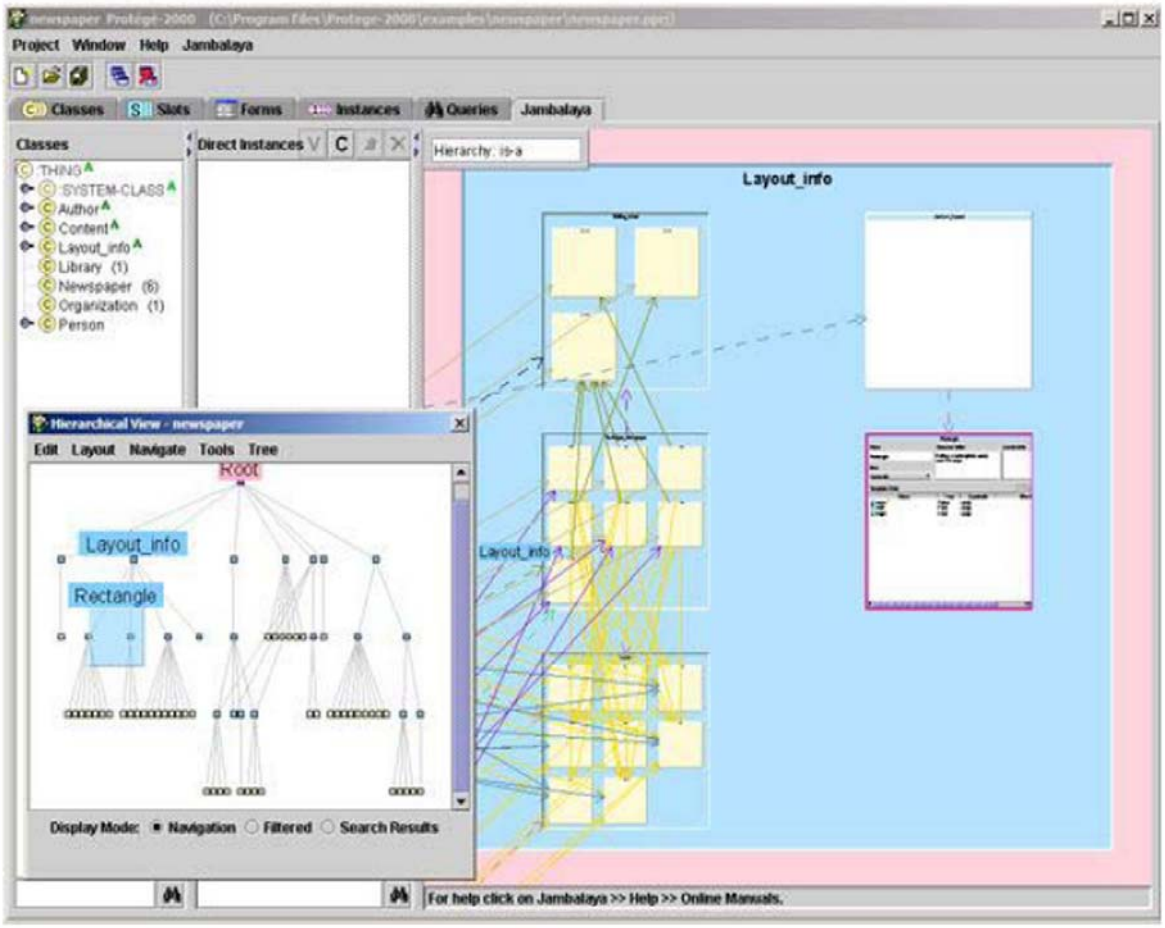
- „BeansGenerator" erstellt aus einem Protégé-Projekt Java-Beans-Klassen, die dann aus einer Java-Applikation heraus auf die

Wissensbasis zugreifen können.

- „DataGenie“ ermöglicht Protégé, aus jeder möglichen Datenbank zu lesen, die JDBC Interface benutzt. Dabei werden die gewählten Tabellen zu Klassen und die Tabellenattribute zu Slots.

- „PROMPT“ hilft bei der Verwaltung mehrerer Protégé-Projekte. Es können verschiedene Ontologien verglichen oder zu einer zusammen gefasst werden. Es ermöglicht ebenfalls die Extrahierung spezieller Teile einer Ontologie.

- „Jambalaya“ ist ein hierarchischer Ontologie-Browser, mit dem bestehende Daten bearbeitet werden können



- „TGVizTab“ veranschaulicht Ontologien in graphischer Form.

- „OntoViz“ bietet eine hoch konfigurierbare graphische Ausgabe von Modellen ähnlich UML.

3. Speicher Plug-Ins („Back-End“ Plug-Ins)

Protégé speichert die Projekte in seinem eigenen Format ab. Mittels eines „Back-End“ Plug-Ins lässt dich dieses durch jedes andere Dateiformat ersetzen. Dadurch wird auch ein Importieren von Daten dieses Formats in Protégé möglich. Daraus resultiert eine große Flexibilität zur Speicherung von Ontologien. Derzeit werden folgende Speicherformate angeboten:

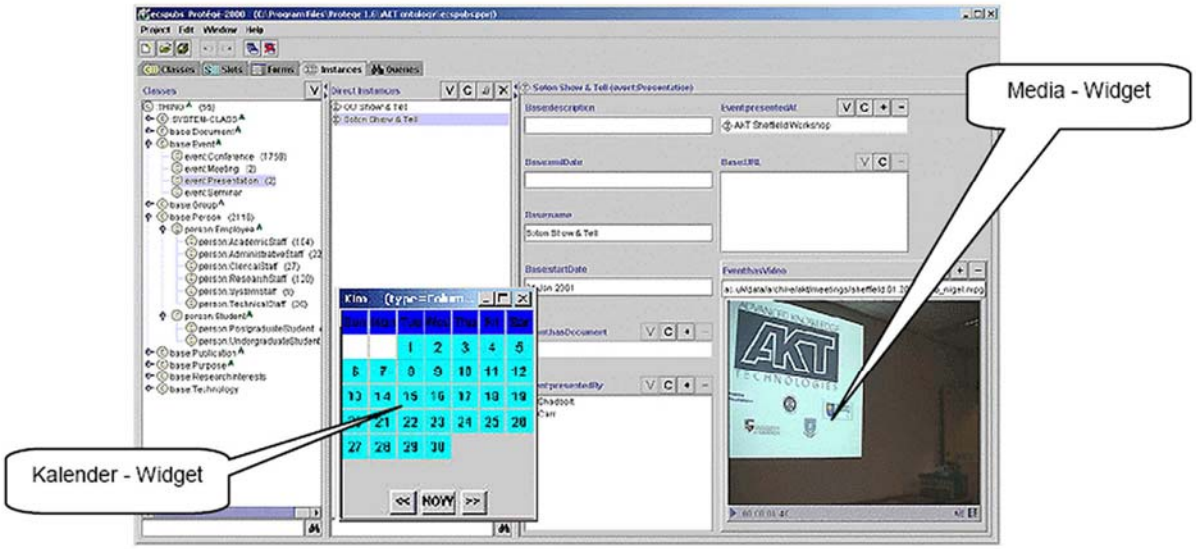
- CLIPS (Standardformat von Protégé)
- UML
- XML
- RDF
- OWL
- DAML+OIL

Weiterhin können Ontologien in Datenbanken gespeichert werden.

4. „Slot-Widget“ Plug-Ins

Wie bereits beschrieben, sind „Slot-Widget“ Plug-Ins graphische Elemente zur Ein und Ausgabe von Slot-Werten im Formular des „Instances“-Tabs, wie z.B. Textfelder oder Slider.

Neben den bereits in Protégé enthaltenen „Default-Widget-Types“ und „Additional-Widget-Types“ können für bestimmte Datentypen weitere „Slot-Widgets“ aus der Plug-In-Bibliothek eingebunden werden, wie z.B. Widgets für Datums- und Zeiteingaben oder zur Einbindung und Darstellung von Media-Dateien.



5. Einbinden eines Plug-Ins

Beim Starten von Protégé werden alle JAR files im "plugins" Ordner des Protégé Stammverzeichnisses zu dessen classpath hinzugefügt. Dadurch kann durch einfaches Hinzufügen eines JAR files in den "plugins" Ordner ein Plug-In in Protégé eingebunden werden.

Ein Plug-In JAR kann mithilfe der Standard Java "jar" Application erstellt werden, wobei ein JAR file ein oder mehrere Plug-Ins enthalten kann.

Das Manifest für das Plug-In JAR muss wie folgt beginnen:

```
Manifest-Version: 1.0
```

Darauf müssen eine Leerzeile und dann die Einträge für die Plug-Ins folgen. Dabei ist unbedingt zu beachten, dass die verschiedenen Einträge mit Leerzeilen getrennt sind und die letzte Zeile im Manifest ebenfalls eine Leerzeile ist.

Beispielsweise könnte man einen Ordner „Mein_Plugin“ in Protégés „plugins“-Ordner erstellen, und dort ein „Tab-Widget“ Plug-In JAR einfügen. Dies geschieht wie folgt:

- im einfachsten Fall befindet sich der Plug-In Quelltext in einer einzigen java-Datei, z.B. „Tolles_Plugin.java“ im Paket „de.lalala.test“
- daraufhin ist eine Textdatei „manifest.mf“ zu erstellen die folgendes beinhaltet:

```
Manifest-Version: 1.0

Name: de/lalala/Tolles_Plugin.class
Tab-Widget: True
```

- Beispielumgebung:

```
grundverzeichnis
|
|- manifest.mf
|- de
   |- lalala
        |- Tolles_Plugin.class
```

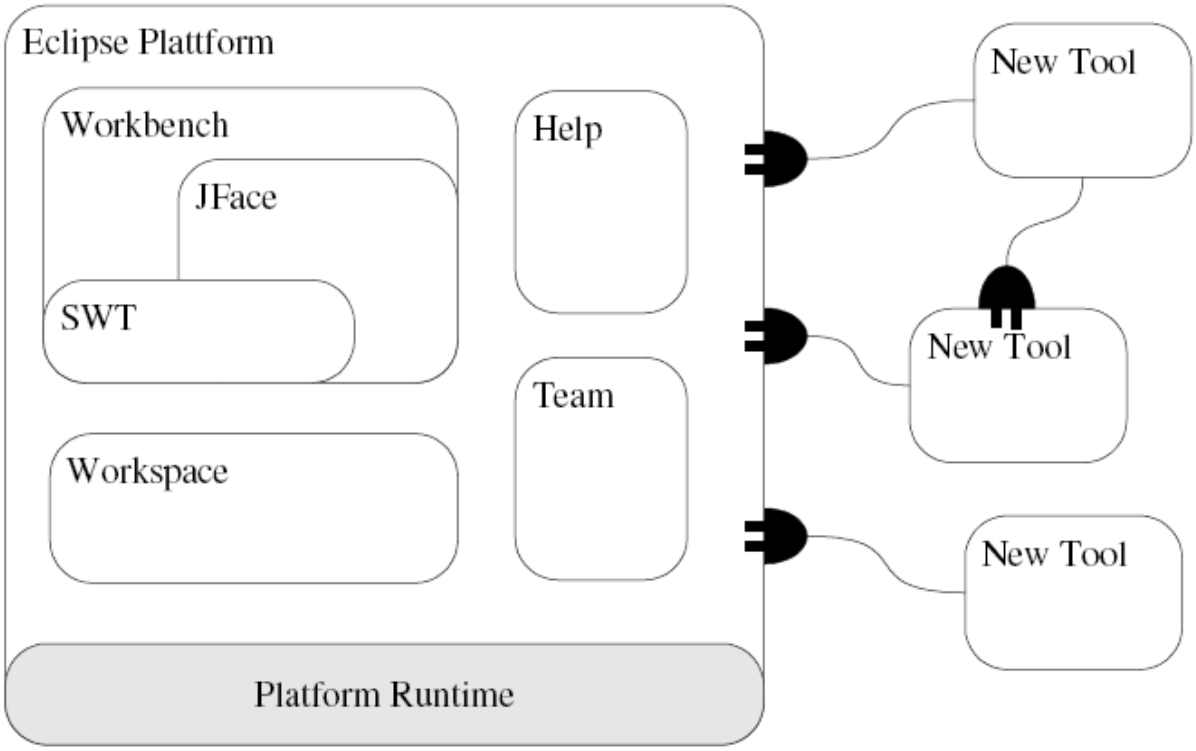
- in einer Konsole kann man nun durch folgendes Kommandos den JAR-file „Plugin.jar“ im Grundverzeichnis erzeugen:
  - cd grundverzeichnis
  - jar cfm ..\Plugin.jar manifest.mf *.*
- kopiert man diesen in den zu Beginn erstellten „Mein_Plugin“-Ordner in Protégés „plugins“-Verzeichnis wird beim Starten von Protégé das Plug-In geladen

Da es sich hier im Beispiel um ein „Tab-Widget“ Plug-In handelt muss man das Tab in Protégé noch unter „Project“ / „Configure“ aktivieren.

→ **Eclipse**

1. Übersicht

Eclipse ist eine Plattform, die von Grund auf zur Erstellung integrierter Web- und Anwendungswerkzeuge entwickelt wurde. Die Plattform an sich besteht nur aus einem sehr kleinen Kern, der sogenannten „Plattform Runtime“. Sie stellt Mechanismen zum Laden und Verwalten der Plug-Ins zur Verfügung. Auch legt sie Regeln fest, wie Plug-Ins miteinander kommunizieren und sich gegenseitig erweitern können. Andere Funktionalität, wie z.B. eine IDE zum Bearbeiten von Java, müssen erst durch Andere implementiert werden.



2. Das Plug-In Model

Da es sich bei Eclipse um ein statisches Plug-In-System handelt, stehen nur Plug-Ins zur Verfügung, die vor dem Start in einem dafür vorgesehen Verzeichnis abgelegt wurden. Es gibt zwar Bestrebungen auch das dynamische Laden von Plug-Ins zu ermöglichen, diese sind aber noch nicht fertig.

Jedes Plug-In besitzt dabei einen Erben der Plug-In-Klasse von Eclipse *'org.eclipse.core.runtime.Plugin'*. Dabei müssen alle Plug-Ins von dieser Klasse abgeleitet werden. Diese Plug-In-Klasse ist für die Konfiguration und Steuerung des Plug-Ins verantwortlich. Zusätzlich zur Plug-In-Klasse gibt es die Datei *plugin.xml*, welche Metainformationen enthält, die Eclipse zum Starten des Plug-Ins benötigt.
Den minimalen Aufbau dieser Datei zeigt das folgende Bild:

```
<?xml version="1.0" encoding="UTF-8"?>
<plugin
  name="JUnit Testing Framework"
  id="org.junit"
  version="3.7"
  provider-name="Eclipse.org">
  <runtime>
    <library name="junit.jar">
      <export name="*"/>
    </library>
  </runtime>
</plugin>
```

Durch diese Manifestdatei ist es Eclipse möglich, das Plug-In zu laden und seine Informationen in der *Pluginregistry* verfügbar zu machen.

In diesem Modell kann ein Plug-In in zwei verschiedenen Verhältnissen zu einem Anderen stehen.

Abhängigkeit : Es gibt dabei die Rollen des *abhängigen Plug-Ins* und des *erforderlichen Plug-Ins*. Dabei unterstützt das erforderliche Plug-In die Funktionen des abhängigen Plug-Ins

Beispiel für eine Abhängigkeit:

Durch das <requires>-Element wird kenntlich gemacht, dass das Plug-In in Abhängigkeit zum *'org.eclipse.ui'*-Basis-Plug-In steht.

```
<?xml version="1.0" encoding="UTF-8"?>
<plugin
   id="com.webtest.ui"
   name="WebTest UI"
   version="1.0.0">
   <runtime>
      <library name="webtestui.jar"/>
   </runtime>
   <requires>
      <import plugin="org.eclipse.ui"/>
   </requires>
</plugin>
```

Erweiterung : Dort gibt es die Rollen des *Host-Plug-In* und des *Erweiterungs-Plug-Ins*. Dabei wird die Funktionalität des Host-Plug-Ins durch das Erweiterungs-Plug-Ins erweitert.

Beispiel für eine Erweiterung:

Im folgenden Bild sehen wir, wie das *'org.eclipse.ui'*-Plugin einen Erweiterungspunkt für Menüeinträge definiert. Das Plug-In, welches nun neue Menüeinträge in die Menüleiste von Eclipse eintragen möchte, muss diesen Erweiterungspunkt benutzen. Dazu verweist es in seiner *plugin.xml* auf diesen Erweiterungspunkt und gibt die im Schema benötigten und optional definierten Informationen an.

```
<?xml version="1.0" encoding="UTF-8"?>
<plugin
 id="org.eclipse.ui"
    name="Eclipse UI"
    version="2.1.0"
    provider-name="Eclipse.org"
    class="org.eclipse.ui.internal.UIPlugin">
 <extension-point id="actionSets" name="Action Sets"
      schema="schema/actionSets.exsd"/>
    <!-- Other specifications omitted. -->
</plugin>
```

3. Installation

Ein Plug-In zu Eclipse hinzuzufügen, bedeutet, alle Ressourcen, die das Plug-In ausmachen, wie z.B. das Manifest, JAR-Dateien und Quellcode, in einen eigenen Ordner im 'plugin'-Verzeichnis des Eclipse-Installationsverzeichnisses zu kopieren. Beim Starten von Eclipse werden dann allerdings, um Overhead zu sparen, nur die Metainformationen der Plug-Ins ausgelesen, um etwa ihre Schnittstellen auch Anderen bekannt geben zu können. Erst wenn die Funktionalität eines Plug-Ins direkt gefordert wird, wird dessen Code auch ausgeführt.