Gruppe: SWP09-7 (Easy Inference Plugin) **Datum:** 18. Mai. 2009

Fremdprojektanalyse

RDF API für PHP (RAP)

Inhaltsverzeichnis

1. Allgemeines 1
2. Produktübersicht 1
3. Grundsätzliche Struktur- und Entwurfsprinzipien für das Gesamtsystem 2
3.1 Allgemeines...2
3.2 Query-Sprachen...2
4. Grundsätzliche Struktur- und Entwurfsprinzipien der einzelnen Pakete 2
4.1 Packages..2
4.2 Modelle in RAP...3
Das MemModel:...3
Das DbModel:...3
Das InfModel:...4
Das InfModelF:...5
Das InfModelB:...5
Das ResModel:...6
Das OntModel:..6
4.2 Inferenzen...6
Erstellen einer subClass-Regel:..7
5. Wiederverwendbarkeit des Codes für das Easy-Inferenz-Plugin..............................10

1. Allgemeines

Die RDF API für PHP, kurz RAP, ist ein Toolkit für Semantic-Web-Programmierer in PHP, welches seit 2002 an der Freien Universität in Berlin entwickelt wurde. Der aktuellste Realase ist die Version 0.9.6 vom Februar 2008.

2. Produktübersicht

Die RAP beinhaltet zwei verschiedene APIs um RDF-Graphen zu verwalten. Während die eine den RDF-Graphen als eine Menge von Statements sieht, betrachtet ihn die andere als eine Menge von Ressourcen.

Für beide APIs sind entsprechende Methoden zur Manipulation und Verwaltung der RDF-Graphen implementiert. Dazu gehören zahlreiche Parser, wie zum Beispiel für RDF/XML und N3, sowie eine SPARQL-Query-Engine und eine RDQL-Query Engine. Zu dem gibt es auch eine Inferenz-Engine, welche RDFS- und auch eine OWL-Regeln unterstützt.

Ein GUI steht zum Darstellen der verschiedenen Modelle der zwei integrierten APIs sowohl für inferierte als auch normale Modelle zur Verfügung. Im Allgemeinen sind die hauptsächlich im Hauptspeicher arbeitenden APIs für große Datenmengen und einen Vielbenutzerbetrieb ungeeignet.

3. Grundsätzliche Struktur- und Entwurfsprinzipien für das Gesamtsystem

3.1 Allgemeines

Die *RDF API* baut sich aus einer im api-Verzeichnis liegenden Datei RdfAPI.php auf, über welche sämtliche wichtige Konstanten und das Package-Model eingebunden werden. Für das Einstellen der Konstanten ist die im gleichen Verzeichnis zu findende Datei constants.php verantwortlich. Dort werden alle Packages, Klassen und Modelle eingebunden, der Zeichensatz eingestellt, sowie Einstellungen für RDF, RDFS, OWL, XML und die Datenbankanbindung vorgenommen.

Jedes Package der API hat sein eigenes Verzeichnis. Jede Klasse ihre eigene php-Datei. Der Quellcode der Klassen ist entsprechend der Standards für phpDoc dokumentiert und ist sehr übersichtlich gehalten und entsprechend gut eingerückt.

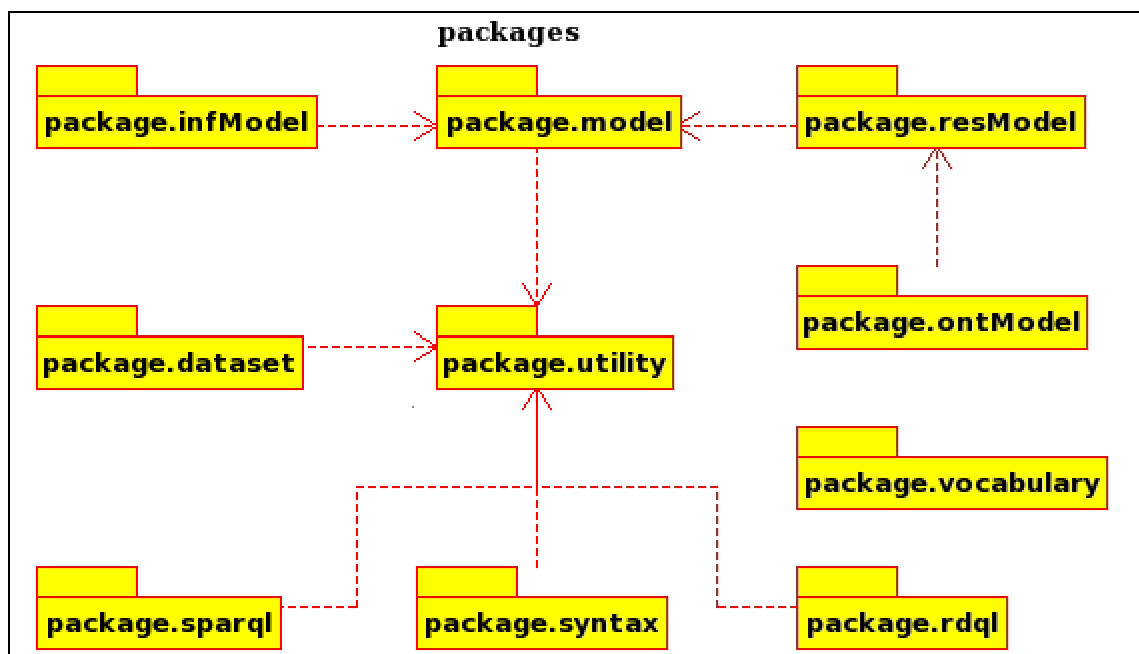
3.2 Query-Sprachen

Mit RAP kann man die Daten der Modelle über Query-Sprachen abfragen. Es wird RDQL und SPARQL unterstützt.

4. Grundsätzliche Struktur- und Entwurfsprinzipien der einzelnen Pakete

4.1 Packages

Die Pakete des RAP sind wie folgt aufgebaut:



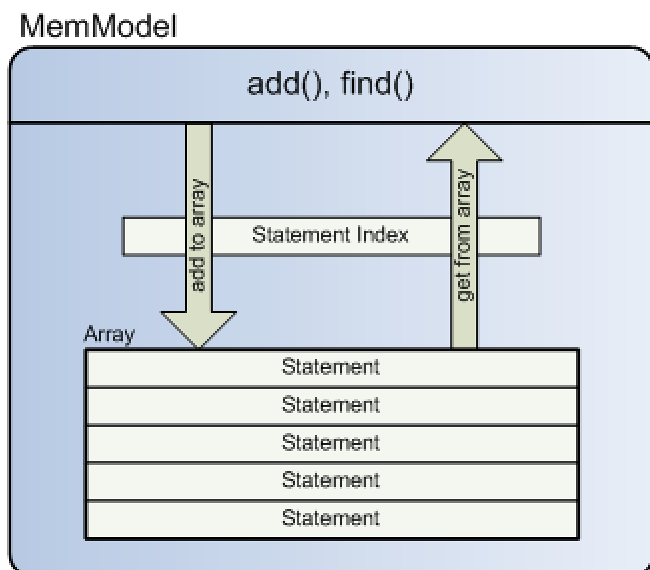
Die Pfeile symbolisieren entsprechend die Vererbungsbeziehungen zwischen Unterklassen der Pakete.

4.2 Modelle in RAP

Die zwei APIs in RAP bieten unterschiedlichste Modelle für die vorhandenen Daten. Betrachten wir zunächst die API, welche den RDF-Graphen als eine Menge von Statements sieht. Für diese API, die von den Entwicklern als *ModelAPI* benannt wird, gibt es vier verschiedene Modelle. Zwei nicht inferierte und zwei inferierte Modelle. Im Folgenden werden nacheinander die Modelle und ihre Eigenheiten beschrieben.

Das MemModel:

Dieses nicht inferierte Modell arbeitet wie der Name schon sagt mit einem im Hauptspeicher gespeicherten RDF-Graphen. Die Statements werden alle in einem Array gespeichert und über einen Index verwaltet. Hinzukommende Statements werden jeweils im Array gespeichert und dann im Index eingetragen um so schnelle Anfragen zu ermöglichen. Dieses Modell ist das schnellste Modell der *ModelAPI*, jedoch für große Datenmengen aufgrund des entsprechend enormen Speicherbedarfs im Hauptspeicher nicht verwendbar.



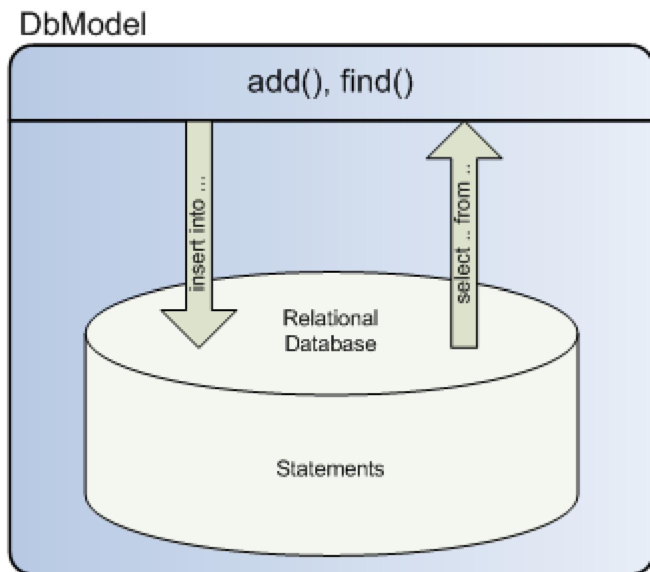
Betrachtet man den Quellcode genauer, so ist das *MemModel* ist eine von der abstrakten Superklasse *Model* abgeleitete Klasse. Die Klasse *MemModel* beherbergt zahlreiche Methoden, welche in der folgenden Übersicht aufgeführt werden.

- Konstruktor und Destruktor zum Anlegen und Freigeben im Hauptspeicher
- Methoden zum Einfügen, Ersetzen und Löschen von RDF-Tripeln und Namespaces
- Such-, Zähl- und Vergleichsmethoden für RDF-Tripel
- Vergleichsmethoden zwischen Instanzen der Klasse *MemModel*
- Methoden zum Verwalten des Modells, wie Setzen einer URI, Anlegen des Index
- Methoden zum Ausgeben als String, HTML oder HTML-Tabelle
- Methoden zum Exportieren
- Methoden zum Ausführen von RDQL- und SPARQL-Queries (zum Ausführen einer SPARQL-Query wird das Modell erst in ein RDF Dataset umgewandelt)

Das DbModel:

Dies ist ein weiteres nicht inferiertes Modell der *ModelAPI* dem eine relationale Datenbank

zugrunde liegt, was auch schon den Hauptunterschied zum zuvor vorgestellten *MemModel* darstellt. Das Modell nutzt die *ADOdb Database Abstraction Library* und unterstützt somit zahlreiche verschiedene Datenbanksysteme. In diesem Modell werden alle Statements in eine einzige Datenbanktabelle geschrieben. Schon dadurch ist zu erkennen, dass der Unterschied zum *MemModel* nicht sehr groß ist. Der Grund für diese Art der Speicherung innerhalb der Datenbank ist laut Entwickler, dass der Zugriff somit deutlich schneller ist. Allerdings kehrt sich dieser Performance-Gewinn bei großen Datenmengen um und somit ist auch dieses Modell ungeeignet für derartige Datenmengen.



Schaut man in den Quellcode, dann stellt man fest, dass das *DbModel* wie auch schon das *MemModel* eine von der abstrakten Superklasse *Model* abgeleitete Klasse ist. Entsprechend sind zahlreiche Methoden gleich oder ähneln einander stark. Hinzukommen hauptsächlich Methoden für die Datenbankschnittstelle. Anstelle des Arrays arbeiten sämtliche modellmanipulierende Methoden auf dem DB-Store, welcher in einer so benannten Klasse implementiert ist.

Das InfModel:

Mit diesem abstrakten Modell kommen wir nun zu den inferierten Modellen der *ModelAPI*. Das Modell wird vom *MemModel* abgeleitet und hat zusätzlich die Fähigkeit weitere Statements zu inferieren. Das *InfModelF* ist dabei nicht sehr schnell, was laut den Entwicklern hauptsächlich auf die Geschwindigkeit von PHP zurückzuführen ist. Dennoch soll es für kleine Datenmengen gut funktionieren. Das Modell unterstützt die RDFS- und OWL-Regeln rdfs:subclass, rdfs:subproperty, rdfs:range, rdfs:domain, owl:sameAs und owl:inverseOf.

Außerdem werden die RDFS-Regeln 6, 8, 10, 12 und 13 unterstützt. Um die Performance des Modells zu steigern, können nicht benötigte Regel abgestellt werden. Alles in allem ist dieses inferierte Modell natürlich extrem speicherintensiv, da es von *MemModel* abgeleitet ist. Die Inferierungsprozesse sorgen für eine zusätzliche Hauptspeicherbelastung.

Vom abstrakten *InfModel* werden die nach ihren unterschiedlichen Inferenzierungsalgorithmen benannten Modelle *InfModelF* und *InfModelB* abgeleitet.

Das InfModelF:

Das „F" im Namen dieses Modells steht für den in diesem Modell verwendeten *forward chaining inference algorithm*. Dieser Algorithmus arbeitet dabei anhand des neu hinzuzufügenden Statements erst die vorhandenen Inferenzregeln ab und speichert dann mit dem neuen Statement auch alle dazugehörigen inferierten Statements ab. Das heißt, das hinzuzufügende Statement steht erst nach dem Abgleich mit den Inferenzregeln zur Verfügung. Somit dauert es eine ganze Weile bis das neue Statement hinzugefügt wurde. Jedoch hat das *InfModelF* damit auch den Vorteil, dass zugehörige Inferenzen ebenfalls gleich vorhanden sind.

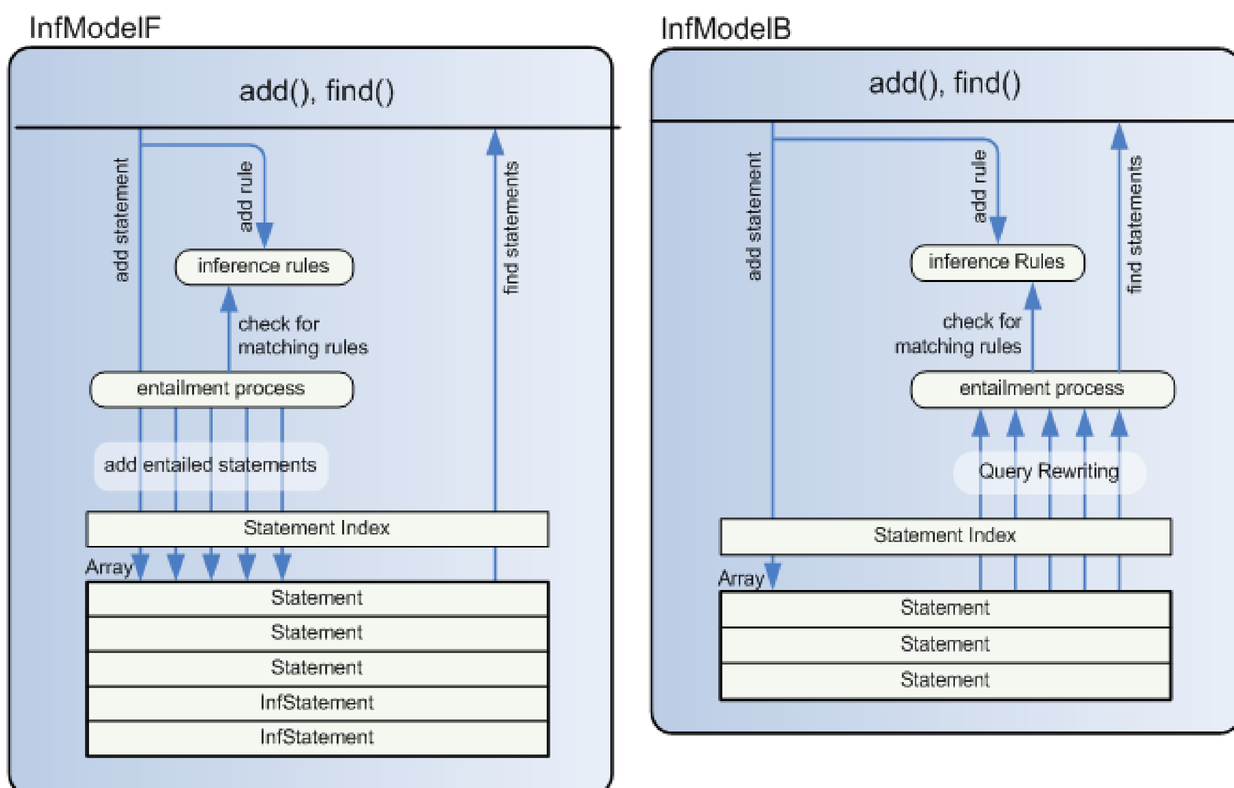
Das *InfModelF* sollte genutzt werden, wenn wenige Veränderungen an der Datenmenge vorgenommen werden und diese aber häufig abgefragt wird.

Das InfModelB:

Dieses Modell ist die Umkehrung des zuvor vorgestellten *InfModelF.* Hier wird ein ***backward** chaining inference algorithm* genutzt. Wie der Name schon sagt arbeitet dieser Algorithmus umgekehrt und fügt erst nach dem hinzufügen des eigentlichen neuen Statements die entsprechenden inferierten Statements hinzu. Dies bringt zwar den Vorteil, dass das neue Statement sofort vorhanden ist, sorgt jedoch auch gleichzeitig dafür, dass das die Inferenzen nicht immer auf dem aktuellsten Stand sind. Im Falle des Löschens oder des Überschreibens eines Statements kommt es somit zu Inkonsistenzen.

Das *InfModelB* sollte genutzt werden, wenn viele Veränderungen an der Datenmenge vorgenommen werden, wenn nur wenige Inferenzregeln angewendet werden und relativ wenige Anfragen auf die Daten stattfinden.

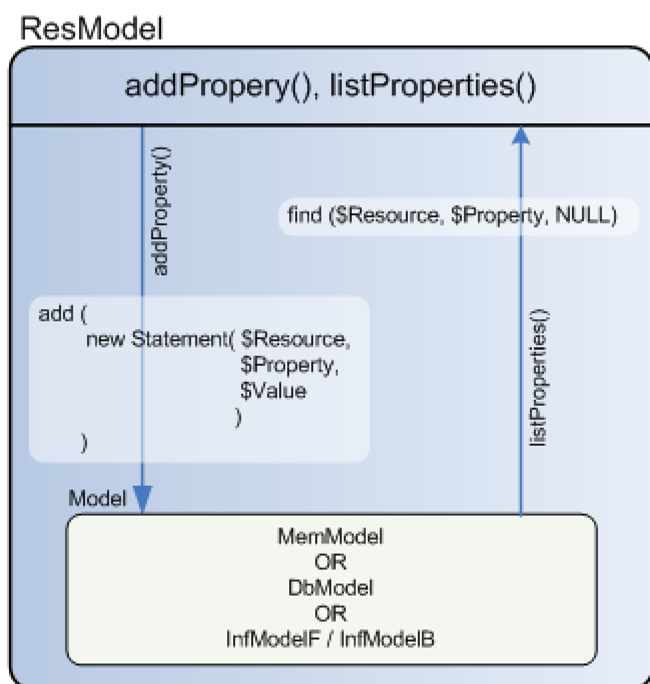
Vergleich zwischen InfModelF und InfModelB:



Soweit zu den Modellen der *ModelAPI*. Die zweite API von *RAP*, die so genannte Res*ModelAPI*, ist auf die *ModelAPI* aufgesetzt und stellt somit eine Art Erweiterung dar. Die Res*ModelAPI* ist der Jena *Model* API, einer weiteren API für Semantic-Web-Technologien sehr ähnlich. Im wesentlichen unterscheidet sich diese API im Vergleich zur zuerst vorgestellten durch die Ansicht des RDF-Graphen als Menge von Ressourcen. Der Aufsatz auf ein Modell der *ModelAPI* sorgt dafür diese Sicht umzusetzen. Im Folgenden werden die zwei Modelle der Res*ModelAPI* vorgestellt.

Das ResModel:

Dieses Modell sieht Statements als Tripel aus einer Ressource, einer Property und dem eigentlichen Value. Es ist deutlich langsamer als das gewählte darunterliegende Modell aus der *ModelAPI*, da die Algorithmen dieses Modells sich derer des darunterliegenden bedienen. Es findet eine Art Übersetzung statt.



Entsprechend der anderen Sichtweise werden im *ResModel* zahlreiche weitere Methoden verwendet. Diese dienen hauptsächlich zum Anlegen von Ressourcen und Properties.

Das OntModel:

Dieses Modell ist abgeleitet vom *ResModel* und stellt eine Erweiterung um RDFS-spezifische Methoden wie AddSubClass(), listSubClasses(), hasSuperProperty(), addDomain() und listInstances() dar.

Das *OntModel* bietet die beste Schnittstelle für eine Arbeit mit RDFS Modellen. Insbesondere wenn inferenziert werden soll, ist dieses Modell mit einem darunterliegenden *InfModelF* oder *InfModelB* am besten geeignet.

4.2 Inferenzen

Inferenzen werden mit Hilfe der Klasse InfModel oder einer ihrer Subklassen realisiert. Es gibt zwei Umsetzungen der Inferenz, durch die Realisierungen InfModelB und InfModelF

gekennzeichnet wie oben beschrieben. Der Ansatz ist einmal, bei jedem Hinzufügen eines Statements die Inferenzen neu zu generieren, und das andere mal werden die Inferenzen bei jeder Abfrage neu berechnet.

Zum Erhalten von Informationen wird i.A. die find-Methode benutzt, die im InfModelB-Fall die private Methode _infFind verwendet um die Inferenzen zu generieren, d.h. inferierte Daten werden direkt als Ergebnisse an Aufrufe der find-Methode sichtbar.

Im InfModelF wird während des Einfügens von neuem Wissen über die add-Methode bereits die private Methode _entailStatementRec verwendet um inferierte Informationen direkt in die Wissensdatenbank einzutragen. Dementsprechend stehen diese dann auch beim Aufruf von find direkt zur Verfügung.

Dann muss allerdings darauf geachtet werden, beim Entfernen von Statements auch die inferierten Regeln mit zu entfernen. Wenn nötig werden dann alle Inferenzen gelöscht und neu generiert.

Allerdings ist bei RAP der Inferenzregelumfang zunächst einmal auf die fest vorgegebenen Inferenzregeln in der Datei InfModel.php beschränkt.

Dabei wird für die Regeln die Klasse InfRule benutzt. Diese besitzt einen Trigger- und ein Entailment-Array sowie Methoden um diese zu verarbeiten.

Der Trigger ist ein Tripel, bestehend aus Subjekt, Prädikat und Objekt, das beim Inferieren benutzt wird um ein Statement zu prüfen, ob die Regel auf ihn zutrifft, indem es mit dem Trigger verglichen wird. Die Elemente des Tripels können entweder eine Instanz von model::Node sein oder null, was bedeutet, dass ein beliebiger Wert im Statement an dieser Stelle stehen kann um den Trigger auszulösen.

Das Entailment-Array repräsentiert ebenfalls ein Tripel, welches für Subjekt, Prädikat und Objekt entweder eine Instanz von model::Node oder einen der Werte '<p>','<s>' oder '<o>' verlangt. Diese Werte geben an, welcher Teil des Statements für die inferierten Tripel als Subject, Prädikat oder Objekt zu benutzen sind.

Vereinfacht kann man sagen, dass der Trigger entscheidet wann inferiert wird und das Entailment wie zu inferieren ist. Das folgende Beispiel soll die Erstellung einer Regeln demonstrieren:

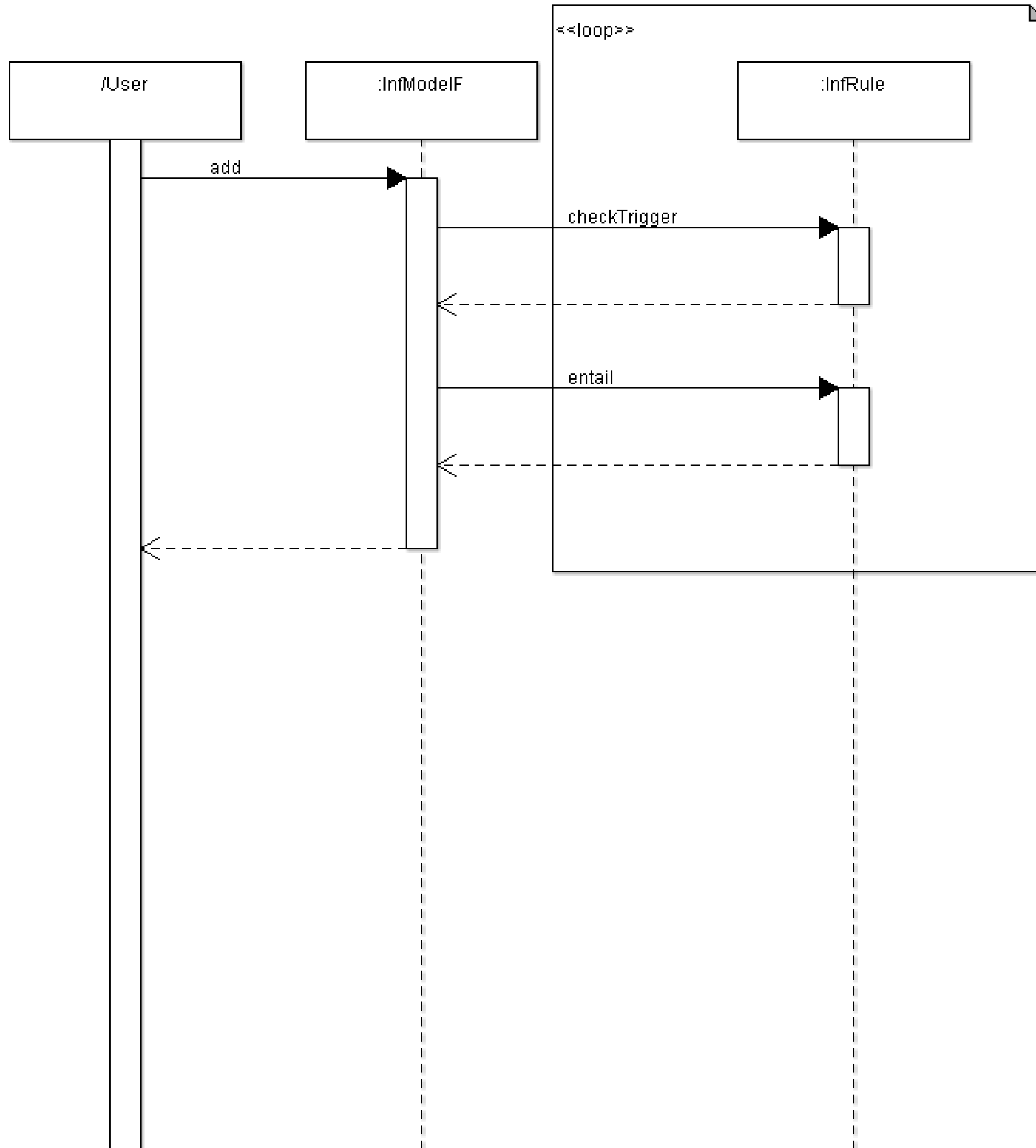
Erstellen einer subClass-Regel:

Annahme: $statement ist vom Typ model::Statement, wobei das Prädikat rdfs:subClass ist, also das Subjekt ist eine Unterklasse von Objekt

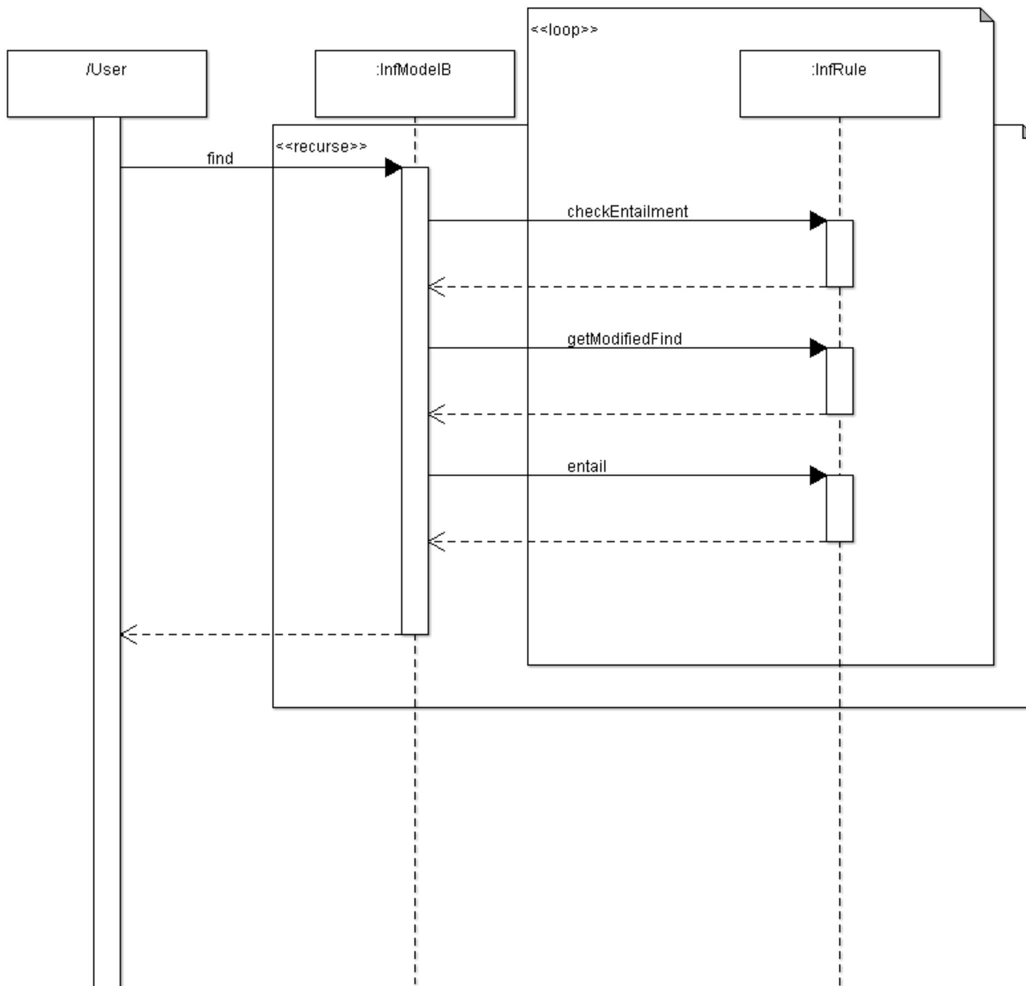
```
$infRule=new InfRule();
$infRule->setTrigger(null, new Resource(RDF_NAMESPACE_URI.RDF_TYPE), $statement->getSubject());
$infRule->setEntailment('<s>', new Resource(RDF_NAMESPACE_URI.RDF_TYPE), $statement->getObject());
```

Wird diese Regel nun angewendet, wird aus jedem Tripel, das beim Prädikat rdf:type besitzt und als Objekt $statement->getSubject() hat, geschlussfolgert, dass das Subjekt des Tripels auch vom Typ statement->getObject() ist.

Anwendung einer Regel von InfModelF(Sequenzdiagramm):



Anwendung einer Regel von InfModelB(Sequenzdiagramm):



5. Wiederverwendbarkeit des Codes für das Easy-Inferenz-Plugin

Der Code von RAP ist für unser Projekt nicht direkt wiederverwendbar, da sich die API zu sehr von Ontowiki und Erfurt unterscheidet. Allerdings sind verschiedene Konzepte die in RAP umgesetzt sind für unser Projekt interessant.

Es könnte sich als vorteilhaft erweisen eine eigene Klasse für Regeln zu implementieren, die sich ähnlich wie die Klasse InfRule verhält. Dadurch könnte man die Realisierung von Regeln von der Hauptkomponente kapseln, wodurch sich die Lesbarkeit und Übersichtlichkeit des Codes verbessern würde.

Ein weiteres Konzept, welches man versuchen könnte in unseren Projekt zu verwirklichen, ist die Anwendung von Regeln beim Hinzufügen oder Löschen von Statements. Da in Ontowiki Events bei diesen Operationen ausgelöst werden, könnte man darauf reagieren und Regeln auf die entsprechenden Statements anwenden um bei hinzuzufügenden Statements gleich Inferenzen zu generieren und mit hinzuzufügen oder bei zu löschenden die nun ungültigen Inferenzen mit löschen. Dadurch können Inkonsistenzen vermieden werden.