d3web.kernel-HowTo

Wie der *d3web.kernel* in Java-Applikationen integriert werden kann

Norman Brümmer, Joachim Baumeister

2. April 2003

Zusammenfassung

Dies ist ein Entwickler-HowTo, das beschreibt, wie der *d3web.kernel* in Java-Applikationen integriert werden kann. Im ersten Abschnitt wird allgemein erklärt, was der *d3web.kernel*, wie er arbeitet und welche die Hauptmechanismen und -objekte sind. Im zweiten Abschnitt werden diese Objekte detaillierter erklärt, so dass der Entwickler ein grundlegendes Verständnis von den wichtigsten Klassen des *d3web.kernel*s bekommt. Im dritten Abschnitt wird anhand eines Szenarios in Form einer Beispielapplikation schrittweise das Vorgehen bei den wichtigsten Mechanismen(wie z.B. Laden von Wissensbasen, starten eines neuen Falls, etc.) erklärt. Im vierten Abschnitt werden die XML-Repräsentationen von Wissens- und Fallbasen vorgestellt. Abschließend wird im fünften Abschnitt erläutert, was zu tun ist, wenn man einen eigenen Problemlöser implementieren möchte.

Inhaltsverzeichnis

1 Was ist d3web?

2 Die grundlegenden Objekte und Konzepte

Die zentralen Klassen zur Repräsentation von Wissensbasisobjekten befinden sich im Paket

`de.d3web.kernel.domainModel`

und seinen Unterpaketen. Eine Wissensbasis wird in *d3web* durch die Klasse `KnowledgeBase` realisiert. Ihre wichtigsten Bestandteile sind Frageklassen und Fragen, Antworten, Diagnosen und Regeln. Ihre Repräsentation wird im Folgenden genauer erklärt:

Für Objekte der Wissensbasis, die eindeutig identifizierbar sein sollen (wie z.B. Frageklassen, Fragen, Diagnosen, einige Antworttypen und Regeln) existiert die abstrakte Klasse `IDObject`. Von ihr erbt direkt die Klasse `RuleComplex`, welche Regeln repräsentiert. Diagnosen, Frageklassen und Fragen haben einen „Namen“, z.B. einen Fragetext oder eine beschreibende Frageklassenbezeichnung. Daher erben die Klassen `Diagnosis` und `QASet` von `NamedObject`, einer Klasse, die `IDObject` erweitert. Die Oberklasse `QASet` wird, wie auch die anderen erwähnten konkreten Klassen später genauer behandelt. Diagnosen und Fragen können einen Wert erhalten. Daher implementieren die Klassen `Diagnosis` und `Question` das Interface `ValuedObject`.

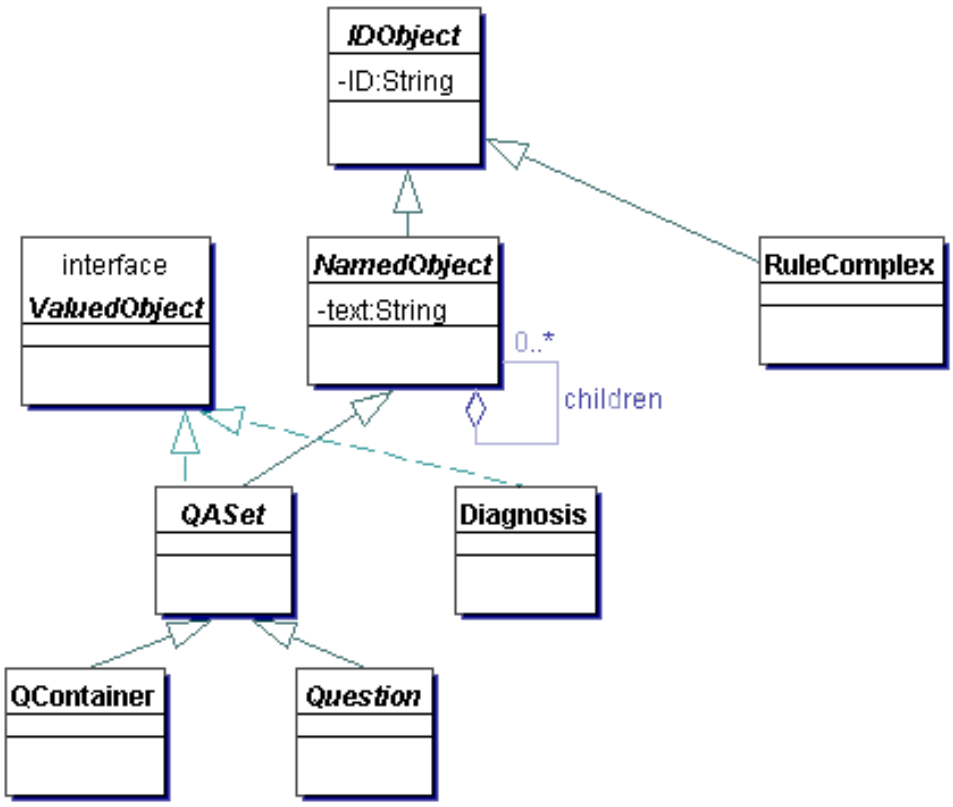

Abbildung 1: de.d3web.domainModel - Die wissensbasisrelevanten Klassen

Wie man in Abbildung 1 sehen kann, können `NamedObject`s wieder `NamedObject`s enthalten, die man über die `getChildren()`-Methode erreichen kann. Andersherum kann man von einem Kindobjekt über `getParents()` auf dessen Elternobjekte zugreifen - eine doppelte Verkettung also. In Tabelle 1 sind die wesentlichen Wissensbasisobjekte und die zugehörigen Klassen gegenübergestellt.

Wissensbasis-Objekt	**d3web-Klasse**
Wissensbasis	KnowledgeBase
Diagnose	Diagnosis
Frageklasse	QContainer
Frage allgemein	Question
Antwort allgemein	Answer
Regel	RuleComplex
Regelkondition allgemein	AbstractCondition
Regelaktion allgemein	RuleAction

Tabelle 1: Gegenüberstellung: Wissensbasisobjekte - d3web-Klassen

2.1 Frageklassen, Fragen und Antworten

In Bezug auf Frageklassen und Fragen hat die oben genannte Eltern-Kind-Verkettung folgende Bedeutung: Eine Frageklasse, realisiert durch `QContainer` kann allgemein `QASets` enthalten; somit also Fragen, das sind `Question`-Objekte, und wiederum Frageklassen. In diesen Fällen handelt es sich einfach um eine hierarchische Beziehung. Hat allerdings ein `Question`-Objekt „Kinder“, so handelt es sich um Folgefragen dieser Frage. Die Semantik ist hier eindeutig, weil eine einfache Frage kein Container ist.
Es gibt eine Reihe verschiedener Fragetypen und dementsprechend viele verschiedene Typen von Antworten. Klassen, die spezielle Antworttypen realisieren, sind im Paket

`de.d3web.kernel.domainModel.answers`

enthalten. Im Folgenden werden die konkreten Frage- und Antworttypen beschrieben:

Choice-Fragen: One-Choice und Multiple-Choice-Fragen werden durch die Klassen `QuestionOC` bzw. `QuestionMC` realisiert. Diese erhalten Werte vom Typ `AnswerChoice`, die entsprechenden Antwort-Implementierungen. Ein Spezielfall der One-Choice-Fragen sind die Ja-Nein-Fragen, realisiert durch `QuestionYN`. Entsprechend existieren Klassen für die Ja- und Nein-Antwort, `AnswerYes` bzw. `AnswerNo`.

Numerische Fragen: Numerische Fragen entsprechen in *d3web* der Klasse `QuestionNum`. Diese bekommen einen Wert vom Typ `AnswerNum`.

Text-Fragen: Text-Fragen sind in *d3web* Objekte vom Typ `QuestionText`. Sie erhalten Werte vom Typ `AnswerText`.

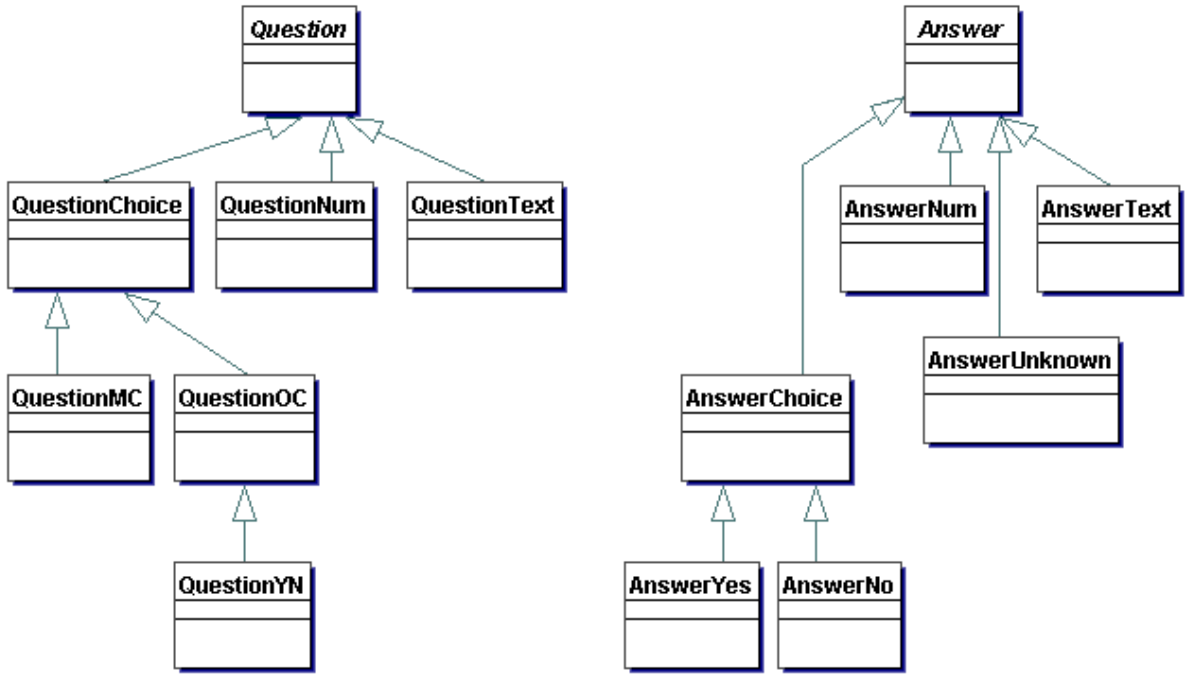

Abbildung 2: Fragen und Antworten

Da alle Fragen unabhängig vom Typ mit „unbekannt“ beantwortet werden können, gibt es eine entsprechende Antwort-Implementierung, `AnswerUnknown`.
In Abbildung 2 sind die beiden Hierarchien nebeneinander zu sehen.
Wie den Fragen die Antworten zugewiesen werden, wird im letzten Abschnitt dieses Kapitels beschrieben.

2.2 Diagnosen

Diagnosen entsprechen in *d3web* `Diagnosis`-Objekten. Da diese `NamedObject`s und damit auch `IDObject`s sind, haben sie sowohl einen Verbalisierungstext als auch eine eindeutige ID. Auch Diagnosen können demnach Hierarchien bilden. Wie man die Eltern- und Kind-Objekte einer Diagnose erreicht, ist bereits am Anfang des Kapitels erklärt worden. Diagnosen können verschiedene Stati annehmen, die von einem Problemlöser berechnet werden: ausgeschlossen (*excluded*), unklar (*unclear*), verdächtigt (*suggested*) und etabliert (*established*). Diese sind abhängig von den Punkten, die sie z.B. durch eine heuristische Regel bekommen. Status und Punkte einer Diagnose werden durch die Klassen `DiagnosisState` bzw. `DiagnosisScore` im Paket `de.d3web.domainModel` repräsentiert. Um die Punkte bzw. den Status einer Diagnose zu erfragen, sollten die Methoden

`getScore(XPSCase, Class)` bzw. `getState(XPSCase, Class)`

der Klasse `Diagnosis` verwendet werden. Als `Class`-Parameter übergibt man die Klasseninstanz desjenigen Problemlösers, der den Diagnosestatus berechnen soll.

2.3 Regeln

d3web verwendet eine Vielzahl von Regeln, um beispielsweise einen geführten Dialog zu steuern (Folgefragen-Regeln, Suppress-Regeln, Contra-Indikations-Regeln) oder um als Funktionsgrundlage von Problemlösern (heuristischer Problemlöser) zu dienen.
Regeln unterscheiden sich nur durch ihre verschiedenen Aktionen. Sie haben ansonsten immer den gleichen Aufbau und folgen der gleichen Funktionsweise. Deshalb wurde das sogenannte *Rule-Pattern* eingeführt, das nun näher betrachtet werden soll. Eine Regel ist immer eine Instanz der Klasse `RuleComplex` und enthält eine Kondition (`AbstractCondition`) und eine Regelaktion (`RuleAction`).

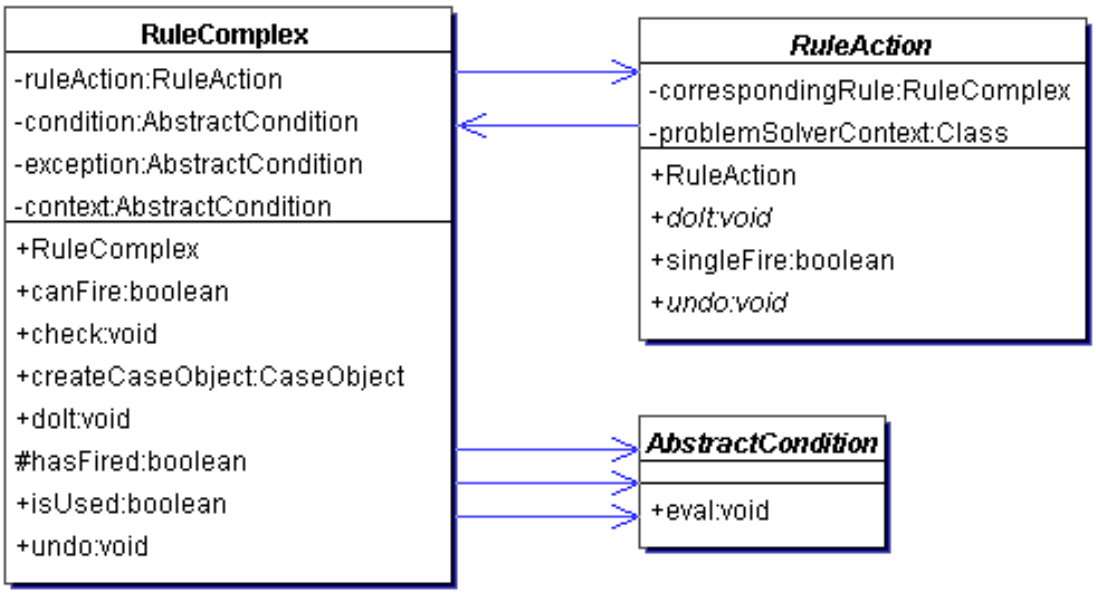

Abbildung 3: Rule-Pattern. Eine Regel mit Regelaktion

In Abbildung 3 ist eine Regel mit Regelaktion `ruleAction` und -Kondition `condition` zu sehen. Desweiteren hat `RuleComplex` die Attribute `context` und `exception` vom Typ `AbstractCondition`, die spezielle Konditions-Objekte sind. Aktionen und Konditionen einer Regel werden Folgenden erläutert.
Über die Methode `canFire(XPSCase)` kann abgefragt werden, ob die Regel feuern kann. Dies wird in Abschnitt 2.3.2 im Detail behandelt. Die Methode `hasFired(XPSCase)` prüft, ob die Regel bereits gefeuert hat - also, ob die Regelaktion im aktuellen Fall (`XPSCase`) schon ausgeführt wurde. `check(XPSCase)` ist eine zusammenfassende Methode, welche die Ausführung und Rücknahme einer Regel steuern kann. Sie führt die Regelaktion aus, wenn die Regel feuern kann und sie noch nicht ausgeführt wurde. Umgekehrt nimmt sie eine Regelaktion zurück, wenn sie bereits ausgeführt wurde, die Regel aber momentan nicht feuern kann (wenn die Kondition nicht erfüllt ist, oder die Ausnahmebedingung erfüllt ist).

2.3.1 Verwendung und Instanziierung von Regeln

Regeln sollten, wenn möglich, nicht manuell über ihren Konstruktor erzeugt werden, sondern durch die Verwendung der entsprechenden Methode in der Klasse `RuleFactory`. Lässt sich die manuelle Erzeugung einer Regel allerdings nicht umgehen, sollte man unbedingt darauf achten, dass erst dann eine Kondition gesetzt wird, wenn die Aktion bekannt ist. Dies ist darin begründet, dass die Aktionen den Problemlöser-Kontext tragen und dieser wichtig für die dynamische Verzeigerung der beteiligten Objekte ist. Diese Anforderung wird jedoch von den jeweiligen Methoden in `RuleFactory` erfüllt. In Abbildung 4 ist die gesamte Hierarchie der bereits implementierten Regelaktionen zu sehen.

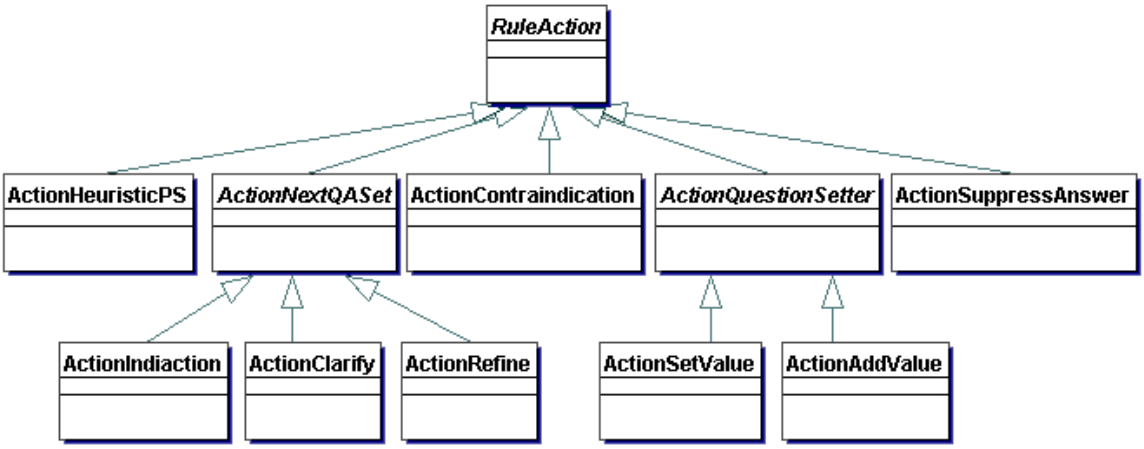

Abbildung 4: RuleAction-Hierarchie: Alle Regelaktionen auf einen Blick

2.3.2 Semantik von Kondition, Ausnahmebedingung und Diagnose-kontext

Konditionen
Die Regelkondition ist die hauptsächliche Bedingung, bei deren Erfüllung eine Regel feuern kann (und somit ihre Aktion ausführt). Eine Kondition muss angegeben werden. Generell kann man sagen, dass Regeln die Form

Wenn KONDITION dann AKTION

haben. Im Folgenden werden noch zwei zusätzliche Bedingungen vorgestellt, mit denen Einschränkungen zur Kondition gemacht werden können.

Ausnahmebedingungen
Eine Ausnahmebedingung ist eine *optionale* Bedingung, die bei Erfüllung das Feuern einer Regel verhindert. Es erscheint etwas verwunderlich, dass zusätzlich zur Kondition eine Ausnahmebedingung (Attribut `exception`) im Regelobjekt existiert. Auf den ersten Blick könnte man annehmen, dass diese durch die Kombination

Kondition UND NICHT Ausnahmebedingung

abgebildet werden könnte. In *d3web* basieren jedoch die meisten Konditionen auf Antworten auf im Dialog gestellten Fragen. Sind Fragen (noch) nicht beantwortet, so kann die umschließende Kondition nicht ausgewertet werden und es wird eine `NoAnswerException` geworfen. Die Teilkondition *NICHT Ausnahmebedingung* könnte im ungünstigsten Fall also nicht evaluiert werden. Ausnahmebedingungen beinhalten somit meist selten beantwortete Fragen, deren Auftreten jedoch zum Verhindern der Regelfeuerung führen soll.

Diagnose-Kontext
Der Diagnose-Kontext (Attribut `context`) ist eine *optionale* Bedingung, die bewirkt, dass eine Regel nur dann in Betracht gezogen wird, wenn die in der Kontext-Kondition angegebenen Diagnosen etabliert sind. Realisiert wird dies durch ein `AbstractCondition`-Kompositum, deren Terminale ausschließlich aus `CondDState`-Objekten bestehen.

Zusammenfassend kann man sagen, dass eine Regel genau dann feuert, wenn:

- die Kondition erfüllt ist,
- der Diagnose-Kontext erfüllt ist (falls vorhanden),
- und die Ausnahmebedingung nicht erfüllt ist (falls vorhanden).

2.3.3 Implementierungsaspekt: Das Regelkonditionen-Kompositum

Die abstrakte Klasse `AbstractCondition` bildet das Topelement aller Regelkonditionen. Sie beinhaltet die wichtigen Methoden `getTerminalObjects()` und `eval(XPSCase)`. Die Methode `getTerminalObjects()` liefert alle in dieser Kondition enthaltenen Terminale, also atomare Konditionen. Für die Auswertung der Kondition bzw. der darin enthaltenen Teilkonditionen ist die Methode `eval(XPSCase)` zuständig.

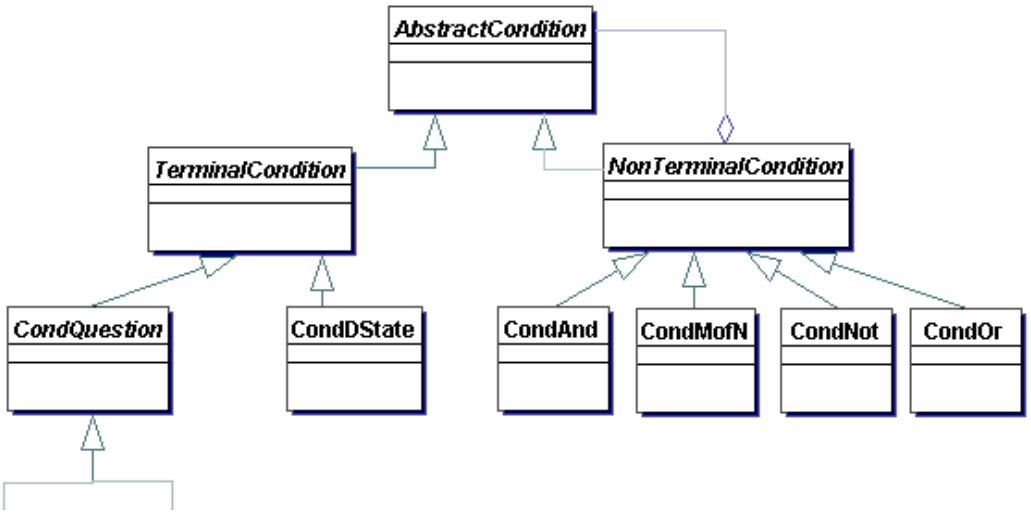

Abbildung 5: Das Regelkonditionen-Kompositum mit einigen konkreten Konditions-Implementierungen

Das `AbstractCondition`-Framework ist als Kompositum-Muster (Composite Pattern) implementiert. Dabei stellt `AbstractCondition` die Komponente (component), alle Erben der Klasse `TerminalCondition` die Blätter (leaf) und alle Nachkommen von `NonTerminalCondition` das Kompositum (composite) dar. Dieses Muster ist in Abbildung 5 dargestellt.

2.3.4 Regelaktionen als zentraler Zustand (State) von RuleComplex

Die Aktion einer Regel wird in einem Zustandsobjekt `RuleAction` gespeichert (siehe Abbildung 4). Die Methode `doIt(XPSCase)` führt die gewünschte Aktion aus. Um nicht-monotones Schließen zu unterstützen, wird die Methode `undo(XPSCase)` zur Rücknahme der Aktion angeboten.
Eine Besonderheit ist die Methode `singleFire()`: Sie steuert, ob eine Regel die spezifizierte Aktion jedes Mal ausführen darf, wenn die Regel überprüft wird, oder ob die Regel nur einmal ihre Aktion feuern darf, nämlich genau dann, wenn sie das erste Mal positiv überprüft wird. Die Default-Einstellung ist *singleFire=true*.

2.3.5 Dynamische Verzeigerung der beteiligten Objekte (Observer)

Regeln werden im allgemeinen in Abhängigkeit vom Status anderer Objekte ausgeführt und verändern üblicherweise den Status anderer Objekte. Damit bei der Veränderung eines beteiligten Objekts schnell alle davon betroffenen Regeln gefunden werden können, ist es sinnvoll, die Regeln als Beobachter (Observer) bei den entsprechenden Objekten anzumelden. In *d3web* meldet sich eine Regel bei den beteiligten Objekten (vom Typ `NamedObject`) über deren Methode `addKnowledge(..)` an. Ändert sich der Wert des `NamedObject`s, so können effizient alle angemeldeten Regeln überprüft werden. Dieser Mechanismus ist für alle Objekte sinnvoll, die in der Regelkondition enthalten sind. Für die Objekte in der Regelaktion existiert der gleiche Mechanismus. In diesem Fall ist er jedoch nicht für die effiziente Abarbeitung der Regeln gedacht, sondern dient der *Erklärung* des Zustands eines bestimmten Objekts. Will man eine Erklärung zum aktuellen Zustand eines Objekts bekommen (d.h. das Zustandekommen des aktuellen Werts nachvollziehen), dann können hierüber schnell alle Regeln angezeigt werden, welche überhaupt das Objekt beeinflussen können. Wie die interne Speicherung von Wissen in `NamedObject`s erfolgt, wird in Abschnitt 2.5 erklärt.

2.4 Problemlöser

Einige Problemlöser sind bereits im *d3web.kernel* implementiert. Diese, im Folgenden kurz erklärten, arbeiten mit Regeln. Es werden zwar für jeden Problemlöser die Regelaktionsklassen genannt, aber es sei nochmals darauf hingewiesen, dass es am sichersten ist, entsprechende Regeln über die Methoden der Klasse `RuleFactory` zu erzeugen.

`PSMethodHeuristic` ist die Implementierung eines heuristischen Problemlösers, der abhängig von Symptomwerten auf Diagnosen Punkte verteilt. Er befindet sich im Paket

`de.d3web.kernel.psMethods.heuristic`

Die Klasse für die entsprechende Regelaktion heißt `ActionHeuristicPS`.

`PSMethodNextQASet` ist die Implementierung eines Problemlösers, durch den abhängig von Symptomwerten Fragen oder Frageklassen indiziert werden können (Folgefragen). Dadurch ist ein Einfluss der Steuerung eines geführten Dialogs möglich. Er befindet sich im Paket

`de.d3web.kernel.psMethods.nextQASet`.

An dieser Stelle befindet ebenfalls ich die entsprechende allgemeine Regelaktionsklasse `ActionNextQASet`. Davon erbend existieren spezielle Indikationsklassen:
`ActionClarify` zur Klärung von Diagnosewerten, `ActionIndication` zur Indizierung von Folgefragen und `ActionRefine` zur Bestätigung etablierter Diagnosen.

`PSMethodContraIndication` stellt einen Problemlöser dar, der abhängig von Symptomwerten Fragen und Frageklassen blockieren kann, so dass diese in einem geführten Dialog nicht gestellt werden. Er befindet sich im Paket

`de.d3web.kernel.psMethods.contraIndication`

Die entsprechende Regelaktion wird durch `ActionContraIndication` implementiert.

`PSMethodQuestionSetter` ist ein Problemlöser über den in Abhängigkeit bestimmter Symptomwerte Werte anderer Symptome setzen oder addieren kann. Er wird im Allgemeinen zur Merkmalsabstraktion verwendet. In abstrakter Weise wird die entsprechende Regelaktion durch `ActionQuestionSetter` dargestellt.
Von dieser gibt es zwei Ausprägungen: `ActionSetValue` zum Setzen von Symptomwerten und `ActionAddValue` zum Addieren neuer Werte auf die bereits vorhandenen (was für numerische und Choice-Werte sinnvoll ist). Die entsprechenden Klassen befinden sich im Paket

`de.d3web.kernel.psMethods.questionSetter`.

`PSMethodSuppressAnswer` repräsentiert einen Problemlöser, der abhängig von Symptomwerten Antwortalternativen von Choice-Fragen unterdrücken kann. Die entsprechende Regelaktionsklasse `ActionSuppressAnswer` befindet sich, wie auch der Problemlöser im Paket

`de.d3web.kernel.psMethods.suppressAnswer`.

Dieser Problemlöser wird im Dialog zur Reduzierung von Antwortalternativen und damit zur Optimierung des Dialogs eingesetzt.

2.5 Zugriff auf Wissen in *d3web*

Jedes Wissensbasisobjekt, das von `NamedObject` erbt, wie zum Beispiel `Question` oder `Diagnosis`, enthält intern gespeichertes Wissen. Welche Art von Wissen dort gespeichert ist und wie es erreichbar ist, wird in diesem Abschnitt ausführlich behandelt.

2.5.1 Interne Speicherung von Wissen

Allgemein wird Wissen in *d3web* durch Klassen repräsentiert, die das Interface `KnowledgeSlice` implementieren. Häufig wird Wissen in Form von Regeln gespeichert (`RuleComplex` implementiert `KnowledgeSlice`). Es kann sich aber auch um Zusatzwissen wie z.B. Gewichte (`Weight`), Ähnlichkeiten (`QuestionComparator`) oder Abnormitäten (`Abnormality`) aus dem Projekt *d3web-SharedKnowledge* handeln, die hier allerdings nicht weiter behandelt werden.
In einem `NamedObject` wird Wissen in einer `Map` gespeichert. Deren Key ist der Problemlöser-Kontext (`Class`-Objekt des entsprechenden Problemlösers) und der zu diesem Key gehörige Wert ist eine weitere `Map`. Diese zu einem Problemlöser-Kontext gehörige `Map` hat Keys vom Typ `MethodKind` und als Werte jeweils eine Liste von dazu verfügbaren `KnowledgeSlices`. Bei Regeln kann der `MethodKind` `FORWARD` für vorwärts verkettetes oder `BACKWARD` für rückwärts verkettetes Wissen sein. Was das bedeutet wird im nächsten Unterabschnitt anhand eines einfachen Beispiels erklärt. Handelt es sich nicht um eine Regel, kann es auch andere `MethodKind`-Instanzen geben. Allgemein sieht der Aufbau der Speicherstruktur in einem `NamedObject` wie in Abbildung 6 skizziert aus.

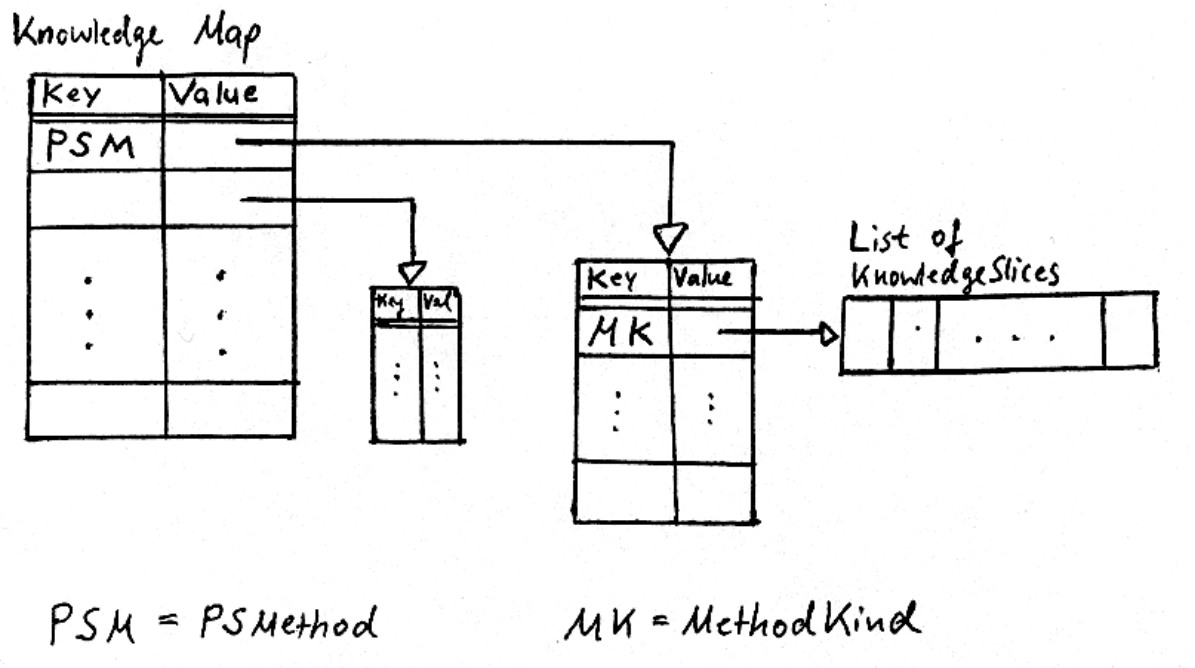

Abbildung 6: Interne Speicherung von Wissen in einem `NamedObject`

`NamedObject` besitzt Methoden zum Hinzufügen und Abfragen von Wissen:

`addKnowledge(Class context, KnowledgeSlice slice, MethodKind kind)` fügt ein `KnowledgeSlice slice` zu einem PSMethod-Kontext `context` mit gegebenem `MethodKind kind` ein.

`getKnowledge(Class context, MethodKind kind)`
liefert eine Liste von `KnowledgeSlices` zum spezifizierten PSMethod-Kontext `context` und dem angegebenen `MethodKind kind`.

Im nächsten Unterabschnitt werden diese Methoden im Beispiel nochmals verdeutlicht.

2.5.2 Ein Beispiel anhand einer Regel

Im Beispiel aus Abbildung 7 wird eine heuristische Regel *r1* mit der Frage *Auspuffrohr-Farbe (QAuspuff)* und der Diagnose *Luftfilter verschmutzt (DLuftfilter)* betrachtet. Weil es sich hier um eine heuristische Diagnoseregel handelt, wird *r1* in beiden `NamedObjects`, *QAuspuff* und *DLuftfilter* in die `Map` unter heuristischem Wissen aufgenommen. Daher ist der verwendete Key `PSMethodHeuristic.class`. Die Frage *QAuspuff* kommt in der Kondition von *r1* vor, die Diagnose *DLuftfilter* in der Aktion von *r1*. Deshalb wird in der `MethodKind-Map` der Frage `QAuspuff` die Regel *r1* unter `MethodKind.FORWARD` eingetragen und in der entsprechenden `Map` der Diagnose `DLuftfilter` unter `MethodKind.BACKWARD`.

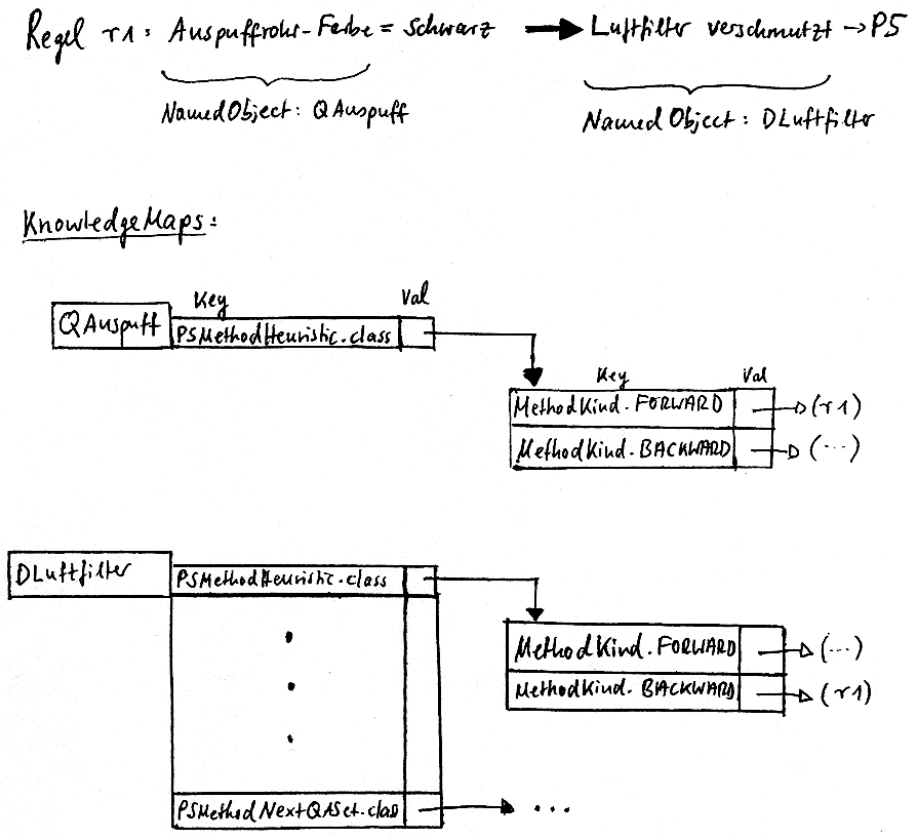

Abbildung 7: Wissensspeicherung in `NamedObject` - Beispiel anhand einer Regel *r1*

2.6 Fälle

Ein Fall ist gleichbedeutend mit dem Durchspielen eines Problems, d.h. Eingabe von Antworten auf Fragen und Herleiten von Lösungen (Diagnosen). Die das Interface `XPSCase` implementierende Klasse `D3WebCase` welche die zentralste Klasse des *d3web.kernel* ist, repräsentiert einen Fall. Über diesen erfolgt die Steuerung von neuen Werten, die Wissensbasisobjekte während des Durchspielens eines Falls erhalten können.

Erzeugung eines neuen Falles
Die Standardmethode zur Erzeugung eines neuen `XPSCase` ist die Verwendung der Factory-Klasse `CaseFactory`. Diese bietet verschiedene Erzeugungsmethoden; unter anderem die Methode `createXPSCase(KnowledgeBase)`. Wie man ein `KnowledgeBase`-Objekt erzeugen, also eine Wissensbasis laden kann, wird im nächsten Abschnitt erklärt.

Setzen von Werten über den `XPSCase`
Damit das Setzen von Werten (z.B. das Beantworten von Fragen oder Addieren von Diagnosepunkten) zentral gesteuert und propagiert werden kann, geschieht es über den `XPSCase`. So werden benutzerabhängige Werte kontrolliert gesetzt und sichergestellt, dass alle Regeln, deren Konditionen von diesen Änderungen betroffen sind, überprüft werden. Um Werte zu setzen, verwendet man eine der folgenden Methoden:

```
setValue(ValuedObject o, Object[] ans),
setValue(ValuedObject o, Object[] ans, Class c),
setValue(ValuedObject o, Object[] ans, Class c, RuleComplex r).
```

Hierbei ist das `ValuedObject` das betreffende Wissensbasisobjekt (z.B. Question).
Das `Object`-Array muss die zu setzenden Werte enthalten (z.B. `Answers`). Dieser Parameter wird als Array übergeben, da beispielsweise eine MC-Frage mehr als eine Antwort erhalten kann. Zusätzlich kann noch ein Problemlöser-Kontext `c` angegeben werden, so dass z.B. beim Setzen von Diagnosewerten ein vom Problemlöser abhängiger Wert gesetzt wird. Desweiteren ist es möglich eine Regel `r` zu übergeben. Das ist vor allem dann wichtig, wenn man sich merken will, durch welche Regeln welche Symptomwerte überschrieben wurden. Dazu gibt es im `domainModel`-Paket die Klasse `SymptomValue`. Sie realisiert ein Paar, das aus einer Regel und dem Symptomwert besteht, den ihre Aktion überschrieben hat. In `CaseQuestion` im `domainModel`-Unterpaket `dynamicObjects` wird eine History zu diesen Paaren gehalten.

3 Eine Beispielapplikation

In diesem Abschnitt werden die üblichen Mechanismen bei der Benutzung des *d3web.kernel* erklärt. Es wird jeweils Beispielcode angegeben, der zusätzlich andeutet, welche Klassen bevorzugt verwendet werden sollten.
Zu Beginn wird gezeigt, wie eine Wissensbasis geladen und gespeichert werden kann. Im darauffolgenden Abschnitt werden Möglichkeiten zum Erzeugen von neuen Fällen aufgezeigt. Anschließend wird demonstriert, wie man vom Fallobjekt an die Wissensbasis gelangt und aus dieser Wissensbasisobjekte abfragt. Im letzten Unterabschnitt werden die wesentlichen Schritte zum Laden und Speichern von *d3web*-Fällen erklärt.

3.1 Laden und Speichern von Wissensbasen

Im Projekt *d3web-Persistence* sind einige Klassen zum Laden und Speichern von Wissensbasen enthalten. Die Wesentliche dieser Klassen ist der `PersistenceManager` im Paket

`de.d3web.persistence.xml`.

Mit seiner `load(URL)`-Methode wird eine Wissensbasis, die als *jar*-Archiv vorliegen muss, von der angegebenen URL geladen, und mit `save(KnowledgeBase, URL)` wird eine Wissensbasis an die spezifizierte Stelle als jar-Archiv geschrieben. Die jar-URLs sind von folgender Struktur:

jar:<protokoll>:<pfad-zum-jar>!/

Also zum Beispiel *jar:file://knowledgebases/kfz.jar!/*.
Es wird automatisch eventuell vorhandenes Zusatzwissen mitgeladen und geschrieben, wenn entsprechende `PersistenceHandler` implementiert und im `PersistenceManager` mittels `addPersistenceHandler(PersistenceHandler)` registriert wurden. Standardmäßig registriert ist der `BasicPersistenceHandler`, der das Basiswissen (Diagnosen, Fragen, Regeln,...) lädt und speichert. Der Code zum Laden einer Wissensbasis sieht z.B. so aus:

```
PersistenceManager pm = new PersistenceManager();
KnowledgeBase kb = pm.load(new URL("jar:file:/c:/.../kbases/kfz.jar!/"));
```

3.2 Erzeugen und Bearbeiten eines neuen Falls

Grundsätzlich steht zum Erzeugen eines neuen Fallobjekts die Fabrik-Klasse `CaseFactory` aus dem Paket

`de.d3web.kernel.domainModel`

zur Verfügung. Um einen leeren Fall zu erzeugen, sollte man die statische Methode

`createXPSCase(KnowledgeBase)`

verwenden, die eine Instanz von `D3WebCase` liefert.
Eine weitere Möglichkeit bietet die Methode

`createAnsweredXPSCase(KnowledgeBase, Class, DialogProxy)`.

Sie liefert einen bereits vorbelegten Fall, der mit allen im `DialogProxy` enthaltenen `Answer`-Objekten beantwortet wurde. Der `Class`-Parameter bestimmt dabei den zu verwendenden `DialogController`-Typ. Üblicherweise werden die Instanzen

`OQDialogController.class` oder `MQDialogController.class`

verwendet. Ein `DialogProxy` ist ein Stellvertreter-Objekt (siehe Proxy-Entwurfsmuster), das zu einer gegebenen Frage-ID nach Antworten sucht. Dabei fragt der Proxy alle angemeldeten Clients (z.B. Datenbank, Dialog,...) und liefert die zuerst gefundene Antwort zurück (`null`, falls keine Antwort gefunden wurde). Die Clients sind vom Typ `DialogClient`. Für die meisten Anwendungen sollte der Client `ShadowMemory` genügen, der die ihm zugefügten Objekte im Arbeitsspeicher hält. Die folgenden Beispiel-Zeilen zeigen, wie man einen vorbelegten Fall generieren kann:

```
DialogProxy proxy = new DialogProxy();
ShadowMemory shm = new ShadowMemory();
shm.addAnswers("Frage1-ID ", answerList);
proxy.addClient(shm);

XPSCase newCase =
  CaseFactory.createAnsweredXPSCase(kb, MQDialogController.class, proxy);
```

Möchte man Fragen manuell beantworten oder auch andere Werte (wie z.B. Diagnose-Punkte) setzen, verwendet man die `setValue(..)`-Methoden, die in Abschnitt 2.6 beschrieben werden. In diesem Beispiel beantworten wir eine Frage `question1` der Antwort `answer1` ohne Angabe eines speziellen Problemlöser-Kontexts (also wird per default `PSMethodHeuristic.class` verwendet):

```
newCase.setValue(question1, new Object[]{answer1});
```

3.3 Abfragen von Informationen aus dem Fall

Wenn man einen Fall, also eine Instanz von `XPSCase` vorliegen hat, kann man daraus auf einige Informationen zugreifen. Die wesentlichen werden in diesem Abschnitt vorgestellt.
Die zum Fall gehörige Wissensbasis erreicht man über `getKnowledgeBase()`. In dieser kann man z.B. nach Wissensbasis-Objekten suchen (`search(..)`-Methoden).
Desweiteren können über `getAnsweredQuestions()` die bereits beantworteten und über `getQuestions()` alle Fragen abgefragt werden. Ebenso gelangt man unter Verwendung von `getDiagnoses()` an alle Diagnosen. Möchte man nur Diagnosen mit bestimmtem Status erhalten, sollte man die Methode `getDiagnoses(DiagnosisState)` verwenden. Diese liefert alle Diagnosen, die bezüglich irgendeines Problemlösers den angegebenen Status bekommen haben. Hat man über eine der `search...()`-Methoden ein Wissensbasisobjekt erhalten, kann man aus diesem mittels `getKnowledge(..)` Wissen abfragen (z.B., Regeln durch die eine Diagnose ihre Punkte bekommen hat). Im folgenden Beispielcode wird gezeigt, wie man Wissensbasis-Objekte suchen und aus ihnen Wissen abfragen kann.

```
KnowledgeBase kb = newCase.getKnowledgeBase();
Diagnosis diag = kb.searchDiagnoses("diag1-ID");
List rules = (List)
  diag.getKnowledge(PSMethodHeuristic.class, MethodKind.BACKWARD);
```

Hier wird zuerst die Wissensbasis aus dem Fallobjekt geholt und aus dieser durch Angabe der ID einer Diagnose das entsprechende `Diagnosis`-Objekt mit `searchDiagnosis(String id)` gesucht. Anschließend werden alle heuristischen Herleitungsregeln aus dieser Diagnose

abgefragt (daher `MethodKind.BACKWARD`).
Wie das Wissen in den Objekten gespeichert ist und welche Möglichkeiten zur Abfrage bestehen, wird in Abschnitt 2.5 ausführlich beschrieben.

3.4 Laden und Speichern von Fällen

Durch die Klassen im Projekt *d3web-CaseRepository* besteht die Möglichkeit, *d3web*-Fälle zu laden und zu speichern. Es ist empfehlenswert, dazu die Klasse `SAXCaseRepositoryHandler` aus dem Paket

`de.d3web.caserepository.sax`

zu verwenden, wie es im folgenden Beispielcode gezeigt wird.

Erzeugen des Handlers

```
SAXCaseRepositoryhandler handler = new SAXCaseRepositoryhandler();
```

Laden von Fällen

```
List caseList = handler.load(kb, new URL("jar:file:/c:/.../cbr/cases.jar!/"));
```

Nun liegt eine Liste von `CaseObject`s vor. Nachdem man ein solches `CaseObject` aus der Liste entnommen hat, kann man sich mit `getQuestions()` die Fragen und dann über `getAnswers(Question)` die Antworten zu einer Frage holen. Damit können dann z.B. Clients für den `DialogProxy` gefüllt werden (siehe Abschnitt 3.2).

Speichern von Fällen
Durch den Aufruf:

```
handler.saveToFile(new File("cases.xml"), caseList);
```

Werden die angegebenen Fälle in das XML-File *cases.xml* geschrieben. Möchte man aus dem Fallspeicher ein DOM-Dokument, also ein `org.w3c.dom.Document` generieren, so verwende man folgenden Aufruf:

```
Document doc = handler.save(caseList);
```

Allerdings kann die Erzeugung einer DOM-Struktur bei großen Fallspeichern sehr lange dauern.

4 XML-Repräsentation von Wissen in *d3web*

In diesem Abschnitt werden die XML-Schemata zu Wissensbasis- und Fallspeicher-XML-Dokumenten vorgestellt.

4.1 XML-Schema für Wissensbasen

In Abbildung 8 ist das XML-Schema für Wissensbasen (Basiswissen) graphisch dargestellt.

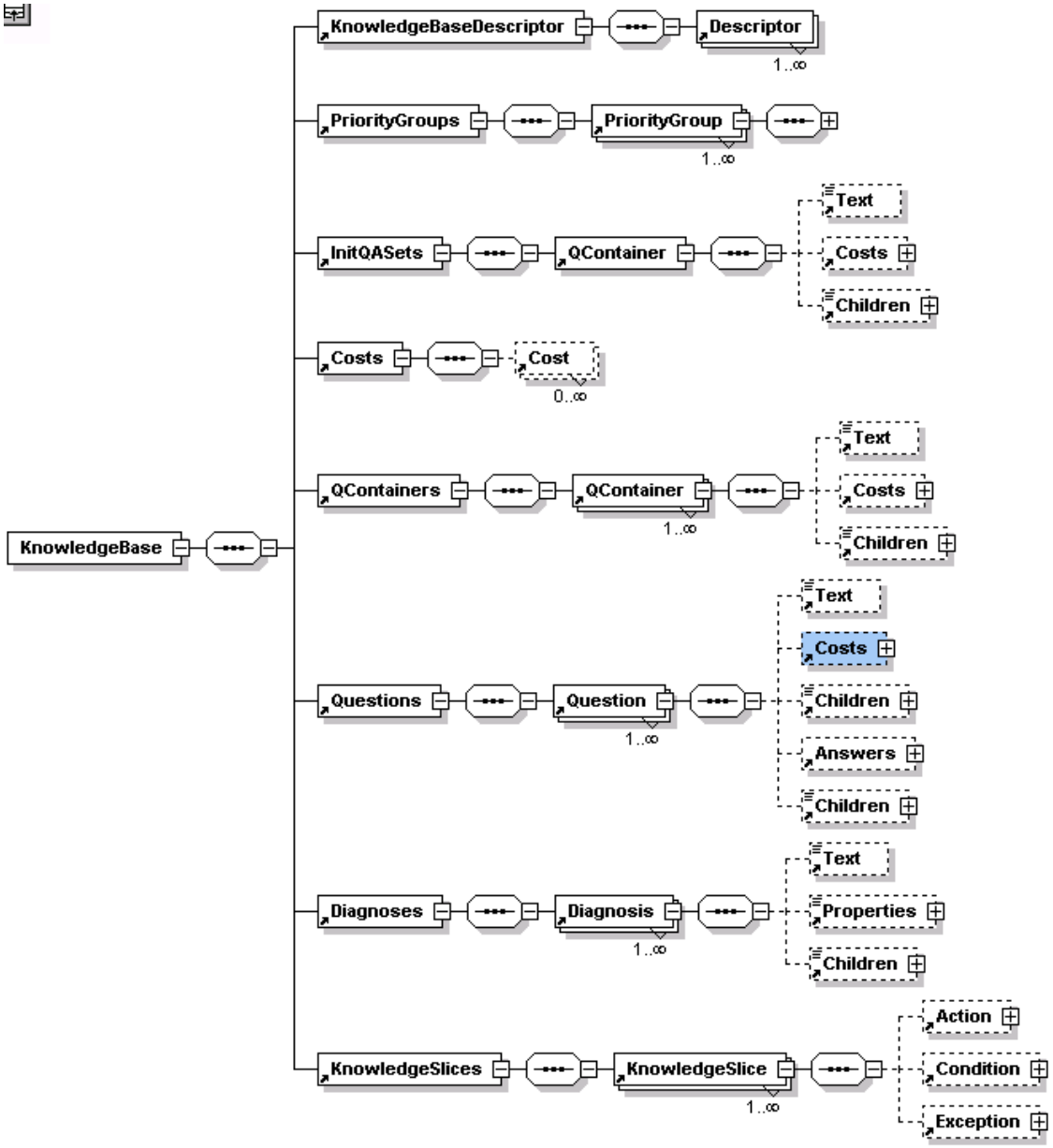

Abbildung 8: Graphische Darstellung des XML-Schemas für Wissensbasen

KnowledgeSlice ist meistens vom Typ *RuleComplex* (*type='RuleComplex'*). Nur dann hat *KnowledgeSlice* die Untertags *Action, Condition* und *Exception*. Es kann aber auch vorkommen, dass

KnowledgeSlice vom Typ *Num2ChoiceSchema* (*type='schema'*) ist. Dann hat es die Untertags *Question* und *LeftClosedInterval*.

4.2 XML-Schema für Fallspeicher

In Abbildung 9 ist das XML-Schema für Fallspeicher graphisch dargestellt.

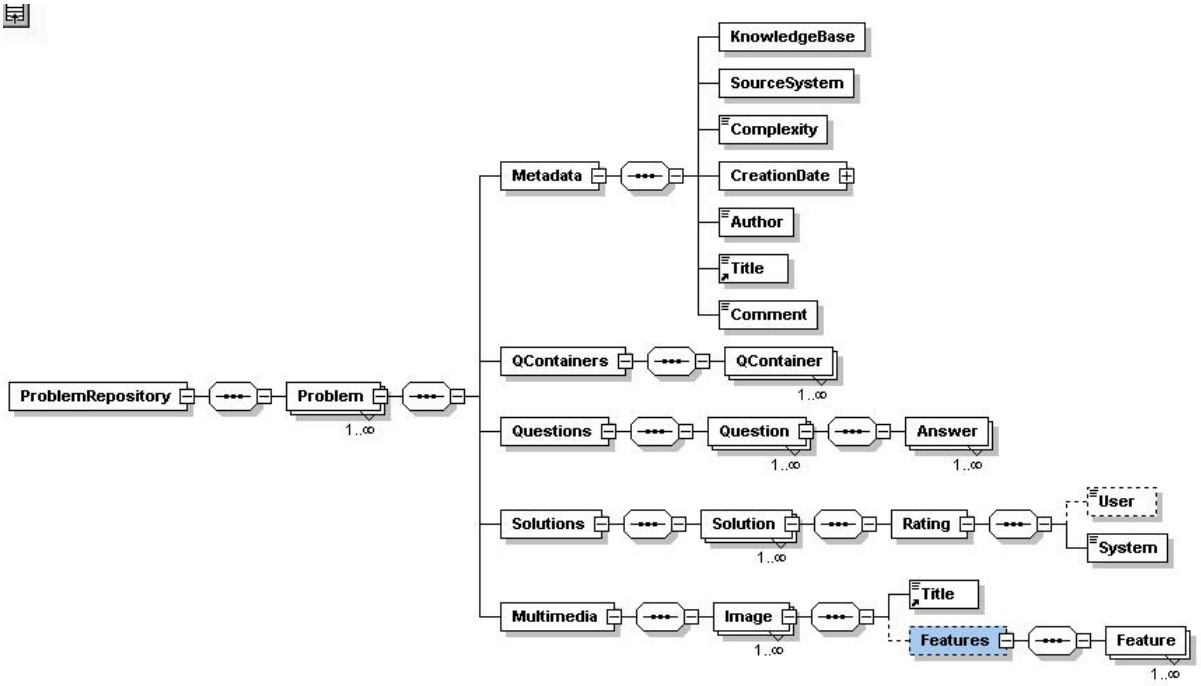

Abbildung 9: Graphische Darstellung des XML-Schemas für Fallspeicher

Ein Fall wird durch den Tag *Problem* dargestellt. *Metadata* steht für die Metadaten zum Fall (z.B. wer ihn erstellt hat oder zu welcher Wissensbasis er gehört).
QContainers enthält Frageklassen (Tag *QContainer*), *Questions* repräsentiert Fragen (Tag *Question*) und die wiederum Antworten (*Answer*). Diagnosen werden durch den Tag *Solution* dargestellt und unter *Solutions* gesammelt. Es werden nur Wissensbasis-Objekte gespeichert, die (bestimmte) Werte im Fall bekommen haben. Das bedeutet, das nur beantwortete Fragen und nicht-unklare Diagnosen (also `DiagnosisState` ist *excluded, suggested* oder *established*) gespeichert werden.
Unter *Multimedia* können Multimadia-Informationen (z.B. URLs zu Bildern im Fallspeicher-jar (Tag *Image*)) gespeichert werden.

5 Implementierung eines Problemlösers für d3web

In diesem Abschnitt wird gezeigt, welche Schritte notwendig sind, um einen neuen Problemlöser für *d3web* zu schreiben.

5.1 Erstellen der notwendigen Klassen und Anmelden des neuen Problemlösers

Um den Problemlöser in den *d3web-kernel* integrieren zu können, muss dieser als eine das Interface *PSMethod* implementierende Klasse geschrieben werden. Eine starke Vereinfachung bietet der `PSMethodAdapter`, der für die meisten Methoden bereits eine Default-Implementierung bereitstellt. Sowohl `PSMethod` als auch der `PSMethodAdapter` sind im Paket

`de.d3web.kernel.psMethods`

enthalten. Leitet man seinen Problemlöser von `PSMethodAdapter` ab, müssen nur zwei wichtige Methoden implementiert werden, die nun näher erläutert werden:

`public DiagnosisState getState(XPSCase, Diagnosis)`
: Diese Methode bestimmt, inwiefern die Diagnosepunkte auf Diagnosestati abgebildet werden sollen. Mehr Informationen dazu findet man in Abschnitt 2.2.

`public void propagate(XPSCase, NamedObject, Object[])`
: Ändert sich der Wert eines `NamedObject` (z.B. Frage oder Diagnose), so ruft der Fall (`XPSCase`) diese Methode bei jedem angemeldeten Problemlöser auf, so dass jeder Problemlöser für sich entscheiden kann, was aufgrund dieser Wertänderung geschehen soll. Beim heuristischen Problemlöser beispielsweise werden alle beteiligten Regeln überprüft und somit der Feuer-Mechanismus angestoßen.

Soll der neue Problemlöser mit Regeln arbeiten, muss eine Regelaktion definiert werden. Diese wird durch die Klasse `RuleAction` dargestellt. Von dieser Klasse muss man die neue Regelaktionsklasse ableiten und die Methoden `doIt(XPSCase)` und `undo(XPSCase)` implementieren, die beschreiben, was geschehen soll, wenn eine Regel mit dieser Aktion feuert, bzw. zurückgezogen wird. Mehr Informationen zu Regeln und Aktionen sind in Abschnitt 2.3 zu finden.
Um den neuen Problemlöser im Fall anzumelden, so dass dieser über das Bekanntwerden von neuen Werten (z.B. Etablieren einer Diagnose oder Beantworten einer Frage) benachrichtigt wird, muss die `D3WebCase`-Methode `addUsedPSMethod(PSMethod)` verwendet werden. Diese hängt den Problemlöser einfach an die interne Liste der zu verwendenden Problemlöser an.

5.2 Paketstruktur

Es ist sinnvoll, ein Unterpaket von

`de.d3web.kernel.psMethods`

zu erzeugen und dort den Problemlöser sowie die Regelaktionsklassen unterzubringen. Dies wäre dann konform zur bereits bestehenden Problemlöser-Paketstruktur im *d3web.kernel*.