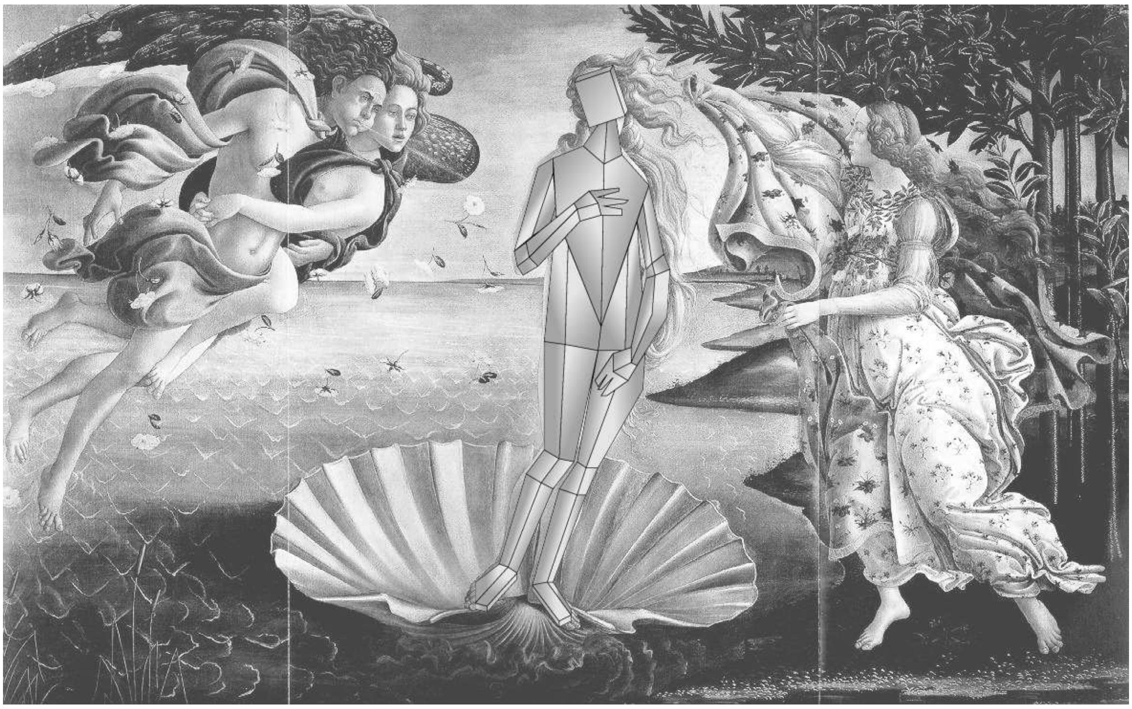
Teil II

Benutzungshandbuch BotIshelly



Inhaltsangabe Teil II

Kapitel 1

Einleitung und Überblick

Nicht Kunst und Wissenschaft allein,
Geduld will bei dem Werke sein.
(Mephisto)

Johann Wolfgang Goethe
Faust I, Hexenküche

Bei BotIshelly handelt es sich um eine Java–Klassenbibliothek, mit deren Hilfe Sie eine Hülle (engl. *shell*) für einen Internetagenten, auch *WebBot* genannt, zusammenbauen können. Die Bestandteile dieser Hülle nehmen Ihnen schon sehr viel Verwaltungsarbeit ab, die für die meisten Einsatzgebiete gleich ist, sind aber darauf angewiesen, daß Sie die Shell mit dem für Ihr Gebiet wichtigen Wissen und den notwendigen Werkzeugen versehen. Nur dadurch kann aus der Shell auch ein Agent werden.

Wenn Sie z. B. einen Agenten für die Suche nach bestimmten Softwarepaketen schreiben möchten, so müssen Sie zum einen den Bereich beschreiben und für den Agenten verständlich modellieren, in dem die Suche laufen soll. Das kann die Modellierung der Webseiten von Universitäten sein, oder vielleicht die Seiten von Softwareherstellern. Zum anderen müssen Sie den Agenten mit Werkzeugen (*Operatoren*) ausstatten, um Webseiten analysieren und verarbeiten zu können. So kann ein Operator, der Informationen aus den Seiten einer Universität holt, sich völlig von einem unterscheiden, der auf kommerziellen Seiten arbeitet. Generell gilt, je allgemeiner Sie Ihren Agenten gestalten wollen, desto größer ist der Aufwand, den Sie in die Modellierung des Weltwissens und der Operatoren stecken müssen!

Um eine Suchanfrage zu bearbeiten, verfolgt BotIshelly ein einfaches und intuitives Ablaufschema, das von Ihnen bei Ihren eigenen Recherchen im Internet vielleicht auch schon angewandt worden ist. In der nachstehenden Tabelle werden diese Schritte direkt Klassen von BotIshelly zugeordnet, die selbst (oder davon abgeleitete Klassen) für die entsprechende Bearbeitung zuständig sind:

1.	Entgegennehmen der Benutzeranfrage	`io.InputmaskServlet`
2.	Auswerten und Strukturieren der Benutzeranfrage	`informationexchange.PlanInformation`
3.	Erstellen eines Plans	`planner.Planner`
4.	Ausführen des Plans	`planner.PlanExecution`
5.	Auswerten und Strukturieren der Ergebnisse	`informationexchange.PlanInformation`
6.	Rückgeben der Ergebnisse an den Benutzer	`io.OutputmaskServlet`
7.	(evtl.) Lernen aus der Suche	`learners.Learner`

Das `InputmaskServlet` hat neben der Eingabeanforderung vom Benutzer noch eine zweite, wichtige Aufgabe: das Erzeugen und Starten einer Instanz der Klasse `systemcontrol.FlowControl`. Die `FlowControl` realisiert die zentrale Systemkontrolle für eine Suche und ist u. a. zuständig dafür, daß Planer und Planausführer erzeugt, mit Informationen versehen und angestoßen werden, sowie für die Verwaltung der zur Verfügung stehenden Systemresourcen. Die Hauptklassen von BotIshelly, also Klassen, die direkt von der Systemkontrolle instantiiert werden, wie `Planner` und `PlanExecution`, kommunizieren nie unmittelbar miteinander, sondern immer durch das Eventhandling der Systemkontrolle

Einige wichtige Klassen, ohne die wohl kaum eine Suche Ihres Agenten Erfolg haben wird, sind in obigem Ablaufschema nicht genannt. Diese Klassen beinhalten das oben angesprochene Weltwissen, sowie die Operatoren, um mit der Welt, die Sie beschrieben haben, umgehen zu können. Dabei müssen Sie das Wissen in die Klassen `knowledge.T_Box` und `knowledge.A_Box` nur füllen, wohingegen Sie die Operatoren aus der Klasse `operators.Operator` tatsächlich ableiten und implementieren. Da die Operatoren oft Seiten oder Dateien aus dem Internet holen oder auch klassifizieren müssen, gibt es noch zwei Untergruppen von Klassen, die teilweise Bestandteil von BotIshelly sind, teilweise von Ihnen bearbeitet oder gar neu geschrieben werden müssen, abhängig vom Aufgabengebiet Ihres Agenten. Zur ersten Gruppe gehören die Klassen, die Daten aus dem Internet beschaffen. Zu diesen zählen z. B. die Hierarchie der von `dataprovider.net.NetService` abgeleiteten Klassen, die eine beliebige URL aus dem Netz laden, oder die zu `dataprovider.search.SearchService` gehörenden Ableitungen, die auf die Abfrage von bestimmten Suchmaschinen spezialisiert sind. Zur zweiten Gruppe gehören die Klassen, die Objekte der Welt (d. i. die Suchdomäne), z. B. URLs, klassifizieren können. Diese Klassen leiten sich aus `dataanalysis.classifiers.Classifier` ab.

Dieses Handbuch stellt Ihnen eine Beschreibung der Klassen von BotIshelly zur Verfügung, die über die Javadoc–Dokumentation[1] hinausgeht. Sie sollten die einzelnen Kapitel alle einmal gelesen haben, bevor Sie sich daran begeben, einen eigenen Agenten zu entwickeln. Um Ihnen das Wiederauffinden von wichtigen Textstellen zu erleichtern, benutzen wir drei Arten von Randmarkierungen:

1. **Unbedingt notwendig!**
   In der Textpassage werden Vorgänge beschrieben, die Sie unbedingt machen müssen, wollen Sie einen Agenten schreiben. Dabei kann es sich um Kleinigkeiten, wie das Setzen einer Variable handeln, oder um so komplexe Dinge, wie das Füllen der T–Box.

2. **Achtung!**
   Dies Icon markiert Stellen, an denen Sie etwas beachten müssen. So sind z. B. Hinweise darauf markiert, was passieren kann, wenn Sie an einer bestimmten Variablen drehen.

3. **Ich will mehr!**
   Diese dritte Markierung schließlich zeigt Ihnen, an welchen Stellen von BotIshelly Sie ansetzen können, um tiefergehende als die Standardeingriffe am System vorzunehmen. Dazu zählen z. B. die Erstellung zusammengesetzter Operatoren, oder die Neuentwicklung eines Planers.

Wenn es wichtig ist, zwischen Instanzen und Klassen zu Unterscheiden, benutzen wir eine Kursivschrift für *`Java-Instanzen`*, eine Schreibmaschinenschrift für `Klassen`. Quellkode und Methodennamen sind auch in Schreibmaschinenschrift gehalten.

Wir haben uns dazu entschieden, Sie in diesem Handbuch entweder direkt oder als *Shellbenutzer* anzureden. Mit „Shellbenutzer“ meinen wir eine Person, die die Bilbiothek BotIshelly dazu benutzt, einen Agenten zu schreiben. Im Gegensatz dazu bezeichnen wir Personen, die den fertigen Agenten später benutzen, als *Agentenbenutzer*.

[1] In der Javadoc–Dokumentation finden sie eine ausführliche Beschreibung der einzelnen Klassen und ihrer Schnittstellen.

Kapitel 2

Benötigte Tools und Libraries Dritter

Zu einer Revolution der deutschen Arbeiter kommt es wahrscheinlich erst, wenn Neckermann die aufblasbare Taschenbarrikade anbietet.
Binsenweisheit

2.1 Tools

Java Die Programmierung und die Ausführung des Java-Codes erfolgten mit dem Java 2 SDK, Version 1.2.1. Unter [Sun, 2000] findet sich hier stets die aktuelle Version. Grundsätzliches über Java beantwortet die Site [Javasoft, 2000].

JSDK Das Java Servlet Development Kit wird von uns in der Version 2.0 eingesetzt. Verwendet wird es von der Systemkontrolle/IO, um die Ausgaben zum Browser via Servlets zu realisieren. Weitere Informationen dazu gibt es unter [jwebserver, 2000].

2.2 Libraries

Java Network FTP Library Die Java Network FTP Library wird von der Datenbeschaffung benutzt, um den FTPNetService zu realisieren. Sie unterliegt der GPL und ist zu finden unter [Sheng–Te, 2000]. Eine Versionsnummer ist nicht angegeben.

COM.STEVESOFT.PAT Dieses Pakte wird ebenfalls von der Datenbeschaffung benutzt, das Reguläre Ausdrücke unter Java ermöglicht. Wir haben die Version 1.3.2 eingesetzt; aktuelle Updates scheint es nicht mehr zu geben. Die URL [Stevesoft, 1998] bietet neben verschiedenen Tutorials und Beispielprogrammen auch Zugriff auf das Paket.

2.3 Benötigte Hardware

Für BotIshelly wird folgende Hardware empfohlen:

UltraSparc 10 mit 300Mhz Taktfrequenz

128 MB Hauptspeicher

500MB freier Plattenplatz

Auf anderen Plattformen wurde das korrekte Arbeiten der Shell nicht getestet. Für den Multiuser-Betrieb wird erheblich mehr Hauptspeicher benötigt.

2.4 Installation der Shell

Um BotIshelly zu installieren, führen Sie bitte folgende Schritte durch:

Zunächst wird ein installiertes JDK (Java Developement Kit) ab Version 1.2.0 benötigt (Java 2). Um die vorgegebene Ein-/Ausgabeschnittstelle von BotIshelly zu benutzen, wird außerdem das JSDK2.0 (Java Servlet Developement Kit) benötigt. Beides ist auf `http://java.sun.com` erhältlich.

Nun müssen die Dateien der Shell in ein beliebiges Verzeichnis kopiert werden. Dafür bietet sich z. B. das Homeverzeichnis an. Es entsteht ein Verzeichnisbaum, dessen oberstes Verzeichnis `sourcen` heißt. Der Java-Classpath muß nun zusätzlich auf das Verzeichnis `Shell` zeigen, welches ein Unterverzeichnis von `sourcen` ist, damit der Java-Compiler die entsprechenden Dateien findet. Ist Ihr Homeverzeichnis `/home/foo` und haben sie dort BotIShelly ausgepackt, so muss der Classpath zusätzlich den Pfad `/home/foo/sourcen/Shell` enthalten.

Sie sollten sich nun im Verzeichnis `sourcen` ein Unterverzeichnis anlegen, in dem sich nur die modifizierten Java-Dateien der Shell befinden. Dieses könnte z.B. den Namen `MyShell` tragen. So können Sie sicherstellen, daß die Java-Dateien der Shell unberührt bleiben. Sie müssen dann allerdings sicherstellen das der Java-Classpath nun zuerst auf `MyShell` zeigt und danach erst auf `Shell`. So berücksichtigt der Java-Compiler erst die Java-Dateien aus ihrer angepassten Shell und danach aus BotIshelly. Dies hat den Vorteil, das sie mehrere Agenten parallel entwickeln können, ohne die Übersicht zu verlieren.

Im Verzeichnis Shell befindet sich auch noch ein `Makefile`. Die Entwickler von BotIshelly haben folgende Befehle bereits implementiert: `make all` compiliert alle Java-Dateien. `make run` startet den Servletrunner. `make newrun` tut das selbe wie `make run`, mit dem Unterschied, daß zusätzlich das Logfile gelöscht wird.

Danach kann der Agent getestet werden, in dem in einem Javascipt-fähigem Webbrowser die URL `http://rechnername:8080/servlet/agent` aufgerufen wird. Wichtig ist, daß die Dateien `Shell/config.sys` und `Shell/io/servlet.properties` angepaßt werden, da diese hartcodierte Pfadangaben enthalten.

Kapitel 3

Systemkontrolle und I/O

Was wär ich
ohne dich,
Freund Publikum?
All mein Empfinden Selbstgespräch,
all meine Freude stumm.
Johann Wolfgang Goethe
Der Autor

3.1 Aufgabe und Definition

Da I/O und Systemkontrolle stark integriert sind, werden sie in einem Kapitel beschrieben. Die I/O ist die Schnittstelle zum Agentenbenutzer, d.h. alle Interaktionen zwischen dem Benutzer und dem Agenten werden nur über die `I/O` stattfinden. Die Aufgabe der `I/O` ist es, Eingaben vom Benutzer entgegenzunehmen, sie zur weiteren Verarbeitung an die zuständigen Klassen weiterzuleiten, die Ergebnisse zur Ausgabe vorzubereiten und ausgeben. Außerdem sollte der Benutzer über die Fehler in der Eingabe oder über das Scheitern der Suche in Kenntnis gesetzt werden.

Die Systemkontrolle stellt die Kommunikation zwischen verschiedenen Programmteilen wie z.B. dem Plann[er und der Planausführung sicher, verwaltet die Suchergebnisse, verteilt die Plan- und SuchIDs und stellt das Eventhandling zur Verfügung.

3.2 Funktionsbeschreibung

Initialisierung und Aufbau der Eingabemaske

Der Suchagent wird gestartet, indem der Agentenbenutzer die entsprechende URL in sein Broeser eingibt. Damit wird automatisch der `InputmaskServlet` initialisiert, der über die Methode `createSessionObjects` in der `Flowcontrol` alle für die Suche notwendigen Objekte wie Instanzen von `Input`, `StartConcept`, `T_Box`, `A_Box`, `OperatorDB`, `Results`, `Output`, `ClassifiererDB` und `Planarchive` instantiiert, und sie in das MainTable-Objekt schreibt, falls MainTable noch leer ist, d.h. die gerade erzeugte Flowkontroll ist die erste und einzige in diesem Moment. Sind alle Objekte in MainTable bereits vorhanden, werden sie ausgelesen. Die Daten in MainTable können dann auch von anderen Servlets benutzt werden.

Die `MainTable` besteht aus globalen A– und T–Box, ClassifiererDB, OperatrDB und einer Liste. In jedem Element dieser Liste werden für eine Suche aktuellen Instanzen von SearchID, ReferenceTable, Results, StartConcept, Planarchive und Input gespeichert. Dadurch wird Multiuserfähigkeit erreicht, d.h. die anderen Servlets werden vom InputmaskServlet unter Eingabe der SearchID aufgerufen, und sie suchen dann in MainTable nach „ihren“ Daten.

Die `ClassifiererDB` wird mit Hilfe von `readObject` aus der Datei geladen, die in der `Flowcontrol` angegeben wird. Die `OperatorDB`, `A_Box` und `T_Box` werden entweder als Objekt geladen oder, falls eine entsprechende Datei nicht existiert, aus einer Textdatei gelesen (vergleiche auch Abschn. 4.2 und 4.3). Falls eins von den o.g. Objekten nicht geladen werden konnte, kann die Suche nicht ausgeführt werden und wird abgebrochen. Der Agentenbenutzer bekommt eine ensprechende Fehlermeldung.

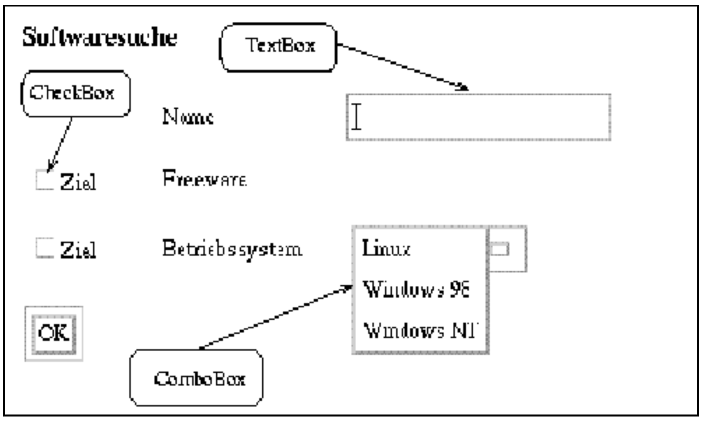
War das Laden von o.g. Objekten erfolgreich, wird die Eingabemaske generiert. Dafür wird die Methode `generateInputMask` im `StartConcept` aufgerufen. Die Methode geht alle Eingabeelemente durch und ruft in jedem Element `generateElement` auf. Diese Methode erstellt das Element zur Ausgabe im HTML-Format.

Es gibt drei Typen von Eingabeelementen :

1. TextBoxElement : Kann eine Zielcheckbox haben. Wird diese gewähl, handelt es sich bei dieser Eingabe um das Suchziel. Außerdem besteht dieser Element aus Namen und einem einfachen Texteingabefeld in dem der Benutzer den Suchstring eingibt, oder Eingaben über das Suchziel macht.

2. CheckBoxElement : Besteht aus einem Namen und einer Zielcheckbox.

3. TComboElement : Genau wie das erste Element kann dieses eine Zielcheckbox enthalten, muß aber nicht. Außerdem besteht es aus einem Namen und einer Combobox (eine Liste von auswählbaren Einträgen).



Ein Beispiel dafür, wie die Elemente in einem Browser dargestellt werden könnten!

Suchzyklus

Hat der Agentenbenutzer seine Suchanfrage in die Eingabefelder eingetragen, drückt er den OK-Button am Ende der Seite. Damit startet er `ProcessInputServlet`, das die Methode `startSearch` in dem Objekt der `Flowcontrol` startet und ein neues Fenster öffnet, in dem am Ende der Suche die Ergebnisse präsentiert werden. Für die Ausgabe ist ein Objekt der Klasse `OutputmaskServlet` verantwortlich.

In dem ersten Fenster erscheint ein Button “Zwischenergebnisse“. Wird dieser gedrückt, erscheinen die zur Zeit vorhandenen Suchergebnisse.

Nach dem Erzeugen des OutputmaskServlets wird das eine Suche eindeutig identifizierende SearchID erzeugt. Für jeden zu startenden Objekt der Klasse `Planner` wird eine PlanID und ein PlanInformation–Objekt ergeugt. Dafür wird jeweils eine Kopie der T–Box und der A–Box angelegt. In der Methode `startSearch` werden zuerst zwei Methoden aus dem `StartConcept` aufgerufen:

1. `extractGoal` markiert in der T–Box alle als Ziele markierten Benutzereingaben.

2. `markInstance` markiert in der A–Box alle dem Benutzer bekannten Merkmale des Suchobjektes. Falls eine Checkbox gewählt wurde, wird die gleichnamige Instanz in der A–Box gefunden, und mit Hilfe der `markAsConstraint` markier. Andernfalls wird eine neue Instanz in der A–Box erzeugt und markiert.

Die Systemkontrolle wird Planer eines Typs immer nur hintereinander ausführen, nicht gleichzeitig, da die Planer dann alle denselben Plan herausfänden. Planer verschiedenen Typs können und sollen gleichzeitig ausgeführt werden.

Für die `Systemkontrolle` wird intern eine Liste von Objekten der Klasse `PlnaObject` erzeugt, wobei jedes Objekt ein PlanInformation–Objekt und den damit gestarteten Planer enthält, damit die Systemkontrolle die verschiedenen Planer verwalten kann.

Suchergebnisse

Jedes einzelne Suchergebnis besteht aus einem Objekt der `NetEntity` (URL und eine Kurzbeschreibung) und einer Zahl, genauer einem Ranking, das die Güte des Ergebnisses beschreibt. Für jeden erfolgreich abgearbeiteten Gesamtplan wird eine Liste der Ergebnisse erzeugt. Sind alle erstellten Pläne ausgeführt, werden diese Listen in eine Gesamtliste – ResultsList geschrieben.

Bevor die Ergebnisse ausgegeben werden, werden die doppelt vorkommenden Einträge entfernt. Sie werden durch einen Vergleich der URLs erkannt. Danach werden die Ergebnisse gemäß ihrem Ranking sortiert, damit der Benutzer die besseren Treffer am Anfang der Seite sieht.

Werden Zwischenergebnisse verlangt, so werden die zu diesem Zeitpunkt im `ResultsList`–Objekt vorhandene Einträge sortiert und ausgegeben. Die Ausgabe wird immer in der Klasse `Output`, durch die Methode `generateOutputMask` erzeugt.

Ende der Suche

Wird das Outputservlet gestartet, so holt es alle notwendigen Objekte wie ResultsList, Output und Flowcontrol aus dem Session-Objekt. Danach wartet es bis alle Planer und Planausführungen fertig sind, und ruft dann die Methode `generateOutputMask` des `Output`–Objektes auf. Falls die Suche ohne Ergebnisse blieb, wird eine entsprechende Meldung ausgegeben. Sonst bereitet es die Ergebnisse zur Ausgabe vor (entfernt doppelte Einträge und sortiert alle Elemente), und gibt eine fertige HTML-Seite zurück. Das Servlet gibt diese Seite aus und ruft die Methode `endSearch` in `Flowcontrol`–Objekt auf. Diese Methode speichert alle Objekte, die während der Suche geändert wurden: A–Box, T–Box, Operatorendatenbank, Klassifiketorendatenbank und Planarchiv, wobei der Dateiname für das Planarchiv mit der SuchID übereinstimmen muß. Alle anderen Objekte werden ebenfalls als Objekte gespeichert, auch dann, wenn sie zuvor aus Textdateien eingelesen wurden.

Außerdem wird der Lerner gestartet. Damit ist die Suche komplett abgeschlossen.

3.3 Implementierung eines Agenten

Nachfolgend sind für jede Klasse alle Änderungen aufgelistet, die der Shellbenutzer vornehmen muß, um einen lauffähigen Agenten zu erstellen. Dabei entspricht die Abschnittüberschrift jeweils dem Namen (inkl. Pfad) der anzupassenden bzw. zu ändernden Klasse.

io.Input

Das konkrete Aussehen der Eingabemaske wird je nach Aufgabe der einzelnen Agenten variieren. Die Schnittstelle besteht defaultmäßig aus einem Formular mit Eingabefeldern, das mit Hilfe eines Java-Servlets erstellt wird. Der Shellbenutzer kann die Eigabemaske jedoch beliebig gestalten. Dazu muß zuerst die Methode `createStartConcept` überschrieben werden.

Als erstes müssen die Konzepte aus der `T_Box` geholt werden, zu denen Eingabeelemente in der Eingabemaske angezeigt werden sollen. Z.B.:

```
Concept C1 = tbox.lookup("operating_system");
Concept C2 = tbox.lookup("name");
Concept C3 = tbox.lookup("status")
```

Dann müssen die Eingabeelemente erzeugt werden. Alle Argumente werden im Konstruktor übergeben.

- `TextBoxElement`:

  ```
  TextBoxElement TE1 =
        new TextBoxElement("Name", true, C2);
  ```

  Die Argumente sind:

  1. Label – Beschriftung des Eingabefeldes
  2. hasGoalButton: `true`, falls das Element eine Zielcheckbox haben soll, sonst `false`.
  3. Konzept, wobei es sich um ein in der `T_Box` definiertes Konzepte handeln muß.

- `CheckBoxElement` :

  ```
  CheckBoxElement Ch3 =
         new CheckBoxElement(true, "Freeware", C1);
  ```

  Die Argumente:

  1. defaultValue: `true` oder `false`. Ist die variable auf `true` gesetzt, wird die Zielcheckbox markiert.
  2. Label (s.o.)
  3. Konzept (s.o.)

- `TComboBoxElement` :

  ```
  LinkedList V2 = new LinkedList(); V.add("Unstable");
  V.add("Stable"); ComboBoxElement CE1 =
         new ComboBoxElement("Linux",V, "OS", true, C1);
  ```

Die Argumente:

1. defaultValue: String, der dem Benutzer als vorgewählter Eintrag presentiert wird
2. values: Liste der ComboBox–Einträge
3. Label (s.o.)
4. hasGoalButton (s.o.)
5. Konzept (s.o.)

Danach müssen alle erzeugten Elemente in die Liste `Elements` eingetragen werden:

```
Elements.add(TE1); Elements.add(CE1); Elements.add(Ch3);
```

Jetzt kann ein `StartConcept`–Objekt erzeugt werden:

```
StartConcept Start = new StartConcept("DebianSearch", Elements);
```

„DebianSearch" ist die Überschrift der Eingabemaske.

io.StartConcept

Sie können vor oder nach der Eingabemaske beliebigen Text plazieren. Beispiel:

```
public void generateInputMask() {
  String HtmlText = "";

  inputmask = "<html><head><meta http-equiv="expires"
               content="0">"+header+"</head><body>

 ------ Hier können Sie Ihren Text einfügen. ------

  <form METHOD=GET ACTION=
             "/servlet/ProcessInputServlet"><table>";

  for (int i=0; i<elements.size(); i++) {
    Element dummy = (Element)elements.get(i); switch(dummy.typ) {
      case 1: {
      TextBoxElement TextBoxDummy = (TextBoxElement)elements.get(i);
      HtmlText = HtmlText+TextBoxDummy.generateElement(i);
      break;
    } case 2: {
      ComboBoxElement ComboBoxDummy = (ComboBoxElement)elements.get(i);
      HtmlText = HtmlText+ComboBoxDummy.generateElement(i);
      break;
    } case 3: {
      CheckBoxElement CheckBoxDummy = (CheckBoxElement)elements.get(i);
      HtmlText = HtmlText+CheckBoxDummy.generateElement(i);
      break;
    } }
  } inputmask = inputmask+HtmlText+"</table>"+
    "<INPUT TYPE=SUBMIT VALUE=OK></form>
```

```
  -----  Oder hier  ------


     </body></html>";
 }
```

In der Methode `extractGoal` werden die Ziele der Suche in der `T_Box` markiert. In der Shell werden nur die Konzepte markiert. (Vergleiche Abschnitt 8.2)

Falls Sie auch Rollen markieren möchten, müssen Sie die Methode `markAsTarget` benutzen:

```
tbox.markAsTarget(tbox.lookup("archive_url"));
tbox.markAsTarget(tbox.lookup("package"));
```

systemcontrol.Flowcontrol

Als erstes müssen Sie Dateinamen für `A_Box`, `T_Box`, `OperatorDB`, `ClassifierDatabase` und die Proxi-Werte setzen. Für die `OperatorDB` muß außerdem auch eine Textdatei definiert werden. Falls die `OperatorDB` noch nicht existiert, wird sie aus dieser Textdatei ausgelesen. Sie können die Dateinamen aus einer Konfigurationsdatei einlesen. Hier ist ein Ausschnitt aus einer Konfigurationsdatei:

```
Working Directory = /home/pg343/malzahn

# relative oder absolute Pfade der zu ladenden Dateien

A_Box=sourcen/integration-shell/DebianSuchagent.abox
T_Box=sourcen/integration-shell/DebianSuchagent.tbox
OperatorDBDate=sourcen/integration-shell/OperatorDB.opdb
OperatorDBObject=sourcen/integration-shell/OperatorDB.opdbObject
ClassDatabase=/home/pg343/share/DebianSuchagent/DebianSuchagent.cldb
```

So kann in der Methode `createSessionObjects` eine Zeile aus der Konfigurationsdatei eingelesen werden:

```
if (line.startsWith("Working Directory"))
  {
    wd=(line.substring
          (line.indexOf("=")+1,line.length())).trim();
    LogService.log(100,this,
      "current Working Directory set to:"+wd);
  }
```

systemcontrol.event.ShellEventAdapter

Die Eventhändling wird im nächsten Kapitel ausführlich beschrieben. Sie können die maximale Anzahl der Instanzen pro Operatorvorbedingungsprädikat bestimmen, die bei der Suche berücksichtigt werden. Momentan befindet sich die Variable `MaxInstance` in der Konfigdatei, und ist auf

10 gesetzt. (siehe Abschnitt II D.1) Diese Beschränkung der Belegungskonbinationen dient der Reduzierung der Laufzeit für die Suche.

Wird das Event `planExecuteFinishedEvent` ausgelöst, kann der Agentenbauer frei entscheiden, ob derselbe Planner oder ein anderer Plannertyp den nächsten Teilplan erstellen soll. Im ersten Fall soll der Programmcode nicht geändert werden. Sonst muß die Zeile

```
PO.getPlan().myResume();
```

durch

```
String Class = (PO.getPlan().getClass()).toString();
PO.getPlan().myDestroy();
if (Class == PlannerTypA) {
   PlannerTypeC plan1 = new PlannerTypeC(planInfo);
   PO.setPlan(plan1);
   }
else {
   PlannerTypeA plan1 = new PlannerTypeA(planInfo);
   PO.setPlan(plan1);
   }
plan1.start();
```

o. ä. ersetzt werden.

Andere Eingabemasken

Möchten Sie statt unserer Benutzereingabeschnittstelle eine andere benutzen, müssen Sie folgendes beachten.

Ihre Schnittstelle sollte eine Instanz der Klasse `Flowcontrol` erzeugen, die Methode `createSessionObjects` starten z.B.

```
Flowcontrol flow = new Flowcontrol(); Object objs[] =
flow.createSessionObjects();
```

Dabei müssen Sie beachten, daß in dieser Methode Objekte der Klassen `Input`, `Output` und `StartConcept` erzeugt werden, und ein Aufruf von `createStartConcept` erfolgt. Ihre Klassen müssen d ieselbe Funktionalität wie die, von uns zu Verfügung gestellten aufzuweisen.

Danach müssen Sie die Eingabemaske generieren.

Hat der Agentenbenutzer die Maske ausgefült und abgeschickt, müssen die von ihm gemachten Eingaben in der A–Box mit `markAsConstraint` und die Ziele in der T–Box mit `markAsTarget` markiert werden. Außerdem muß die Suche mit einem Aufruf von `startSearch` in einem Objekt der Kalsse `Flowcontrol` gestartet werden. Wichtig ist, daß auch diese Methode dieselbe Instanz von `StartConcept` benutzt, so daß diese Methode evtl. zum Teil geändert werden muß. Nachdem die Suche beendet wurde, muß die Ausgabemaske generiert und ausgegeben werden.

Kapitel 4

Eventhandling

> Die Menschen sind da, um einander zu helfen, und wenn man eines Menschen Hülfe in rechten Dingen nötig hat, so muß man ihn dafür ansprechen. Das ist der Welt Brauch und heißt noch lange nicht betteln.
>
> *Jeremias Gotthelf*

Planer, Planausführer und die Systemkontrolle kommunizieren mit einander mit Hilfe von Events. Hat ein `Planner`–Objekt einen Teilplan erstellt, so löst er ein Event aus. Folgende Events sind möglich:

1. `planReadyFinishedEvent` : Der Planer hat einen letzten Teilplan erstellt. Nach dessen erfolgreicher Ausführen müssen die Ergebnisse vorliegen.

2. `planReadyUnfinishedEvent` : Nach der Ausführung des erstellten Teilplans muß ein `Planner`–Objekt mit dem vorhandenen Wissen weiterplanen.

3. `planReadyAborted` : Der Planer konnte mit den vom Benutzer eingegebenen Daten keinen Plan erstellen. Die Suche wird abgebrochen. Der Planer wird angehalten und gelöscht.

Wurde ein Teilplan erstellt, also eines der ersten beiden Events ausgelöst, wird der Planer angehalten und mit dem aktuellen `PlanInformation`–Objekt eine Planausführung gestartet.

Die Planausführung gibt das Ende der Ausführung durch Events bekannt. Das `planExecuteFinishedEvent` gibt an, daß die Planausführung erfolgreich verlaufen ist. Falls der ausgeführte Teilplan nicht der letzte Teilplan des Gesamtplans war, wird ein Planer mit dem `PlanInformation`–Objekt angestoßen. Der gestartete Planer kann derselbe sein, der den vorherigen Teilplan erstellt hat, oder ein anderer Planertyp. Die Entscheidung liegt bei dem Shellbenutzer.

Sonst wird der Planer, der diesen Teilplan erstellt hat, zerstört, die gewonnenen Ergebnisse werden in eine Liste geschrieben. Das `PlanInformation`–Objekt wird in das Planarchiv eingetragen. Außerdem wird überprüft, ob alle Planer bereits dei Suche komplett abgeschlossen haben. In diesen Fall gilt die Suche als beendet.

Das `planExecuteFinishedBackEvent` wird nur dann aufgerufen, wenn der ausgeführte Teilplan nicht der letzte war, und es ist unbedingt notwendig, daß derselbe Planer auch weiterplant.

Das `planExecuteAbortedEvent` gibt an, daß die Planausführung aus irgendeinem Grund scheiterte. Dieser Planer wird als fertig markiert d.h. er hat diese Suche abgeschlossen, und wird nicht mehr gestartet. Falls alle anderen Planer auch markiet sind, wird die Suche beendet.

Ein Planer oder eine Planausführung könnten außerdem auch ein `planExecuteFinishedEvent` auslösen, falls ein letzter Teilplan erstellt und ausgeführt wurde. Der Planer, der diesen teilplan erstellt hat, wird dann ebenfalls zerstört, und die vorhandenen Ergebnisse werden in die Liste der Suchergebnisse eingetragen.

Will ein Planer oder eine Planausführung ein Event auslösen, muß er zuerst ein Event erzeugen:

```
 ShellEvent e = new ShellEvent(this, getPlanInfo());
```

Dann muß er einen ensprechenden Listener registrieren:

```
ShellEventListener sel =
     (ShellEventListener)Flowcontrol.ShellEventListeners.elementAt(0);
```

Jetzt kann das Eventauslösende Objekt beliebige Events erzeugen:

```
sel.planReadyAborted(e);
```

Kapitel 5

T– und A–Box

Wissen ohne Ordnung ist Hausrat auf einem Leiterwagen

Jakob Lorenz

5.1 Überblick und Definition

Für die Wissensrepräsentation wird das Modell einer T- und A-Box verwendet. Die T–Box ist der terminologisch Teil, in der die Begriffe definiert und in eine Begriffsstruktur eingeordnet werden. Die T–Box stellt das konzeptionelle Weltwissen des Suchagenten dar und muss vor der Initialisierung modelliert werden. Deshalb ist der Inhalt der T–Box ist entscheidend für die Sucherfolge des Agenten.

Die A–Box ist der assertionale Teil der Wissensrepräsentation, d.h. in der A–Box werden die Individuen oder Instanzen und die tatsächlich bestehende Rollen gespeichert. Auch die A–Box muß vor der ersten Suche mit einigen Instanzen gefüllt werden, damit der Planer einen Suchplan entwickeln kann.

Termini der T–Box:

Def.: Ein Konzept repräsentiert eine Menge von Instanzen.

Def.: Ein Konzept K heißt primitives Konzept , wenn seine Menge von Instanzen eine Untermenge der Schnittmenge seiner Oberkonzepte $K_1, ..., K_n$ ist. Sei U die Instanzenmenge von K und $O_1, ...O_n$ die Mengen von $K_1, ..., K_n$, dann gilt $U \subseteq O_1 \cap ... \cap O_n$.

Def.: Ein Konzept heißt definiertes Konzept, wenn seine Menge von Instanzen gleich der Schnittmenge seiner Oberkonzepte $K_1, ..., K_n$ ist. Sei U die Instanzenmenge von K und $O_1, ...O_n$ die Mengen von $K_1, ..., K_n$, dann gilt $U \subset O_1 \cap ... \cap O_n$.

Def.: Eine Rolle ist eine binäre Relation zwischen Konzepten.

Def.: Eine *isA*–Beziehungen zwischen Konzepten ist keine Rolle im oben genannten Sinne, sondern stellt die Konzepthierachie dar.

Termini der A–Box:

Def.: Eine Instanz ist eine Entität eines Konzeptes. Instanzen können gleichzeitig zu verschiedenen Konzepten gehören.

Def.: Eine konkrete Rolle ist eine tatsächlich bestehende Rolle zwischen zwei Instanzen und wird häufig auch als Rolleninstanz bezeichnet.

5.2 T–Box

Die T–Box wird als Graph dargestellt, wobei die Knoten die Konzepte bilden und die Kanten die möglichen Rollen zwischen den Objekten. Ausserdem gibt es eine ausgezeichnete Kantenmenge, die die *isA*–Beziehungen zwischen den Konzepten darstellt und die Hierachie zwischen Oberkonzepten und Unterkonzepten repräsentieren. Sie alleine bilden mit den Konzepten einen Baum mit `anything` als Wurzel, unter welches jedes Konzept subsummiert wird. Es existiert ein Subsummationstest, der die Konzepte `a` und `b` darauf überprüft, ob `a` `b` subsummiert. Wenn das der Fall ist, ist `b` spezieller als `a` und `a` ist das Oberkonzept von `b`. Dann ist `b` ein primitives Konzept.
Wird das Konzept `b` von den Konzepten $a_1, ..., a_n$ subsumiert, wobei die Konzepte $a_1, ..., a_n$ sich nicht subsummieren und `b` genau den Konzepten $a_1, ..., a_n$ entspricht, ist `b` ein definiertes Konzept. Zusätzlich subsummiert sich jedes Konzept auch selbst.
Die Rollen in der T–Box müssen eindeutige Namen haben, weil sowohl die Startinstanzen, als auch die Zielinstanz zu mehren Konzepen gehören können, und somit nur der Rollennamen die Eindeutigkeit garantiert.

Außerdem werden in der T-Box Konzepte als Ziele der Suche markiert. Instanzen, die unter dieses Konzepte fallen, bilden die Suchergebnisse. Aus diesen Anforderungen folgt auch, daß zu jeder Suche eine eigene T–Box erzeugt werden muß.

Beispiel

Ein Ausschnitt einer T–Box für einen Software–Suchagenten könnte aussehen wie in Abbildung II 5.1:

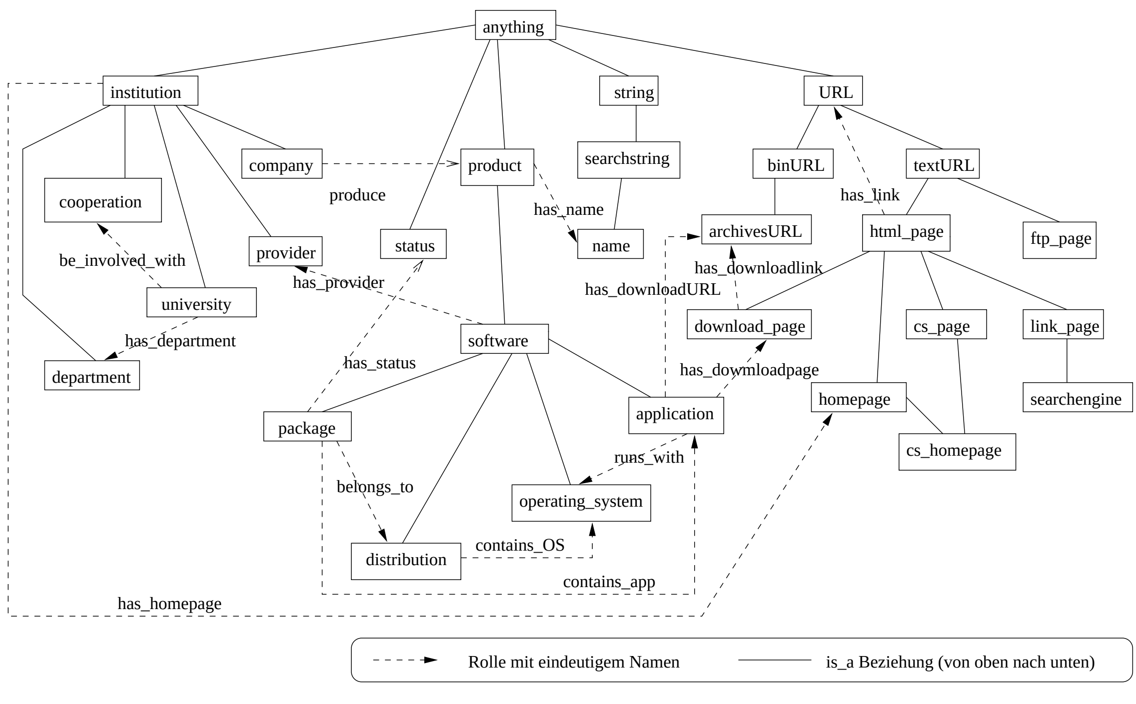


Abbildung 5.1: Mögliches Aussehen der T–Box für einen Softwaresuch–Agenten

Dieses kleine Beispiel zeigt, wie umfangreich eine T–Box, und wie mühsam eine graphische Darstellung werden kann. Dennoch ist es ratsam, eine solche Darstellung zu erstellen, weil so am schnellsten deutlich wird, wo Modellierungsfehler sind, z.B. wenn ein Konzept keinerlei *isA*–Beziehungen hat, nicht einmal zu `anything`, oder wenn es durch die modellierten Rollen möglich wird, unsinnige konkrete Rollen zu den Instanzen zu bilden, was nicht erwünscht ist.

Die drei linken Äste des Hierachiebaums ist abhängig vom jeweiligen Suchgebiet, auf den Rest wird wohl kein Suchagent, der im Internet suchen soll, verzichten können.

An diesem Beispiel kann auch noch einmal der Unterschied zwischen einem primitiven und einem definierten Konzept deutlich gemacht werden. *`homepage`* ist ein primitiven Konzept, weil *`homepage`* zwar ein Unterkonzept von *`html_page`* ist, also jede *`homepage`* ist auch eine *`html_page`*, aber nicht jede *`html_page`* ist auch eine *`homepage`*. Dagegen gehört eine Instanz nur dann zum Konzept *`cs_homepage`*, wenn sie sowohl eine *`homepage`* und als auch eine *`cs_page`* ist.

Dateiformat der T–Box

Um dem Shell–Benutzer eine etwas komfortabelere Möglichkeit zu bieten, die T–Box anzulegen, ist es sinnvoll, die Dateien in einer lesbaren Form anzulegen. Deshalb kann der Inhalt der T–Box mit Hilfe einer Textdatei eingegeben werden. Das Dateiformat wird zeilenweise verarbeitet, so daß in jeder Zeile genau ein Kommando stehen darf. Zudem muß folgend Syntax beachtet werden:

- Groß- und Kleinschreibung wird nicht unterschieden.
- Kommentare werden mit # eingeleitet. Alles was in der Zeile nach # steht, wird ignoriert.
- Zwischen den Bezeichnern muß immer mindestens ein Leerzeichen stehen.
- Primitive Konzepte werden in der Form $A :< B$ bzw. $A :< B$ *and* C definiert. Dabei ist das Konzept A das primitive Konzept, welches von dem Konzept B bzw. von den Konzepten B und C subsummiert wird.
- Definierte Konzepte werden in der Form $A := B$ bzw. $A := B$ *and* C definiert. Dabei ist das Konzept A das definierte Konzept, welches von dem Konzept B bzw. von den Konzepten B und C subsummiert wird.
- Das alle Konzepte subsummiernde Oberkonzept heißt *anything* und das inverse kein Konzept subsummiernde Konzept heißt *nothing*.
- Rollen werden in der Form $R:\ A => B$ definiert, mit R ist die zu definierende Rolle, A das Konzept, in dem die Rolle definiert ist, und B das Konzept, auf das die Rolle verweist.
- Fehlerhafte Zeilen werden beim Laden mit einer Fehlermeldung aufgelistet und bei der weiteren Abarbeitung ignoriert.

Zurück zum Beispiel in Abb. II 5.1

Entsprechend zu der Modellierung sehen Einträge in der T–Box–Textdatei dann so aus:

```
# Konzepte die Computer betreffen

# primitive Konzepte
operating_system :< software
application :< software
cs_page :< html_page
homepage :< html_page

# definiertes Konzept
cs_homepage := cs_page and homepage
```

```
# Rollen
runs_with : application => operating_system
contains_OS : distribution => operating_system
```

5.3 A–Box

Die A–Box ist eine Liste, die als Elemente die Instanzen enthält. Es können nur solche Instanzen in die A–Box geschrieben werden, die zu einem Konzept aus der T–Box gehören. Eine Instanz kann mehr als einem Konzept zugeordnet werden, wobei die Konzepte nicht in Oberkonzept–Unterkonzept–Relation zueinander stehen müssen. Jede Instanz hat eine Liste von Konzepten, denen sie zugeordnet wird, und zudem hängt an jedem Konzept ein Wahrscheinlichkeitswert, der angibt, mit welcher Wahrscheinlichkeit die Instanz unter dieses Konzept fällt. Der Wahrscheinlichkeitswert ist ein *double* im Intervall von [0,1].

Außerdem hat jede Instanz eine Liste mit den konkreten Rollen, die zu dieser Instanz existieren. Auch an den konkreten Rollen hängt ein Wahrscheinlichkeitswert im Intervall von [0,1], der angibt, mit welcher Wahrscheinlichkeit die Rolle der Instanz zugeordnet werden kann. Es können ähnlich wie bei den Konzepten nur dann konkrete Rollen angelegt werden, wenn es in der T–Box eine Rolle mit diesem Namen gibt und sowohl die Startinstanz, als auch die Zielinstanz in ihrer Konzeptliste die Konzepte haben, zwischen denen die Rolle in der T–Box definiert ist.

Neben der eindeutigen Id–Nummer und dem Namen der Instanz, der nicht eindeutig sein muß, kann ein dazugehörender Inhalt vom Typ ***NetResult*** gespeichert werden. Zusätzlich hat jede Instanz noch weitere Attribute, wie z.B. `Instance.date`, das das Eintragsdatum festhält, oder `Instance.counter`, der angibt, wie oft eine Instanz zur Planerstellung benutzt wurde, und noch einige andere, die für den Planer oder den Lerner wichtig sind.
Genau wie bei der T–Box wird es mehrere A–Boxen gleichzeitig geben, eine globale, die über alle Suchen persistent ist und die vor der Initialisierung des Suchagenten mit Inhalt gefüllt werden muß. Für jede Suchanfrage wird eine Kopie der globale A–Box erzeugt, die sogenannte suchglobale A–Box, in der die Instanzen, die durch die Anfrage hinzu gekommen sind markiert werden. Für jeden Planer wird eine Kopie der suchglobale A–Box angelegt, die durch einen Planausführer mit neu dazukommenden Instanzen angefüllt wird. Ist eine Suche abgeschlossen, überträgt anschliessend der Instanzenlerner die neu hinzugekommenden Instanzen, soweit es sinnvoll erscheint, in die globale A-Box.

Dateiformat der A–Box–Datei

Wie bei der T–Box, ist es auch bei der A–Box möglich, die globale A–Box mit Hilfe einer Textdatei zu füllen. Die Textdatei wird dann durch den Aufruf von `A_Box.load` in eine leere A–Box eingelesen.
Da die Textdatei zeilenweise verarbeitet wird, sind beim Erstellen einige Regeln zu beachten:

1. Kommentare werden mit # eingeleitet. Alles was in einer Zeile nach einem # steht, wird ignoriert.

2. Jede einzelne Instanz wird durch <Instance> eingeleitet und durch <\Instance> beendet. Die einzelnen Komponenten, wie z.B. der Namen der Instanz werden danach zeilenweise eingegeben.

3. Ähnliches gilt für die Aufzählung von Rollen, sie beginnt mit `<roles>`, dann kommen zeilenweise die einzelnen Rollen und endet mit `<\roles>`. Auch die Konzepte beginnen mit `<concepts>`, dann kommen zeilenweise die Konzepte, evtl. der Wahrscheinlichkeitswert in eckigen Klammern. Die Aufzählung der Konzepte wird mit `<\concepts>` abgeschlossen.

Zurück zum Beispiel in Abb. II 5.1

Instanzen der A–Box könnten dann z.B. folgende sein:

```
# im Bereich Betriebssysteme
<Instance>
name: linux
<concepts>
operating_system [1]
<\concepts>
<\Instance>

<Instance>
name: debian # Linux-distributionen
<concepts>
distribution
<\concepts>
<roles>
debian contains_OS linux [0.8]
<\roles>
<\Instance>
```

Wird die A–Box mittels der Textdatei aufgefüllt, muß der Instanzenname eindeutig sein, weil die Id–Nummer erst nach dem Aufruf von `A_Box.load` vergeben wird und die Instanz erst ab diesem Zeitpunkt durch ihre Id identifiziert wird. Der Shellbenutzer muß beim Eintragen der Instanzen darauf achten, daß das oder die Konzepte in der T-Box vorhanden sind und daß in der T-Box die entsprechenden Rollen existieren. Wird bei den Konzepten und konkreten Rollen kein Wahrscheinlichkeitswert angegeben, wird er beim laden automatisch auf 1 gesetzt. Sind Sie sich aber nicht sicher, ob ein Konzept oder eine konkrete Rolle tatsächlich zutrifft, können Sie auch einen beliebegen anderen *double*–Wert im Intervall von [0,1] in eckigen Klammern an die konkrete Rolle oder das Konzept schreiben.

Die Instanzattribute `Instance.counter`, `Instance.date`, `Instance.delete`, usw. werden beim Erzeugen der Instanz automatisch gesetzt. Wird die A-Box erneut gespeichert, werden `Instance.id`, `Instance.date` und `Instance.counter` mitgespeichert. Dann ist es möglich, daß doppelte oder kein Instanzennamen existiert, die Instanzen können aber anhand ihrer Ids eindeutig identifiziert werden.

Instanzen, die weder einen Namen noch eine Id haben, werden beim Laden nicht erzeugt. Das gilt auch für Instanzen, bei denen kein Konzept angegeben ist, oder das angegebene Konzept nicht in der T-Box existiert. Dagegen werden Instanzen, deren Rollen in der T-Box nicht vorhanden sind, erzeugt, die konkreten Rollen aber nicht angelegt. In diesen Fällen gibt es dann eine Fehlermeldung.

Die Funktion `A_Box.save` sollte nicht zum Speichern der globalen A–Box benutzt werden, weil der `Instance.contens` nicht mit in die Textdatei gespeichert wird.Die Funktion `A_Box.save` dient lediglich dazu, die A–Box wieder in die lesbare Form der Textdatei zu überführen. Möchte man später neue Instanzen in die globale A–Box schreiben, kann man das auch wieder mit Hilfe einer neuen Textdatei tun. Die neuen Instanzen werden dann durch `theGlobalA_Box.load` zu den schon existierenden Instanzen dazu geladen. Sie müssen bei dieser neuen Textdatei nicht darauf achten, ob Sie Instanzennamen vergeben, die in `theGlobalA_Box.load` schon vorkommen, nur innerhalb der neuen Textdatei dürfen Instanzennamen nicht doppelt vergeben werden.

Eigene Änderungen der Implementierung

Es empfiehlt sich nicht, die Wissensrepräsentation zu verändern, da sie ein zentraler Bestandteil der Shell ist und eng mit den Plannern, den Planausführern und der Systemkontrolle zusammenarbeitet. Sollten Sie das denoch wünschen, müssen Sie die Schnittstellen, die im Java–Doc beschrieben sind, einhalten.

Wollen Sie die Syntax der T–Box–Textdatei, bzw. der A–Box–Textdatei ändern, müssen Sie die Methoden `T_Box.load`, `T_Box.save`, `A_Box.load` und `A_Box.save` entsprechend anpassen.

Kapitel 6

Planarchiv

Ich habe mir die Zeitungen vom vorigen Jahr binden lassen. Es ist unbeschreiblich, was für eine Lektüre dieses ist: 50 Teile falsche Hoffnung, 47 Teile falsche Prophezeiungen und 3 Teile Wahrheit. Diese Lektüre hat bei mir die Zeitungen von diesem Jahr sehr herabgesetzt; denn ich denke: Was diese sind, das waren jene auch.

Georg Christoph Lichtenberg

In das Planarchiv schreibt die Systemkontrolle die Planinfo–Objekte, die zusammen einen Suchverlauf repräsentieren. Diese Pläne werden in einer globalen Datei gespeichert und werden von den Lernern benutzt.

Kapitel 7

Operatoren

Zum Erfolg gibt es keinen Lift, man muß die Treppe benützen.

Emil Oesch

7.1 Operatoren im Überblick

Wenn Sie etwas erreichen wollen, müssen Sie normalerweise auch etwas dafür tun. Um etwas machen zu können, müssen meistens gewisse Vorbedingungen erfüllt sein. Ist schließlich Ihre Arbeit nicht ganz für die Katz', dann kommt auch noch etwas Produktives dabei heraus.

Wollen Sie z. B. einen Bilderrahmen mit einem Nagel an einer Wand befestigen, so geht das nur, wenn Sie auch einen herumliegenden Bilderrahmen, einen herumliegenden Nagel, eine freie Wand und natürlich einen Hammer haben. Geht alles glatt, so haben Sie anschließend keinem herumliegenden Bilderrahmen mehr, keinen herumliegenden Nagel und auch keine freie Wand. Geblieben ist Ihnen nur der Hammer und die Information, daß der Rahmen jetzt mit dem Nagel an der Wand befestigt ist. Bei gleichen Anfangsbedingung kann es aber auch sein, daß Sie nach der Ausführung Ihres Vorhabens eine immer noch freie Wand, einen herumliegenden Bilderrahmen, einen krummen, herumliegenden Nagel und einen dicken Daumen haben. Den Hammer haben Sie immer noch.

An diesem kleinen Beispiel ist deutlich zu sehen, daß die Ausführung einer Aktion nach folgendem Schema abläuft:

1. Was brauche ich?
2. Habe ich alles?
3. Ausführen!
4. Was habe ich jetzt?

Wenn Sie eine Operation (eine (oft) kompliziertere Aktion oder Handlung) machen sollen, so können Sie normalerweise sagen, was Sie alles brauchen und was Sie dann mit den einzelnen Sachen machen würden. Das ist völlig unabhängig davon, ob Sie das dann auch wirklich tun. Genausogut können Sie im vorhinein sagen, was als Ergebnis herauskommen könnte. Das eine Operation durchaus mehr als ein Ergebnis haben kann, haben wir in obigem Beispiel gesehen!

Was Sie nicht allgemein formulieren können ist, ob nun in dem Moment, in dem Sie die Operation ausführen wollen, auch alle Vorbedingungen erfüllt sind. Genausowenig können Sie, auch wenn alle Vorbedingungen erfüllt sind, wissen, was nachher bei Operationen mit mehreren möglichen Ausgängen herauskommt.

Die Beschreibung dessen, was eine Operation an Voraussetzungen hat, was mit den notwendigen Dingen dann gemacht würde und welche Ergebnisse dabei herauskommen könnten, steckt in der Klasse `Operator`. Leider ist diese Klasse abstrakt, so daß Sie, lieber Shellbenutzer, erst einmal Operatoren bauen müssen, die auf dem Weg zur Lösung genau Ihres Problems Ihnen die Steine aus dem Weg räumen können.

Anders ausgedrückt: wenn Sie einen eigenen Agenten schreiben möchten, so sind die Operatoren die Stelle, an denen Ihnen BotIshelly die wenigste Arbeit abnehmen kann, und Sie am meisten kaputt machen können!

Gehen wir es also an! ;-)

7.2 Prädikate und ihre Argumente

Prädikate sind Aussagen über irgendetwas. Das, worüber Aussagen getroffen werden, nennen wir auch die Argumente eines Prädikats. Setzen wir zwei Argumente über ein Prädikat in Beziehung, so sprechen wir auch von einer Rolle zwischen den beiden Argumenten. Aus der Mathematik kennen Sie die Rolle „kleiner gleich“. Prinzipiell kann man eine Aussage treffen wie $a < b, \quad a, b \in \mathbf{N}$, mehr programmiersprachlich aufgeschrieben als `Less (a : Nat, b : Nat)`. Dabei bezeichnen wir `Less` als den Namen des Prädikats, `a` und `b` als die Namen von Prädikatargumenten und `Nat` als Konzept oder Typ, auf den die möglichen Prädikatargumente eingeschränkt sind. Sie können z. B. auch notieren `Prime (a : Nat)`, wenn Sie ausdrücken wollen, daß es sich bei `a` um eine Primzahl handelt, oder auch `Nat (a : Nat)`, um `a` als natürliche Zahl auszuzeichnen.

Dererlei Prädikate können Sie in der Praxis nur anwenden, wenn Sie die Variablen auch mit Instanzen belegen können. Ein Prädikat, dessen Argumente vollständig mit Instanzen belegt sind, nennt man instantiiert. So sind z. B. `Less (a=1: Nat, b=3: Nat)` oder auch `Less (a=3: Nat, b=1: Nat)` vollständig instantiiert, obwohl letztere Aussage natürlich falsch ist.

Im vorangehenden Beispiel bezeichnen wir `Less` als den Namen des Prädikats (oder auch der Rolle, da das Prädikat zweistellig ist), `a` und `b` als Namen der Prädikatargumente, die Zahlen 1 und 2 als Instanzen der Prädikatargumente und die beiden `Nat` als Konzepte der Prädikatargumente.

BotIshelly stellt entsprechend die Klassen `PredArgument` und `Predicate` im Paket `operators` als Prädikatargumente und Prädikate zur Verfügung. Bei der Erzeugung einer Java–Instanz ***PredArgument*** ist darauf zu achten, daß das im Konstruktor

```
public PredArgument(Concept type, String name)
```

übergebene Konzept aus der T–Box stammen muß, die gerade benutzt wird. Sie dürfen sich die Java–Instanz ***Nat*** eines Konzeptes `Nat` also nie selber durch einen Aufruf von `new` erzeugen, sondern müssen immer mit Hilfe von z. B. *T_Box*`.lookup("Nat")` die von der T–Box angebotene Instanz ***Nat*** nehmen.

Ein Prädikat läßt sich durch Angabe seiner Stelligkeit (engl. *arity*) und des Prädikatnamens über den Konstruktor

```
public Predicate(int predArity, String predName)
```

erzeugen. Hier liegt es in Ihrer Verantwortung, darauf zu achten, daß sie nur Namen eintragen, die auch so als Namen von Konzepten oder Rollen in der T–Box vorkommen!

Als kleines Beispiel sei hier noch das Erzeugen einer Rolle für die kleiner–Beziehung aufgezeigt:

```
Predicate predicate = new Predicate (2, "less");
Concept natural = tbox.lookup ("natural");
predicate.setPredArg (new PredArg (natural, "a"), 0);
predicate.setPredArg (new PredArg (natural, "b"), 1);
```

Bitte beachten Sie, daß zu diesem Zeitpunkt noch keine Instanzen in den PredArgs belegt oder gar erzeugt sind!

7.3 Vorbedingungen und Nachbedingungen

`[Nagel (nagel : Nagel), Wand (wand : Wand), Rahmen (rahmen : Rahmen), Hammer (hammer : Hammer)]` wäre eine passende Vorbedingung für einen Operator, der einen Bilderrahmen an einer Wand befestigen kann, wobei die Liste so zu lesen ist, daß alle Prädikate erfüllt sein müssen (Konjunktion). Denkbar sind natürlich auch andere boolesche Verknüpfungen, wie Disjunktionen oder Negationen. Aufgrund der Komplexheit läßt BotIshelly aber nur Konjunktionen von Prädikaten als Vor– und Nachbedingungen zu.

Obwohl sich Vor– und Nachbedingungen recht ähnlich sind, gibt es doch einige Unterschiede. Z. B. gibt es genau eine Vorbedingung für einen Operator. Nehmen wir an, Sie haben einen Operator mit einem Stein als Vorbedingung, der damit eine Scheibe einschmeißen kann. Wollen Sie den Operator erweitern, so daß Sie jetzt auch mit einem Hammer eine Scheibe einschmeißen können, so können sie als Vorbedingung [Hammer oder Stein] *nicht* formulieren. Sie können das Problem auf zwei Arten angehen:

1. Sie schreiben einfach einen neuen Operator, indem sie den mit dem Stein kopieren und als Vorbedingung jetzt Hammer eintragen.

2. Sie ändern die T–Box so ab, daß Sie ein gemeinsames Oberkonzept für Hammer und Stein (z. B. SchwererGegenstand) einfügen. Entsprechend würde dann die Vorbedingung für den Operator jetzt `[SchwererGegenstand (schwererGegenstand : SchwererGegenstand)]` lauten.

Am Ausgang des Bildaufhängens haben wir gesehen, daß es mehr als eine Nachbedingung geben kann. Zusätzlich dazu, daß nur Konjunktionen erlaubt sind, schränkt BotIshelly hier weiter dadurch ein, daß die Nachbedingungen nur exklusiv–oder miteinander verknüpft sein dürfen. Wenn Sie schreiben wollen, daß der Daumen blutet, das Bild hängt oder beides passiert ist, so müssen Sie drei Nachbedingungen (vereinfacht)

1. `[BildAnWand (wand : Wand, bild : Bild)]`

2. `[DaumenBlutet (daumen : Daumen)]`

3. `[BildAnWand (wand : Wand, bild : Bild),`
   `DaumenBlutet (daumen : Daumen)]`

schreiben. Sollte die erste Nachbedingung erfüllt sein, so können Sie daraus nicht schließen, daß der Daumen nicht blutet. Sie wissen nur, daß der Daumen aufgrund der Ausführung des Bildaufhäng–Operators nicht blutet. Falls das Prädikat `DaumenBlutet` vor der Anwendung des Operators schon erfüllt war, so ist es danach natürlich immer noch erfüllt.

Ein Operator kann bei der Ausführung dreierlei Sachen mit Prädikaten und Rollen machen:

1. Ein `Predicate` der Vorbedingung wird nicht verändert.

2. Ein `Predicate` der Vorbedingung wird gelöscht.

3. Ein `Predicate` wird neu erzeugt.

Aus diesem Grund sind einzelne Nachbedingungen nicht nur Listen von Prädikaten, sondern bestehen aus zwei Listen, einer Add– und einer Deleteliste. In der Addliste einer Nachbedingung sind alle diejenigen Prädikate und Rollen eingetragen, die durch den Operator neu erzeugt wurden. In der Deleteliste stehen dann natürlich diejenigen, die durch den Operator gelöscht wurden. Da der Operator sein Wissen *nur* aus den Vorbedingung schöpfen kann, dürfen in den Deletelisten der Nachbedingungen verständlicherweise auch nur Prädikate oder Rollen auftauchen, die schon in der Vorbedingung standen. Implizit vorausgesetzt wird, daß ein `Predicate` der Vorbedingung, daß nicht in der Deleteliste einer Nachbedingung auftaucht, beim Erreichen genau dieser Nachbedingung auch nicht verändert worden ist.

BotIshelly bietet für Nachbedingungen die Klasse `Postcondition` im Paket `operators` an, die über den leeren Konstruktor `Postcondition()` erzeugt wird. Die beiden Methoden `addAddListPred (Predicate p)` und `addDeleteListPred (Predicate p)` dienen zum Manipulieren der Add– und Deleteliste der einzelnen Nachbedingung. Die Add– und Deletelisten müssen natürlich disjunkt sein, da es keinen Sinn macht, als Ergebnis eines Operators ein Prädikat gleichzeitig zu erzeugen und zu löschen.

Wenn Sie in den Vor– und Nachbedingungen eines Operators die Namen für die Prädikatargumente vergeben, müssen Sie einige Einschränkungen beachten, die sich aus der Implementierung der Planausführung ergeben: Einstellige Prädikate müssen zwingend wie das Konzept des zugehörigen Prädikatargumentes heißen. Daraus folgt, daß es zu jedem Konzept auch nur genau einen Prädikatnamen geben kann. Bei zweistelligen Prädikaten sind Sie in der Wahl des Namens allerdings frei. Z. B.:

```
[Nat(nat1: Nat), Less (nat2: Nat, prime: Prime)]
```

Eine weitere Einschränkung ist, daß in einem Prädikat der Vorbedingung die Namen der Prädikatargumente nicht schon *alle* links von diesem vorgekommen sein dürfen. Denn eine Vorbedingung der Art

```
[Nat(nat1: Nat), Nat(nat2: Nat), Less (nat1: Nat, nat2: Nat)]
```

in der die Prädikatargumentnamen `nat1` und `nat2` von `Less` schon beide links von `Less` auftauchen, sorgt bei der Ausführung dafür, daß der Planer, der von links nach rechts die Vorbedingung zu belegen versucht, `Less(nat1, nat2)` schon für belegt hält. Dann wird aber durch den Planausführer nicht mehr überprüft, ob das Prädikat `Less` für `nat1` und `nat2` überhaupt Gültigkeit hat! Es sei an dieser Stelle noch einmal darauf hingewiesen, daß diese Einschränkungen sich aus unserer Implementierung des Planausführers ergeben.

Erlaubt dagegen ist

```
[Nat(nat1: Nat), Less (nat1: Nat, nat2: Nat)]
```

da hier beim Belegen von `Less` zwar `nat1` schon durch das linksstehende Prädikat belegt wurde, beim Belegen von `nat2` durch den Planausführer aber eine Überprüfung der Gültigkeit von `Less` stattfindet. *Ausreichend* und auch zu bevorzugen ist an dieser Stelle aber eine Vorbedingung der Form

```
[Less (nat1: Nat, nat2: Nat)]
```

da der Planausführer bei der Belegung von `nat1` und `nat2` natürlich nur Instanzen auswählt, die auch unter das Konzept `Nat` passen und damit per se das Prädikat `Nat` erfüllen.

Aus obigen Einschränkungen folgt, daß Sie z. B. für einen Operator, der als Vorbedingung $a < b$ und $a < c$ und $b < c$ fordert, diese nur schreiben können als

```
[Less (natA: Nat, natB: Nat), Less (natA: Nat, natC: Nat),
 Less (natB: Nat, natD: Nat)]
```

Den Vergleich, ob `natC` und `natD` identisch sind, müssen Sie innerhalb der Ausführungsmethode des Operators selbst ausführen!

7.4 Operatoren an sich

Nachdem Sie einen aus der abstrakten Klasse `Operator` abgeleiteten Operator mit Hilfe von

```
Operator.setPrecond(Predicate[] prec)
Operator.setPostcond(Postcondition[] postc)
```

erfolgreich mit Vor– und Nachbedingungen[1] versehen haben, kommt nun der schwierigere Teil: die Methode `eval()`. Diese Methode ist nämlich abstrakt und muß von Ihnen überschrieben werden, bevor jemals eine Instanz eines abgeleiteten Operators erzeugt werden kann.

Was `eval` machen soll, ist einfach zu beschreiben: es nimmt sich die Instanzen aus den Prädikatargumenten der Vorbedingung und verarbeitet sie so lange, bis eine Nachbedingung dabei herauskommt, für die eventuell neue Instanzen erzeugt werden müssen. Bevor Sie `eval` wieder verlassen, müssen Sie mit `setPostcondFullfilled (int i)` eine der Nachbedingungen als erfüllt kennzeichnen, wobei die erste Nachbedingung den Index 0 hat.

Wie Sie nun erreichen, daß `eval()` etwas Vernünftiges aus den Vorbedingungen macht, ist ganz alleine Ihr Problem! Hier hängt es von Ihrem programmiertechnischen Können und Überblick ab, wie schnell und wie stabil und ob überhaupt jemals etwas Sinnvolles aus Ihrem Operator herauskommt! Wir wünschen Ihnen dabei auf jeden Fall viel Spaß und Vergnügen! :-)

Bei der Bereitstellung von Konstruktoren müssen Sie beachten, daß Sie immer auch einen Konstruktor mit leerer Parametermenge implementieren (z. B. `MyOperator ()` in der Klasse `MyOperator`. Dieser wird für das Einlesen der Operatorendatenbank (siehe II 8.2) gebraucht.

7.5 Sourcecode mit tausend Worten: ein Beispiel

Um Ihnen einmal zu zeigen, wie ein einfacher Operator aussehen kann, der eine Suchmaschine anfragt, geben wir hier ein kleines Beispiel. Der folgende Operator `NathanSearchOperator` hat die Aufgabe, die Suchmaschine *Nathan*[1] zu befragen. Dazu verlangt der Operator als Vorbedingung ein erfülltes Prädikat

```
searchstring (x : searchstring)
```

wobei `searchstring` ein in unserer T–Box vorkommendes Konzept ist. Der Operator soll weiter zwei Nachbedingungen haben, von denen die erste nur aus einer leeren Add– und Deleteliste besteht, für den Fall, daß der Operator scheitert. Die zweite Nachbedingung hat eine leere Deleteliste und fügt über die Addliste ein Prädikat der Form

[1] Die Klasse `Postcondition` enthält eine jeweils als `Predicate[]` realisierte Add– und Deleteliste

[1] www.nathan.de

```
urlmenge (y : urlmenge)
```

hinzu.

Der Rumpf des Operators ist recht einfach zu realisieren, da in aller Regel nur der Konstruktor und die Methode `eval` implementiert werden müssen:

```
package operators;
  import exceptions.*;
  import informationexchange.LogService;
  import knowledge.*;
  import dataprovider.net.*;
  import dataprovider.search.*;
  import dataprovider.enumerationobjects.*;
  import java.util.LinkedList;

public class NathanSearchOperator
  extends Operator
{
  public NathanSearchOperator () {
    // ...
  }

  public void eval ()
    throws OperatorFailedException {
    // ...
  }
}
```

Die richtige Implementierung des Konstruktors garantiert immerhin schon, daß die Verbindung nach Außen klappt. Alle Planer könnten mit einem solchen Operator arbeiten, da sie ja nur interessiert, was vorne reingeht und was hinten wieder herauskommt. Deswegen ist das Schreiben des Konstruktors, sind Sie sich erst einmal über die gewünschten Vor– und Nachbedingungen im Klaren, eine eher einfache und mechanische Arbeit.

```
public NathanSearchOperator () {
```

Kümmern wir uns zuerst um die Vorbedingung. Davon brauchen wir genau eine:

```
  precond = new Predicate[1];
```

In der Beschreibung des Nathan–Operators haben wir uns ein einstelliges Vorbedingungsprädikat vom Konzept `searchstring` überlegt, daß wir jetzt direkt in der Vorbedingungsliste (in der Implementierung von BotIshelly als Array realisiert) erzeugen:

```
  precond[0] = new Predicate(1,"searchstring");
```

Jetzt müssen wir nur noch ein Prädikatargument mit einem Namen `x` erzeugen und an die einzige Stelle unseres einstelligen Prädikats einfügen (Array–Index 0):

```
  PredArgument x = new PredArgument("x");
  precond[0].setPredArg(x, 0);
```

Das *concept* des Prädikatarguments haben wir nicht zugewiesen, da dieses von der Operatorendatenbank aus dem Namen des einstelligen Prädikats automatisch geschlossen und eingesetzt wird. Genau darum können Sie für Namen einstelliger Prädikate ja auch nur gültige Konzeptnamen benutzen!

Nachdem wir jetzt die Vorbedingung haben, erzeugen wir zuerst eine Nachbedingungslist mit Platz für zwei Nachbedingungen

```
    postconds = new Postcondition[2];
```

und dann zuerst eine Nachbedingung ohne (also mit leeren) Add– und Deletelisten

```
    postconds[0] = new Postcondition();
```

und dann die zweite Nachbedingung mit dem Prädikat `urlmenge` in der Addliste

```
    postconds[1] = new Postcondition();
    Predicate pred = new Predicate(1, "urlmenge");
    PredArgument y = new PredArgument("y");
    pred.setPredArg(y, 0);
    postconds[1].addAddListPred(pred);
  }
```

Das war es auch schon für den Konstruktor. Eigentlich ganz einfach ...

Wesentlich spannender wird es jetzt bei der Methode `eval` die, wir erinnern uns, die ganze Arbeit machen muß. Denn ansonsten kann der Planer einen noch so hübschen Plan aufstellen, der Planausführer kann mit Operatoren, die nichts oder andere Sachen machen, nichts Sinnvolles produzieren.

```
  public void eval ()
    throws OperatorFailedException {
```

Da wir ja nur mit konkreten Instanzen arbeiten können, müssen wir uns erst einmal die (wir haben nur eine) Instanz der Vorbedingung holen. Der String, den wir suchen wollen, steht im Namen der ersten *Instance* im ersten *Predicate* der Vorbedingung, auf den wir mit folgender Kodezeile „direkt“ zugreifen können:

```
    String searchquery = precond[0].getPredArg(0).getInstance().name;
```

Um überhaupt Suchen zu können, müssen wir uns erst einen auf Nathan passenden Suchservice besorgen und uns natürlich auf eventuelle Ergebnisse vorbereiten:

```
    NathanSearchService  service = new NathanSearchService();
    SearchNetResult results;
```

Jetzt sind wir auch schon bereit und können unsere Suchanfrage endlich starten. Danach müssen wir darauf gefaßt sein, diverse Exceptions, die bei der Benutzung des Internets auftauchen können, abzufangen. Das sollte Ihnen aus Java aber bekannt sein, weswegen wie hier nicht weiter darauf eingehen. Anzumerken ist allerdings, daß, falls die Suchmaschine keine Ergebnisse liefert, die leere, erste Nachbedingung (mit dem Array–Index 0) als erfüllt markiert wird:

```
    try {
      results = service.doQuery( searchquery );
    } catch (SearchServiceNoConnectException e1) {
      throw new OperatorFailedException
                                ("Could not connect to SearchEngine");
    } catch (SearchServiceNoResultsException e2) {
      try {
        setPostcondFullfilled(0);
      } catch (exceptions.OperatorNoSuchPostconditionException e3) {
        LogService.logFatal
             (this,"Exception "+e3+": non-existing postcondition!");
      };
      return;   /* "dreckiger" Sprung aus dieser Methode */
    };
```

Das „dreckige“ `return` ist nötig, damit die Methode `eval` im Falle eines Scheiterns sofort verlassen wird.

Nehmen wir einmal an, daß unser Operator nicht gescheitert ist! Die Addliste hat nur ein Prädikat, das wir uns erst einmal „aus den Tiefen“ holen. Zusätzlich erzeugen wir eine neue Java–Instanz der Klasse `Instance`, mit genau dem Konzept des Prädikatarguments in der Addliste:

```
    Predicate postPred =
         (Predicate) postconds[1].getAddList().listIterator().next();
    Instance postPredInstance =
           new Instance(results,postPred.getPredArg(0).getConcept());
```

Jetzt bleibt uns eigentlich nur noch, die neue Instanz auch in das Prädikatargument einzutragen und die zweite Nachbedingung als die erfüllte zu markieren. Den Array–Index 1 haben wir natürlich erzeugt (siehe den Konstruktor weiter oben), müssen aber trotzdem die Exception abfangen, die geworfen würde, wenn der Index nicht existieren sollte:

```
    postPred.getPredArg(0).setInstance(postPredInstance);

    try {
      setPostcondFullfilled(1);
    } catch (exceptions.OperatorNoSuchPostconditionException e4) {
        LogService.logFatal
             (this,"Exception "+e4+": non-existing postcondition!");
    };
  }
```

Das war's! Allerdings müssen wir Sie darauf hinweisen, daß der obige Operator ein sehr einfacher Vertreter seiner Gattung ist. Trotzdem hoffen wir, daß Sie nun eine ungefähre Vorstellung von dem haben, was Sie bei der Programmierung von Operatoren erwartet!

7.6 Spezielle Operatoren

Classifier–Operatoren

Die Klasse `ClassifierOperator` leitet sich direkt aus `Operator` ab und stellt eine Klasse von Operatoren dar, die wohl mit am häufigsten zur Anwendung kommen wird: Operatoren, die eine Eingabe daraufhin untersuchen, ob sie unter ein bestimmtes Konzept fallen. Ein derartiger Operator muß zu jedem Konzept geschrieben werden, zu dem hin Weltobjekte klassifiziert werden sollen; mit Ausnahme von `anything` und `nothing` in der Regel also zu allen Konzepten.

Glücklicherweise weiß jedes T–Box–Konzept, durch welchen Klassifizierer es eine Instanz auf Zugehörigkeit testen kann. Darum ist es möglich, einen allgemeinen `ClassifierOperator` vorzugeben, der nur noch im Konstrukturaufruf

```
ClassifierOperator(Concept postcondConcept,
                   Concept precondConcept)
```

mit den Konzepten des (im Erfolgsfalle) Ergebnisses und der Eingabe gefüttert werden muß.

Der Operator, den Sie erhalten, hat als Vorbedingung ein Prädikat des Konzeptes `precondConcept` und zwei Nachbedingung: ein Prädikat des Konzeptes `postcondConcept` in der Addliste bzw. eine leere Nachbedingung, falls die Eingabe nicht unter das `postcondConcept` klassifiziert werden konnte.

Die `eval()`–Methode ist für alle Classifier–Operatoren gleich und muß von Ihnen nicht neu geschrieben werden. Eigentlich müssen Sie nicht einmal die Classifier–Operatoren selbst schreiben, da Ihnen diese Arbeit von der Operatorendatenbank abgenommen wird (vgl. II 8.3).

Compound–Operatoren

Manchmal kann es sinnvoll sein, Folgen von Operatoren zusammenzufassen. Das könnte z. B. eine immer wieder auftretende Folge bestimmter Operatoren sein. Der Klasse `CompoundOperator` stehen durch Implementierung des `List`–Interfaces Methoden zur Verfügung, um auf den Compound–Operator wie auf eine Liste zuzugreifen. Dadurch ist es möglich, Operatoren der Reihenfolge, in der sie ausgeführt werden sollen, in den Compound–Operator zu speichern. Diese Art von Operatoren soll auch vom Operatorenlerner erzeugt werden.

Bei der Erstellung eines Compound–Operators ist zu beachten, daß ihm an Informationen nur die Dinge zur Verfügung stehen, die er durch die Vorbedingung erhält und solche Informationen, die er selber erzeugt. Die Methode `eval()` arbeitet die enthaltenen Operatoren der Reihe nach ab (muß von Ihnen also nicht mehr geschrieben werden), ohne Vor– oder Nachbedingungen zu belegen. Diese Belegung liegt ganz allein in der Verantwortung des Bauers des Compound–Operators. Also in Ihrer oder in der des Operatorenlerners. Dazu müssen Sie im Konstruktor des Compound–Operators eine Instanz, die z. B. in einem Teiloperator erzeugt wird, in einem anderen als Vorbedingung auftaucht erzeugen und an beiden Stellen in die Prädikatargumente der Vor– und Nachbedingung einhängen. Dazu kann es nötig sein, daß Ihre Teiloperatoren vor dem Erzeugen einer neuen Instanz überprüfen, ob die Instanz vielleicht schon erzeugt ist! Durch das Neuerzeugen einer Instanz würde nämlich der Instanzenfluß unterbrochen und der Compound–Operator immer scheitern!

Die Teiloperatoren werden in serieller Reihenfolge ausgeführt. Es ist nicht möglich, Verzweigungen auszuführen. Das bedeutet aber auch, daß die Anzahl der Nachbedingungen des zuletzt ausgeführten Operators gleich der Anzahl der Nachbedingungen des gesamten Compound–Operators sein muß! Denn es gibt für den Compound–Operator keine Möglichkeit, noch mal abschließend das Ergebnis des letzten Teiloperators zusammenzufassen o. ä. Der Index der im letzten Teiloperator gültigen Nachbedingung ist gleichzeitig auch der Index der gültigen Nachbedingung des gesamten Operators.

Der Compound–Operator scheitert spätestens dann, wenn eine seiner Komponenten scheitert.

Kapitel 8

Operatorendatenbank

Wollten wir alle Herren sein, wer sollte da die großen Säcke tragen?

Aus Dänemark

8.1 Operatorendatenbank im Überblick

Die `OperatorDB` ist eine aus `dataanalysis.databases.GenericDatabase` abgeleitete Datenbank, die die von Ihnen, dem Shellbenutzer, geschriebenen Operatoren enthält. Der Agent kennt nur die Operatoren, die in der `OperatorDB` enthalten sind. Darum muß die Datenbank vor dem ersten Lauf eines Agenten erst einmal mit zu den Anforderungen des Agenten passenden Operatoren gefüllt werden. Die einzelnen Operatoren werden durch eine ID unterschieden, die innerhalb der Datenbank eindeutig ist. Wenn Sie ein und den selben Operator aus Versehen zweimal in die Datenbank speichern, so haben sie den Operator nachher auch doppelt; unter verschiedenen IDs, da diese von der DB fortlaufend vergeben werden. Die Operatoren, die in der Datenbank stehen, sind als Muster zu verstehen, von denen an den Agenten nur jeweils eine Kopie auf Anforderung weitergereicht wird.

8.2 Die Datenbank als Datenbank

Die Operatorendatenbank besitzt zwei Konstruktoren:

```
OperatorDB(T_Box tBox, Concept c, boolean onlySpecialized)
OperatorDB(String file, T_Box tBox, Concept c, boolean onlySpecialized)
```

Die drei Parameter `tBox`, `c` und `specialized` werden im Abschnitt II 8.3 erläutert und dienen der automatischen Generierung der Classifier–Operatoren. Die T–Box dient zusätzlich dazu, die Operatoren aus einer Textdatei (siehe folgenden Absatz) erzeugen zu können.

Wird dem Konstruktor zusätzlich noch ein Dateiname als `String` übergeben, so versucht die Datenbank, ihre Operatoren aus der Datei zu laden. Bei der Datei handelt es sich um eine Textdatei, die zeilenweise wechselnd die Klassennamen und das Gewicht (siehe nächsten Absatz) der Operatoren enthalten. Um z. B. aus dem Paket `operators` die Operatorklasse `PrintlnOperator` mit einem Gewicht von 3.0 zu generieren, so müssen in der entsprechenden Datei die folgenden beiden Zeilen stehen:

```
operators.PrintlnOperator
3.0
```

Das Generieren aus der Datei funktioniert natürlich nur, wenn die Systemvariable `CLASSPATH` das Verzeichnis enthält, in dem sich das Paket `operators` befindet.

Das Gewicht eines Operators gibt an, wie „erfolgsversprechend" die Anwendung dieses Operators ist und soll dem Planer die Entscheidung zwischen Operatoren erleichtern. Eine Lernkomponente kann aufgrund von Erfahrungen, die mit einzelnen Operatoren gemacht wurden, dieses Gewicht verändern, wobei größere Werte einer höheren Erfolgswahrscheinlichkeit entsprechen. Um das Gewicht eines Operators nachträglich zu ändern, sollte immer die Methode `updateWeightMap (double newWeight, Operator op)` in der `OperatorDB` benutzt werden. Dadurch wird sichergestellt, daß Änderungen am Gewicht eines Operators sich auch auf die Datenbank auswirken, so daß ein später angeforderter Operator derselben ID auch das neue Gewicht hat.

Die aus der Datei geladenen Operatoren werden über den leeren Konstruktur erzeugt, weshalb alle aus `Operator` abgeleiteten Operatoren diesen implementieren müssen. Dabei werden die IDs an die Operatoren nach Reihenfolge des Einlesens aufsteigend vergeben. Die Nummer eines Operators könnte sich also bei einem Neueinlesen ändern, so daß bei der Benutzung der Lernerklassen darauf zu achten ist, mit welcher Version der Operatorendatenbank das entsprechende Planarchiv erzeugt wurde. Eine Änderung der IDs ist allerdings nur zu erwarten, wenn Sie die Datenbank wirklich neu aufbauen und die Operatoren in anderer Reihenfolge angeben. Denn beim Löschen eines Operators aus der DB wird die Stelle einfach freigehalten, so daß keine Verschiebung der IDs eintritt; genausowenig tritt beim Einfügen eines neuen Operators eine Verschiebung ein, da der Operator am Ende der DB eingefügt wird.

Nachdem man die Datenbank gefüllt hat, sei es über eine Datei oder durch Benutzen von `addOperator (Operator newOp)`, wird die Hauptaufgabe der DB darin liegen, auf Anfrage Operatoren zu liefern. Die beiden wichtigsten Methoden dazu sind:

```
public Operator getOperator (long id)
public Operator getNewOperator (long id)
```

Von diesen möchte man in der Regel die zweite aufrufen, da sie eine Kopie des in der Operatorendatenbank unter der Nummer ID gespeicherten zurückliefert, nicht das Original. Der Unterschied ist wichtig, da ein Operator durchaus häufiger als nur einmal angewendet werden kann. Würde auf dem Original gearbeitet, so machten sich Änderungen an der einen Stelle eines Plans gleich an allen Stellen bemerkbar, an denen eine Referenz auf dieses Original steht. Da die Operatoren an anderen Stellen normalerweise auch in anderem Kontext stehen, ist dieser Effekt nicht erwünscht.

Zur Erleichterung stehen speziell für den Planer noch Methoden zur Verfügung, auf die bei der Dokumentation der Klasse `Planner` (siehe II 10.4) noch hingewiesen wird.

8.3 Classifier–Operatoren leichtgemacht

Wie schon angedeutet, die Classifier–Operatoren, die für die einzelnen Konzepte der T–Box gebraucht werden, müssen Sie nicht selber schreiben: diese Arbeit wird Ihnen von der Operatorendatenbank abgenommen. Dazu dienen die drei Parameter `T_Box tBox`, `Concept c` und `boolean onlySpecialized` in den beiden Konstruktoren von `OperatorDB`.

In der übergebenen T–Box finden sich alle Konzepte und Rollen, die wiederum die entsprechenden Klassifizierer kennen, die für sie zuständig sind. Dadurch wird die automatische Generierung der Classifier–Operatoren erst möglich.

Klassifizierer bekommen Instanzen eines Konzeptes als Eingabe und stellen fest, ob es sich hierbei um Instanzen eines Unterkonzeptes handelt. Es wird also immer von einem Oberkonzept zu einem

Unterkonzept hin klassifiziert. Prinzipiell ist zu jeder Kombination aus Ober– und Unterkonzept ein Klassifizierer denkbar. Eine `Ente` kann unter das Konzept `Wasservogel` fallen, das wiederum Unterkonzept zu `Vogel` ist. Statt zwei Klassifizierer `Vogel`⟶`Ente`, `Wasservogel`⟶`Ente` zu bauen, kann man sich auf den allgemeineren ersten Fall beschränken, da damit unter anderem auch Wasservögel als Enten klassifiziert werden können. Gibt es ein gemeinsames Oberkonzept aller Konzepte, zu denen hin klassifiziert werden soll, so reicht es also, wenn A–Box–Instanzen dieses Oberkonzeptes Vorbedingung der Klassifikatoroperatoren sind. Da `anything` alle Konzepte subsumiert, gibt es ein solches Oberkonzept immer. Natürlich muß zu jedem Unterkonzept dieses Oberkonzeptes ein Klassifikator existieren! Daher ist es sinnvoll, das Oberkonzept so speziell wie möglich zu wählen. Da in der Internetdomäne nur URLs als Weltobjekte neu erzeugt werden können, reicht die Wahl des Konzepts `URL` als Obermenge aus. Dieses eine Oberkonzepte wird der Operatorendatenbank im Konstruktor übergeben.

Würde als Oberkonzept `Vogel` übergeben, so erzeugt die Datenbank die Klassifikationsoperatoren zu jedem Unterkonzept von `Vogel`; also auch `Vogel`⟶`Wasservogel` und `Vogel`⟶`Ente`. In manchen Domänen kann es ausreichend sein, nur zu den Blättern des Subsumptionsgraphen hin zu klassifizieren. Wenn Sie also wissen, daß Sie immer A–Box–Instanzen zu den Klassen `Ente`, `Gans`, `Schwan`, usw. haben werden, niemals aber eine A–Box–Instanz zu `Wasservogel`, dann brauchen Sie auch keinen Klassifikationsoperator, der von `Vogel` nach `Wasservogel` klassifiziert (und damit auch keinen Klassifikator). Wenn Sie im Konstruktor der Datenbank `onlySpecialized` mit dem Wert `true` übergeben, so werden nur Klassifikationsoperatoren erzeugt, die vom gemeinsamen Oberkonzept hin zu den spezialisiertesten Unterkonzepten — den Blättern im T–Box–Graphen — gehen. Die nicht erzeugten Klassifikationsoperatoren kennt nun auch der Planer nicht, was den Planungsvorgang beschleunigen kann.

Zu Klassifikatoren, siehe Kapitel II 13, zum Planer Kapitel II 10.

Kapitel 9

Planinformation

> Es genügt nicht, zum Fluß zu kommen mit dem Wunsche, Fische zu fangen. Man muß auch das Netz mitbringen.
>
> *Aus China*

9.1 Planinformation im Überblick

Das Planinformationsobjekt dient zum einen dem Daten– und Informationsaustausch zwischen Systemkontrolle, Planern, Planausführung und Planarchiv. Zum anderen werden durch dieses Objekt die Planungsziele erzeugt und die Ergebnisse der Suche gefiltert.

Die Mächtigkeit dieses Objekts können Sie gut an der Breite des Kontruktors erkennen:

```
PlanInformation (long planID, long searchID,
                 A_Box localABox, T_Box tBox,
                 OperatorDB operatorDB)
```

Zusätzlich zu diesen Angaben wird vom Planer im Planinformationsobjekt der auszuführende Plan abgelegt, der nach erfolgreicher Ausführung und vor dem Abspeichern im Planarchiv in einen Lernbaum umgewandelt wird. Ergänzt werden diese „größeren“ Objekte noch durch einige boolesche Flags, die Auskunft über das Ergebnis der Planausführung geben und/oder bestimmen, wie mit dem Planinfo weiter verfahren werden soll.

Erzeugt wird ein `PlanInformation`–Objekt erstmalig in der Klasse `systemcontrol.Flowcontrol`, die auch die eindeutigen IDs für den planerstellenden Planer (`planID`) und die Suche (`searchID`) generiert.

9.2 Zielextraktion

Die Eingabedaten, die der Agentenbenutzer später an Ihren Agenten schickt, müssen für einen Planer erst in eine sinnvolle Form gebracht werden. Dazu ist es notwendig, die beiden Variablen `targetPredicates` und `startPredicates`, die als `Predicate[]` definiert sind, zu belegen. Dazu werden innerhalb des Konstruktor von `PlanInformation` zwei Methoden aufgerufen:

```
generateTargetPredicates();
generateStartPredicates();
```

Die erste Methode versucht, aus den über `tBox.getTargetConcepts()` und `tBox.getTargetRoles()` beschafften Informationen über die Benutzereingabe die Prädikate zu erzeugen, die der Planer mindestens mit einem Plan erreichen muß. Leider ist eine automatische Generierung der von der Suchdomäne und den Benutzereingaben abhängigen Zielprädikate zum jetzigen Zeitpunkt nicht möglich, so daß Sie diese Methode überschreiben müssen, damit der von Ihnen zusammengebaute Agent auch weiß, wonach er suchen soll!

In der zweiten Methode werden die Instanzen, die sich in der A–Box schon befinden und somit das Wissen beschreiben, was der Planer voraussetzen darf, in Prädikate umgewandelt. Bitte beachten Sie an dieser Stelle, daß Planer und Planausführer dazu ausgelegt sind, auf Instanzen der Klasse `Predicate` zu arbeiten, die A–Box aber nur auf Instanzen der Klasse `Instance`. Die Umwandlung ist also unbedingt nötig! Diese Methode muß von Ihnen aber nicht überschrieben werden.

9.3 Ergebnisfilterung

Nachdem die Planausführung erfolgreich beendet wurde, sollten die Ergebnisse, die dann hoffentlich vorhanden sind, dem Benutzer auch in irgendeiner Form zur Verfügung gestellt werden. Dazu besorgt sich ***FlowControl*** die Ergebnisse aus dem Planinformationsobjekt über die Methode `getResults()`. Diese Methode ruft intern `verify()` auf, wobei die Implementierung letzterer bei Ihnen liegt!

`verify` belegt die Variable `goalInstances`, ein Array von `Result`, das nachher von `getResults` zurückgegeben wird. Die Aufgabe von `verify` ist es nun, aus der A–Box die Instanzen herauszufiltern, von denen Sie glauben, daß sie der Benutzer gesucht haben könnte. Eine einfache Heuristik wäre, immer die ersten zehn gefundenen Instanzen zurückzuliefern. Sie können die Ergebnisse auch noch einmal klassifizieren und sie dann sortiert nach Relevanz (was auch immer Sie dann als Relevanz definieren) zurückliefern. Auf jeden Fall ist es Ihre Aufgabe, diese Methode für Ihren Agenten entsprechend zu überschreiben!

9.4 Der Planbaum

Der Planbaum stellt eine rekursive Struktur zur Verfügung, in der Operatoren in den Knoten stehen. Jeder Knoten hat so viele Kanten wie Nachbedingungen, wobei bei der Planausführung jeweils der Operator als nächstes ausgeführt wird, der an der Kante der erfüllten Nachbedingung des vorher ausgeführten Operators hängt.

Jeder Planer muß in der Lage sein, einen Planbaum zu erzeugen, auf dem ein Planausführer dann arbeiten kann. Zum erzeugen stehen als wichtigste Methoden der Konstruktor

```
public PlanTree (Operator operator)
```

sowie

```
public PlanTree setNext (PlanTree next, int i)
```

zur Verfügung. Durch den Parameter `i` von `setNext` wird als Index der Nachbedingung des aufrufenden Planbaumknotens interpretiert und der übergebene Planbaum `next` an die entsprechende Stelle eingehängt.

Jede Kante einer Nachbedingung verfügt im Planbaum über zwei boolesche Werte, von denen der eine vom Planer, der andere von der Planausführung gesetzt wird. Ersterer, `goalReached`, kennzeichnet, daß bei Erreichen dieser Nachbedingung im Plan genügend Instanzen bei der Ausführung gesammelt wurden, um das oder die Ziele zu erreichen. Für den Planausführer ist das

das Zeichen, da daß die Suche erfolgreich abgeschlossen wurde. Zweitere Variabe, `goalPath`, wird vom Planausführer gesetzt, wenn die Nachbedingung auf einem Pfad liegt, der zum Ziel geführt hat. In einem Planbaum können durchaus mehrer Pfade als zum Ziel führend markiert sein, da der Planausführer durch Backtracking mehrmals auf unterschiedlichen Wegen zum Ziel kommen kann. Zur Markierung gibt es die Methode `markGoalPath()`, die den Pfad von der erfüllten Nachbedingung bis abwärts zur Wurzel markiert und gleichzeitig bei den Instanzen, die in den Knoten über die PredArgs in den Operatoren hängen, das Flag `successfull` auf `true` gesetzt.

Der Planbaum ist nicht dazu gedacht, in das Planarchiv gespeichert zu werden. Darum muß er vor dem Abspeichern des Planinformationsobjektes mit Hilfe von `PlanInformation.createLearnTree()` in einen Lernbaum umgewandelt werden.

9.5 Der Lernbaum

Der Lernbaum ist eine Variante des Planbaums, die sich von diesem hauptsächlich dadurch unterscheidet, daß sie keine Instanzen mehr enthält. In den Knoten des Lernbaums hängen sogenannte Lernoperatoren, abgespeckte „echte“ Operatoren. Der Lernbaum wurde eingeführt, um nur die Informationen zu speichern, die die Lernerklassen wirklich brauchen.

Da der Lernbaum über die Möglichkeit verfügt, sich als XFig–Grafik auszugeben, kann er von Ihnen sehr gut dazu benutzt werden, Kontrollausgaben der vom Planer erzeugten oder vom Planausführer ausgeführten Pläne zu machen. Der Aufruf von `exportFig ("Kontrolle")` erzeugt so z. B. die Files Kontrolle.fig und Kontrolle.txt, wobei die Nummern in der grafischen Ausgabe den Knotennummern in der textuellen Ausgabe entsprechen. In der grafischen Ausgabe entspricht ein grün gefüllter Kreis einer Nachbedingung, die zum Ziel führte. Alle anderen sind rot. Ist ein Nachbedingungskreis von einem weiteren Kreis umgeben, so ist für diese Nachbedingung das Flag goalReached gesetzt.

Kapitel 10

Planer

> Pläne machen ist mehrmalen eine üppige, prahlerische Geistesbeschäftigung, dadurch man sich ein Ansehen von schöpferischem Genie gibt, in dem man fordert, was man selbst nicht leisten, tadelt, was man doch nicht besser machen kann, und vorschlägt, wovon man selbst nicht weiß, wo es zu finden ist.
>
> *Immanuel Kant*
> Prolegomena 4, 10

10.1 Planer im Überblick

Wenn Sie ein Ziel vor Augen haben, dann können Sie mehrer Arten versuchen, es zu erreichen. Eine Variante wäre, blindlings loszusuchen und zu hoffen, daß Sie schon irgendwann über das eigentlich Gesuchte stolpern werden. Eine andere Möglichkeit ist, sich vorher Gedanken darüber zu machen, wie das Ziel am Besten zu erreichen ist und dem Plan, den Sie dabei erstellen, dann zu folgen. Oft wird es so sein, daß das Gelingen des Plans von allerlei Bedingungen abhängt. Wenn Sie auf eine Art und Weise nicht zum Ziel gelangen, so kommen Sie eventuell auf andere Weise zum Ziel. Manchmal ist es dann notwendig, daß Sie Ihren Plan umschreiben, da Ereignisse eingetreten sind, die Sie nicht vorhergesehen haben. Natürlich kann es auch sein, daß Sie feststellen, daß das von Ihnen gesteckte Ziel gar nicht zu erreichen ist. Stellen Sie das schon bei der Planung fest, so brauchen Sie gar nicht erst loszulaufen.

Einen solchen Plan zu erstellen, das ist die Aufgabe, die eine von der abstrakten Klasse `Planner` abgeleitete Klasse erledigen soll. Der fertige Plan, also die Anweisung, was in welchem Falle zu machen ist, wird dann an den Planausführer gegeben, der den Plan in die Tat umzusetzten versucht. Der Plan, den der Planer erzeugt, muß als `PlanTree` (siehe II 9.4) vorliegen, damit der Ausführer etwas damit anfangen kann.

10.2 Die abstrakte Klasse `Planner`

Aus der abstrakten Klasse `Planner` im Paket `planner` muß der von Ihnen benutzte Planer abgeleitet sein. Als einzige — wenn auch mächtige — Informationsquelle, dient Ihrem Planer dabei das Planinformationsobjekt, was im Konstruktor des Planers übergeben wird. Hier findet der Planer die A– und T–Boxen, die für die Suche relevant sind, hier findet sich die Operatorendatenbank, mit deren Hilfe Operatoren für den Planbaum ausgewählt werden können.

Um zu sicherzustellen, daß eine Systemkontrolle mehrere Planer gleichzeitig beschäftigen kann, ist `Planner` von `Thread` abgeleitet. Daraus folgt aber auch, daß Sie entscheiden müssen, wie oft und an welchen Stellen in Ihrem Planer Sie den Wert der booleschen Flags `destroyThread` und `suspendend` abfragen und darauf so reagieren, daß Ihrem Planer keine Informationen verloren gehen können. Die Zuordnung der Belegung der Flags zu den Aktionen, die ausgeführt werden wollen, sehen so aus:

`destroyThread == true`	$\longrightarrow$	destroy
`suspended == false`	$\longrightarrow$	resume
`suspended == true`	$\longrightarrow$	suspend

Nachdem Ihr Planer von der Systemkontrolle erzeugt wurde, wird die Methode `eval()` aufgerufen, die als Rückgabewert ein `PlanInformation` enthält, daß dann (hoffentlich) einen ausführbaren Plan enthält. Aus diesem Grund müssen Sie in einem eigenen Planer diese Methode überschreiben und mit Sinn, Verstand und wahrscheinlich auch Intuition füllen! Standardmäßig gibt diese Methode nämlich `null` zurück, was Ihnen auf dem Weg zum Ziel nicht viel helfen wird ;-)

Zwei Beispiele für eine Implementierung von `eval` finden Sie in den Klassen `PlannerTypeA` und `PlannerTypeC` im Paket `planner.plannertyes` (s. a. II 10.5 und II 10.6), anhand derer Sie auch die Benutzung des Eventhandlings sehen können.

10.3 Keine Verbindung? Weiterreichen von Instanzen

Ein Planer, der Operatoren zu einem Plan zusammenbaut, möchte in aller Regel, daß der Planausführer dort später Daten fließen läßt. Diesen Datenfluß muß aber bereits der Planer sicherstellen, da der Ausführer naturgemäß keine Ahnung davon hat, was sich ein Planer beim Aufstellen des Plans so alles gedacht hat. Im Normalfall wird der Ausführer den Ersteller des Plans nicht einmal kennen.

Wenn Ihr Planer einen Operator anwendet, dann soll das Ergebnis meistens weiterverwendet werden; vor allem dann, wenn Ihr Operator neue Weltobjekte erzeugt. Darum muß der Planer schon konkrete Java–Instanzen der Klasse `Instance` erzeugen, auch wenn er sie noch nicht mit Inhalt füllt oder füllen kann. Damit diese Instanz von Operator A zu einem Operator G weitergereicht werden kann, muß der Planer diese Instanz schon in die entsprechenden Prädikatargumente einhängen. Dadurch ist sichergestellt, daß der Planausführer, wenn er die Instanz in Operator A mit Inhalt füllt, beim Erreichen von Operator B schon eine Referenz auf eine Instanz vorfindet, nämlich genau die aus Operator A. Dann wird der Ausführer auch nicht mehr versuchen, dieses Instanz mit einem anderen Inhalt zu füllen.

Wenn Sie also einen eigenen Planer schreiben wollen, dann müssen Sie darauf achten, daß der Instanzenfluß gewährleistet ist!

10.4 Hilfestellung durch die Operatordatenbank

Eine Hilfestellung erhalten Sie allerdings durch die Operatorendatenbank, die Ihnen einige Funktionen zur Verfügung stell, mit denen Sie eine Auswahl aus den möglichen Operatoren treffen können. Da Sie Operatoren mit Ihren Vor– und Nachbedingungen zu einem Planbaum zusammenstecken müssen, können Sie sich mit den beiden Methoden

```
getAllOperators4PrecondPred (Predicate p)
getAllOperators4PostcondPred (Predicat p)
```

einen `ListIterator` auf alle Operatoren zurückgeben lassen, die das Prädikat `p` in der Vor– bzw. Nachbedingung haben.

Es kann sinnvoll sein, zuerst die Operatoren mit der besten Gewichtung anzuwenden. Auf diese können Sie mit der Methode

```
getBestOperators ()
```

zugreifen, die einen `ListIterator` über die Operatoren, absteigend sortiert nach Gewichtung, zurückliefert. Die Methode `getBestOperators` liefert übrigens schon Klone der Operatoren aus der Datenbank, Sie müssen also nicht anhand der ID mit `getNewOperator (ID)` noch eine Kopie anfordern. Die Verwendung der Operatoren mit höchster („bester") Gewichtung stellt auf einfache Weise sicher, daß Operatoren, die durch die Lernkomponenten als „erfolgreich" bewertet wurden, häufiger angewendet werden, da sie weiter vorne in der Liste stehen. So kann Ihr einmal geschriebener Planer auf die Ergebnisse der Lerner reagieren, ohne daß Sie den Planerkode ändern müssen.

10.5 Planer Typ A

Die Klasse `PlannerTypeA` implementiert einen Planer, der jeweils einen „Plan" erstellt, der aus einem Operator besteht, diesen zur Ausführung bringt und aufgrund des Ergebnisses den nächsten Schritt plant. Dieser Planer ähnelt reaktiven Planern, unterscheidet sich von diesen aber deutlich durch die mangelnde Fähigkeit, Sensoren zur Orientierung einsetzen zu können. Klassische Sensoren („Welches Signal zeigt die Ampel?") haben im Internet scheinbar keine Entsprechung.

Der Planer Typ A verfolgt eine sehr einfache Strategie, um das Ziel zu erreichen: die Operatoren werden in der Reihenfolge ihrer Gewichtung ausprobiert. Nach einer gewissen Anzahl von Schritten wird versucht, einen speziellen Operator anzuwenden, der als Vorbedingung die Zielprädikate hat. Kann dieser Operator von der Planausführung erfolgreich instantiiert werden, so war der Plan erfolgreich und die Suche kann beendet werden. Die Suche ist auch beendet, allerdings abgebrochen, wenn der Planer feststellt, daß er alle möglichen Operatoren hintereinander ausgeführt hat, *ohne* daß sich sein Wissen über die Welt verändert hat. Da das Internet für die Dauer der Suche als statisch angesehen wird, würde auch keine weitere Anwendung der Operatoren ein neues Ergebnis liefern.

Bemerkenswert ist der Planer vom Typ A, weil er in seiner Strategie einfach, in seiner Handhabung aber recht anspruchsvoll ist. Im Gegensatz zu „echten" Planern, die einen Plan erstellen und damit fertig sind, ist der Typ A–Planer auf die Ergebisse angewiesen, die der Planausführer liefert. Da der Aufruf des Planausführers nur über die Systemkontrolle geschehen kann, benutzt der Planer die Events

```
planReadyUnfinished
planExecuteFinished
```

um die gewünschte Weiterverarbeitung zu kontrollieren. Dabei löst das Event `planReadyUnfinished` als Reaktion aus, daß der Planer suspended und die Planausführung mit dem Planinformationsobjekt gestartet wird. Der Planer hat vorher im Planinformationsobjekt das Flag `backToSender` gesetzt, was später den Planausführer veranlassen wird, das Info–Objekt über die Systemkontrolle wieder an den ursprünglichen Planer zurückzuschicken.

Das zweite Event, `planExecuteFinished`, besagt der Systemkontrolle, daß der Planer Typ A seine Aufgabe erfüllt hat. Die Systemkontrolle beendet den Planer und kann daraufhin entscheiden, ob der fertige Plan ein Teilplan aus einem größeren Kontext war und einen anderen Planer mit der Weiterbearbeitung beschäftigen (vgl. dazu auch den Abschnitt über das Eventhandling in der Systemkontrolle).

10.6 Planer Typ C

Der Planer Typ C ist eine Variante von GraphPlan [Blum und Furst, 1997], der als derzeit schnellster Planalgorithmus für STRIPS–beschränkte klassische Planungsdomänen gilt. Der Hauptunterschied zum klassischen GraphPlan besteht in der Variante, die `PlannerTypeC` benutzt (PG 343–GP), besteht in der Hinzunahme dynamisch wachsender Objektwelten und dem Zulassen disjunktiver Nachbedingungen.

Um Operatoren mit mehreren Nachbedingungen anwenden zu können, werden diese intern zu Nachbedingung–vielen Operatoren aufgesplittet, wobei die i–te Nachbedingung auf den i–ten Operator abzielt. Durch diese Modifikation greift der ursprüngliche GP–Algorithmus.

Nach Erstellen des Plans wird das Event `planReadyFinished` ausgelöst, was die Weiterreichtung des Planinformationobjektes an eine Planausführung veranlaßt.

Kapitel 11

Planausführung

> Es ist im Leben wie im Schachspiel: Wir entwerfen einen Plan; dieser bleibt jedoch bedingt durch das, was im Schachspiel dem Gegner, im Leben dem Schicksal zu tun belieben wird. Die Modifikationen, welche hierdurch unser Plan erleidet, sind meistens so groß, daß er in der Ausführung kaum noch an einigen Grundzügen zu erkennen ist.
>
> *Arthur Schopenhauer*
> Aphorismen zur Lebensweisheit V, 48

11.1 Planausführung im Überblick

Aus eigener Anschauung wissen Sie wahrscheinlich, daß derjenige, der einen Plan aufstellt, diesen nicht unbedingt auch persönlich ausführen muß. Bei BotIshelly ist es so, daß der Planer keinerlei Möglichkeit hat, selber den entworfenen Plan auszuführen. Hier gibt es die Klasse `PlanExecution`, die ein Planinformationsobjekt entgegennimmt und den dort enthaltenen Planbaum abzuarbeiten versucht.

Grob vereinfacht füllt die Planausführung dazu die Vorbedingung des ersten Operators mit Instanzen aus der A–Box und führt ihn dann aus. Als nächstes wird überprüft, welche Nachbedingung gültig geworden ist und entsprechend zum nächsten Operator verzweigt, bei dem das Spiel wieder von vorne losgeht. Im günstigsten Fall erreicht der Planer eine Nachbedingung, für die `goalReached` gesetzt ist. Dann ist das Ziel erreicht und die Präsentation der Ergebnisse kann von der Systemkontrolle veranlaßt werden.

11.2 Die Klasse `PlanExecution`

Die Klasse `PlanExecution` ist zur Abwechslung mal nicht abstrakt, und Sie müssen auch keine `eval`– oder ähnliche Methode selber schreiben! Der Planausführer von BotIshelly arbeitet mit den Planbäumen ganz zufriedenstellend zusammen ;-)

`PlanExecution` ist aus `Thread` abgeleitet und wird von der Systemkontrolle mittels `run()` gestartet. Vorher muß der Planausführer natürlich über den Konstruktor

```
PlanExecution (PlanInformation planInfo, int maxInstances)
```

erzeugt werden. Alle Informationen, die den Plan und das Wissen betreffen, findet der Planausführer im Planinformationsobjekt. Die zusätzlich übergebene Beschränkung `maxInstances` gibt an, wieviele Instanzen höchstens bei einem Backtracking in Betracht gezogen werden. Dadurch wird verhindert, daß ein Operator, der als Vorbedingung `anything` hat, mit einer unter Umständen sehr großen, kompletten A–Box belegt wird und dadurch Ihren Rechner (oder anderer Leute Rechner) lahmlegt.

Wenn Sie den Planausführer nur ein wenig verbessern wollen, so können Sie am ehesten an den Methoden

```
LinkedList selectInstances (Concept c)
LinkedList selectConcreteRoles (Role r)
```

ansetzen. Diese beiden Methoden besorgen sich aus der A–Box die `maxInstances` ersten Konzepte bzw. konkreten Rollen, die zurückgeliefert werden. Hier können Sie versuchen, aus den Instanzen eine Auswahl zu treffen, die z. B. die mit den höchsten Konfidenzwerten bzgl. des zu belegenden Konzeptes oder der Rolle.

Erreicht der Planausführer eine Nachbedingung, deren `goalReached` gesetzt, ist, so könnte er aufhören und den Pfad im Planbaum als zum Ziel führend markieren. Allerdings wird vom Benutzer in aller Regel mehr als nur ein Ergebnis erwartet. Darum wird an der Stelle ein Backtracking eingeleitet und der Plan bis zu der Stelle zurückgerollt, an der noch eine ausgewählte Instanz nicht benutzt worden war. Dabei müssen Sie natürlich, wollen Sie den Planausführer umschreiben, darauf achten, daß die A–Box auch nach dem Backtracking noch konsistent ist. Insbesondere müssen Sie auf die Instanzen achten, die durch die Add– oder Deletelisten eines Operators hinzugekommen oder gelöscht worden sind!

Die in BotIshelly gewählte Implementierung des Planausführers stellt sicher, daß nur für Instanzen an den Stellen, an denen sie zum ersten Mal in einem Pfad im Planbaum auftauchen, über mögliche Belegungen iteriert wird. Alle weiteren Vorkommen einer Instanz sind aufgrund der vom Planer zu gewährleistenden Gleichheit der Java–Referenzen in den entsprechenden Prädikatargumenten für den Planer schon belegt und werden nicht neu iteriert. Wenn Sie eine eigene Planausführung schreiben wollen, dann müssen Sie hierauf achten! (Vgl. dazu Abschnitt II 10.3)

11.3 Hilfestellung durch die Instanzen und die A–Box

Um das im vorhergehenden Abschnitt angesprochene Problem, hinzugekommene Instanzen löschen zu müssen, gelöschte Instanzen wieder hinzufügen zu müssen, kommen Ihnen die Klassen `Instance` und `A_Box` ein Stückchen entgegen. In `Instance` gibt es das boolesche Flag `delete`, das von der Planausführung gesetzt oder wieder gelöscht werden darf. Eine Instanz, die innerhalb einer Deleteliste auftaucht, müssen Sie also nicht wirklich löschen, sondern können sie nur als gelöscht markieren. Wenn Sie dann ein Backtracking ausführen, setzten sie das Flag auf `false` und Sie haben wieder den alten Zustand erreicht. Sind Instanzen neu hinzugekommen, so markieren Sie sie im Backtrackingschritt als deleted.

Die A–Box reagiert auf das Flag `deleted` in so weit, als Sie von der A–Box keine Konzepte oder konkreten Rollen zurückgeliefert bekommen, in denen eine der Instanzen als gelöscht markiert ist. Nach dem Backtracking verhält sich die A–Box also so, als wären die Instanzen wirklich gelöscht worden. Wie ist das nun mit Instanzen, die über eine Addliste hinzugekommen sind und deren `delete` im Laufe eines Backtrackings auf `true` gesetzt worden ist? Das ist auch kein Problem, da es hier zwei Möglichkeiten gibt:

1. Die Instanz lag nicht auf einem Pfad, der zum Ziel geführt hat.
2. Die Instanz lag auf einem Pfad, der zum Ziel geführt hat.

In erstem Fall ist es nicht schade um die Instanz, da sie uns nicht weitergebracht hat. Im zweiten Fall jedoch ist gleichzeitig für die Nachbedingung, in der die Instanz entstanden ist, das Flag `goalPath` im Planbaum gesetzt. Dadurch wird die A–Box beim Abspeichern daran gehindert, diese Instanz wirklich zu löschen.

11.4 Mengen von Instanzen

Manchmal möchte man Operatoren schreiben, die eine Menge von Instanzen zurückliefern. So ist es z. B. sinnvoll, wenn ein Operator, der eine Suchmaschine befragt, alle Suchergebnisse zurückliefert. Die Prädikatargumente sind in BotIshelly in der Lage, Mengen von Instanzen zu speichern.

Da der Planer die Pläne anhand abstrakter Instanzen erzeugt, ist die Mengenwertigkeit der Prädikatargumente für ihn nicht interessant. Darum ändert sich auch nichs in der Formulierung der Vor– und Nachbedingungen eines Operators. Anders liegt der Fall beim Planausführer. Dieser muß sehr wohl darauf achten, ob ein Operator in einem Prädikatargument eine oder mehrere Instanzen abgelegt hat!

Neue Instanzen für die A–Box — und damit auch Mengen von Instanzen — können nur innerhalb von Operatoren erzeugt werden. Wenn der Operator nur eine Instanz für ein Prädikatargument erzeugt, dann wird mit der Methode *PredArgument*.`setInstance` diese Instanz gesetzt. Mengen von Instanzen können durch wiederholtes Aufrufen von *PredArgument*.`addInstance` hinzugefügt werden.

Kapitel 12

Datenbeschaffung

Wer kein Messer hat, kann kein Brot schneiden
aus Spanien

Die Klassen im Paket `dataprovider` stellen für einen Agenten die Schnittstellen zum Netz dar. Grundlegende Funktionen sind fertig imlpementiert. So ist z. B. für Agenten, die auf HTML–Dokumenten arbeiten, die via HTTP geladen werden, keine weitere Programmierarbeit nötig.[1] Es existieren auch Klassen, deren Instanzen die Abfrage von Suchmaschinen im WWW ermöglichen.

Ferner gibt es die Möglichkeit, Dateien aus dem lokalen Dateisystem zu laden, um z. B. die Klassifikatoren mit lokalen Daten trainieren zu können, ohne dafür einen lokalen HTTP–Server einrichten zu müssen.

12.1 Wichtige Klassen im Überblick

In den folgenden Abschnitten werden verschiedene, wichtige Klassen der Datenbeschaffung vorgestellt, und es wird beschrieben, wie diese zu benutzen sind.

`NetService` Die Net–Services stellen die Möglichkeit bereit, den Inhalt einer URL herunterzuladen. Zur Zeit werden die Protokolle HTTP und FTP, sowie der Zugriff auf lokale Dateien unterstützt.

`NetEntity` Diese Klasse stellt die Superklasse, den Grundtyp aller Objekte dar, die etwas aus dem Netz repräsentieren, wie z. B. Links, HTML–Seiten u. ä..

`NetResult` In Objekten dieses Typs werden die von einem *`NetService`* heruntergeladenen Dateien bzw. Dokumente abgelegt. Über verschiedene Methoden kann auf Teilaspekte eines Dokumentes zugegriffen werden, wie z. B. auf Titel und Links einer HTML–Seite.

`SearchService` Mit den Search–Services können Suchmaschinen auf einfache Art und Weise abgefragt werden.

Wenn die vorhandenen Klassen bzw. deren Funktionalität nicht ausreichen, so gibt es in einem gesonderten Abschnitt Tips, wie und wo Erweiterungen möglich sind, und was dabei zu beachten ist.

[1] bezogen auf die Datenbeschaffung natürlich

12.2 `dataprovider.net.NetService` — Realisierung von Transferprotokollen

Die Net–Services stellen die Funktionalität bereit, Daten aus dem Netz durch Angabe ihrer URL herunterzuladen. In der aktuellen Version von BotIshelly werden die drei Protokolle HTTP, FTP und File[1] unterstützt, die mit Hilfe der drei Klassen `HTTPNetService`, `FTPNetService` und `FileNetService`[2] benutzt werden können. Alle Protokolle sind als Subklassen der abstrakten Klasse `NetService` realisiert. Dies sollte auch für weitere, neu zu programmierende Protokollservices so gehandhabt werden.

Um zu einer URL, deren Protokoll bekannt ist, den passenden Service zu erhalten, kann direkt ein Objekt der entsprechenden Klasse instanziieren werden, z. B.:

```
HTTPNetService  service = new HTTPNetService();
```

Wenn das Protokoll nicht bekannt ist, bzw. noch nicht feststeht, kann die statische Methode `NetService.getNetService` benutzt werden. Diese Klassenmethode liefert den zur übergebenen URL passenden Service zurück oder wirft eine Exception, falls deren Protokoll nicht unterstützt wird. Im Programmcode könnte dies wie folgt aussehen:

```
NetService  service;
String url;

try {
  // Funktioniert
  url = "http://www.nathan.de/";
  service = NetService.getNetService( url );

  // Funktioniert auch
  url = "file:/a/b/c.html";
  service = NetService.getNetService( url );

  // Funktioniert nicht
  url = "news:fbi-news";
  service = NetService.getNetService( url );
}
catch ( NetServiceNoSuchServiceException e ) {
  System.out.println( "Unbekanntes Protokoll in URL " + url );
}
```

Nachdem ein entsprechender Service instanziiert worden ist, kann dessen Methode `getNetResult` aufgerufen werden, um den Inhalt einer übergebenen URL herunterzuladen. Vorher kann via `setTimeout` ein Timeout (in Sekunden) gesetzen werden, der festlegt, nach welcher Zeit eine Verbindung, die nicht antwortet, abgebrochen wird.[3] Sollte die gewünschte Information passwortgeschützt sein, müssen über die Methoden `setUser` und `setPassword` der Benutzername, respektive das Passwort gesetzt werden.

Das folgende Beispiel zeigt, wie eine Datei, die auf einem FTP–Server liegt, heruntergeladen werden kann. In diesem Beispiel würde allerdings eine Exception geworfen werden, da gzip'te Postscriptdateien von BotIshelly nicht unterstützt werden.

[1] File ist natürlich kein echtes Protokoll. Der Einfachheit halber wird hier jedoch keine (weitere) Unterscheidung vorgenommen.

[2] Der widersprüchliche Name kommt daher, daß lokale Dateien in der Modellierung noch nicht vorgesehen waren.

[3] Bei einem *`FileNetService`* wird der Timeout ignoriert.

```
FTPNetService service = new FTPNetService();
NetResult result;

try {
  service.setUser ("anonymous");
  service.setPassword ("user@somewhere.com");
  service.setTimeout (30);
  result = service.getNetResult("ftp://ftp.de/pub/memo47.ps.gz");
}
catch (Exception e) {
  // hat wohl etwas nicht funktioniert...
}
```

12.3 `dataprovider.net.NetEntity` — Der kleinste Nenner im Datenaustausch

Die abstrakte Klasse `NetEntity` stellt die Superklasse für Objekte dar, die etwas aus dem Netz modellieren, und die zum Informations- und Datenaustausch zwischen den unterschiedlichen Funktionseinheiten eines Agenten vorgesehen sind. Beispiele sind Links, HTML–Seiten oder auch Bild- und Binärdateien. Unter anderem kommunizieren die Datenbeschaffung, die Klassifikatoren und die A–Box(-en) mit Hilfe dieser Klasse.

Eine ***NetEntity*** verkörpert also ein Objekt, das dem Netz "entsprungen" ist, und dessen "Ursprugsort", nämlich die URL, die Methode `getURL` zurückliefert. Damit eine ***NetEntity*** z. B. als Ergebnis einer Suche auch ausgegeben bzw. angezeigt werden kann, existiert die abstrakte Methode `getShortDescription`, die eine für diesen Zweck sinnvolle Beschreibung liefern sollte.

Bei einer zielgerichteten Suche ist es wichtig, herausfinden zu können, ob eine ***NetEntity*** überhaupt etwas mit dem Suchziel zu tun hat, bzw. aus welchen Gründen auch immer für wichtig oder unwichtig erachtet werden sollte. Es muß also die Möglichkeit der Klassifikation bestehen. Aus diesem Grund sind die Methoden `addClassification`, `getClassification` und `getAllClassificators` implementiert. Außerdem ist die abstrakte Methode `getText` vorhanden, die eine textuelle Beschreibung der ***NetEntity*** liefern sollte, die den Klassifikatoren eine möglichst gute Präzision ihrer Entscheidungen ermöglicht. Nachdem die Klassifikatoren ihre Ergebnisse über `addClassification` eingetragen haben, kann auf diese schließlich im weiteren Programmablauf via `getClassification` und `getAllClassificators` zugegriffen werden.

12.4 `dataprovider.net.NetResult` — Grundtyp aller Repräsentationen von Dateien bzw. Dokumenten

Die abstrakte Klasse `NetResult` ist die Superklasse aller möglichen, von BotIShellYunterstützten Dokument- bzw. Dateitypen. Sie ist folgerichtig eine Subklasse von `NetEntity`. Das Ergebnis der Methode `getNetResult` eines Net–Services ist — falls keine Exception geworfen wurde — eine Instanz einer Subklasse von `NetResult`, z. B. ein ***HTMLTextNetResult*** bei HTML–Dokumenten. Zur Verdeutlichung die Hierarchie der Klassen[1]:

Sollen neue Dokumenttypen unterstützt werden, so muß eine entsprechende, eventuell neu zu erstellende Subklasse von `NetResult` benutzt werden. In der Methode `getNetResult` eines jeden

[1]Ein dünner Rahmen zeigt an, daß es sich um eine abstrakte Klasse handel.

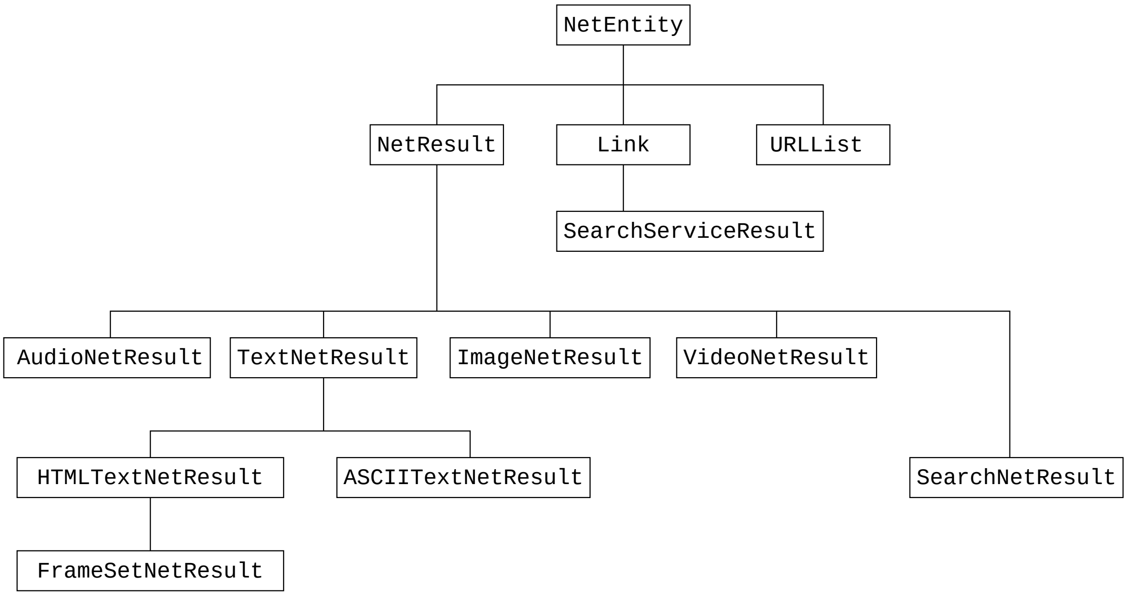


Abbildung 12.1: Hierarchie der Subklassen von `NetEntity`

Net–Services, der diesen Typ unterstützen soll, muß dies vorgesehen werden, d. h. der Programmcode dieser Methode muß erweitert werden. Wie dies genau zu geschehen hat, wird noch erleutert werden.

Die Klasse `NetResult` definiert verschiedene Methoden, um auf Einzelaspekte der dargestellten Datei zugreifen zu können:

`getDownloadDate` Liefert das Datum (als *java.util.Date*), an dem die Übertragung abgeschlossen wurde.

`getSubmittedDate` Liefert das Datum der letzten Änderung der Datei, soweit das benutzte Protokoll eine solche Information vorsieht, sonst das gleiche Datum wie `getDownloadDate`.

`getRaw` Mit dieser Methode kann der “rohe”, ungeänderten Inhalt der heruntergeladenen Datei als *String* angefordert werden. Sie wird selten gebraucht werden, da z. B. zum Ermitteln der Links einer HTML–Seite bereits eine Methode existiert.

`getShortDescription` Diese Methode liefert eine Kurzbeschreibung des Inhaltes eines *NetResult*. Z. B. bei HTML–Dateien die ersten 300 Zeichen des Textes der Seite.

`getText` Von `NetEntity` geerbte Methode, die die Information liefert, nach der klassifiziert wird.

`getURL` Ebenfalls von `NetEntity` geerbte Methode. Liefert die URL, mit der der *NetResult* geladen wurde.

12.4.1 Die Klasse `HTMLTextNetResult`

Die Klasse `HTMLTextNetResult` soll noch besondere Erwähnung finden, da sie zum einen besondere Funktionen anbietet und zum anderen HTML wohl das meistbenutzte Dokumentenformat im WWW ist. So kann die Methode `getLinks` dazu verwendet werden, alle Links einer HTML–Seite zu ermitteln:

```
HTMLLinkEnumeration enumeration;
HTTPNetService service = new HTTPNetService;
HTMLTextNetResult result;
Link link;

try {
  // Datei runterladen
  result = (HTMLTextNetResult) service.getNetResult("http://www.linux.com");

  // Enumeration erzeugen
  enumeration = result.getLinks();

  // Die einzelnen Ergebnisse ausgeben
  while (enumeration.hasMoreElements()) {
    link = (Link) enumeration.nextElement();
    System.out.println ("URL=" + link.getURL() );
    System.out.println ("Text=" + link.getLinkText() );
  }

  // Zum Schluß noch den Titel der Seite anzeigen
  System.out.println ("Titel der Seite: " + result.getTitle());
}
```

Die Methode `getLinks` liefert eine ***Enumeration*** für alle Links einer HTML–Seite. Die einzelnen Objekte in dieser Aufzählung sind vom Typ `Link`. Auf die URL und den Link–Text eines ***Link*** kann über die Methoden `getURL` und `getLinkText` zugegriffen werden.

Bei HTML–Seiten kann es auch sinnvoll sein, den Titel einer Seite zu ermitteln. Dieses ist über die Methode `getTitle` möglich.

Im WWW ist es inzwischen üblich, in HTML–Seiten sogenannte "Frames" zu benutzen, was zur simultanen Darstellung von mehreren HTML–Dateien in einem WWW–Browser–Fenster führt. Die Angabe einer URL kann bei einer Suche per Hand also zur Darstellung weiterer Seiten führen, die zur Informationsgewinnung dienen können, ohne daß deren URL je sichtbar wird. Diesem Sachverhalt trägt die Klasse `FrameSetNetResult` Rechnung, welche eine Subklasse von `HTMLTextNetResult` ist. Bei Instanzen dieser Klasse, wurden alle HTML–Dateien, die ein WWW–Browser anzeigen würde, geladen. Die Methoden `getText` und `getLinks` verhalten sich entsprechend.

12.5 dataprovider.search.SearchService — Abfragen von Suchmaschinen leicht gemacht

Über die Search–Services wird eine Schnittstelle zu Suchmaschinen bereitgestellt. Alle verfügbaren Suchmaschinenschnittstellen sind dabei Subklassen der abstrakten Klasse `SearchService`. In der aktuellen Version von BotIshelly sind die Suchmaschinen AltaVista und Nathan abfragbar (www.altavista.com, www.nathan.de).

Eine exemplarische Abfrage der Suchmaschine AltaVista zeigt der folgende Beispielcode:

```
SearchResultEnumeration enumeration;
AltavistaSearchService service = new AltavistaSearchService();
SearchServiceResult result;
SearchNetResult results;
```

```
try {
  results = service.doQuery ("java compiler");
  enumeration = results.getResults();

  while (enumeration.hasMoreElements()) {
    result = (SearchServiceResult) enumeration.nextElement();

    System.out.println("URL=" + result.getURL());
    System.out.println("Desc=" + result.getShortDescription());
    System.out.println("Title=" + result.getText());
  }
}
catch (Exception e) {
  // wohl ein Fehler aufgetreten...
}
```

Zuerst muß eine Instanz der entsprechenden `SearchService`–Subklasse erzeugt werden, in diesem Fall also von `AltavistaSearchService`. Danach wird deren Methode `doQuery` aufgerufen, der die einzelnen Suchworte durch Freizeichen getrennt übergeben werden. Die Search–Services führen dann auf diesen Suchworten eine UND–verknüpfte Anfrage durch.

Das Ergebnis der Methode `doQuery` ist ein *`SearchNetResult`*. Um die einzelnen Suchergebnisse (i. d. R. zehn Stück) zu erhalten, muß zunächst über die Methode `getResults` eine *`SearchResultEnumeration`* erzeugt werden. Wie bei einer *`java.util.Enumeration`* vorgesehen, kann nun mit den Methoden `hasMoreElements` und `nextElement` auf die einzelnen Objekte der Aufzählung, in diesem Fall also die einzelnen Suchergebnisse, zugegriffen werden. `nextElement` liefert hier Instanzen von `SearchServiceResult`.

Bei einem *`SearchServiceResult`* kann schließlich über die Methoden `getURL`, `getText` und `getShortDescription` auf die URL, den Linktext und die Kurzbeschreibung des Suchergebnisses zugegriffen werden.

12.6 Modifikationen & Erweiterungen

12.6.1 Die Klasse `HTMLConvert`

Mit den Klassenmethoden `convertEntities` und `convertToEntities` kommen Sie in der Regel nicht in Kontakt, sie seien hier aber erwähnt, da sie bei Erweiterungen der Shell vielleicht nützlich sein können.

Entities[1] werden in Markup–Sprachen verwendet, um Zeichen zu kodieren. Insbesondere in HTML finden diese Verwendung: Da es z. B. nicht erlaubt ist, das Zeichen ´`&`´ zu verwenden, muß stattdessen ´`&`´ benutzt werden.

Sollten in einem *`String`* solche Entities vorkommen, können diese mit der Klassenmethode `convertEntities` aufgelöst werden. Sollte der umgekehrte Weg gewünscht werden, um z. B. einen *`String`* mit Sonderzeichen korrekt in einem Browser auszugeben, realisiert die Klassenmethode `convertToEntities` eben dieses.

12.6.2 `NetService`: Unterstützung eines neuen Protokolls

Sollten neue Protokollservices hinzugefügt werden, so muß sich dies auch in der Klassenmethode `NetService.getNetService` widerspiegeln, damit deren Funktionalität erhalten bleibt, d. h. der

[1] Hier nicht in Zusammenhang mit der Klasse `NetEntity`

Programmcode dieser Methode muß erweitert werden. Die Stelle, an der dies zu geschehen hat ist deutlich durch Kommentare gekennzeichnet.

Um ein neues Protokoll zu unterstützen, muß zunächst die entsprechende Klasse als von `NetService` erbende Klasse realisiert werden. Danach muß in dieser neuen Klasse die Methode

```
public NetResult getNetResult (String url)
  throws NetServiceException, MalformedURLException
```

implementiert werden, der eine URL als *`String`* übergeben wird und die ein *`NetResult`* zurückliefert.

Als Subklasse von `NetService` erbt die neue Klasse auch die `String`–Variablen *`user`* und *`password`* (für passwortgeschützte Dokumente; siehe die Methoden `setUser` und `setPassword`) sowie die `int`–Variable *`timeout`*, die den Timeout der Datenverbindung in Sekunden angibt. Beim Implementieren der Methoden sollten diese drei Variablen natürlich beachtet und ihrer Bestimmung entsprechend genutzt werden.

Zuerst sollte geprüft werden, ob die URL überhaupt korrekt ist. Im Fehlerfall sollte eine `MalformedURLException` ausgelöst werden, was z. B. mit der Zeile

```
new java.net.URL (url);
```

möglich ist, da der `URL`–Konstruktor genau diese Exception werfen kann/darf.[2]

Danach wird die Datei heruntergeladen und in einem *`String`* abgespeichert. Je nach Protokoll muß dann noch entschieden werden, welchen Typ der heruntergeladene Inhalt hat. Möglichkeiten wären der Content–Type bei HTTP–Verbindungen, die Dateiendung bei FTP–Verbindungen oder die Betrachtung der ersten 16 Bytes (siehe `file`–Kommando bei UNIX-Systemen). Außerdem müssen zwei Zeitpunkte (als *`java.util.Date`*) bekannt sein: das Datum der letzten Änderungen (für `submittedDate`) und das Datum, an dem die Übertragung abgeschlossen wurde (für `downloadDate`). Mit diesen Daten wird dann die Klassenmethode `composeNetResult` aufgerufen. Als erster Parameter muß dieser Methode ein String übergeben werden, der den Typ der geladenen Datei beschreibt. Bisher werden einige MIME–Typen und einige Dateinamenendungen erkannt. Welche Typen erkannt werden, wird von der Abbildung bestimmt, die durch die statische *`TreeMap`* `typeMap` in der Klasse `NetService` induziert wird. Das Ergbenis dieser Methode kann schließlich als Ergebnis der Methode `getNetResult` zurückgeben werden. Die Exceptions die `composeNetResult` werfen kann, können — und sollten — dabei einfach weiterpropagiert werden.

Als gutes Beispiel für die Implementierung der Methode `getNetResult` empfiehlt sich das Studium der Datei `HTTPNetService.java`, in der z. B. das Last–Modified–Attribut als `submittedDate` genutzt wird.

12.6.3 `NetResult`: Hinzufügen eines neuen Dokumenttyps

Um einen neuen Dokumenttyp hinzuzufügen, muß zuerst eine neue Klasse in die `NetEntity`– bzw. `NetResult`–Hierarchie eingesetzt werden. Danach sind die geerbten, abstrakten Methoden (z. B. `getText`) zu implementieren. U. U. kann es auch sinnvoll sein, geerbte Methoden zu überschreiben, oder auch zu überladen.

Damit die vorhandenen Net–Services Ihren neuen Dokumenttyp benutzen, d. h. Instanzen Ihrer Klasse erzeugen, können, müssen in der Klasse `NetService` folgende Erweiterungen getätigt werden:

[2] D. h. keine `try`/`catch`–Klammerung um diesen Aufruf.

1. Die MIME–Typen und die Dateinamenendungen, die dem neuen Dateityp entsprechen, müssen in die `typeMap` eingetragen werden, indem sie auf einen ***Integer*** abgebildet werden, der einen noch nicht benutzten Wert repräsentiert. Dies passiert in einem statischen Codeblock von `NetService`, z. B.:

   ```
   // MIME-type für HTTPNetService
   typeMap.put("text/vnd.wap.wml", new Integer(333) );
   // file-extension für FTPNetService
   typeMap.put("wml",              new Integer(333) );
   ```

2. In der Methode `composeNetResult` muß obiger Integerwert einen "Switch–Case" einleiten, in dessen Anweisungsteil dann eine Instanz Ihrer neuen Klasse erzeugt und zurückgeben werden muß, z. B.:

   ```
   case 333:
     return new WMLNetResult(url, rawContent, download, lastModified);
   ```

12.6.4 `SearchService`: Hinzufügen einer neuen Suchmaschinenschnittstelle

Um eine neue Suchmaschine hinzufügen zu können, muß diese zwei Bedingungen erfüllen:

1. Sie muß über eine GET–Anfrage abfragbar sein.
2. Zu jedem Suchergebnis muß sie eine URL, einen Titel und eine Kurzbeschreibung liefern.

Nehmen wir eine fiktive Suchmaschine, deren Suchergebnisseiten so aussehen:

```
<html>
...

<dl>
<dd><a href="http://www.test.de/">Erste Seite</a>
<dt>Kurbeschreibung zur ersten Seite<br>

<dl>
<dd><a href="http://www.test2.de/">Zweite Seite</a>
<dt>Kurbeschreibung zur zweiten Seite<br>
...

</html>
```

Wie bei allen (bekannteren) Suchmaschinen üblich, sind die Suchergebnisse nach einem Schema aufgebaut (bei Suchmaschinen, für die das nicht gilt, muß ein neuer Parser geschrieben werden, der die abstrakte Klasse `ParserService` implementiert). Bei obigem Beispiel gilt, daß die URL immer zwischen `<dd><a href="` und `"` steht. Für die drei Bestandteile URL, Titel und Kurzbeschreibung müssen Sie nun nach solchen umschließenden Zeichenketten suchen.

Ihre neue Klasse muß von der abstrakte Klasse `SearchService` erben:

```
public class NewSearchService extends SearchService
{
    public SearchNetResult doQuery(String queryString)
```

```
                 throws SearchServiceNoResultsException,
                        SearchServiceNoConnectException
    {
        SimpleParserService  parser = new SimpleParserService();
        SearchServiceResult  result;
        ParserStore          store;
        NetResult            netresult;
        String               url;

        //...
    }
}
```

Danach definieren Sie einen neuen Parser, dem Sie reguläre Ausdrücke für die umschließenden Zeichenketten (Format: RegulärerAusdruckLinks|Variable|Regulärer Ausdruck rechts) übergeben können. In unserem Beispiel:

```
try {
  parser.setParser( new String[]  { "<dd><a href=\"|URL|\"",
                                    ">|TITLE|</a>",
                                    "<dt>|DESC|<br>" } );
}
catch( ParserServiceMalformedParserException e ) {
   // ...
}
```

Als nächstes müssen Sie die URL erzeugen, mit der die Suchmaschine via GET–Request abgefragt wird. Achten Sie dabei vor allem darauf, daß Sie gemäß CGI–Konvention kodieren (ein Leerzeichen wird zu `+`, etc.). Wie Sie das mit Hilfe der `Transformer`–Klasse bewerkstelligen können, lesen Sie einfach im Quellcode der Klasse `AltavistaSearchService` nach.

```
url = "http://www.somewhere.com/cgi-bin/query.cgi?QUERY=" + queryString;
```

Dann können Sie den Inhalt dieser URL herunterladen und darauf den Parser anwenden:

```
/* Erstmal nen Service ermitteln - sollte HTTPNetService sein */
try { netservice = NetService.getNetService( url ); }
catch( Exception e ) { throw new SearchServiceNoConnectException(); }

/* Runterladen - zumindest versuchen */
try { netresult = netservice.getNetResult( url ); }
catch( Exception e ) { throw new SearchServiceNoConnectException(); }

/* Inhalt der HTML-Seite */
try{ content = netresult.getRaw(); }
catch( NetResultReloadFailedException e ) {}

/* Parsen */
try { store = parser.parseString( content ); }
catch( ParserServiceUnknownParserException e ) {}

/* Ein SearchNetResult erzeugen */
result = SearchNetResult( store, url, content,
                          netresult.getDownloadDate(),
                          netresult.getSubmittedDate() ) );
```

Das *SearchNetResult* enthält jetzt die Suchergebnisse. Wie bereits beschrieben, kann jetzt mit der Methode `getResults` eine *Enumeration* über die einzelnen Suchergebnisse erhalten werden. Im übrigen muß an dieser Stelle noch geprüft werden, ob überhaupt Suchergebnisse gefunden wurden (ein *SearchNetResult* kann auch mit null Suchergebnissen erzeugt werden). Falls keine Ergebnisse vorliegen, muß die Methode `doQuery` die Exception `SearchServiceNoResultsException` auslösen.

Kapitel 13

Datenanalyse

> Die Natur hat uns zwar viele Kenntnisse versagt, sie läßt uns über so manches in einer unvermeidlichen Unwissenheit, aber den Irrtum verursacht sie doch nicht. Zu diesem verleitet uns unser eigener Hang, zu urteilen und zu entscheiden auch da, wo wir wegen unserer Begrenztheit zu urteilen und zu entscheiden nicht vermögend sind.
>
> *Immanuel Kant*
> Logik 8

13.1 Aufgabe der Datenanalyse

Die Datenanalyse und damit die Klassifikatoren beschäftigen sich in der ersten Linie mit der Analyse bzw. der Klassifizierung von Texten, die von der Datenbeschaffung übergeben wurden.

Für die Klassifizierung werden Klassifizierer bereitgestellt auf die später noch im einzelnen eingegangen wird. Damit Sie auch noch eigene Ideen einbinden und verwirklichen können, besteht zusätzlich die Möglichkeit eigene oder neu entwickelte Klassifikatoren in das System zu integrieren. Wie dies genau geschieht wird Ihnen noch näher erläutert. Zunächst wird die Funktionalität der einzelnen Komponenten der Datenanalyse beschrieben.

13.2 Arbeitsweise

Die Datenanalyse stellt verschiedene Klassifikatoren zur Verfügung, die sich im wesentlichen mit der Analyse von Texten befassen. Die bereits implementierten Klassifikatoren sind der Bayes–Klassifikator (Klassifizierung von Text mit wahrscheinlichkeitstheoretischen Ansätzen), der k–NN-Klassifikator (Entwickelt in der Statistik, benutzt die Definition eines Abstandsmasses) und der Rocchio–Klassifikator (Vektorraum–Retrieval-Modell, hierbei werden Dokumente als Wortvektoren dargestellt).

Für speziellere Aufgaben gibt es zusätzlich noch den `NetEntityClassifier`, den `LinkPageClassifier` und den `URLClassifier`. Die zuletzt genannten Klassifikatoren zeichnen sich dadurch aus, daß sowohl das Training, wie auch die Klassifikation auf eine eindeutige Aufgabe hin orientiert sind.

Alle Klassifikatoren können in ihrer Arbeitsweise trainiert werden, indem sie bereits bewertete Trainingsmengen (Dokumente) klassifizieren und einordnen. Wichtig ist hierbei, daß sowohl positive wie auch negative Beispiele eingesetzt werden, um den größtmöglichen Lernerfolg zu erzielen. Die Trainingsmengen können von Ihnen zusammengestellt und bewertet werden. Je genauer Sie an diese Aufgabe herangehen, desto besser werden die Ergebnisse.

Während der Klassifizierung wird neben dem Feature–Wörterbuch (Grundlage für den `Documentvector`) auch noch das Negativ–Wörterbuch(Stop–Wort–Liste) und eventuell das Synonym–Wörterbuch eingesetzt.

Die Wörter des Negativ–Wörterbuchs werden bei der Umwandlung eines Textes in einen Vektor nicht berücksichtigt, während die Wörter aus dem Synonym–Wörterbuch dabei helfen können die Suche mit verwandten Wörtern erweitert durchzuführen. Weitere Informationen zum Thema Vektorisierung finden sie auf der Seite 198.

Auch der Vektorisierer, der für die Erstellung eines Vektors aus einem Text verantwortlich ist, steht den Klassifikatoren als wichtiges Hilfsmittel zur Verfügung. Viele Klassifikatoren nutzen die Darstellung von Texten als Vektoren.

Nachdem die Systemkontrolle die `ClassifierDatabase` geladen hat, können die Klassifikatoren eingesetzt werden. Der Planer hat hierbei die Möglichkeit für die Durchführung seines Plans einen konkreten Klassifikator aus der Klassifikatoren–Datenbank (`ClassifierDatabase`) zu laden (Bsp.: `get(Klassifikatorname)`). Nachdem die Auswahl getroffen ist kann die Klassifizierung beginnen.

Soviel zu der allgemeinen Arbeitsweise der Klassifikatoren, in den nun folgenden Kapiteln wird Ihnen nun genauer erklärt werden, wie das Training eines Klassifikators abläuft, die Klassifizierung der einzelnen Dokumente funktioniert und welche Hilfsmittel Ihnen bereits zur Verfügung stehen.

Für die Implementierung eines neuen Klassifikators müssen die ab Seite 202 aufgeführten Punkte berücksichtigt werden.

13.3 Die Konstruktion von Klassifikatoren

Zunächst betrachten wir die Klasse, die allen hier implementierten Klassifikatoren, zugrundeliegt. Die Klasse `Classifier` stellt das Grundgerüst für die Klassifikatoren dar. Jeder `Classifier` hat die Möglichkeit auf den `Vectorizer` oder das `Dictionary` zurückzugreifen. Der `Vectorizer` ist ein wesentliches Hilfsmittel für die Klassifizierer. Der Vektorisierer transformiert die textuelle Darstellung eines Dokuments in einen vektoriellen Wert, den `Documentvector`. Dabei hat der Vektorisierer zwei unterschiedliche Aufgaben, die sich bereits anhand des Methodenaufrufs erkennen lassen.

Befindet sich der Klassifikator im Trainingsmodus, so erfolgt der Aufruf durch `vectorize(newDocument, true)`. Der boolesche Parameter läßt hierbei erkennen, daß sich der Vektorisierer zur Zeit im Trainingsmodus befindet. Was bedeutet nun Trainingsmodus? In der Trainingsphase ist der Klassifikator bemüht so viele Merkmale wie möglich aus den Beispieldatenmengen herauszufiltern. Merkmale bedeutet in diesem Kontext, die unterschiedlichen Wörter aus den Dokumenten zu sammeln und in einem Wörterbuch dem sogenannten Feature–Wörterbuch einzubinden.

Da sich der `Documentvector` an diesem speziellen Wörterbuch orientiert sollte eine Veränderung der Wörterbücher während der eigentlichen Klassifikationsphase unterbunden werden, dies wird durch den Methodenaufruf `vectorize(newDocument, false)` erreicht.

Um die Funktionsweise und den Aufbau des `Documentvectors` zu verdeutlichen, stellen wir Ihnen die grundlegenden Aspekte vor. Wie bereits erwähnt untersteht die Klasse `Documentvector` in erster Linie dem Vektorisierer. In diesem Zusammenhang werden alle wichtigen Methoden zur

Verfügung gestellt, die es erlauben ein Dokument in einen Vektor zu konvertieren und auch Berechnungen zwischen zwei Vektoren durchzuführen. Als Beispiele sind hier eine kleine Auswahl vorhandener Methoden aufgeführt: `setElement`, `addElement`, `dimension`, `dotProduct`, `euklidLength`.

Aus Platzgründen wird der `Documentvector` spärlich repräsentiert, d.h. Positionen die einen Wert von 0.0 beinhalten werden nicht explizit gespeichert. Dies bedeutet auf der anderen Seite, daß alle unbesetzten Stellen des Vektors automatisch den Wert 0 enthalten.

Die Dokumentvektoren sind auch für die Klassifizierer von größter Bedeutung, da fast alle Berechnungen der Abstände oder der Ähnlichkeit über Vektoren ermittel werden. Wenn Sie einen neuen Klassifizierer implementieren möchten, so wäre es sinvoll, wenn sie sich einen Überblick über die Methoden der Klassen `Vectorizer` und `Documentvector` verschaffen. Vielleicht kann hierdurch ein zusätzlicher Programmieraufwand verhindert werden, indem bereits vorhandene Methoden eingesetzt werden.

13.3.1 Aufbau eines Dokumentvektors

Der Dokumentvektor ist das Ergebnis der Vektorisierung eines Textes (String). Bei den hier vorgestellten Klassifikatoren wird der Dokumentvektor wie folgt konstruiert:
Der übergebene String (Dokument) wird in seine Wörter zerlegt. Dabei wird ein `StringTokenizer` eingesetzt, der dafür sorgt, daß aus dem Text, sämtliche Notation herausgefilter wird und nur noch die Wörter selbst zurückbleiben.

Für jedes einzelne Wort wird zunächst überprüft, ob es im Negativ–Wörterbuch (Stopwort–Wort–Liste) vorkommt. Ist dies der Fall, so wird das aktuelle Wort bei der Vektorisierung nicht berücksichtigt. Im anderen Fall, wird das Wort ins Feature–Wörterbuch eingetragen. Anhand der Position, an der das Wort ins Wörterbuch eingetragen wurde, erfolgt der Eintrag in den Dokumentvektor. Kommt ein Wort häufiger im Text vor, so wird der Wert an der Stelle im Dokumentvektor um eins erhöht, ein erneuter Eintrag ins Feature–Wörterbuch erfolgt nicht.

Wenn der Klassifikator sein Training durchführt, durchläuft ein Text nach dem anderen den Vektorisierer. Dabei wächst das Feature–Wörterbuch und somit auch die Dimension des Dokumentvektors kontinuierlich an. An dieser Stelle sei nochmals darauf hingewiesen, daß die Anpassung der Wörterbücher nur im Traingsmodus der Klassifikatoren möglich ist.

Um den Aufbau eines Dokumentvektors zu veranschaulichen folgt ein Beispiel:

Übergebener Text: „Die Klasse Dokumentvektor untersteht in erster Linie dem Vektorisierer. In diesem Zusammenhang werden alle wichtigen Methoden zur Verfügung gestellt, die es erlauben ein Dokument in einen `Documentvector` zu konvertieren.“

Bei der Zerlegung des Textes werden nur die Wörter berücksichtigt, die Notation in dem Dokument (String) wird vernachlässigt. Für die Konstruktion des Dokumentvektors stehen drei verschiedene Wörterbücher zur Verfügung.

- **Feature–Wörterbuch**: Grundlage für den `Documentvector`, da er sich an den Positionen der Wörter in diesem Wörterbuch orientiert.
- **Negativ–Wörterbuch**: Dieses Wörterbuch wird auch häufig Stop–Wort–Liste genannt. Alle Wörter, die in diesem Wörterbuch aufgeführt sind, werden bei der Klassifizierung nicht berücksichtigt. (mögliche Beispiele: und, der, ein usw.)
- **Synonym–Wörterbuch**: Hier besteht die Möglichkeit bei der Klassifizierung auch verwandte Wörter mit einzubeziehen, dabei erhofft man sich eine erhöhte Treffsicherheit bei der Klassifizierung.

Zustand der Wörterbücher vor der Vektorisierung:

Wörterbuch	Inhalt
Feature–Wörterbuch	—
Negativ–Wörterbuch	ein, einen, die, dem, diesem, es, untersteht, gestellt, erlauben, in, zur, werden, erster, alle, wichtigen

Zustand der Wörterbücher nach der Vektorisierung:

Wörterbuch	Inhalt
Feature–Wörterbuch	Klasse, Dokumentvektor, Linie, Vektorisierer, Zusammenhang, Methoden, Verfügung, Dokument, konvertieren
Negativ–Wörterbuch	ein, einen, die, dem, diesem, es, untersteht, gestellt, erlauben, in, zur, werden, erster, alle, wichtigen

Resultierender Dokumentvektor:

Position	Wort	Anzahl
1	Klasse	1
2	Dokumentvektor	2
3	Linie	1
4	Vektorisierer	1
5	Zusammenhang	1
6	Methoden	1
7	Verfügung	1
8	Dokument	1
9	konvertieren	1

(1, 2, 1, 1, 1, 1, 1, 1, 1) ist der Dokumentvektor des übergebenen Dokuments.

Nachdem nun die Klasse `Documentvector` eingehend beschrieben wurde, können wir uns nun dem Teil widmen, der diesen Vektor einsetzt, der Klasse `Vectorizer`. Bei der Vektorisierung können drei verschiedene Wörterbücher eingesetzt werden (Feature–, Negativ– und Synonym–Wörterbuch), deren Grundgerüst aus der Klasse `Dictionary` stammt, und die alle notwendigen Methoden beinhaltet, die beim Erzeugen und Bearbeiten von Wörterbüchern notwendig sind. Die Wörterbücher sind transient definiert um den Speicherplatz effektiver nutzen zu können. Im einzelnen können die Bücher über den gesetzten Namen aus der `DictionaryDatabase` aufgerufen werden.

Um den Trainingsmodus eines Klassifikators von der eigentlichen Klassifikation zu trennen, hilft, wie oben im Text bereits angedeutet, ein boolescher Parameter in der `vectorize`-Methode aus der Klasse `Vectorizer`.

Die Methode `vectorize` erhält als zweiten Parameter neben dem zu klassifizierenden Text auch noch einen booleschen Wert. Befindet sich der Klassifikator im Trainingsmodus so wird `vectorize` mit dem Zusatz `true` aufgerufen. Somit wird die Anpassung der Wörterbücher ermöglicht. Dies bedeutet, daß mit jedem neuen Dokument aus der Trainingsmenge auch eine Veränderung der Wörterbücher (insbesondere des Feature-Wörterbuchs) einhergeht.

Da sich aber die Dimension des Dokumentvektors an der Anzahl der Wörter im Feature-Wörterbuch orientiert, muß eine weitere Anpassung der Wörterbücher während der eigentlichen Klassifizierung unterbunden werden. Dies wird erreicht indem die `vectorize`-Methode nun mit dem zu klassifizierenden Dokument und dem Zusatz `false` aufgerufen wird. In diesem Fall ist es nicht mehr möglich, daß sich die Wörterbücher automatisch anpassen und somit die Dimension des Vektors verfälschen könnten.

Bevor wir zur Konstruktion neuer Klassifikatoren kommen, werden Ihnen die bereits implementierten Klassifikatoren kurz vorgestellt.

13.3.2 Rocchio-Klassifkator

Der Rocchio–Algorithmus wurde als Methode zum Relevance Feedback im Rahmen des Information–Retrieval–Systems SMART entwickelt. Der Algorithmus basiert auf dem Vektorraum–Retrieval–Modell. In diesem Modell werden Dokumente ebenso wie Anfragen als Wortvektoren repräsentiert. Weiterhin wird ein Abstandsmaß definiert, welches die (semantische) Ähnlichkeit dieser Vektoren messen soll. Wird eine Anfrage an das Retrieval–System gestellt, werden die Dokumente aus der Datenbank anhand ihrer so definierten Ähnlichkeit zur Anfrage sortiert. Es wird davon ausgegangen, daß Dokumente um so wahrscheinlicher relevant in Bezug auf die Anfrage sind, je ähnlicher sie zu ihr sind.

Ablauf des Rocchio–Algorithmus:

- Dokumente werden vektorisiert (Vektoren enthalten die TF-Werte der Worte).
- Nachdem alle Dokumente vektorisiert wurden beginnt die Gewichtung über einen zweiten Faktor: Inverse Document Frequency

  $$IDF(w) = log\frac{n}{DF(w)}$$

  (n: Gesamtzahl der betrachteten Dokumente, DF: Anzahl der Dokumente, die das Wort w mindestens einmal enthalten). Um die Anzahl der Dokumente herauszufinden, die das Wort w enthalten, müssen das Feature–Wörterbuch und sämtliche Vektoren durchlaufen und überprüft werden. Anhand des Feature–Wörterbuchs kann ein Array erzeugt werden, daß die IDF–Werte zu den Wörtern abspeichert.
- TFIDF–Gewichtung berechnen: Je häufiger ein Wort in einem Dokument vorkommt, desto wichtiger (TF–Teil) und wenn das Wort in vielen Dokumenten vorkommt desto unwichtiger kann es sein (IDF–Teil).
- Um den d–Vektor zu erhalten muss nun die TF–Repräsentation des Textes mit der IDF–Repräsentation der Wörter aus dem Feature–Wörterbuch elementweise multipliziert werden.
- Um die Ähnlichkeit zwischen Dokumenten herauszufinden wird der Kosinus der zugehörigen Vektoren benutzt.

13.3.3 Bayes-Klassifikator

Der Bayes–Klassifikator basiert auf einer Modellierung von Text mit wahrscheinlichkeitstheoretischen Methoden. Es wird davon ausgegangen, daß Texte von verschiedenen Wahrscheinlichkeitsverteilungen erzeugt werden. Diese Verteilungen sind sehr komplex und lassen sich direkt nicht handhaben. Deshalb werden vereinfachende Annahmen gemacht, die es ermöglichen, die Verteilungen auf einem Computer darzustellen und aus Trainingsdaten zu schätzen.

Quelle: [Joachims, 1997]

13.3.4 k–NN-Klassifikator (k–Nearest–Neighbor–Verfahren)

> Bei dem k–Nearest–Neighbor-Verfahren handelt es sich um einen Klassifikatorenalgorithmus, der in der Statistik entwickelt wurde. Im Unterschied zu dem naiven Bayes'schen Klassifikator werden keine Verteilungsannahmen gemacht. Vorwissen kann allerdings an anderer Stelle, nämlich bei der Definition des sogenannten Abstandsmaßes eingebracht werden. Dieses Abstandsmaß (wird häufig mit dem Cosinus zweier Dokumentvektoren berechnet) wird dazu benutzt, ähnliche Situationen aus der Vergangenheit zu finden. Es sollte so beschaffen sein, daß Dokumente, die anhand dieses Maßes ähnlich sind, möglichst zur selben Klasse gehören. Die Grundidee des k–Nearest–Neighbor–Verfahrens ist es, ein neues Beispiel so zu klassifizieren wie die ähnlichsten k Beispiele, für die man die Klassifikation kennt.

Quelle: [Joachims, 1997]

Wie groß k im Einzelfall sein soll, kann von Ihnen selbst bestimmt werden. Mit der Methode `setK (int i)` aus der Klasse `KNNClassifier` wird der Wert für k gesetzt. Wird k nicht explizit übergeben, so erhält k automatisch den Wert fünf. Um den aktuellen Wert von k auszulesen kann die Methode `getK()` aufgerufen werden.

Da die Datenanalyse nicht immer komplizierte Klassifizierer benötigt, gibt es noch drei speziellere Klassifikatoren, die für bestimmte Einsatzgebiete benutzt werden können.

Da wäre zum einen der `LinkPageClassifier`. Dieser Klassifikator ist in der Lage, HTML-Seiten anhand der in der Seite vorkommenden Links zu klassifizieren. Sollte die Mehrheit der Links zu denen im Training angegebenen Konzepten gehören, so wird die Seite als positiv eingestuft.

Als nächstes gibt es dann noch den `NetEntityTypeClassifier`, der anhand des Objekt-Typs `NetEntity` klassifiziert. Sollte der Klassenname der übergebenen `NetEntity` oder einer Elternklasse dieser `NetEntity` einem Name entsprechen, mit dem der Klassifizierer trainiert wurde, so wird 1.0 eingetragen, ansonsten 0.0. Wird der Klassifizierer z.B. mit `dataprovider.net.TextNetResult` trainiert, so werden alle davon erbenden Klassen sowie die Klasse selbst als positiv klassifiziert.

Zum Schluß gibt es noch den `URLClassifier`. Hier werden URL´s anhand von Wortvorkommen klassifiziert. Alle Worte die in den Beispielen vorkommen, werden auf Vorkommen in der URL der übergebenen `NetEntity` überprüft. Sollte mindestens eines der Worte vorkommen, trägt der Klassifikator 1.0 ein, ansonsten 0.0. Die Groß–/Kleinschreibung wird nicht beachtet.

Falls Ihnen diese Klassifikatoren nicht ausreichen, so besteht die Möglichkeit weitere Klassifizierer zu implementieren. Was Sie dabei beachten müssen zeigt Ihnen das unten aufgeführte Beispiel mit den aufgeführten Erklärungen.

13.3.5 Implementierung neuer Klassifikatoren

Bei der Implementierung eines neuen Klassifikators muß zunächst darauf geachtet werden, daß es mindestens zwei Methoden gibt. Dies wäre zum einen die Trainingsfunktion (`public int train ()`) und zum anderen die eigentliche Klassifikationsmethode (`public NetEntity classify (NetEntity document)`).

Die Trainingsdaten werden von der Datenbeschaffung bereitgestellt und sind unter `untrainedExamples` (aus der Klasse classdataprovider.net.NetEntity) abgelegt. Diese Beispielmenge besteht aus mehreren Strings, wobei jeder einzelne ein Dokument darstellt der zusätzlich noch eine Bewertung enthält, die das Beispiel als positiv oder negativ einordnet.

Diese vorherige Bewertung der einzelnen Beispiele muß von Ihnen vorgenommen werden, und je gewissenhafter Sie bei Durchführung vorgehen, desto besser sind die Ergebnisse bei der Klassifikation. Um Beispiel hinzuzufügen wird die Methode `addExample` (Übergabe eines Strings oder eines Objekts vom Typ `NetEntity`) aus der Klasse `Classifier` benutzt.

Dies wäre der klassische Weg. Im Laufe der Entwicklung dieses Agenten, wurden zur schnelleren Einbindung von Beispieldaten Hilfsmittel (Tools) entwickelt, die Ihnen die Einbindung von Trainingsmengen und die von Ihnen vorzunehmende Bewertung der einzelnen Bespiele erleichtern sollen. Aus diesem Grund verweise wir auf das Kapitel Tools.

Sind Trainingsbeispiele vorhanden so können diese dafür benutzt werden über den Vektorisierer (Methode: `vectorize (String document, boolean autoAddWord)`), die Wörterbücher für ihre spezielle Aufgabe anzupassen. Um diese Anpassung zu erreichen muß dem Vektorisierer neben dem Text auch noch ein `true` mitgeliefert werden. `true` sorgt dafür, daß das Feature–Wörterbuch angepaßt wird, da gerade das Feature–Wörterbuch für den Aufbau des Dokumentvektors verantwortlich ist.

Die untrainierten Beipiele aus der Menge `untrainedExamples` der Klasse `dataprovider.net-.NetEntity` müssen aus Speicherplatzgründen nach dem durchgeführten Training gelöscht werden (`untrainedExamples.clear()`). Dabei besteht die Möglichkeit als Rückgabewert die Anzahl der trainierten Beispiele auszulesen.

Nachdem sämtliche Dokumente vektorisiert wurden, kann damit begonnen werden, die Texte jeweils als positives oder negatives Beispiel einzuordnen. Um eine Grenze zu setzen, ab der ein Dokument als positiv oder negativ bezeichnet wird, kann die Methode `setThreshold` aus der Klasse `Classifier` benutzt werden. Hierbei muß beachtet werden, daß der Schwellwert zwischen 0.0 und 1.0 liegt und daß die Dokumente mit einer Bewertung über diesem Grenzwert als zugehörig (also positiv) eingestuft werden. (Der Aufbau eines Dokumentvektors wird ab Seite 199 beschrieben).

Für einige Klassifikatoren ist es wichtig, daß die Vektoren der übergebenen Beispiele in TF–IDF–Form vorliegen. Die Berechnung wurde bereits im Ablauf des Rocchio-Algorithmus beschrieben (Seite 201).

An dieser Stelle ist das Training abgeschlossen. Die gewonnenen Daten (Wörterbücher und Dokumenvektoren) können nun bei der folgenden Klassifizierungsmethode eingesetzt werden.

Die Klassifikationsmethode (`public NetEntity classify (NetEntity document)`) bekommt von der Datenbeschaffung ein Dokument übergeben, das klassifiziert werden soll. Werden unzulässige Werte eingetragen oder eine Exception eingeleitet, so kann der Klassifizierer seine Analyse nicht durchführen. Daraus folgt, daß der Klassifikator kein Ergebnis eintragen kann. Alle transienten Variablen müssen mit der Methode `checkVariables()`

überprüft werden. Die benötigten Wörterbücher und Vektorisierer können aus der `ClassifierDatabase` geladen werden. Zunächst wird das übergebene Dokument vektorisiert. Dabei muß darauf geachtet werden, daß die Vektorisierungsfunktion mit dem Zusatz `false` gestartet wird. `false` sorgt dafür, daß eine Anpassung der Wörterbücher nun nicht mehr stattfindet. (Möglicherweise ist auch an dieser Stelle eine Umwandlung des Vektors in eine TF–IDF–Form notwendig. Hierfür können die bereits in der Trainingsmethode benutzten Funktionen eingesetzt werden.)

Nach der Vektorisierung stehen alle Mittel zur Verfügung, die für eine Einordnung des neuen Dokuments benötigt werden. In der Klasse `Dokumentvector` sind Methoden abgelegt, die die Berechnungen zwischen verschiedenen Vektoren erleichtern.

Nachdem die Klassifikation durchgeführt wurde kann das Ergebnis mit dem Befehl `document.addClassification (getElementname(), classifyResult)` gespeichert werden.

13.4 Grobstruktur eines neuen Klassifikators

Hier bekommen Sie einen Überblick, wie die Struktur eines Klassifikators auszusehen hat. Es sind dabei nur die wesentlichen Punkte aufgeführt, die Ihnen helfen sollen einen eigenen, neuen Klassifizierer in das System einzubinden.

```
// Source file: sourcen/dataanalysis/classifiers/newClassifier.java

package dataanalysis.classifiers;

import dataprovider.net.NetEntity;
import dataanalysis.docvectors.*;
import dataanalysis.databases.*;
import dataanalysis.dictionaries.*;
import informationsexchange.LogService.*;

public class newClassifier {

// hier koennen alle lokalen Variablen definiert werden

// Konstruktor
public newClassifier() {
 super();
 ...}
// Konstruktor mit Datenbankuebergabe
public newClassifier(ClassifierDatabase theDB)
 super(theDB);
 ...}

public int train(){
...
// Ueberpruefung, ob die transienten Variablen gesetzt wurden
error = checkVariables();
...
// Nachdem alle Trainingsdokumente vektorisiert wurden, koennen
// die Dokumente geloescht werden
untrainedExamples.clear();

// Da die Woerterbuecher nun angepasst sind, koennen die notwendigen
// Berechnungen nun durchgefuehrt werden
...
return (Anzahl-Dokumente);
}

public NetEntity classify(NetEntity document){
// das Dokument wird anhand der Daten aus dem Trainings eingeordnet
// und klassifiziert
...
// Eintragung des Klassifikationsergebnisses
document.addClassification(getElementname(),classifiesValue);
...
return (document);
}
}//newClassifier
```

13.5 Operationen auf der ClassifierDatabase

Zum Schluß noch ein Beispiel, wie Sie einen Klassifizierer aus der ClassifierDatabase laden oder speichern können.

```
public void load(String fname)
  throws ClassNotFoundException, IOException
{
  FileInputStream fs = new FileInputStream(fname);
  ObjectInputStream is =  new ObjectInputStream(fs);
  theDB = (ClassifierDatabase)is.readObject();
  is.close();
};

public void save(String fname)
  throws IOException
{
  FileOutputStream fs = new FileOutputStream(fname);
  ObjectOutputStream os = new ObjectOutputStream(fs);
  os.writeObject(theDB);
  os.close();
}
```

Benutzung im Programm:

```
try {
  load("Daten.dat");
} catch(Exception e) {
  e.printStackTrace();
  System.exit(2);
};
```

Kapitel 14

Instanzenlerner

Menschen von dem ersten Preise
lernen kurze Zeit und werden weise.
Menschen von dem zweiten Range
werden weise, lernen aber lange.
Menschen von der dritten Sorte
bleiben dumm und lernen Worte.

Konfuzius

Der Instanzenlerner wird nach der Beendigung einer Suchanfrage von der Systemkontrolle aufgerufen, und hat in erster Linie die Aufgabe den Inhalt der verschiedenen lokalen A–Boxen zu untersuchen und in der globalen A–Box zusammenzuführen. Um die lokalen A–Boxen zu Verfügung zu haben, arbeitet der Instanzenlerner auf dem Planarchiv, das unter anderem die lokalen A–Boxen enthält.
Als erstes ruft die Methode `InstanceLearner.update` die Methode `A_Box.update` einer solcher lokalen A–Box auf. Dieser Aufruf bewirkt, daß die lokale A–Box bereinigt wird. Das heißt, alle Instanzen, die nur ***anything*** oder den unter ***startConcepts*** zusammengefaßten Konzepten zugeordnet werden konnten, werden gelöscht. Genau so wie alle Instanzen, deren *used*–Falg auf false steht und alle Instanzen, deren *delete*–Falg auf true und das *successful*–Flag auf false seht. Im zweiten Schritt durchläuft der Instanzenlerner die lokale und die globale A–Box parallel und fügt neue Instanzen hinzu, bzw. erhöht bei schon vorhandenen Instanzen den `Instance.counter`–Wert um eins und setzt bei `Instance.date` das aktuelle Datum. Zum Schluß löscht der Instanzenlerner alle lang nicht mehr benutzten Instanzen. Dazu berechnet er anhand des *delete–time*-Wertes das Datum, ab wann Instanzen als lang nicht mehr benutzt gelten.
Der Einsatz des Instanzenlerners hat den Effekt, daß die Suchzeit bei ähnlichen Anfragen sehr viel geringer wird, weil in der A–Box schon mehr Informationen vorhanden sind und vom ***Planer*** bei der Erstellung vom Pläner berücksichtigt werden können.
Wird der Instanzenlerner gestartet, werden die von Ihnen einstellbaren Variablen aus der Konfigurationsdatei geladen. Diese Konfigurationsdatei muß von Ihnen vor dem ersten Start des Agenten angelegt werden.

Dateiformat der Konfigurationsdatei

Alle Befehle müssen der Form $< A > = < B >$ entsprechen und in einer Zeile sehen.Zwischen Gross- und Kleinschreibung wird nicht unterschieden.

Die eine zu setzende Variable ist die *deleteTime*, die die Zeitspanne angibt, ab der unbenutzte Instanzen in der globalen A-Box gelöscht werden sollen, damit die A–Box im Laufe der Zeit nicht zu groß wird. Möglich sind Angaben in years, months, weeks, days, hours, minutes, seconds. Die Einheiten können auch, durch Kommata getrennt, kombiniert werden. Wird keine Einheit angegeben, ist days die default–Einheit.
Die andere Variable bezieht sich auf Konzepte, deren Instanzen nicht von den lokalen A–Boxen in die globale übernommen werden sollen. In der Konfigurationsdatei heißen diese Konzepte ***startConcepts***, und sind z.B. die Benutzereingaben, die unter das Konzept searchstring fallen.

Beispiel

Beispielhafte Configdatei des Instanzenlerners

# deleteTime = 8 weeks
# deleteTime = 3 days, 7 hours
deleteTime = 2 months

startConcepts = keyword, searchstring

Kapitel 15

Operatorlerner

Lernst Du wohl,
wirst Du gebratener Hühner voll.
Lernst Du übel,
mußt Du mit der Sau zum Kübel

Martin Luther
Tischreden

Überlegungen zum Operatorlerner

Der Operatorlerner ist **nicht** implementiert!
Im Gegensatz zum Instanzenlerner soll der Operatorlerner nicht nach jeder Suchlanfrage aufgerufen, weil der Operatorlerner die Daten mehrerer Suchanfragen miteinander vergleichen können muß. Deshalb soll er ein eigenständiges Programm sein, das vom Shellbenutzer gestartet werden muß.

Der Operatorlerner soll auch auf dem Planarchiv arbeiten, das die gesammelten Informationen vorangegangener Suchanfragen enthält, und müßte Methoden zur Verfügen haben, um diese Informationen analysieren zu können.

Die eine Idee für den Operatorlerner ist, daß er die Gewichtungen der einzelnen Operatoren neu berechnet, je nachdem wie erfolgreich der Operator in den vorangegangenen Suchen war. Die andere Idee ist, geeignete Reihenfolgen, in denen die Operatoren hintereinander ausgeführt werden sollen, zu finden und sie mit Hilfe der Compound–Operatoren den Planern zu Verfügung zu stellen.

Das Problem ist nun, Kriterien zu bestimmen, mit denen man die Güte von Operatoren berechnen kann. Die Tatsache, daß sie zu Plänen gehört haben, die zum Erfolg geführt haben, ist wahrscheinlich nicht ausreichend, da der Erfolg einer Suchanfrage nicht ausschließlich von den Operatoren abhängt. Bei den Suchmaschinen–Operatoren könnte man die Anzahl der Ergebnisse und deren Konfidenzwert zur Berechnung der Güte heranziehen. Aber auf Grund der unzureichenden Datenmenge, die zur Analyse zur Verfügung standen, war es uns nicht möglich zu fundierten Erkenntnissen zu kommen.

Anhang A

Das Logbuch: die Klasse LogService

Es ist von großem Vorteil, die Fehler, aus denen man lernen kann, recht frühzeitig zu machen.
Winston Churchill

Die Klasse `LogService` stellt eine einfache Möglichkeit dar, während eines Programmlaufs Informationen — welche sich typischerweise auf eben jenen beziehen — in Form von Zeichenketten in eine Datei — im folgenden „Logdatei" genannt — zu schreiben. Sinn und Zweck solcher „Logbucheintragungen" sind i. a. das bessere Verständnis von komplexen/unüberschaubaren Programmabläufen und das Lokalisieren von Fehlern im Programm.

Die Klasse `LogService` ist als `abstract` definiert, weil sie ausschließlich statische Methoden, Variablen und Konstanten enthält (sprich Klassenmethoden, -variablen und -konstanten), und eine Instanziierung damit nicht nötig und deshalb auch nicht sinnvoll ist.[1]

Ein Eintrag in die Logdatei muß eine Priorität besitzen, welche sich in der Regel auf die Wichtigkeit der Information beziehen sollte. Sind Logdateieinträge für eine Fehlersuche bestimmt, so kann die Priorität auch als Synonym für eine bestimmte Methode o. ä. "missbraucht" werden. Die Klassenmethode `log`, der als erster Parameter eine Zahl vom Typ `int` und als zweiter Parameter ein *`String`* übergeben wird, nimmt die Eintragung in die Logdatei vor. Dabei ergibt sich der Eintrag aus der Verkettung von der Priorität, dem aktuellen Systemdatum und der eigentlichen Information. Sollte die Logdatei noch nicht bzw. nicht mehr existieren, so wird sie neu erzeugt. **Bei massiver Nutzung dieses Services, zum Beispiel zur Fehlersuche, kann die Logdatei unter Umständen (sehr) schnell (sehr) groß werden, so daß auf rechtzeitiges, manuelles Löschen zu achten ist.** Aus diesem Grund existiert die Klassenvariable `logLevel`. Deren Funktionalität besteht darin, daß Logdateieinträge, die eine geringere Priorität als deren Wert besitzen, nicht an die Logdatei angehangen werden.[2] Es ist somit nicht nötig, Programmcode, der Logdateieinträge zur Fehlersuche erzeugt, zu ändern, wenn diese nicht (mehr) gewünscht werden.[3] Um die Nutzung dieser Funktionalität zu erleichtern, existieren Klassenkonstanten, die als Werte für `logLevel` gedacht sind. Prioritäten, die bei der Fehlersuche eingesetzt werden, sollten aus $[0, 99]$ sein, wenn diese Konstanten benutzt werden. Initialer Wert von `logLevel` ist der

[1] außer zur abkürzenden Schreibweise vielleicht . . .
[2] Sie werden einfach „vergessen", enden also direkt in der „garbage collection".
[3] Die Funktionsaufrufe finden nach wie vor statt, was den Programmablauf dann unnötigerweise verlangsamt.

Wert der Klassenkonstante `DEBUG`. Der Wert von `logLevel` kann auch über die Konfigurationsdatei gesetzt werden.

Neben der Methode `log` gibt es u. a. noch die Methoden `logInfo`, `logWarning`, `logError` und `logFatal`, bei denen die Priorität automatisch auf den Wert der entsprechenden Klassenkonstante gesetzt wird. Ferner existiert zu jeder der genannten Methoden eine überladende Mehode, der zusätzlich ein Objekt übergeben werden muß. Bei diesen Methoden wird vor die übergebene Nachricht der Klassenbezeichner des übergebenen Objekts gesetzt.

Der Name der Logdatei wird durch die Klassenvariable `logfileName` bestimmt, welche mit der Zeichenkette "`agent.log`" initialisiert ist. Über die Konfigurationsdatei (s. II D.1)kann diese Einstellung überschrieben werden. **Sollte ein Fehler[4] bei einer Logdateieintragung auftreten, wird das Programm mit einer entsprechenden Nachricht auf der Standardausgabe abgebrochen.** Für die genauen Signaturen der Methoden und die Namen der Konstanten sei auf die JavaDoc–HTML–Dokumentation der Klasse `LogService` hingewiesen.

[4] genauer: eine IO-Exception

Anhang B

Die Klassifikatoren unterstützende Werkzeuge

B.1 dbExplorerConsole

Das Tool `dbExplorerConsole` ist ein menügesteuertes Programm, mit dem man Datenbanken (`ClassifierDatabase`) erzeugen, anzeigen und bearbeiten kann. Die Entwicklung dieses Tools ist noch nicht abgeschlossen, es ist aber schon eine funktionierende Version vorhanden. Für die Benutzung ist lediglich die Kenntnis der Eigenschaften der `Classifier` wichtig. Diese Eigenschaften können in der Javadoc–Dokumentation zu BotIshelly nachgelesen werden.

B.2 ClassifierGenerator

B.2.1 Einleitung

Mit dem Tool `ClassifierGenerator` sind Sie in der Lage, die Klassifiziererdatenbank auf eine einfache Art und Weise zu erzeugen oder zu erweitern. Die Informationen mit welchen Beispielen und Werten die Klassifizierer in die Datenbank eingefügt werden sollen entnimmt dieses Tool im wesentlichen dem `ClassifierConfigFile`, dass sich im Paket `dataanalysis.tools` befindet. Weiteres zum konkreten Inhalt der Konfigurationsdatei können Sie der Javadoc-Dokumentation entnehmen.

B.2.2 Aufrufsyntax

Das Tool wird mit (mindestens) zwei Parametern und eventuellen Optionen aufgerufen. Aufgerufen wird das Tool wie folgt:

```
java classifiergenerator.ClassifierGenerator [-v] [-f]
        Databasefilename Classifiername...
```

- **Parameter Databasefilename**
  Hier geben Sie den Dateinamen der `ClassifierDatabase` an, in der die Klassifikatoren gespeichert werden sollen. Existiert unter diesem Dateinamen keine `ClassifierDatabase`, so wird eine neue `ClassifierDatabase` erzeugt.

- **Parameter Classifiername...**
  Hier geben Sie den oder die Namen der Klassifizierer an, die erzeugt werden sollen. Die Endung „.examples“ wird automatisch entfernt, d.h. durch den Parameter „test.examples“ wird ein Klassifizierer mit dem Namen „test“ erzeugt. Sie können also immer die Dateinamen der Konfigurationsdateien verwenden, wenn die Konfigurationsdateien die Endung „.examples“ besitzen.

- **Option -v**
  Wird diese Option angegeben, dann wird der Inhalt der `ClassifierDatabase` nach dem Abarbeiten aller Klassifizierer auf dem Bildschirm ausgegeben. Ansonsten wird nur eine Zusammenfassung angezeigt, wie viele Klassifizierer hinzugefügt wurden.

- **Option -f**
  Wird diese Option angegeben, dann werden eventuell schon in der `ClassifierDatabase` vorhandene Klassifizierer gleichen Namens ersetzt.

B.2.3 Aufrufbeispiele

- `java classifiergenerator.ClassifierGenerator -v -f`
  `testDB.cldb productpage.examples`
  Es wird eine `ClassifierDatabase` mit dem Namen „testDB.cldb“ verwendet (ggf. erzeugt). In dieser Datenbank wird ein Klassifizierer mit dem Namen „productpage“ erzeugt, wobei ein eventuell schon vorhandene Klassifizierer mit diesem Namen ersetzt wird. Die Daten zur Erzeugung werden aus der Datei „productpage.examples“ verwendet. Am Ende wird eine Zusammenfassung und der Inhalt der Datenbank angezeigt.

- `java classifiergenerator.ClassifierGenerator`
  `testDB.cldb test.examples foo.examples`
  Es wird eine `ClassifierDatabase` mit dem Namen „testDB.cldb“ verwendet (ggf. erzeugt). In dieser Datenbank werden zwei Klassifizierer erzeugt. Sollte z.B. schon ein Klassifizierer mit dem Namen „test“ vorhanden sein, wird eine Fehlermeldung ausgegeben und mit dem nächsten Klassifizierer fortgefahren. Am Ende wird nur die Zusammenfassung angezeigt.

B.3 GetLocalExamples

B.3.1 Einleitung

Mit dem Tool `GetLocalExamples` sind Sie in der Lage, die Beispiele für die Klassifikatoren Ihres Agenten auf dem lokal verfügbaren Speicherplatz (z.B. Festplatte) abzulegen.

Durch diese Maßnahme wird das spätere Training der Klassifikatoren erheblich beschleunigt und der Netzverkehr bei der (empirischen) Einstellung der Klassifikatorparameter erheblich gesenkt. Desweiteren können so Testläufe mit dem Tool `ClassifierTester` innerhalb einer kurzen Zeit durchgeführt werden.

B.3.2 Benutzung

Dieses Tool erzeugt jeweils aus einem gegebenen `ClassifierConfigFile` eine lokalisierte Version, die im Unterverzeichnis „lokal“ abgespeichert wird. Die Beispiele in der Datei werden in einem Unterverzeichnis abgespeichert. Geben Sie z.B. eine Datei „test1.examples“ vor, so wird im Unterverzeichnis „lokal“ ein Unterverzeichnis „test1“ erstellt, in dem die Beispiele aus dem `ClassifierConfigFile` abgespeichert werden. Das angepasste `ClassifierConfigFile` wird im

Unterverzeichnis „lokal“ abgespeichert. Sollte dieses schon vorhanden sein, so werden nur die Beispiele geholt, die noch nicht lokal vorhanden sind. Es ist also einfach möglich eventuelle Serverausfälle zu überbrücken, indem Sie mehrere Durchläufe zu verschiedenen Zeiten machen.

B.3.3 Aufrufsyntax

Das Tool wird mit einem Parameter und eventuellen Optionen aufgerufen. Aufgerufen wird das Tool wie folgt:

```
java classifiergenerator.GetLocalExamples [-r] [-v] [-a]
        [-n<num>] ConfigFilename
```

- **Parameter ConfigFilename**
  Hier geben Sie den Dateinamen des `ClassifierConfigFile` an, dessen Beispiele geholt werden sollen.

- **Option -r**
  Wird diese Option angegeben, so wird nicht der formatierte Text, sondern der Quelltext der Beispiele (also z.B. HTML-Code) abgespeichert. Bei gesetzter Option wird die Methode `getRaw()` der jeweiligen *NetEntity* aufgerufen, ansonsten die Methode `getText()`.

- **Option -v**
  Wird diese Option angegeben, so werden detaillierte Ausgaben auf dem Bildschirm ausgegeben. Jede Beispiel–URL wird dann ausgegeben.

- **Option -a**
  Wird diese Option angegeben, so wird nicht automatisch „.html“ an jede geholte Beispiel–URL angehangen. Zur Zeit ist das Anhängen von „.html“ notwendig, da die Beispiele nur so wieder verarbeitet werden können (bei file–URL's muss die Endung „.html“ vorliegen, um `HTMLTextNetResults` zu erhalten.).

- **Option -n<num>**
  Wird diese Option angegeben, so werden immer **num** Beispiele gleichzeitig geholt. Dadurch wird die Geschwindigkeit im Gegensatz zum sequentiellen Abruf erheblich erhöht. Als praktikabel hat sich ein Wert von 20 herausgestellt.

Anhang C

Generierung von Operatoren

C.1 OperatorGenerator

Dieses Werkzeigt ermöglicht es dem Shellbenutzer schnell und einfach Operatorcoderahmen zu erzeugen. Das lästige und fehleranfällige erzeugen der Konstruktoren entfällt dadurch.

Das Werkzeug wird mit folgender Kommandozeile gestartet:

```
java operatorgenerator/OperatorTextFileReader <Operator-Signaturdatei>
```

Das Werkzeug legt dann für jeden in der Operator–Signaturdatei beschriebenen Operatoren eine Datei <Operatorname>.java an. Bestehende Dateien gleichen Namens werden *ohne* Rückfrage *überschrieben.*

Die Operator–Signaturdatei muß dabei folgendes Format haben:

Jeder neue Operatorabschnitt wird mit der Zeile @name <Operatorname> Danach folgen die Vorbedingungsprädikate, die zeilenweise eingeleitet durch @precondition aufgezählt werden. Anschliessend wird mit @postcondition ein Nachbedingungsabschnitt eingeleitet. Nach dieser Zeile folgen die zu dieser Nachbedingung gehörenden Add- und Deletelisteneinträge jeweils zeilenweise mit den Tags @addlistitem bzw. @deletelistitem eingeleitet. Eine weitere Nachbedingung wird wieder mit dem @postcondition-Tag begonnen. Zeilen die mit einem # beginnen werden ignoriert.

Hinter den Add- oder Deletelistenprädikaten sind durch einen „:“ abgetrennt die Buchstaben „y“ oder „n“ je Prädikatsargument einzutragen.„y“ bedeutet, daß diese Argument neu vom Operator erzeugt wird und daher vom Planer als dynamisches Weltobjekt behandelt werden muß. Die genaue Syntax entnehmen Sie bitte Anhang II D.4.
Beispiel:

```
# blubber-Operator
@name blubber
@precondition be_involved_with(a,b)
@precondition archive_url(a)
@postcondition
@addlistitem has_downloadurl(a,b):n,n
@addlistitem distribution(a):n
@addlistitem contains_software(a,b):n,n
@deletelistitem be_involved_with(a,b):n,n
@deletelistitem belongs_to(a,b):n,n
```

Der blubber–Operator ist also ein Operator mit dem Namen *blubber*, zwei Vorbedingungsprädikaten und genau einer Nachbedingung. Die Nachbedingung besteht einer Addliste mit drei Einträgen und einer Deleteliste mit zwei Einträgen.

Zu beachten ist, daß Prädikatsargumente mit gleichem Namen im Konstruktor denselben ***PredArguments*** entsprechen. Das bedeutet, im obigen Beispiel, daß nur genau zwei ***PredArguments*** erzeugt werden: das ***PredArgument*** mit dem Namen *a* und das ***PredArgument*** mit dem Namen *b*. Natürlich macht es keinen Sinn in einer Nachbedingung ein Prädikatargument mit dem Namen eines Prädikatarguments aus der Vorbedingung erzeugen zu wollen.

Abweichend von der in Anhang II D.4 genannten Syntax können bei Vorbedingung die Kennzeichen `:n` oder `:n,n` bzw. `:n` oder `:n,n` entfallen, da sie hier bedeutungslos sind. Vorbedingung können keine Objekte erzeugen.

Fehlerhafte Zeilen in der Operator–Signaturdatei ignoriert das Hilfsprogramm unter Angabe der fehlerhaften Zeilennummmer.

C.1.1 grafische Oberfläche

Es gibt eine prototypische Implementation eines grafischen Frontends zu dem beschriebenen Operatorgenerator. Dieses ist aber weder benutzerfreundlich noch komplett fertig gestellt. Dem experimentierfreudigen Leser wollen wir aber die Möglichkeit nicht vorenthalten. Fertig kompiliert kann man es mit

```
java operatorgenerator <A-Box-TextDatei>
```

starten. In diesem Tool können dann Operatoren zusammengeklickt werden. Da die Arbeiten an diesem Frontend nicht abgeschlossen wurden und es nicht fester Bestandteil von BotIshelly ist, wird für eine weitere Dokumentation auf die Sourcecode–Dokumentation verwiesen.

Anhang D

Beschreibungssprachen und Dateiformate

Neuerdings habe ich mir's zur Richtschnur gemacht: In Sachen, die ich nicht verstehe, und es tut einer etwas, das ich nicht begreife, so macht er's dumm und greift's ungeschickt an; denn das, was schicklich und recht ist, begreift man auch in unbekannten Dingen; wenigstens muß es einer einem leicht und bald erklären können

Goethe, an Charlotte von Stein
12.9.1780

D.1 Konfigurationsdatei des Agenten

In der Konfigurationsdatei stehen alle wichtigen Daten, die für eine Suche notwendig sind.

Die von uns zu Verfügung gestellte Konfigurationsdatei heißt config.sys.

Falls Sie eine andere Konfigurationsdatei benutzen möchten, müssen Sie darauf achten, daß Sie in der Klasse `systemcontrol.Flowcontrol` den Namen der Konfigurationsdatei ändern. Folgendes müssen Sie unbedingt beachten: Kommentare werden mit `#` eingeleitet und dürfen nur am Anfang einer Zeile stehen. Abkürzungen wie $HOME bitte NICHT verwenden. Außerdem ist die Datei case sensitiv.

Hier ist ein Beispiel für eine Konfigurationsdatei.

```
# config.sys Version 0.2

# Globale Konfigurationsdatei fuer BotIShelly

# Working Directory wird vorangestellt, wenn bei Pfadangaben ein relativer
# Pfad angegeben wird

Working Directory = /home/pg343/schroet3
```

Falls Sie relative Pfade benutzen, sollten Sie das beim Laden der Daten in der `Flowcontrol` beachten und diese ergänzen. Zurück zu unserem Beispiel:

```
# relative oder absolute Pfade der zu ladenden Dateien

A_BoxDatei = sourcen/integration-shell/DebianSuchagent.abox
A_BoxObject = sourcen/integration-shell/DebianSuchagent.aboxObject
T_Box = sourcen/integration-shell/DebianSuchagent.tbox
OperatorDBDatei = sourcen/integration-shell/OperatorDB.opdb
OperatorDBObject = sourcen/integration-shell/OperatorDB.opdbObject
ClassDatabase = /home/pg343/share/DebianSuchagent/DebianSuchagent.cldb

# Bool der definiert, ob die Classifier DB am Ende der Suche
gespeichert werden soll, oder nicht

SaveClassDB = false

# Einstellungen fuer die Log-Datei

# wird LogPath nicht angegeben, wird der Default benutzt ($HOME/agent.log)
logfileName = agent.log

# wird LogLevel nicht angegeben, wird der LogLevel auf 0 gesetzt
logLevel = 0

# Proxy Settings

proxySet  = true
proxyHost = fbi-www
proxyPort = 3128

# maximale Anzahl der Instanzen pro Operatorvorbedingungspr"adikat bestimmen,
die bei der Suche ber"ucksichtigt werden

MaxInstance = 10
```

Es darf in Ihrer Konfigurationsdatei nichts davon fehlen, was Sie in unserem Beispiel sehen. Natürlich können Sie diese Datei beliebig erweitern, sofern Sie die syntaktische Einsränkungen beachten.

D.2 Konfigurationsdatei für die Operatorendatenbank

Die Konfigurationsdatei für die Operatorendatenbank ist vielleicht die schlichteste und am wenigsten benutzerfreundliche Konfigurationsdatei in BotIshelly. Dafür wird sie aber auch nur einmal im gesamten Agentenleben gebraucht. Ihre Aufgabe ist es die Operatorendatenbank mit den geschriebenen Operatoren zu füllen. Die Syntax dieser Datei ist einfach:

In der ersten Zeile steht der Name der Operatorklasse einschließlich des Paketnamens, in dem er sich befindet. In der darauffolgenden Zeile steht die Gewichtung dieses Operators mit der er initial in die Datenbank eingehen soll. Die Gewichtung ist ein double-Wert.

Kommentare und Leerzeile sind in dieser Datei nicht erlaubt.

Fehlerhafte Zeilen werden ignoriert und ihre Zeilennummer im Agentenlog angegeben.

D.3 Syntax für ClassifierConfigFile

D.3.1 Regeln

Die folgenden Regeln gelten für das `ClassifierConfigFile`:

- Führende Leerzeichen in einer Zeile werden entfernt.
- Tab-Zeichen werden in Leerzeichen umgewandelt.
- Leerzeilen werden entfernt.
- Beispiele die Leerzeichen enthalten sind in Anführungszeichen (") einzuschließen.
- Kommentare werden durch das Zeichen „#" als erstes Zeichen der Zeile eingeleitet.
- Soll ein Beispiel „asis" an den Klassifizierer gegeben werden, so muss das erste Zeichen in der Zeile ein „@" sein.
- Schlüsselwörter beginnen mit dem Kommentarzeichen gefolgt von beliebig vielen Leerzeichen und anschließend dem Schlüsselwort. Das mögliche Argument wird durch ein „="–Zeichen vom Schlüsselwort getrennt.
- Folgt ein `double`–Wert auf das Beispiel (durch Leerzeichen getrennt), so wird dieser als Bewertung des Beispiels verwendet, ansonsten der Wert, der durch das Schlüsselwort `DEFAULT` festgelegt wurde.

D.3.2 Schlüsselwörter

Es gibt die folgenden Schlüsselwörter (Groß–/Kleinschreibung wird unterschieden!):

- `CREATED` : gibt das Erstellungsdatum der Datei an.
- `DEFAULT=f` : ein `double`–Wert, der als Standard–Klassifikation benutzt wird. Ist f kein gültiger `double`–Wert, so wird dieser ignoriert. Standardwert ist 1.0.
- `END` : bricht das Einlesen nach dieser Zeile ab.
- `NEEDS=x` : bezeichnet einen Namen eines benötigten Klassifizierers. Wird noch nicht beachtet.
- `NEGATIVDICT=x` : den Namen des Negativ–Wörterbuches (Stopwortliste) des Klassifizierers. Standardwert ist „Global_Stoplist".
- `THRESHOLD=f` : ein `double`–Wert, der den Threshold des Klassifizierers festlegt. Standardwert ist 0.5.
- `TYPE=x` : legt den Typ des Klassifizierers fest (Klassenname).
- `VECTORIZER=x` : den Namen des Vektorisierers des Klassifizierers.
  Standardwert ist „<Klassifizierername>_Vectorizer".
- `WORDDICT=x` : den Namen des Feature–Wörterbuches des Klassifizierers. Durch dieses Standardwert ist „<Klassifizierername>_Words".

D.3.3 Beispiel für ein ClassifierConfigFile

```
#  Ein Beispiel
#  TYPE=dataanalysis.classifiers.DummyClassifier
#  DEFAULT=1.0
#  THRESHOLD=0.5
#  NEEDS=a_classifier_name
#  VECTORIZER=a_classifier_name_Vectorizer
#  WORDDICT=a_classifier_name_Features
#  NEGATIVDICT=Global_Stoplist
#
@"ein String als Beispiel" 0.75
# Beispiel-URL mit Benutzung des DEFAULT-Wertes
http://localhost/
#
# Ende der Datei
# END
```

D.4 Operator–Signaturdateisyntax

Was ein <gültiger Java-Variablenbezeichner> ist entnehmen sie bitte der Java-Dokumentation. Was eine <Zeichenkette> oder ein <Zeilenumbruch> ist sollte dem Leser bekannt sein. Die <Operatorsignaturdatei> soll als Textfile geschrieben werden.

```
<Name>                   ::=<gültiger Java-Variablenbezeichner>
<Kommentar>              ::=#<Zeichenkette><Zeilenumbruch>|
                            <Kommentar><Kommentar>
<neu erzeugt>            ::=n|y
<einstelliges Prädikat> ::=<Name>(<Name>):<neu erzeugt>
<zweistelliges Prädikat>::=<Name>(<Name>,<Name>):<neu erzeugt>,<neu erzeugt>
<Prädikat>               ::=<zweistelliges Prädikat>|
                            <einstelliges Prädikat>
<Vorbedingung>           ::=[@precondition <prädikat>]|
                            <Vorbedingung><Vorbedingung>|
                            <Kommentar><Vorbedingung>
<Addlisteneintrag>       ::=@addlistitem <prädikat>|
                            <Kommentar><Addlisteneintrag>
<Deletelisteneintrag>    ::=@deletelisteitem <prädikat>|
                            <Kommentar><Deletelisteneintrag>
<Nachbedingung>          ::=@postcondition {<Zeilenumbruch><Addlisteneintrag>}
                                           {<Zeilenumbruch><Deletelisteneintrag>}|
                            <Nachbedingung><Zeilenumbruch><Nachbedingung>|
                            <Kommentar><Nachbedingung>
<Operatorname>           ::=@name <Name>|
                            <Kommentar><Operatorname>
<Operatorsignatur>       ::=<Operatorname><Zeilenumbruch><Vorbedingung>
                                          <Zeilenumbruch><Nachbedingung>|
                            <Kommentar><Operatorsignatur>
<Operatorsignaturdatei> ::=<Operatorsignatur>{<Zeilenumbruch><Operatorsignatur>}
```

Index