Technische Universität Darmstadt

Fachbereich Informatik
Knowledge Engineering

Prof. Dr. Johannes Fürnkranz

Diplomarbeit:

Aufbau einer on-line Datenbank für Machine Learning Datenbanken

---

Betreuer: Prof. Dr. Johannes Fürnkranz
Verfasser: Thomas Wanderer, Matrikelnummer 991397

Eidesstattliche Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe angefertigt habe und mich anderen als der im beigefügten Verzeichnis angegebenen Hilfsmittel nicht bedient habe. Alle Seiten, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht.

_____

(Datum, Ort) (Unterschrift)

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

1.1 Maschinelles Lernen

Das Gebiet des Maschinellen Lernens beschäftigt sich mit Algorithmen, die ihre Leistungsfähigkeit bezüglich dem Lösen einer Aufgabe, durch Erfahrungen mit dem Lösen der Aufgabe verbessern können. Ein solcher Algorithmus nimmt als Input Trainingsdaten entgegen und erzeugt daraus, in Abhängigkeit der gemachten Erfahrungen (bereits gesehene Trainingsdaten), eine Funktion, die die Aufgabe löst. [MIT]
Die Methoden des Maschinellen Lernens werden beispielsweise im Data-Mining verwendet, wo das Trainieren (Lernen) von Klassifizierern eine der grundlegenden Anwendungen ist. [WITT]
Abbildung 1 verdeutlicht diese Lernaufgabe.

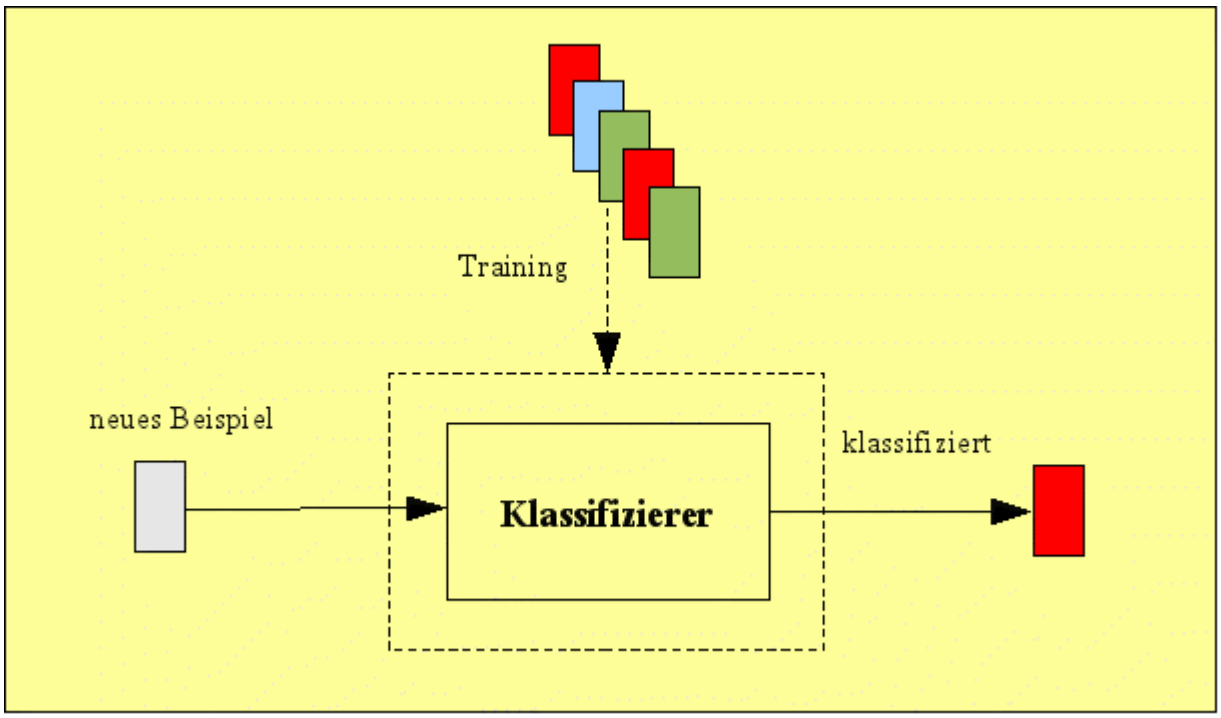

Abbildung 1: Trainieren eines Klassifizierers [WEB]

Der Lernalgorithmus erhält in einer Lernphase Trainingsbeispiele, woraus er eine Funktion (Klassifizierer) generiert, die die gesehenen Beispiele verallgemeinert, sodass diese die Fähigkeit besitzt ein neues Beispiel einer Klasse zu zuordnen. In dem hier aufgeführten Beispiel spricht man vom überwachten Lernen, da die Trainingsbeispiele mit einem Label versehen sind, d.h. das Ergebnis der Funktion (die Klasse, dem das Beispiel angehört) für ein Trainingsbeispiel ist bekannt.

1.2 Trainingsdaten

Die Trainingsdaten, wie in Abbildung 1 dargestellt, stellen den Input des Lernalgorithmus dar. Sie repräsentieren die Erfahrungen aus denen ein Lernalgorithmus lernen soll. Die Menge aller Trainingsbeispiele bezüglich eines Lernproblems, wird im Folgendem als Datensatz bezeichnet.

1.2.1 Merkmale von Datensätzen

Ein Datensatz besteht aus einer Menge von Instanzen, wobei jede Instanz ein Trainingsbeispiel eines Lernproblems ist, d.h. ein individuelles und unabhängiges Beispiel der zu erlernenden Funktion. Die Anwendung der gelernten Funktion (Klassifizierer) aus Abbildung 1 auf eins der Trainingsbeispiele liefert folglich die Klasse, dem das Beispiel angehört.
Eine Instanz besteht aus einer Menge von Werten festgelegter Attribute, wobei die Attribute die Merkmale sind, durch die eine gemachte Erfahrung beschrieben werden kann. In Bezug auf das Lernproblem aus Abbildung 1 sind die Attribute zur Beschreibung der Trainingsbeispiele genau die Größen (im Optimalfall), die die Klassenzugehörigkeit bestimmen.
Abbildung 2 zeigt einen Datensatz bestehend aus drei Instanzen, welche durch die Werte fünf verschiedener Attribute charakterisiert sind.

attr01	attr02	attr03	attr04	attr05
4	1.2	klein	23/11/2001	ja
7	6.5	mittel	02/03/1997	nein
16	3.1	groß	01/11/2007	ja

Abbildung 2: Datensatz

Die Namen der Attribute des Datensatzes stehen im Tabellenkopf. Die Zeilen unterhalb des Tabellenkopfs stellen jeweils eine Instanz dar. Innerhalb einer Spalte befinden sich in den Instanzen die Werte des selben Attributs. Da die Anzahl der Attribute eines Datensatzes konstant ist, ist die Anzahl der Werte aller Instanzen eines Datensatzes gleich. Im Datenbankkontext stellt ein Datensatz (alle Instanzen) eine Relation dar, während die Attribute das Relationenschema bilden.
Der Wert eines Attributs ist ein Maß des Merkmals, das durch das entsprechende Attribut ausgedrückt wird. In dem Datensatz aus Abbildung 2 könnten beispielsweise die Werte des Attributs *attr01* Celcius-Werte sein, als konkrete Messung des Merkmals Temperatur.
Hinsichtlich der Werte, die die Attribute annehmen können, lassen sich verschiedene Datentypen unterscheiden. Die Bezeichnungen für Attribute eines Typs und die Unterscheidungskriterien der Datentypen können dabei auf unterschiedliche Weise festgelegt werden.
Im Rahmen dieser Arbeit werden die folgenden fünf grundlegenden Attributtypen unterschieden:

- nominal - nominelle Attribute
- numeric - numerische Attribute
- string - Attribute mit String-Werten
- date - Attribute, die ein Datum darstellen
- relational - relationale Attribute

Nominelle Attribute können Werte einer definierten endlichen Menge verschiedener String-Werte annehmen. Subtypen nomineller Attribute sind Attribute vom Typ boolean und ordinal. Während die Werte-

mengen boolescher Attribute aus genau zwei Elementen besteht, lassen sich Werte ordinaler Attribute hierarchisch ordnen. Ein Beispiel für eine Wertemenge eines ordinalen Attributs ist die Menge *{klein, mittel, groß}* mit $klein < mittel < groß$.
Numerische Attribute haben Zahlenwerte (real oder integer) und die Werte der Attribute vom Typ string sind Zeichenketten. Im Unterschied zu den nominellen Attributen ist die Menge der möglichen Attributwerte nicht vordefiniert. Attribute vom Typ date repräsentieren ein Datum in Form eines Strings. Die Werte relationaler Attribute sind wiederum eine Menge von Werten verschiedener definierter Attribute. Diese Attribute entsprechen ebenfalls jeweils einem der oben aufgeführten Typen. [WITT]
Neben der Darstellung der Trainingsbeispiele in Form einer Menge von Attributwerten, können die Trainingsbeispiele auch durch Prädikatenlogik formuliert sein. Datensätze in dieser Darstellungsform werden im Rahmen dieser Arbeit nicht berücksichtigt.
Der Klassifizierer aus Abbildung 1 wurde auf einer Menge von Trainingsbeispielen trainiert, sodass dieser alle gesehenen Beispiele richtig klassifizieren kann. Um den Klassifizierer zu evaluieren, werden weitere, noch nicht "gesehenen" Trainingsdaten benötigt. Datensätze für Evaluierungszwecke werden folgend als Testdatensatz bezeichnet. Diese müssen die gleiche Struktur haben, wie die Datensätze, die in der Lernphase verwendet wurden.

1.2.2 Verwendbarkeit von Datensätzen

In Abhängigkeit des Lernalgorithmus (bzw. dessen Implementierung) müssen die verwendeten Datensätze gegebenenfalls über bestimmte Eigenschaften verfügen. So können beispielsweise einige Lernalgorithmen zur Erzeugung eines Klassifizierers nur für Datensätze verwendet werden, wenn diese gelabelt sind, d.h. wenn diese über ein ausgewiesenes Attribut verfügen (Klassenattribut), welches die Klassenzugehörigkeit eines Beispiels angibt. Mit Bezug auf die Eigenschaften der Trainingsdaten unterscheidet man verschiedene Lernformen [ML]:

- *Supervised Learning* - Das Ergebnis der zu lernenden Funktion ist für jedes Beispiel bekannt (Beispiele sind gelabelt). Das Klassifikationsproblem aus Abbildung 1 ist ein Beispeil für Supervised Learning.
- *Semi-Supervised Learning* - Es existiert nur für einen Teil der Trainingsbeispiele ein Label.
- *Unsupervised Learning* - Außer den Trainingsbeispielen liegen keine weiteren Informationen vor (keine Labels). Solche Trainingsdaten lassen sich beispielsweise für Clustering Algorithmen verwenden.
- *Reinforcement Learning* - Das Ergebnis der Zielfunktion steht nur in einem indirekten Zusammenhang zu einem einzelnen Trainingsbeispiel. Beispiel: Beim Lernen eines Brettspiels, wird dem Lerner erst nachdem alle Spielzüge ausgeführt wurden, das Ergebnis der Zielfunktion mitgeteilt (gewonnen oder verloren). Dadurch lässt sich die Richtigkeit eines einzelnen Spielzuges nicht am Ergebnis des Spiels festmachen.

Der Datentyp der Attribute eines Datensatzes nimmt ebenfalls Einfluss auf die Verwendbarkeit von Lernalgorithmen. So sind beispielsweise bestimmte Algorithmen zur Erzeugung von Entscheidungsbäumen nur bedingt auf Datensätze anwendbar (nach Aufbereitung der Daten), wenn diese numerische Attribute enthalten. Auf der anderen Seite erfordern Datensätze mit nominellen (Klassen-)Attributen eine Datenaufbereitung, um diese für Algorithmen zur Erzeugung von Funktionen für numerische Vorhersagen zu verwenden.

Des weiteren sind fehlende Werte in den Datensätzen in Abhängigkeit des eingesetzten Algorithmus entsprechend zu behandeln, und die Anzahl der Instanzen kann die Performanz von Lernalgorithmen (z.B. Lazy Learner) negativ beeinflussen, so dass deren Verwendung für Datensätze mit sehr vielen Instanzen fraglich ist.
Insgesamt ist festzustellen, dass die Einsatzfähigkeit und die Tauglichkeit eines Lernalgorithmus von den Eigenschaften des Datensatzes abhängen.

1.3 Inhalt und Motivation der Arbeit

Der Inhalt der Arbeit besteht in der Entwicklung und Implementierung einer Web-Applikation, die Datensätze des Maschinellen Lernens verwaltet und ein Web-Interface zur Verfügung stellt, das den Zugriff auf die Datensätze ermöglicht. Als Ausgangspunkt der Arbeit dient das UCI-Repository[1]. Dieses Repository verwaltet zahlreiche Datensätzen, welche nach verschiedenen Gesichtspunkten (z.B. Anzahl der Instanzen, Anzahl der Attribute, ...) ausgewählt werden können. Ein Nachteil dieser Datensatzsammlung besteht darin, dass die Datensätze in unregelmäßigen und teilweise willkürlichen Formaten vorliegen. Daraus resultiert eine aufwendige Aufbereitung der Datensätze, um diese für bestimmte Lernalgorithmen zu verwenden.
Ziel der Arbeit ist es, eine möglichst effiziente Suche nach geeigneten Datensätze zu realisieren und die Datensätze in verschiedenen gebräuchlichen Dateiformaten bereitzustellen, um eine Datenaufbereitung vor der weiteren Verwendung zu vermeiden. Des weiteren soll die Möglichkeit bestehen, neue Datensätze über ein Web-Interface in die bestehende Sammlung zu integrieren. Die Meta-Daten eines Datensatzes sollen dabei automatisch generiert und gespeichert werden, um diese in einer Suchmaske als Auswahlkriterium zur Verfügung zu stellen.

[1] Das UCI-Repository ist eine Sammlung von Datensätzen für das Maschinelle Lernen, die von dem *Center for Machine Learning and Intelligent Systems* der *Univerity of California, Irvine* im WWW zur Verfügung gestellt ist. http://mlearn.ics.uci.edu/MLRepository

2 Dateiformate für Datensätze

Es existieren verschiedene Dateiformate, um Datensätze des Maschinellen Lernens zu speichern und zu verarbeiten. In den folgenden Kapiteln werden die für die Arbeit relevanten Formate vorgestellt.

2.1 ARFF-Format

Eine Datei im ARFF-Format (Attribute-Relation File Format), welches speziell für die Weka-Software[2] entwickelt wurde, ist eine Textdatei mit der Endung *.arff*. In Abbildung 3 ist der Inhalt einer ARFF-Datei zu sehen.

```
% Beispiel für eine ARFF-Datei
@relation Beispiel
@attribute name01 {klein, mittel, groß}
@attribute name02 numeric
@attribute name03 string
@attribute name04 date dd/MM/yyyy
@attribute name05 relational
  @attribute name05a integer
  @attributename05b real
@end name05
@data
klein, 4, tisch, 20/08/2005, "2, 5.7"
groß, ?, stuhl, 13/12/2006, "8, 45.1"
mittel, 9, fenster, 07/06/2007, "32, 6.0"
```

Abbildung 3: ARFF-Datei

Eine Arff-Datei gliedert sich in zwei Teile. Der erste Teil ist der sogenannte Header und der zweite Teil beinhaltet die eigentlichen Daten. Kommentare werden durch das % Zeichen eingeleitet, welche in einer beliebigen Zeile der Datei vorkommen dürfen.
Der Header besteht aus einer Deklaration des Namens und aus Deklarationen der Attribute des Datensatzes. Die Namensdeklaration bildet die erste Zeile (unberücksichtigt der Kommentarzeilen) einer ARFF-Datei und hat die Form: *@relation <Name>*. Das Schlüsselwort *@relation* zeigt die Namensdeklaration an und wird von dem Namen des Datensatzes in Form eines Strings gefolgt. In den darauf folgenden Zeilen werden die Attribute deklariert. Eine Attributdeklaration hat die allgemeine Form: *@attribute <Name> <Attributtyp>*. Neben dem Schlüsselwort *@attribute* und dem Namen des Attributs

[2] Die Weka-Software ist eine freie Software, die eine in Java implementierte Sammlung von Algorithmen des Maschinellen Lernens enthält, welche über eine grafische Oberfläche direkt auf Datensätze angewendet werden können oder aus eigenen Java-Code heraus aufgerufen werden können. [WEKA]

beinhaltet eine Attributdeklaration eine Angabe des Attributtyps. Das ARFF-Format unterstützt alle der grundlegenden Datentypen, wie sie im Kapitel 1.2.1 beschrieben sind.
Die Angabe des Attributtyps besteht entweder in Form eines der Schlüsselworte *numeric*, *string*, *date* oder *relational*, oder aus einer Aufzählung aller möglichen Werte des Attributs bei nominellen Attributen.
Für numerische Attribute können in Abhängigkeit der Zahlenwerte alternativ zu *numeric* auch die Schlüsselworte *real* oder *integer* verwendet werden. Bei nominellen Attributen besteht die Typangabe aus einer Aufzählung aller möglichen Werte, die das Attribut annehmen kann. Die Deklaration des Attributs namens *name01* der ARFF-Datei aus Abbildung 3 zeigt eine solche Aufzählung. Bei Attributen, die ein Datum repräsentieren, kann optional nach dem Schlüsselwort *date*, das in den Instanzen zu verwendende Datumsformat in Form eines String angegeben werden. Dieser String muss dem Format entsprechen, welches von der Java-Klasse *SimpleDateFormat*[3] benutzt wird. Ist das Datumsformat nicht explizit angegeben, müssen die Werte eines Attributs vom Typ *date* in den Instanzen das Format (SimpleDateFormat) *yyyy-MM-dd'T'HH:mm:ss* haben. In Abbildung 3 ist beispielhaft ein *date* Attribut mit dem Format *dd/MM/yyyy* zu sehen.
Attribute des Typs *relational* werden wie das Attribut namens *name05* aus Abbildung 3 deklariert. Die Deklaration wird durch das Schlüsselwort *@attribute* eingeleitet und durch das Schlüsselwort *@end* abgeschlossen. Dazwischen befinden sich Attributdeklarationen, die den bereits beschriebenen Attributdeklarationen entsprechen.
Im Anschluss an die Attributdeklarationen werden die eigentlichen Daten aufgeführt, welchen das Schlüsselwort *@data* vorangestellt ist. Für jede Instanz wird jeweils eine separate Zeile verwendet. Die Attributwerte sind durch Kommata voneinander getrennt und müssen in der Reihenfolge erscheinen, wie sie deklariert sind. Die Werte relationaler Attribute sind durch doppelte Anführungszeichen einzuschließen. Fehlende Attributwerte sind in den Instanzen durch ein Fragezeichen dargestellt. Das ARFF-Format sieht keine Möglichkeit vor Klassenattribute zu benennen. [ARFF]

2.2 Sparse ARFF-Format

Das sparse ARFF-Format unterscheidet sich von dem ARFF-Format nur im Datenteil, während die Namens- und Attributdeklarationen im Header in beiden Formaten identisch sind.
Das sparse ARFF-Format verzichtet auf eine Darstellung von Daten mit dem Wert 0. Alle anderen Werte sind mit Angabe ihres Indexes (ihre Stelle innerhalb der Instanzen) durch Kommata getrennt aufgeführt, wobei der Index durch ein Leerzeichen separiert vor dem Attributwert steht. Der Index startet bei 0 und zeigt an, welchem Attribut der Wert zu zuordnen ist. Jede Instanz ist durch geschweifte Klammern eingeschlossen. In Abbildung 4 sind die Instanzen des selben Datensatzes, einmal im ARFF-Format (oben) und einmal im sparse ARFF-Format (unten) dargestellt. Zu beachten ist, dass die Darstellung der Werte eines relationalen Attributs ohne Index erfolgt[4]. Diese Attributwerte werden wie im ARFF-Format dargestellt.
Wie im Beispiel aus Abbildung 4 gezeigt, werden fehlende Werte im sparse ARFF-Format ebenfalls durch ein Fragezeichen abgebildet. [ARFF]

2.3 XRFF-Format

Das XRFF-Format (eXtensible attribute-Relation File Format) ist eine auf XML (eXtensible Markup Language) basierende Erweiterung des ARFF-Formats. Die Struktur und der Inhalt einer XRFF-Datei

[3] http://java.sun.com/j2se/1.5.0/docs/api/, Klasse: java.text.SimpleDateFormat
[4] Die Dokumentation der ARFF-Formate ist an dieser Stelle nicht eindeutig, so dass hier diejenige Darstellung gewählt wurde, die mit der Weka-Software verarbeitet werden kann.

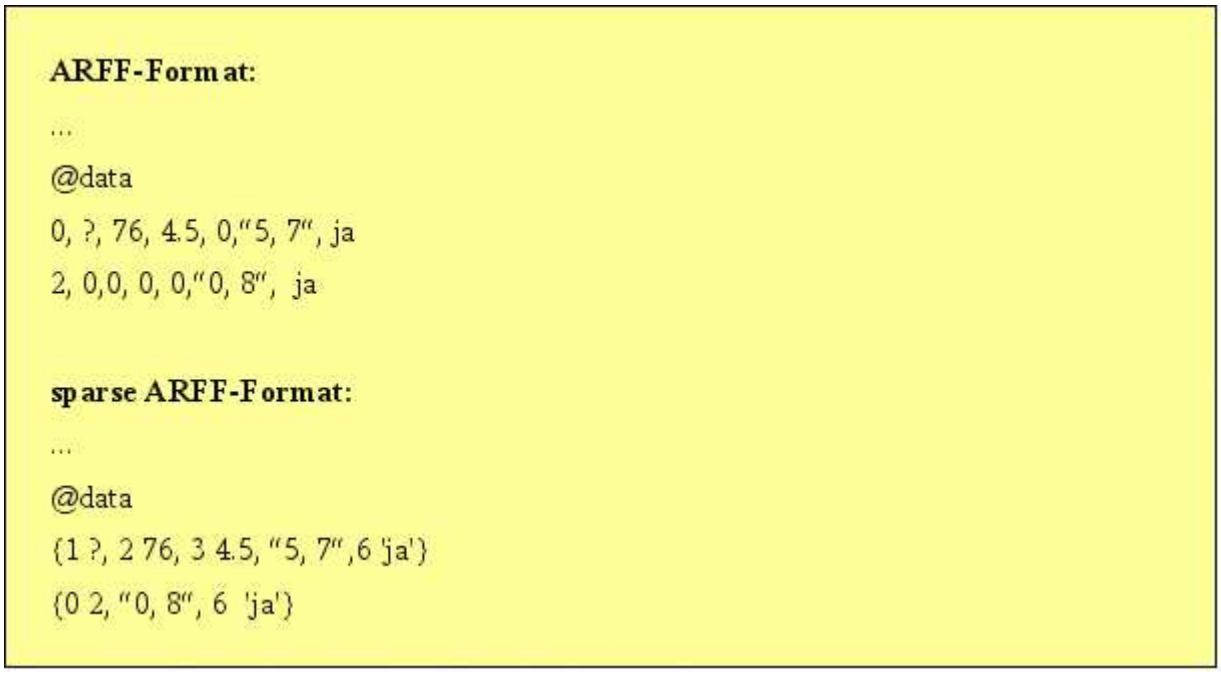

Abbildung 4: Gegenüberstellung sparse ARFF-Format und ARFF-Format

ist im wesentlichen durch die DTD (Document Type Definition) aus Abbildung 5 festgelegt.
Das Wurzelelement einer XRFF-Datei ist ein *dataset* Element, welches die beiden Kindelemente *header* und *body* besitzt. Das *header* Element enthält analog zu dem Header einer ARFF-Datei die Attributdeklarationen, und das *body* Element beinhaltet die Instanzen des Datensatzes.
Das *attributes* Element ist ein Container für alle Attributdeklarationen, während mit einem *attribute* Element ein einzelnes Attribut beschrieben werden kann. Der Typ eines Attributs des Datensatzes wird durch das *type* Attribut des *attribute* Elements benannt. Im XRFF-Format werden die selben Attributtypen unterstützt wie im ARFF-Format. In Folge dessen kann das *type* Attribut einen der nachstehenden Werte haben: *numeric*, *date*, *nominal*, *string* oder *relational*. Durch das Attribut *class* besteht die Möglichkeit, festzulegen, welches Attribut das Klassenattribut des Datensatzes ist. In der XRFF-Datei aus Abbildung 6 ist das Attribut mit dem Namen *attr04* als Klassenattribut festgelegt. Der Default-Wert des *class* Attributs ist *no*.
Bei der Deklaration von Attributen des Typs *nominal* sind die Elemente *labels* und *label* zu verwenden, um die Wertemenge nomineller Attribute zu definieren. Zur Definition des Formats eines Datums bei Attributen des Typs *date*, besitzt das *attribute* Element das optionale Attribut *format*. Dieses enthält einen String, der das Datumsformat angibt. Der String muss, wie im ARFF-Format ,die Regeln zur Formatbeschreibung der Java-Klasse *SimpleDateFormat* befolgen.
Das *attribute* Element relationaler Attribute hat als Kindelement ein *attributes* Element, das wiederum als Container der Attribute des relationalen Attributs dient (siehe *attribute* Element *attr03* in Abbildung 6).
Im Gegesatz zum ARFF-Format besteht im XRFF-Format die Möglichkeit Attribute zu gewichten. Dem Attribut namens attr01 der XRFF-Datei aus Abbildung 6 wurde durch sein *metadata* und *property* Element die Gewichtung *0,6* zu gewiesen.
Das *body* Element einer XRFF-Datei besitzt genau ein *instances* Element, das für jede Instanz des Datensatzes ein *instance* Element besitzt, welches wiederum für jeden Attributwert einer Instanz ein

```
<!DOCTYPE dataset [
  <!ELEMENT dataset (header,body)>
  <!ATTLIST dataset name CDATA #REQUIRED>
  <!ATTLIST dataset version CDATA "3.5.4">

  <!ELEMENT header (notes?,attributes)>
  <!ELEMENT body (instances)>
  <!ELEMENT notes ANY>

  <!ELEMENT attributes (attribute+)>
  <!ELEMENT attribute (labels?,metadata?,attributes?)>
  <!ATTLIST attribute name CDATA #REQUIRED>
  <!ATTLIST attribute type (numeric|date|nominal|string|relational) #REQUIRED>
  <!ATTLIST attribute format CDATA #IMPLIED>
  <!ATTLIST attribute class (yes|no) "no">
  <!ELEMENT labels (label*)>
  <!ELEMENT label ANY>
  <!ELEMENT metadata (property*)>
  <!ELEMENT property ANY>
  <!ATTLIST property name CDATA #REQUIRED>

  <!ELEMENT instances (instance*)>
  <!ELEMENT instance (value*)>
  <!ATTLIST instance type (normal|sparse) "normal">
  <!ATTLIST instance weight CDATA #IMPLIED>
  <!ELEMENT value (#PCDATA|instances)*>
  <!ATTLIST value index CDATA #IMPLIED>
  <!ATTLIST value missing (yes|no) "no">
]>
```

Abbildung 5: DTD für das XRFF-Format

value Element hat. Mit Ausnahme der *value* Elemente relationaler Attribute haben *value* Elemente keine weiteren Kindelemente, sondern beinhalten den Attributwert als Kindknoten. Die *value* Elemente der relationalen Attribute besitzen wie das *body* Element genau ein *instances* Element mit dem bereits beschriebenen Inhalt. Die Reihenfolge mit der die *value* Elemente im *instance* Element vorkommen müssen, ist durch die Reihenfolge der zugehörigen *attribute* Elemente im *header* Element vorgegeben. Das Fehlen von Werten wird durch ein *missing* Attribut mit dem Wert *yes* im *value* Element zum Ausdruck gebracht. Da der Default-Wert des *missing* Attributs per DTD mit *no* definiert ist, muss das *missing* Attribut nur bei fehlenden Werten explizit angegeben werden, wie in Abbildung 6 zu sehen ist.
Der Datensatz aus Abbildung 6 besitzt nur eine Instanz. Entsprechend der *attribute* Elemente hat das *instance* Element vier *value* Elemente als Kinder. Neben den Attributen können im XRFF-Format auch einzelnen Instanzen eine Gewichtung zugeteilt werden. Dies geschieht über das *weight* Attribut eines *instance* Elements, wie in der Instanz des Datensatzes aus Abbildung 6 beispielhaft zu sehen ist. [XRFF]

```
<dataset name="example" version="3.5.3">
  <header>
    <attributes>
      <attribute name="attr01" type="numeric">
        <metadata>
          <property name="weight">0.6</property>
        </metadata>
      </attribute>
      <attribute name="attr02" type="date" format="dd-MM-yyyy"/>
      <attribute name ="attr03" type="relational">
        <attributes>
          <attribute name="rel01" type="numeric">
          <attribute name="rel02" type="numeric">
        </attributes>
      </attribute>
      <attribute name="attr04" type="nominal" class="yes">
        <labels>
          <label>yes</label>
          <label>no</label>
        </labels>
      </attribute>
    </attributes>
  </header>
  <body>
    <instances>
      <instance weight="0.9">
        <value missing="yes"/>
        <value>23-04-2000</value>
        <value>
          <instances>
            <instance>
              <value>12</value>
              <value>2400</value>
            </instance>
          </instances>
        </value>
        <value>yes</value>
      </instance>
    </instances>
  </body>
</dataset>
```

Abbildung 6: Beispiel für eine XRFF-Datei

2.4 Sparse XRFF-Format

Für das XRFF-Format ist ebenfalls eine sparse Darstellung der Daten eines Datensatzes vorgesehen. Da sowohl das sparse XRFF-Format, als auch das XRFF-Format der gleichen DTD zugrunde liegen,

ist die Struktur und der Inhalt der beiden Formate im wesentlichen gleich. Das sparse XRFF-Format unterscheidet sich sich vom XRFF-Format in den folgenden drei Punkten:

- Die *instance* Elemente müssen ein *type* Attribut besitzen mit dem Wert *sparse*. Der Default-Wert für dieses Attribut ist in der entsprechenden DTD mit *normal* definiert.
- Jedes vorhandene *value* Element muss ein *index* Attribut haben, welches die Stelle des Wertes innerhalb der Instanz angibt. Die Stellen einer Instanz, also die Werte des *index* Attributs, beginnen mit dem Wert 1.
- Ist der Attributwert in einer Instanz 0, so wird das entsprechende *value* Element weggelassen.

In Abbildung 7 ist beispielhaft ein Ausschnitt einer sparse XRFF-Datei abgebildet. Hier ist eine Instanz zu sehen, die für die Attribute an den Stellen 2, 3, 6 und 7 den Wert 0 hat. Da aus diesem Auschnitt nicht ersichtlich ist, wie viele Attribute der Datensatz hat, ist nicht auszuschließen, dass weitere Attribute existieren, die in dieser Instanz den Wert 0 haben. [XRFF]

```
...
<body>
  <instances>
    <instance type="sparse">
      <value index="1">23</value>
      <value index="4">1234 </value>
      <value index="5" missing="yes"/>
      <value index="8">2400</value>
    </instance>
  </instances>
</body>
...
```

Abbildung 7: Ausschnitt aus einer sparse XRFF-Datei

2.5 CSV-Format

Das CSV-Format (Comma Separated Value) ist ein Format für Textdateien mit einfach strukturierten Daten. Es ist nicht speziell für Anwendungen des Maschinellen Lernens entwickelt worden. Die Beschreibung des Formats in diesem Kapitel liegt der entsprechenden RFC-Spezifikation[5] zugrunde.[CSV]
Eine CSV-Datei setzt sich aus zwei Teilen zusammen. Der eine Teil besteht aus einer Auflistung der Namen aller Attribute des Datensatzes und der andere Teil enthält die Instanzen des Datensatzes. Ein Beispiel für einen Datensatz im CSV-Format zeigt die Abbildung 8.
In der ersten Zeile einer CSV-Datei sind die Namen der Attribute durch Kommata getrennt aufgeführt. Damit ist die Reihenfolge festgelegt, mit der die Attributwerte in den Instanzen erscheinen müssen. Anschließend sind die Instanzen aufgeführt, wobei jede Instanz, wie im ARFF-Format, in einer separaten Zeile steht. Die Attributwerte sind ebenfalls durch Kommata getrennt. Die Anzahl der Attributwerte muss in jeder Zeile der Anzahl der Attributnamen entsprechen.
Eine explizite Angabe des Datentyps eines Attributs ist in diesem Format nicht möglich. Die Weka-Software unterscheidet jedoch die Typen nominal und numeric in Abhängigkeit der in den Instanzen

[5] RFC (Request for Comments) ist die Bezeichnung für eine Reihe von technischen und organisatorischen Dokumenten bezüglich des Internets. [RFC]

```
name01, name02, name03, name04

12, 234, 453, auto
543, 786, 798, bus
123, 4, 443, fahrrad
```

Abbildung 8: Datensatz im CSV-Format

vorkommenden Werten. Die Attribute *namen01*, *namen02* und *namen03* der CSV-Datei aus Abbildung 8 würden als numerisch interpretiert werden, während dem Attribut *name04* der Typ nominal zugewiesen werden würde. Das Zeichen ? in einer Instanz wird von der Weka-Software als fehlender Wert interpretiert.

2.6 C4.5-Format

Im Rahmen der C4.5-Software, eine Implementierung des C4.5-Algorithmus, wurde das C4.5-Dateiformat definiert. Sämtliche der hier gemachten Angaben beziehen sich auf die aktuellste Version (Release 8) dieser Software. [C45]
Die Darstellung und Beschreibung eines Datensatzes im C4.5-Format verteilt sich auf mehrere Dateien. Eine Datei mit der Endung *.names* enthält die Definition der Attribute des Datensatzes und eine andere Datei, mit der Endung *.data,* beinhaltet den eigentlichen Datensatz. Zum Speichern eines eventuell vorhandenen Testdatensatzes sieht das C4.5-Format eine weitere Datei vor, die mit *.test* endet.
Die **.names* Datei eines Datensatzes enthält eine Reihe von Einträgen zur Definition der Attribute, der Klassen und der Werte des Datensatzes. Der erste Eintrag dieser Datei (Kommentare ausgenommen) besteht aus einer Aufzählung der möglichen Werte des Klassenattributs. Die einzelnen Werte sind durch Kommata getrennt, wie in Abbildung 9 zu sehen ist. In diesem Beispiel gibt es die zwei Klassen *yes* und *no*. In diesem Format können nur Datensätze abgebildet werden, die ein nominelles Klassenattribut haben.

```
yes, no.
temperature: continuous.
humidity: high, normal.
outlook: discrete 4.
id: ignore.
```

Abbildung 9: *.names Datei eines Datensatzes im C4.5-Format

Die folgenden Einträge definieren die Attribute des Datensatzes in der Reihenfolge, in der deren Werte in den Instanzen aufgeführt werden müssen. Eine Attributdefinition hat die Form:
<Attributname> : <Attributbeschreibung>
Der Name des Attributs in Form eines Strings wird von einem Doppelpunkt gefolgt. Die anschließende Attributbeschreibung, bzw. die Angabe des Datentyps eines Attributs, hat eine der aufgeführten Formen:

- *continuous* - für numerische Attribute
- *eine Liste aller möglichen Attributwerte* - für nominelle Attribute

- *discrete n* - für nominelle Attribute, wobei *n* die Anzahl der verschiedenen möglichen Attributwerte ist
- *ignore* - für Attribute, die vom Lernalgorithmus nicht berücksichtigt werden sollen

In Abbildung 9 ist für jeden Attributtyp ein Beispiel gegeben.
Sämtliche Einträge werden durch einen Punkt abgeschlossen. Dieser kann weggelassen werden, wenn nach dem Punkt in der gleichen Zeile keine weiteren Zeichen vorkommen.
Die Instanzen des Datensatzes sind in der **.data* Datei abgelegt und haben die gleiche Form wie die Instanzen im ARFF-Format, mit der Ausnahme, dass der letzte Wert einer Instanz (also der letzte Wert einer Zeile) dem Klassenattribut zu zuordnen ist. Alle anderen Attributwerte werden in der selben Reihenfolge aufgeführt, wie die entsprechenden Attribute in der **.names* Datei definiert sind.
Eine **.test* Datei hat den gleichen Aufbau wie eine **.data* Datei, jedoch werden in solchen Dateien Testdatensätze gespeichert. Fehlende Werte werden sowohl in einer **.data*, als auch in einer **.test* Datei durch ein Fragezeichen dargestellt.
Kommentare sind in allen Dateien einleitend durch ein | Zeichen zu kennzeichnen.

2.7 C5.0-Format

Die C5.0-Software ist ein kommerzieller Nachfolger der C4.5-Software und definiert eine Reihe von Applikationsdateien, von denen hier nur diejenigen behandelt werden, die zur Darstellung der Datensätze verwendet werden und im Rahmen dieser Arbeit von Bedeutung sind. [C50]
Die Darstellung von Datensätzen im C5.0-Format baut auf dem C4.5-Format auf. Es werden ebenfalls die drei Dateien **.names*, **.data* und **.test* unterschieden, die analog zum C4.5-Format die gleiche Bedeutung haben, jedoch weitere Beschreibungsmöglichkeiten für die Datensätze bieten.
Der erste Eintrag einer **.names* Datei im C5.0-Format dient ebenfalls der Beschreibung des Klassenattributs, wobei dieser im Gegensatz zum C4.5-Format verschiedene Formen haben kann:

- *Aufzählung der Werte des Klassenattributs* - wie C4.5-Format
- *<Attributname>* - der String Attributname bezeichnet das nominelle Klassenattribut
- *<Attributename> : <Liste von Schwellenwerten>* - der String Attributname bezeichnet das numerische Klassenattribut und nach dem Doppelpunkt folgt eine Liste von Schwellenwerten; z.B. *age: 10, 20* –> damit sind drei Klassen definiert: age<=10, 10<=age<=20 und age>20

Neben Datensätzen mit nominellen Klassenattributen können in diesem Format auch Datensätze mit numerischen Klassenattributen dargestellt werden, sofern Schwellenwerte vorliegen, die eine endliche Menge von Klassen erzeugen.
Danach kommen die Einträge zur Definition der Attribute in der allgemeinen Form:
<Attributname> : <Attributbeschreibung>
Nachstehende Möglichkeiten der Attributbeschreibung, bzw. der Definition des Datentyps eines Attributs, sind erlaubt:

- *continuous* - für numerische Attribute
- *date* - Attribute vom Typ *date* repräsentieren ein Datum der Form YYYY/MM/DD oder YYYY-MM-DD, wobei YYYY eine vierstellige Jahreszahl ist, MM bezeichnet einen Monat im Jahr und DD steht für einen Tag im Monat

- *time* - Attribute des Typs *time* repräsentieren eine Uhrzeit der Form HH:MM:SS mit Werten von 00:00:00: bis 23:59:59
- *timestamp* - der Datentyp *timestamp* ist eine Kombination der Typen *date* und *time,* und hat die Form 'YYYY/MM/DD HH:MM:SS' oder 'YYYY-MM-DD HH:MM:SS'
- *eine Liste aller möglichen Attributwerte* - für nominelle Attribute; falls die Attributwerte hierarchisch geordnet sind, kann dies durch das Schlüsselwort *[ordered]* angezeigt werden (siehe Abbildung 10)
- *discrete n* - für nominelle Attribute, wobei *n* die Anzahl der verschiedenen möglichen Attributwerte ist
- *ignore* - für Attribute, die vom Lernalgorithmus nicht berücksichtigt werden sollen
- *label* - Attribute vom Typ *label* können verwendet werden, um Instanzen einen Identifikator hinzuzufügen; sie spielen für den Lernalgorithmus jedoch keine Rolle

Des weiteren besteht die Möglichkeit Attribute durch eine Formel zu definieren. Solche Definitionen haben die Form:
<Attributname> := <Formel>
Die Formel wird, nach den in der Mathematik üblichen Regeln, aufgestellt. Diese werden hier jedoch nicht weiter erläutert. Ein einfaches Beispiel für ein Attribut, das durch eine Formel definiert ist, ist in Abbildung 10 (*attribute07*) zu sehen.
Wie im C4.5-Format sind sämtliche Einträge der **.names* Datei mit einem Punkt abzuschließen. Die Dateien **.data* und **.test* zum Speichern der Instanzen eines Datensatzes erfolgt analog zum C4.5-Format, wie im vorherigen Kapitel beschrieben. Die Werte eines Attributs, welches durch eine Formel definiert ist, kommen nicht in den Instanzen vor. Abbildung 10 zeigt den Inhalt einer **.names* Datei im C5.0-Format und beispielhafte Instanzen einer zugehörigen **.data* Datei.

***.names:**

```
class1, class2, class3.
attribute01: continuous.
attribute02: date.
attribute03: time.
attribute04: Peter, Hans, Paul.
attribute05: discrete 2.
attribute06: label.
attribute07:= attribute01 + 2.
```

***.data**

```
4, 2007/12/03, 10:38:00, Paul, Tisch, 100, class1
56, 2006/02/13, 00:34:50, Peter, ?, 101, class2
...
```

Abbildung 10: *.names Datei mit Beispiel-Instanzen im C5.0-Format

3 Anforderung an die Anwendung

Die wesentlichen Bestandteile der Anwendung zur Verwaltung von Datensätzen des Maschinellen Lernens lassen sich in die zwei Hauptkomponenten Datenspeicherung und lesender und schreibender Zugriff auf die Datensätze, unterteilen. In den folgenden Kapiteln werden die Anforderungen an diese Komponenten beschrieben.

3.1 Datenspeicherung

Neben den eigentlichen Datensätzen müssen Meta-Daten über die Datensätze persistent gespeichert werden. Diese Meta-Daten repräsentieren verschiedene Eigenschaften der Datensätze, die verwendet werden, um den Zugriff auf die Datensätze zu organisieren.

3.1.1 Speicherung der Datensätze

Um die Datensätze in den verschiedenen Formaten, wie sie in Kapitel 2 beschrieben sind, zur Verfügung zu stellen, müssen die Datensätze so abgespeichert werden, dass eine Konvertierung in die beschriebenen Formate möglich ist. Dazu sind sämtliche Informationen, die in den einzelnen Formaten enthalten sind, vorzuhalten. Im einzelnen sind das:

1. Name des Datensatzes
2. Namen der Attribute
3. Datentypen der Attribute
4. die Wertemenge nominaler Attribute
5. die Anzahl der verschiedenen Werte der Attribute des Typs string und date
6. das Format mit dem Attribute vom Typ date dargestellt werden
7. die Information, welches Attribut das Klassenattribut ist
8. diskrete Werte numerischer Klassenattribute
9. Gewichtung der Attribute
10. Attributwerte in den Instanzen
11. Gewichtung der Instanzen
12. Referenz der Werte in den Instanzen zu den zugehörigen Attributen, bzw. die Reihenfolge mit der die Attributwerte in den Instanzen auftreten
13. Erläuterungen bzw. Kommentare zu den Datensätzen

Neben den Namen der Attribute, ist in den verschiedenen Dateiformaten deren Datentyp anzugeben. Die Attributtypen, die durch das Speicherformat unterstützt werden müssen, um eine Konvertierung in die genannten Dateiformaten zu ermöglichen, sind in Abbildung 11 dargestellt.
In der Abbildung sind die Attributtypen der verschiedenen Dateiformate zu einer Gruppe zusammengefasst, für die es eine Entsprechung in den anderen Dateiformaten gibt. Eine Gruppe ist durch einen umrandeten Zeilenblock dargestellt. Es ergeben sich sieben Gruppen, wobei die ersten fünf Gruppen den grundlegenden Attributtypen aus Kapitel 1.2.1 entsprechen. Die Attribute vom Typ *label* und *Formel* im

ARFF	XRFF	C4.5	C5.0	CSV
numeric real integer	numeric	continuous	continuous	-
nominal	nominal	nominal (discrete n)	nominal (discrete n)	-
string	string	(discrete n)	(discrete n)	-
date	date	(discrete n)	date time timestamp	-
relational	relational	-	-	-
-	-	-	label	-
-	-	-	Formel	-

Abbildung 11: Gegenüberstellung der Datentypen

C5.0-Format bilden jeweils eine weitere Gruppe, die in den anderen Formaten jedoch keine Rolle spielen. Attribute vom Typ *relational* hingegen, können in anderen Dateiformaten als ARFF und XRFF unbeachtet bleiben. Die Abbildung von numerischen und nominellen Attributen erfolgt in den verschiedenen Dateiformaten auf analoge Weise, durch Angabe eines Schlüsselwortes (*numeric* oder *continuous*) oder durch Angabe der Wertemenge nomineller Attribute. In den Formaten C4.5 und C5.0 können nominelle Attribute alternativ mit *discrete* n deklariert werden, wobei n die Anzahl der möglichen Werte ist, die ein nominelles Attribut annehmen kann. Während Attribute vom Typ *string* in den Formaten ARFF und XRFF durch das Schlüsselwort *string* definiert werden, sind Attribute diesen Typs in den anderen Formaten nicht vorgesehen. In den Formaten C4.5 und C5.0 können solche Attribute durch *discrete n* als nominelle Attribute abgebildet werden. Dazu muss die Anzahl der Werte in den Instanzen solcher Attribute zusätzlich gespeichert werden. Um ein zeitliches Datum in den verschiedenen Dateiformaten abbilden zu können, muss neben dem Typ jener Attribute, auch das Datumsformat, in welchem entsprechende Attributwerte in den Instanzen vorkommen, gespeichert werden. Aus dem Datumsformat muss abgeleitet werden können, welche Zeiteinheiten innerhalb eines Datums verwendet wurden. Für das Datum *05.03.03* beispielsweise, lässt sich ohne weitere Angaben nicht herleiten, ob es sich bei den Zahlen um Monate, Jahre oder andere Zeiteinheiten handelt. Um ein solches Datum durch die dafür vorgesehenen Attributtypen der Dateiformate darzustellen, muss die Information gespeichert werden, welche Zeiteinheiten bei der Darstellung eines Datums verwendet werden. Im C4.5-Format kann ein Datum wiederum nur als nominelles Attribut durch *discrete n* abgebildet werden, wodurch hierbei ebenfalls die Notwendigkeit entsteht, die Anzahl aller Werte in den Instanzen eines solchen Attributs zu speichern. Da im CSV-Format auf eine Unterscheidung verschiedener Attributtypen komplett verzichtet wird, sind zur Abbildung eines Datensatzes in diesem Format keine zusätzlichen Informationen bezüglich der Attributtypen notwendig. Während zur Darstellung eines Datensatzes im XRFF-Format die Information genügt, welches Attribut das Klassenattribut ist, können in den Formaten C4.5 und C5.0 Datensätze nur dargestellt werden, wenn

ein Klassenattribut vom Typ nominal existiert, oder wenn der Zahlenraum numerischer Attribute auf eine endliche Menge von Intervallen abgebildet wird. Um Datensätze mit numerischen Klassenattributen in den Formaten C4.5 und C5.0 darzustellen, müssen also zu diesen Klassenattributen diskrete Werte abgespeichert werden, die den entsprechenden Zahlenraum in geeigneter Weise repräsentieren.
Zur Unterstützung des XRFF-Formats muss die Möglichkeit bestehen, die Gewichtung von Attributen und Instanzen zu speichern. Des weiteren muss durch das Speicherformat gewährleistet sein, dass eine eindeutige Zuordnung der Werte in den Instanzen zu den entsprechenden Attributen durchgeführt werden kann. Dokumentationen und zusätzliche Informationen über die Datensätze sollen ebenfalls durch das Speicherformat aufgenommen werden können, um sie als Kommentare in den verschiedenen Dateiformaten abzubilden.
Außerdem soll das Speicherformat als flexibles Datenaustauschformat verwendet werden können, welches sich hinsichtlich der Konvertierung in weitere Formate leicht erweitern lässt.

3.1.2 Speicherung der Meta-Daten

Die Anwendung soll ein User-Interface bereitstellen mit dem eine Suchanfrage nach geeigneten Datensätzen durchgeführt werden kann. Die Kriterien, die zur Suche nach Datensätzen verwendet werden, bilden bestimmte Eigenschaften der Datensätze ab. Es handelt sich dabei um Meta-Daten der Datensätze, die sich im wesentlichen auf die Verwendbarkeit der Datensätze bezüglich bestimmter Lernalgorithmen beziehen. Die Meta-Daten müssen daher persistent für jeden Datensatz gespeichert werden. Das Format mit dem die Meta-Daten gespeichert werden, muss den Zugriff auf die Meta-Daten in Form einer Suchanfrage zulassen und es muss das Einfügen von Meta-Daten weiterer Datensätze unterstützen. Zusätzlich soll eine zukünftige Erweiterung der Meta-Daten um weitere Eigenschaften möglich sein. Folgende Meta-Daten werden gespeichert:

- Name des Datensatzes
- kurze Beschreibung des Datensatzes
- Name der Datei, die den Datensatz enthält
- Anzahl aller Attribute
- Anzahl der numerischen Attribute
- Anzahl der nominalen Attribute
- Anzahl der boolschen Attribute
- Datentyp des Klassenattributs
- Anzahl der verschiedenen Werte bei nominalen Klassenattributen
- Häufigkeit der größten Klasse (Prozentsatz bezüglich aller Instanzen) bei nominalen Klassenattributen
- Anzahl der Instanzen
- Prozentsatz der fehlenden Attributwerte in den Instanzen
- Anzahl der verfügbaren Testdatensätze
- Dateinamen der Testdatensätze

Die Dateinamen der Datensätze und der Testdatensätze stellen dahingehend eine Ausnahme dar, als dass diese nicht als Suchkriterien verwendet werden, sondern für die Abwicklung eines Datensatz-Downloads benötigt werden.

3.2 Zugriff auf die Datensätze

Neben dem Suchen und Herunterladen von Datensätzen, soll die Anwendung das Integrieren von neuen Datensätzen in die bestehende Datensatzsammlung unterstützen. Es ist also zwischen einem lesenden und schreibenden Zugriff auf die Daten zu unterscheiden. Die Zugriffe sollen dabei on-line mit einem üblichen Browser erfolgen. Daraus resultiert die Notwendigkeit, die Anwendung als Web-Anwendung zu realisieren, deren Dienste über einen Uniform Resource Locator[6] (URL) abgerufen werden können.

3.2.1 Client-Server-Szenario

Im klassischen Client-Server-Modell, wie es in Abbildung 12 dargestellt ist, ist der Browser mit dem der Anwender eine Anfrage stellt der Client und die im Rahmen dieser Arbeit zu entwickelnde (Web-)Anwendung der Server.

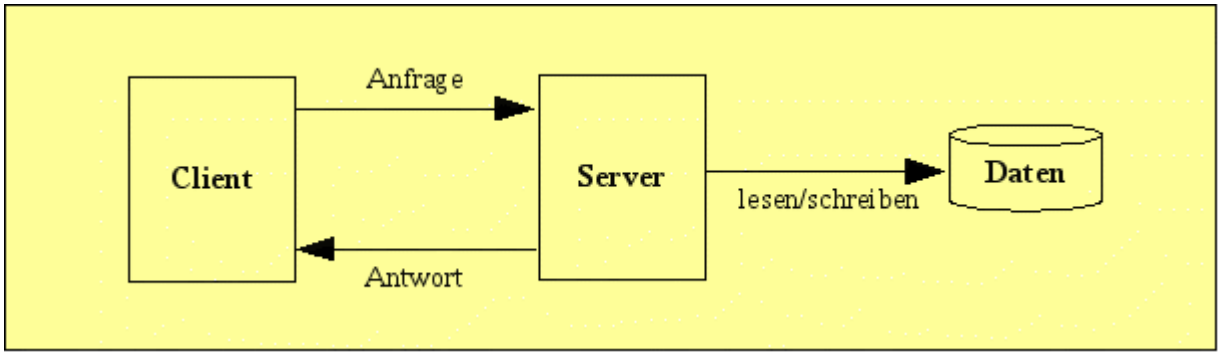

Abbildung 12: Klassisches Client-Server-Model

Der Server, also die Web-Anwendung, muss in diesem Szenario mehrere Anforderungen erfüllen, um die gewünschten Dienste (Bereitstellen und Integrieren von Datensätzen) anzubieten.
Die Anwendung muss Anfragen des Clients über das HTTP-Protokoll[7] (Hypertext Transfer Protocol) entgegen nehmen. Diese Anfrage können Parameter enthalten, die serverseitig validiert und ausgewertet werden müssen, da sie den Programmablauf beeinflussen. Die vom Benutzer angegebenen Eigenschaften der gesuchten Datensätze sind beispielsweise solche Parameter.
Des weiteren muss die Anwendung den lesenden und den schreibenden Zugriff auf die Datensätze durchführen und eine Antwort über das HTTP-Protokoll an den Client senden. Da in manchen Fällen die Antwort an den Client in Abhängigkeit der Benutzereingaben steht und somit keinen statischen Inhalt hat, muss die Anwendung in der Lage sein, dynamische Antworten zu generieren, die ein für den Client

[6] Ein Uniform Resource Locator (URL) ist eine Untermenge der Uniform Resource Identifier (URI). Ein URI ist eine Sequenz von Zeichen, die eine abstrakte oder physische Ressource identifiziert. Ein URL identifiziert eine Ressource, indem er den Zugriffmechanismus auf die Ressource beschreibt, d.h. der Ort der Ressource im Netzwerk. Die allgemeinen Form eines URL ist: protokoll://servername/pfad/resourcenbezeichner. [BONG][URI]

[7] Das HTTP ist ein einfaches zustandloses Protokoll zur Übertragung von Daten über ein Netzwerk. Es kommt hauptsächlich zum Einsatz, um Daten des World Wide Webs (WWW) in eine Web-Browser zu laden. Die Kommunikation zwischen beiden Parteien wird durch den Client (z.B. Web-Browser) eingeleitet, indem dieser durch eine Anfrage (Request) eine Ressource anfordert. Die Antwort (Response) des Servers auf die Anfrage besteht im Zusenden der angeforderten Ressouce an den Client, womit die Kommunikation durch dieses Protokoll abgeschlossen ist. [BERG]

verarbeitbares Format hat (z.B.: HTML-Seite, die der Browser anzeigen kann). Die Antwort an den Client kann aber auch eine Datei sein, die den vom Benutzer ausgewählten Datensatz enthält.

3.2.2 User-Interface

Um eine Kommunikation des Anwenders mit der Anwendung zu ermöglichen, werden Benutzeroberflächen benötigt. Diese müssen durch den clientseitigen Browser angezeigt werden können, damit der Anwender in der Lage ist, das Ein- und Auschecken von Datensätzen zu steuern.
Zum Auffinden von geeigneten Datensätzen ist eine Suchmaske erforderlich, die es dem Anwender ermöglicht, verschiedene Kriterien anzugeben, die die gesuchten Datensätze erfüllen sollen. Die Kriterien entsprechen den Meta-Daten der Datensätze, wie sie inKapitel 3.1.2 aufgelistet sind. Die über die Suchmaske ausgewählten Suchkriterien müssen anschliessend an den Server übermittelt werden, wo eine Suchanfrage generiert und an die abgespeicherten Meta-Daten gestellt wird. Das Ergebnis einer solchen Suchanfrage ist eine Menge von Datensätzen, die die Suchkriterien erfüllen. Die gefundenen Datensätze sollen dann auf einer weiteren Oberfläche angezeigt werden, auf der der Anwender die Datensätze auswählen kann, die heruntergeladen werden sollen.
Das Einchecken von neuen Datensätzen soll ebenfalls durch ein User-Interface unterstützt werden. Neben der Möglichkeit eines Datei-Uploads, muss diese Benutzeroberfläche dem Anwender die Möglichkeit geben, Angaben zum Datensatz zu machen, die nicht durch die Anwendung selbst ermittelt werden können. Welche Angabe das im einzelnen sind, hängt vom Dateiformat ab, welches zum Einchecken verwendet werden kann.

3.2.3 Lesen und Schreiben der Datensätze

Das Lesen von Datensätzen, bzw. der gesamte Vorgang beim Auschecken von Datensätzen, erfordert eine Reihe von Aktionen, die in Abbildung 13 dargestellt sind. Wie im vorangegangenen Kapitel beschrieben, nimmt der Anwender durch Benutzeroberflächen Einfluss auf die Aktionen.
In der Abbildung sind auf der linken Seite, die zum Auslesen von Datensätzen notwendigen Benutzeroberflächen, durch Ellipsen abgebildet. Die rechteckigen Textfelder in der Mitte stellen die erforderlichen Aktionen dar, die sich zu fünf Komponenten zusammenfassen lassen. Die Abfolge mit der die Aktionen auszuführen sind, ist durch die Pfeile mit durchgehender Linie symbolisiert. Die Pfeile mit gestrichelter Linie zeigen auf der einen Seite den Informationsfluss zwischen dem Client und dem Server an, und auf der anderen Seite stellen sie den Zugriff auf die persistenten Daten (Meta-Daten und Datensätze) der Anwendung dar.
Nachdem der Anwender die Such-Parameter in der Suchmaske angegeben und abgesendet hat, müssen die Such-Parameter serverseitig validiert werden. Sollten diese fehlerhaft sein, sodass keine Suche durchgeführt werden kann, muss der Anwender zur erneuten Eingabe der Suchkriterien aufgefordert werden. Sollten die Parameter keine Fehler aufweisen, kann eine Suche nach entsprechenden Datensätzen durchgeführt werden. Dazu sind die Such-Parameter mit den gespeicherten Meta-Daten zu vergleichen. Danach ist das Suchergebnis, also alle in Frage kommenden Datensätze, dem Anwender anzuzeigen, sodass dieser aus allen Treffern, diejenigen Datensätze auswählen kann, die zum Download bereitgestellt werden sollen. Außerdem muss das Dateiformat angegeben werden, in das die Datensätze überführt werden sollen. Mit Hilfe dieser Angaben kann die Transformation der Datensätze erfolgen, damit diese abschließend zum Download zur Verfügung gestellt werden können.

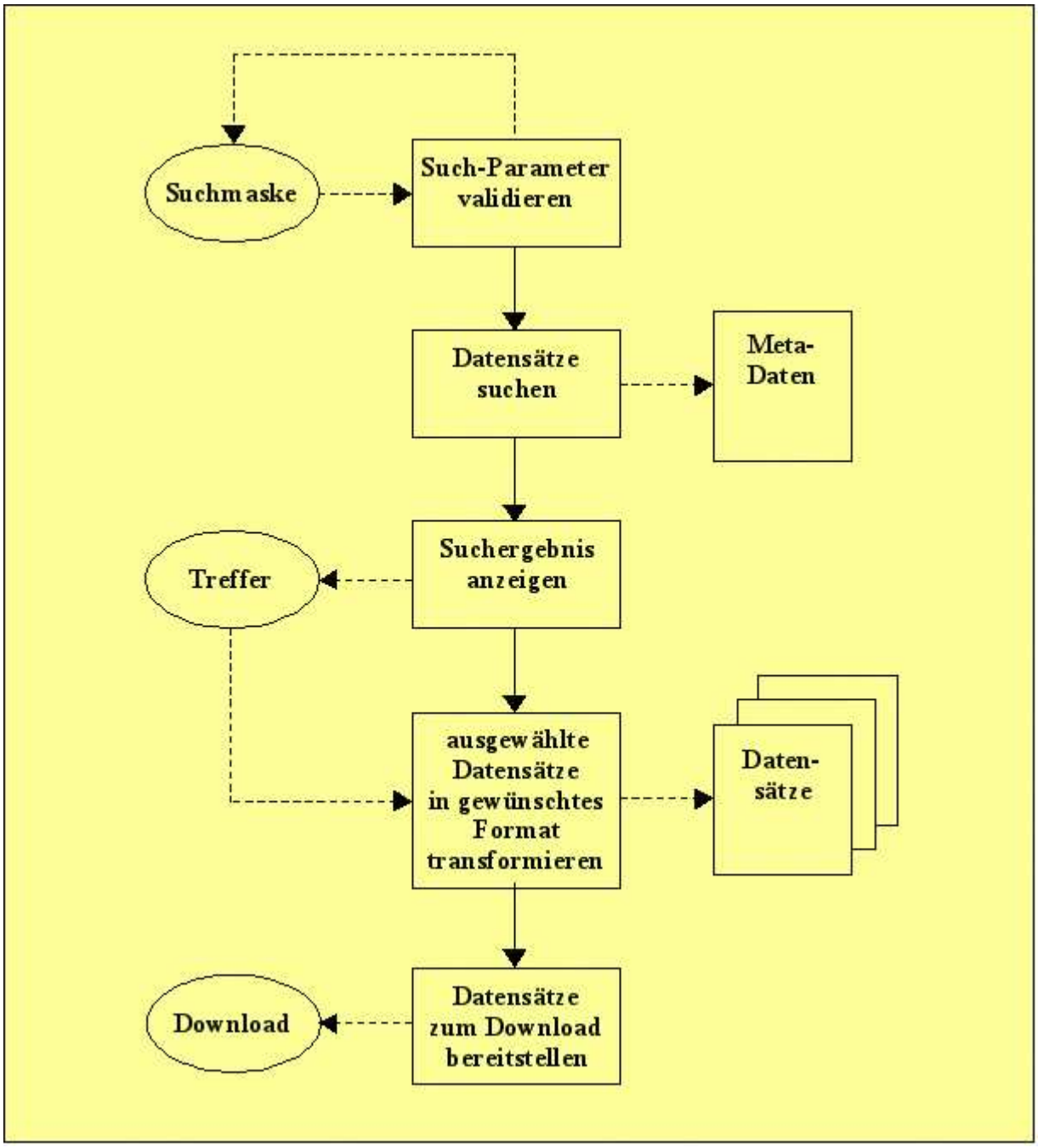

Abbildung 13: Auschecken von Datensätzen

Das Schreiben von Datensätzen (Einchecken von Datensätzen) macht ebenfalls eine Reihe von Aktionen notwendig, die durch eine Interaktion mit dem Anwender eingeleitet werden. Das Einchecken von Datensätzen ist in Abbildung 14 gezeigt, wobei die gleiche Symbolik verwendet wird wie in Abbildung 13.

Als Erstes muss eine serverseitige Validierung der übermittelten Parameter und Datensatzdatei durchgeführt werden. Der Datei-Upload und das Übermitteln der Parameter an den Server werden durch den Anwender über eine Benutzeroberfläche initiiert. Bei der Validierung ist zu prüfen, ob die Upload-Datei, die den Datensatz enthält, die Spezifikation des Dateiformats, welches zum Einchecken von Datensätzen unterstützt wird, einhält. Das ist erforderlich, um sicherzustellen, dass der Datensatz korrekt weiter-

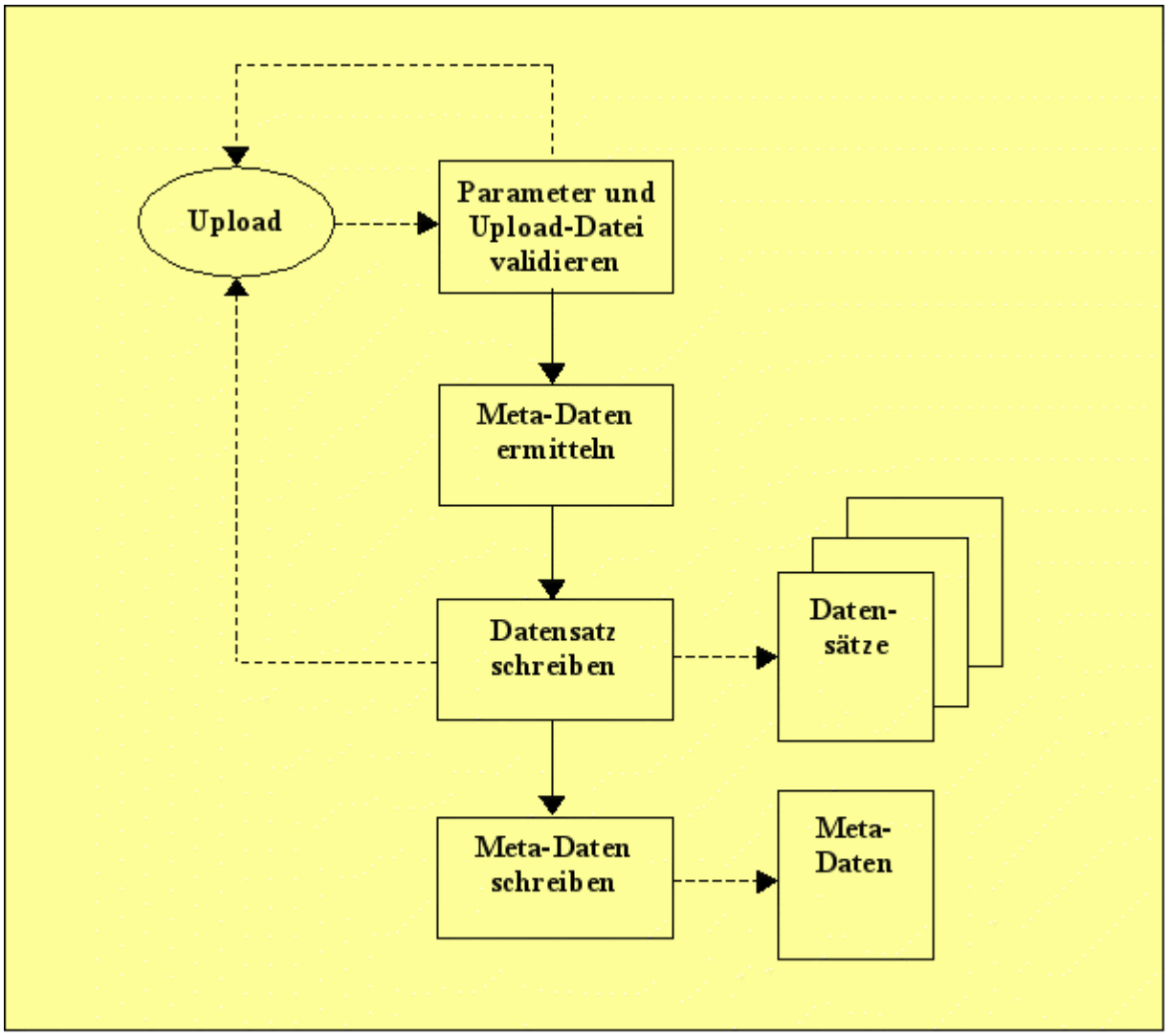

Abbildung 14: Einchecken von Datensätzen

verarbeitet werden kann. Weitere Informationen (Parameter), die für die Verarbeitung des Datensatzes zusätzlich erforderlich sind und durch den Anwender separat angegeben werden müssen, sind ebenfalls auf ihre Richtigkeit hin zu untersuchen.
Werden durch den Validierungsprozess Fehler in der Upload-Datei oder in den Parametern erkannt, muss eine erneute Übermittlung der Daten durch den Anwender erfolgen. Im anderen Fall können die Meta-Daten des einzucheckenden Datensatzes, wie sie in Kapitel 3.1.2 beschrieben sind, ermittelt werden. Dazu ist eine Analyse der Upload-Datei nötig, um die entsprechenden Informationen über den Datensatz zu ermitteln.
Anschließend muss der Datensatz peristent gespeichert werden, um zukünftige Zugriffe auf den Selbigen zu gewährleisten. Da die Datensätze in einem bestimmten Format (siehe Kapitel 3.1.1) abgespeichert werden sollen, muss der Datensatz vor der Speicherung in das definierte Speicherformat überführt werden. Das Schreiben von Testdatensätzen macht eine zusätzliche Kompatibilitätsprüfung notwendig, weil Testdatensätze die gleiche Struktur wie dem zugrunde liegenden Datensatz haben müssen. Ein bereits vorhandener Datensatz ist also mit dem einzucheckenden Testdatensatz hinsichtlich ihrer Struktur zu vergleichen. Sollten die beiden Datensätze nicht kompatibel zueinander sein (z.B. andere Attributtypen),

kann der Testdatensatz nicht eingecheckt werden, sodass ein wiederholter Datei-Upload eines korrigierten Testdatensatzes nötig ist.
Nach dem erfolgreichen Speichern des (Test-)Datensatzes, müssen im Anschluß daran auch die Meta-Daten zum neu abgespeicherten Datensatz gespeichert werden, damit dieser, durch zukünftige Suchanfragen ausfindig gemacht werden kann.

3.2.4 Transformation der Datensätze

Da die Anwendung dem Benutzer die Möglichkeit bieten soll, die ausgewählten Datensätze in verschiedenen Dateiformaten (siehe Kapitel 2) auszuchecken, wird ein Mechanismus benötigt, der die gespeicherten Datensätze in das gewünschte Format transformiert. Diese Transformationskomponente muss also in der Lage sein, einen Datensatz in Form des anwendungsspezifischen Speicherformates als Input entgegenzunehmen, und diesen auf ein anderes, durch den Anwender ausgewähltes Dateiformat, abzubilden. Der Output der Komponente ist dann beispielsweise eine ARFF-Datei, die der Anwender herunterladen kann. Um die Anwendung, um die Fähigkeit der Unterstützung zusätzlicher Dateiformate erweitern zu können, muss die Transfomationskomponente so realisiert werden, dass die Abbildung auf andere, noch nicht unterstützte Dateiformate, leicht ergänzt werden kann.

4 Eingesetzte Technologie

In diesem Kapitel werden die bei der Implementierung der Anwendung eingesetzten Technologien vorgestellt und kurz erläutert. Die Erläuterungen stellen keine vollständigen Beschreibungen der jeweiligen Technologie dar, sondern eine Grundlage für das Verständnis der Implementierung.

4.1 XML-Standards

Neben dem eigentlichen XML-Standard versteht man unter dem Begriff XML eine ganze Standardfamilie zur Beschreibung, Speicherung und Übertragung von Daten. Die XML-Standards werden vom World Wide Web Consortium[8] (W3C) entwickelt und veröffentlicht. In den folgenden Kapiteln werden, die für die Arbeit relevanten XML-Standards beschrieben.

4.1.1 XML

Der XML-Standard definiert Regeln zur Erstellung von maschinen- und menschenlesbaren Dokumenten, deren logischer Aufbau einem hierarchisch strukturierten Baum entspricht.
Ein wesentlicher Bestandteil von XML-Dokumenten sind die Elemente, die entweder durch Start- und Ende-Tags, oder im Falle eines leeren Elements, durch ein Leeres-Element-Tag begrenzt sind. Der Inhalt von nicht leeren Elementen befindet sich zwischen dem Start- und dem Ende-Tag, welcher aus weiteren Elementen und/oder Zeichendaten (die eigentlichen Daten) bestehen kann. Des weiteren dürfen Elemente Attribute haben. Attribute sind Name/Wert-Paare, mit deren Hilfe zusätzliche Daten festgelegt werden können.

```
<Buch status="ausleihbar">
  <Titel>Charting</Titel>
  <Autor>
    <Vorname>Michael</Vorname>
    <Nachname>Saul</Nachname>
  </Autor>
  <ISBN>3-938350-07-5</ISBN>
</Buch>
```

Abbildung 15: Beispiel für ein XML-Dokument

Abbildung 15 zeigt ein einfaches XML-Dokument. Das Element *Buch* ist das Wurzelelement, da es alle anderen Elemente umschließt. Es besitzt ein Attribut mit dem Namen *status*, welches den Wert *ausleihbar* hat. Der Inhalt des *Buch* Elements sind die Elemente *Titel*, *Autor* und *ISBN*. Während die Elemente *Titel* und *ISBN* Zeichendaten (*Charting* und *3-938350-07-5*) als Inhalt haben, beinhaltet das Element Autor weitere Elemente.
Damit XML-Dokumente dem Standard entsprechend wohlgeformt sind, müssen folgende Bedingungen erfüllt sein:

- ein XML-Dokument muss genau ein Wurzelelement haben
- zu jedem öffnenden Tag muss es einen korrespondierenden schließenden Tag geben

[8] Das W3C ist ein internationales Konsortium, welches aus über 350 Mitgliedsorganisationen (private und öffentliche) besteht und sie Aufgabe hat, Web-Standards und -Richtlinien zu entwickeln (http://www.w3c.org).

- es dürfen keine miteinander verschachtelten Tags existieren, z.B.: <tag1><tag2></tag1></tag2>
- ein Element darf keine zwei Attribute mit dem gleichen Namen besitzen

Ein XML-Dokument bezeichnet man als gültig (oder auch valide), wenn es den durch ein XML-Schema (siehe Kapitel 4.1.2) festgelegten Regel, für den Aufbau des Dokuments, entspricht. [XML]

4.1.2 XML-Schema

XML-Schema ist eine Sprache zur Beschreibung eines Typs von XML-Dokumenten, bzw. zur Definition der Struktur von XML-Dokumenten oder der Struktur von Teilen innerhalb von XML-Dokumenten.
Ein XML-Schema Dokument ist selbst ein XML-Dokument (vgl. Kapitel 4.1.1), welches Definitionen von Elementen und Attributen, sowie die Definitionen der zugehörigen Datentypen beinhaltet, und die zulässigen Strukturierungen und Kombinationen der Elemente in einem XML-Dokument festlegt. Der XML-Schema Standard liefert zum einen vordefinierte grundlegende Datentypen (integer, string, ...), und zum anderen stellt es ein Vokabular zur Definition weiterer Datentypen und zur Strukturierung von XML-Dokumenten bereit.
Werden innerhalb eines XML-Dokuments Elemente verschiedener Schemata verwendet, können Namenskonflikte auftreten. Um derartige Konflikte zu vermeiden, können die Typdefinitionen und Elementdeklarationen eines Schemas, einem bestimmten Namensraum (so genannter Ziel-Namensraum) zugeordnet werden. Namensräume werden mittels eines eindeutigen Uniform Resource Identifiers (URI) voneinander abgegrenzt. In Abbildung 16 ist der Inhalt eines XML-Schema Dokuments abgebildet.

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
  xmlns:abc="http://www.beispiel.de"
  targetNamespace="http://www.beispiel.de">

  <simpleType name="persNr">
    <restriction>
      <maxInclusive value="1000"/>
    </restriction>
  </simpleType>

  <element name="Person">
    <complexType>
      <sequence>
        <element name="Name" type="string"/>
        <element name="personalNr" type="abc:persNr"/>
      </sequence>
    </omplexType>
  </element>

</schema>
```

Abbildung 16: Beispiel für ein XML-Schema

Durch das *targetNamespace* Attribut im Wurzelelement *schema*, werden die durch dieses Schema definierten Elemente und Datentypen mit dem (Ziel-)Namensraum *http://www.beispiel.de* verknüpft. Das *simpleType* Element wird verwendet, um einen einfachen Datenytp namens *persNr* zu definieren. Dieser

stellt eine Einschränkung des XML-Schema Basistyp *positiveInteger* dar, mit einem Maximalwert von 1000. Ein XML-Element vom Typ *persNr* darf dementsprechend eine Ganzzahl von 0 bis 1000 enthalten. Die anschließende Elementdeklaration spezifiziert ein *Person* Element als komplexen Typ, mit einer Sequenz aus dem Element *Name* vom Typ *string* und dem Element *personalNr* vom Typ *persNr*. Im Gegensatz zu Elementen eines einfachen Typs, können Elemente eines komplexen Typs weitere Elemente und Attribute besitzen. Die Angabe des Typs in der Deklaration des Elements *personalNr* ist mit dem Präfix *abc* versehen, welches im *schema* Tag an den Namensraum *http://www.beispiel.de* (*xmlns:abc="..."*) gebunden ist. Da dies der gleiche Namensraum wie der Ziel-Namensraum ist, wird ein verarbeitendes Programm in diesem Dokument nach der Typdefinition *persNr* suchen.
Ein XML-Dokument, das einem XML-Schema entspricht, bzw. gegenüber einem XML-Schema valide ist, bezeichnet man auch als Instanzdokument. Abbildung 17 zeigt ein mögliches Instanzdokument in Bezug auf das Schema aus Abbildung 16.

```
<xyz:Person
  xmlns:xyz="http://www.beispiel.de"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.beispiel.de C:\schema.xsd">

  <Name>Helmut Maier</Name>
  <personalNr>563</personalNr>

</xyz:Person>
```

Abbildung 17: XML-Instanzdokument

Entsprechend dem zugrunde liegenden Schema besitzt das Instanzdokument aus Abbildung 17 ein *Person* Element mit den zwei Kind-Elementen *Name* und *personalNr*. Gemäß ihres Datentyps enthält das Element *Name* einen String und das Element *peronalNr* eine positive Ganzzahl, die kleiner als 1001 ist. Das Präfix *xyz* signalisiert einem verarbeitendem Programm, dass das *Person* Element dem Namesraum *http://www.beispiel.de* angehört (*xmlns:xyz="..."*). Das *schemaLocation* Attribut enthält ein Wertepaar bestehend aus einem Namensraum und einem URI. Der URI beschreibt wo ein geeignetes Schemadokument für den angegebenen Namesraum zu finden ist.
In diesem Beispiel dürfen die Elemente *Name* und *personalNr* nicht mit dem Präfix *xyz* versehen werden, da diese im Schema lokal als Kindelemente des Elements *xyz:Person* deklariert sind. [XMLsch][XMLname] [HOLZ]

4.1.3 XPath

Der Zweck der XML Path Sprache (XPath) ist die Adressierung einzelner Teile eines XML-Dokuments, wobei XPath als Grundlage für weitere Standards der XML-Standardfamilie dient.
Das Datenmodell, welches XPath zugrunde liegt, repräsentiert ein XML-Dokument als einen hierarchisch strukturierten Baum, der aus Knoten und Kanten besteht. Es werden aber auch atomare Datentypen wie integer oder string durch dieses Datenmodell unterstützt. Mit Hilfe eines so genannten XPath-Ausdrucks können Positionen und Bewegungsrichtungen in einem solchen Baum beschrieben, sowie einzelne Knoten oder ganze Zweige eines Baumes ausgewählt werden.
XPath unterscheidet sieben verschiedene Knotengattungen: Dokumentknoten, Elementknoten, Textknoten, Kommentarknoten, Processing-Instruction-Knoten, Attributknoten und Namensraumknoten. In Ab-

bildung 18 ist das XML-Dokument aus Abbildung 15 in seiner logischen Struktur als Baum dargestellt.

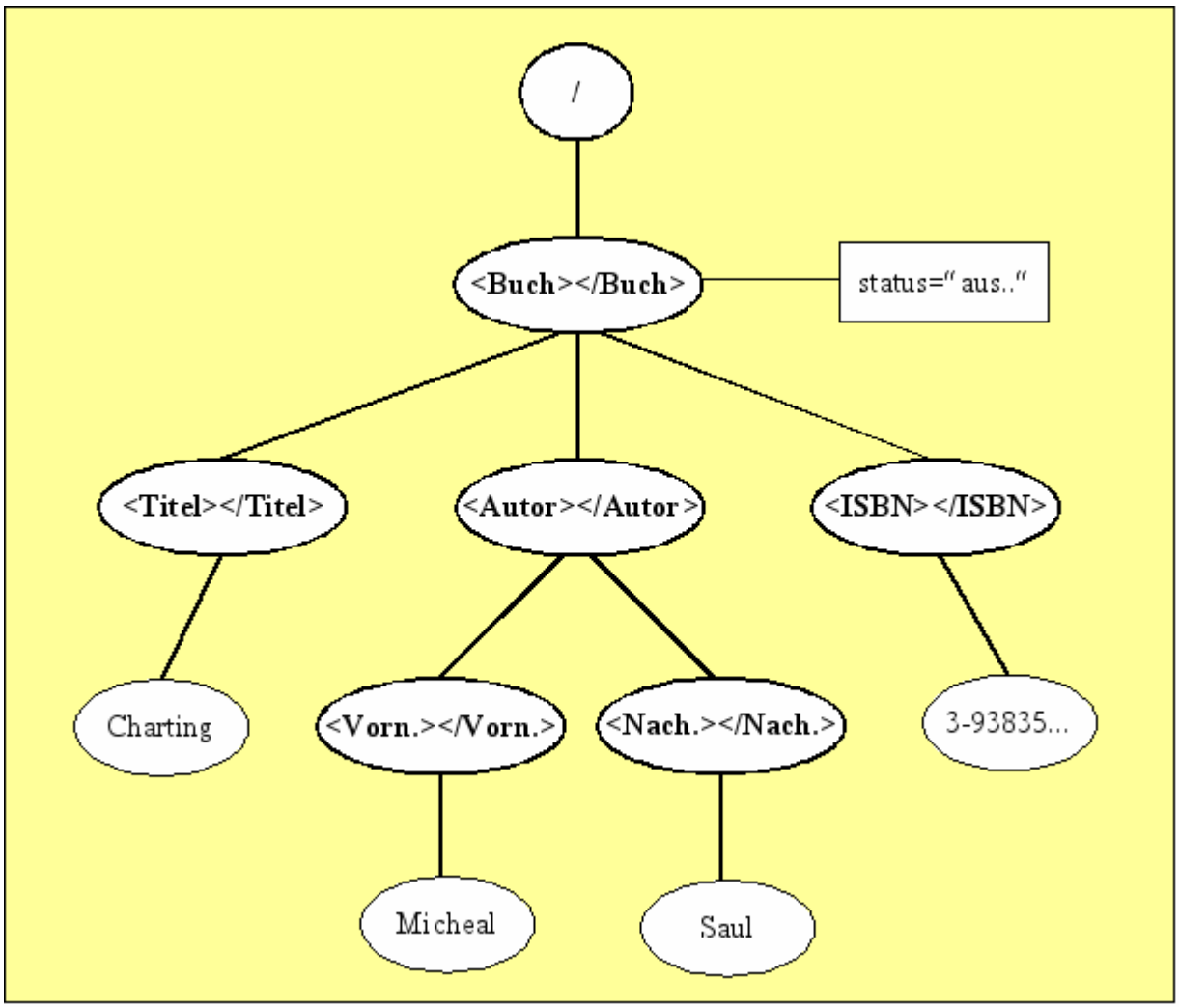

Abbildung 18: Baumstruktur eines XML-Dokuments

Die Wurzel des dargestellten Baums ist ein Dokumentknoten, dessen Inhalt das gesamte Dokument umfasst. Dieser Knoten ist ein abstraktes Konzept, für den es im XML-Dokument keine Entsprechung gibt. Wählt man den Dokumentknoten mit einem XPath-Ausdruck aus, bedeutet das, das gesamte Dokument auszuwählen.
Jedes Element eines XML-Dokuments wird durch einen Elementknoten vertreten. Im Beispiel sind das alle Knoten mit fetter Umrandung, außer dem Dokumentknoten. Elementknoten haben einen Bezeichner in Form eines QNames[9] und können als einzige Knotengattung, neben dem Dokumentknoten, Kinder besitzen. Eine Eltern-Kind-Beziehung zwischen zwei Knoten ist in Abbildung 18 durch eine fett gedruckte Kante dargestellt. Der Knoten *Autor* besitzt beispielweise zwei Kindknoten: den Knoten *Vorname* und den Knoten *Nachname*.
Der Inhalt der Elemente Titel, Vorname, Nachname und ISBN besteht aus reinem Text, welcher in Form von Textknoten abgebildet wird (Knoten mit dünner Umrandung). Da Textknoten nur Zeichendaten

[9]Ein QName (Qualified Name) ist ein Bezeichner, der sich in seiner lexikalischen Form aus einem optionalen Namensraumpräfix und einem lokalen Bezeichner in der Form *Präfix:lokalerBezeichner*, zusammensetzt. Intern arbeiten XML-Anwendungen in der Regel mit sogenannten expandierten QNames. Dabei wird der Präfix durch den URI-String, den er repräsentiert, ersetzt. [BONG]

enthalten, besitzen sie keine hierarchisch geordnete innere Struktur.
Die Attribute von Elementen werden durch Attributknoten (in Abbildung 18 rechteckiger Knoten) dargestellt. Sie besitzen einen QName als Bezeichner und haben einen Wert. Die Beziehungen zwischen Elementknoten und ihren Attributknoten sind per Definition keine vollständigen Eltern-Kind-Beziehungen. Zwar ist ein Elementknoten ein Elternknoten seines Attributknotens, umgekehrt ist der Attributknoten aber nicht Kind des Elementknotens.
Unter dem Begriff XPath-Ausdruck (bzw. Expression) ist alles zu verstehen, was mit Hilfe von XPath formuliert werden kann. Es handelt sich dabei um einen Oberbegriff, der verschiedene Arten von XPath-Ausdrücken zusammenfasst. An dieser Stelle werden nur die so genannten Pfadausdrücke näher beschrieben.
Mit einem Pfadausdruck lassen sich Teile eine XML-Dokuments (bzw. Knoten im repräsentierenden Baum) lokalisieren. Die Lokalisierung kann relativ oder absolut erfolgen. Während der relative Pfadausdruck in Beziehung zu einem aktuellen Kontext (aktueller Knoten) steht, ist der absolute Pfadausdruck dokumentbezogen (bezieht sich auf den Dokumentknoten). In beiden Fällen jedoch, beschreibt der Pfadausdruck einen schrittweisen Weg zwischen zwei Knoten. Daher setzt sich ein Pfadausdruck aus einem oder mehreren Schritten, den Location-Steps, zusammen. Die Location-Steps werden durch das Zeichen / voneinander getrennt.
Ein Location-Step wiederum setzt sich aus zwei Teilen zusammen. Der erste Teil ist die Achse, die Informationen über die Bewegungsrichtung und die Reichweite enthält. Beispiele sind die Child-Achse, Parent-Achse und die Descendant-Achse. Ausgehend vom Elementknoten *Autor* aus Abbildung 18 liegen die Knoten *Vorname* und *Nachname* auf der Child-Achse und der Knoten *Buch* liegt auf der Parent-Achse.
Um aus einer Menge von Knoten, die auf der ausgewählten Achse liegen, eine Untermenge auszuwählen, werden Knotentests verwendet, die den zweiten Teil eines Location-Steps bilden. Zusammengefügt hat ein Location-Step folgende Form: Achse::Knotentest()
Wieder ausgehend vom Knoten *Autor* aus Abbildung 18, wählt der Pfadausdruck *child::Vorname* den Elementknoten *Vorname* aus. In diesem Beipiel bezieht sich der Knotentest auf den Knotenbezeichner. Knotentest bezüglich der Knotengattung sind aber auch möglich.
Optional kann einem Location-Step ein Prädikat angefügt werden, wodurch sich weitere Möglichkeiten ergeben bestimmte Elemente zu extrahieren. Da die Möglichkeiten zur Formulierung von Prädikaten sehr umfangreich sind, werden diese hier nur anhand eines Beispiels erläutert. Um aus einen XML-Dokument wie in Abbildung 18 dargestellt, alle Autoren auszuwählen, die einen bestimmten Nachnamen haben, kann der absolute XPath-Ausdruck */Buch/Autor[Nachname/text()='Müller']* verwendet werden. Das Prädikat in der eckigen Klammer filtert aus allen *Autor* Knoten diejenigen heraus, die einen Kindknoten namens *Nachname* haben, welcher wiederum einen Textknoten besitzt, mit dem Inhalt *Müller*. In dem hier aufgeführten Beispiel wäre die Ergebnismenge des XPath-Ausdrucks jedoch leer, da kein Autor mit dem Nachnamen Müller existiert.
Sämtliche Ergebnismengen eines XPath-Ausdrucks, werden im zugrunde liegenden Datenmodell als Sequenz bezeichnet. Eine Sequenz ist eine in Dokumentenreihenfolge angeordnete Menge von Items. Items können Knoten, aber auch Werte eines atomaren Typs sein. [BONG][XPath][DATA]

4.2 Java-Technologie

Die Java-Technologie setzt sich aus zwei Teilen zusammen: die Programmiersprache Java und eine Java-Plattform. Eine Plattform besteht aus einer Ablaufumgebung und aus Programmierschnittstellen, um Java-Programme auszuführen. Die Java Plattform Standard Edition (Java SE) stellt eine Reihe von

Application Programming Interfaces (API) zur Verfügung und ist für lokale Anwendungen auf Einzelplatzrechnern vorgesehen. Sie bildet außerdem eine Grundlage für weitere Java-Plattformen. Die auf Java SE aufbauende Java Plattform Enterprise Edition (Java EE) stellt zusätzliche APIs zur Verfügung, die für Anwendungen auf Webservern geeignet sind. [JAVA]
In den folgenden Kapiteln werden einige für die Arbeit relevanten Komponenten der Java EE und Konzepte zur Verarbeitung von XML mit Java vorgestellt, wobei die Programmiersprache Java und die grundlegenden Java APIs als bekannt vorausgesetzt werden.

4.2.1 Java EE allgemein

Neben den Java APIs zur Entwicklung von serverseitigen Applikationen enthält Java EE eine Reihe von Spezifikationen, die verschiedene Softwarekomponenten und deren Zusammenspiel definiert. Damit wird ein Rahmen zur Verfügung gestellt, der als Grundlage für die Entwicklung von mehrschichtigen Web-Anwednungen benutzt werden kann.
Für die Ausführung von Java EE Komponenten wird eine spezielle Laufzeitumgebung benötigt, einen sogenannten Java EE Application Server. In Abbildung 19 ist ein Java EE Server im Rahmen des mehrschichtigen Java EE Applikation Modells zu sehen.

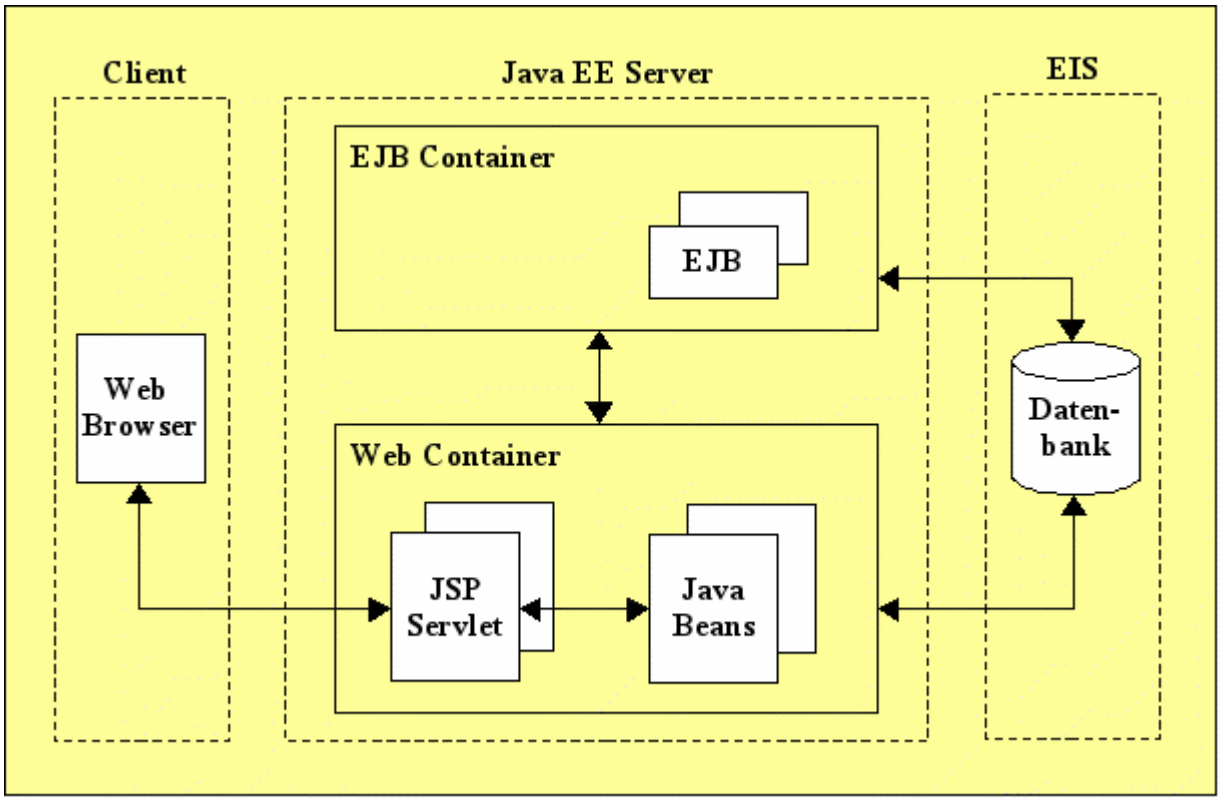

Abbildung 19: Java EE Applikations Modell

Ein Java EE Server teilt sich in die zwei Einheiten Web-Container und Enterprise Java Bean (EJB) Container. Der Web-Container ist die Laufzeitumgebung für Servlets und JavaServer Pages (JSP) einschließlich deren unterstützenden Java-Klassen (Java Beans) und der EJB-Container ermöglicht das Ausführen von EJBs. EJBs sind durch Java EE spezifizierte Komponenten, die verwendet werden können, um Module der Geschäftlogik einer Anwendung aufzunehmen. Für diese Arbeit spielen die EJBs jedoch keine Rolle und werden daher nicht weiter betrachtet.

Im Java EE Applikationsmodell werden im wesentlichen drei Schichten unterschieden. Eine Client-Schicht, eine Java EE Server Schicht und die sogenannte Enterprise Information System (EIS) Schicht. Die Schicht, die durch den Java EE Server realisiert ist, kann weiter in eine Web- und in eine EJB-Schicht unterteilt werden.
Ein auf der Client-Schicht laufender Web-Browser kommuniziert mit dem Server unter Verwendung des HTTP-Protokolls, indem er Komponeten des Web-Containers, wie Servlets oder JSPs, als Zugangspunkte benutzt. Entsprechend der Clientanfrage werden bestimmte Servlets und/oder JSPs zur Ausführung gebracht. Diese können zur Generierung der Anwort an den Client weitere Java Klassen benutzen, die in der Abbildung als Java Beans bezeichnet sind. Die Anwendungslogik kann komplett durch die Web-Komponeten aufgenommen werden, oder aber die Web-Komponeten dienen nur als Zugang zu einer Anwendungslogik, die durch die EJB-Komponenten implementiert ist. Sowohl Web- als auch EJB-Komponenten können auf Einheiten der EIS-Schicht zugreifen. Unter den Einheiten dieser Schicht sind Datenbanken, aber auch Dienste anderer Server zu verstehen. [BERG][J2EETut01]

4.2.2 Servlets

Servlets sind Java Klassen, die serverseitige (Java-)Anwendungen mit der Fähigkeit ausstatten können, Anfragen eines Clients entgegen zunehmen und eine dynamische benutzerabhängige Antwort zu generieren (i.d.R HTML-Seiten). Instanzen solcher Klassen laufen im Web-Container eines Applikationsservers ab und müssen die Schnittstelle *javax.servlet.Servlet* implementieren, welche die Methoden definiert, die der Web-Container verwendet, um mit einem Servlet zu interagieren. Um Anfragen auf Basis des HTTP-Protokolls zu bearbeiten, muss ein Servlet die Klasse *javax.servlet.http.HttpServlet* erweitern. Diese Klasse implementiert das Interface *javax.servlet.Servlet* und stellt Methoden für die Abwicklung einer HTTP-Anfrage bereit.
Der gesamte Lebenszyklus eines Servlets wird durch den Web-Container geregelt. Dieser erzeugt eine Instanz des Servlets und führt die Methoden des Servlets, bei jedem Aufruf eines mit dem Servlet verknüpften URLs, aus. Die zwei wesentlichen Methoden eines HTTP Servlets sind *doGet()* und *doPost()*, welche durch jedes Servlet, dass sich aus der *HTTPServlet* Klasse ableitet, überschrieben werden müssen, um einen GET- und POST-Request[10] in geeigneter Weise zu verarbeiten.
Eine HTTP-Anfrage eines Web-Browsers der Form *get www.beispiel.de/start* veranlasst also den Web-Container eines Servers die Methode *doGet()* des Servlets auszuführen, welches mit der URL *www.beispiel.de/start* verknüpft ist.
Die Erzeugung der Antwort auf eine Anfrage kann durch das Servlet selbst, gegebenenfalls unter Benutzung weiterer Java Klassen erfolgen, oder die Anfrage wird durch das Servlet an andere Komponenten weitergeleitet, die eine entsprechende Antwort generieren. [BERG][HUNT]

4.2.3 JavaServer Page JSP

JSPs bauen auf der Servlet Technologie auf und können im Rahmen einer Web-Anwendung verwendet werden, um Antworten auf Clientanfragen mit statischem und dynamischem Inhalt zu erzeugen. Eine JSP ist ein Textdokument, welches zwei Arten von Text enthält: Statische Daten, die in einem textbasierten Format ausgedrück sind, wie beispielsweise HTML oder XML, und die sogenannten JSP Elemente, die den dynamischen Inhalt der JSP erzeugen. Im Kontext dieser Arbeit kann eine JSP als ein HTML-Dokument angesehen werden, welches JSP Elemente enthält, die den dynamischen Teil der an den Client zusendende

[10] Das HTTP-Protokoll unterscheidet mehrere Request-Methoden, die den Server zur Durchführung bestimmter Aktionen veranlassen. Die Methoden GET und POST veranlassen den Server bestimmte Daten an den Client zu senden, wobei die POST-Methode geeignet ist, um große Datenmengen mit der Anfrage an den Server zu senden. Bei diesen Daten kann es sich beispielsweise um Parameter eines HTML-Formulars handeln. [HUNT]

HTML-Seite, darstellen.
Bei einem Request nach einer JSP wird der Java Code, der sich in den JSP Elementen verbirgt, im Web-Container des Applikations Servers zur Ausführung gebracht. Das Ergebnis der Ausführung, welches auch wiederum HTML-Text sein kann, wird in den statischen Teil der JSP integriert und die vollständige HTML-Seite wird als Antwort an den Client gesendet. Auf der Clientseite kommt also reiner (X)HTML-Text an, der im Browser angezeigt werden kann.
Im Gegensatz zu einem Servlet hat eine JSP den Vorteil, dass das Erscheinungsbild einer JSP sich an das eines HTML-Dokuments anpasst und somit besser geeignet ist, die Schnittstellen zum Benutzer zu implementieren, da diese, wie bereits erwähnt, in HTML formuliert sein müssen.
In Abbildung 20 ist im oberen Teil eine einfache JSP zu sehen, wobei die JSP Elemente fett gedruckt sind.

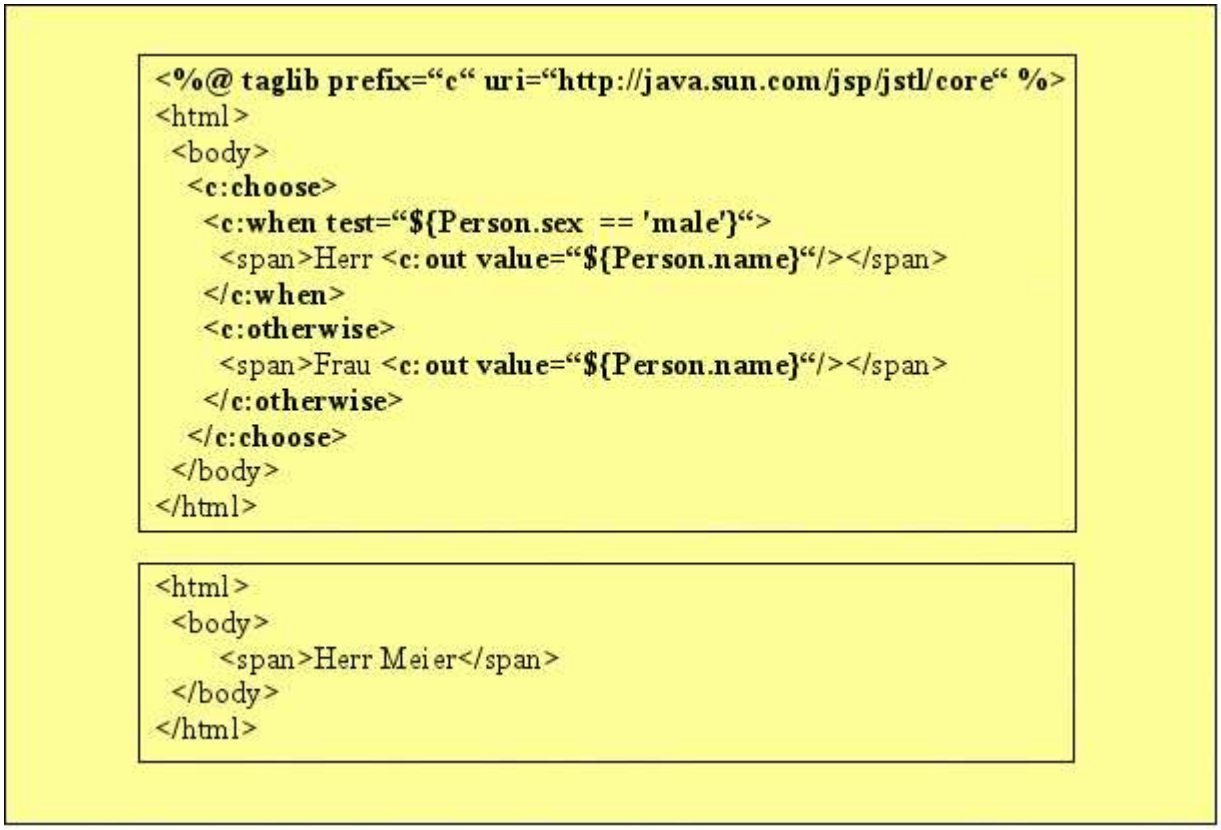

Abbildung 20: Beispiel JSP

Man unterscheidet vier Typen von JSP Elementen: Direktiven, Aktionselemente, Skriptelemente und sogenannte Expression Language (EL) Ausdrücke.
Durch Direktiven werden Informationen über die JSP selbst definiert. Dazu zählt beispielsweise die Information, welche Java Klassen importiert werden sollen, oder ob (bzw. welche) weitere JSPs eingebunden werden sollen. Die Direktive in der ersten Zeile der JSP aus Abbildung 20 zeigt an, dass in diese JSP Aktionselemente einer bestimmten Tag-Bibliothek zum Einsatz kommen.
Aktionselemente führen eine Aktion auf Daten aus, wie sie zum Zeitpunkt der Anforderung der JSP durch den Browser, vorliegen. Zum Beispiel kann eine Datenbankanfrage mit Hilfe, der durch den Request mit gelieferten Parameter, durchgeführt werden. In der JSP-Spezifikation sind eine Reihe von Standard-Aktionselementen definiert (z.B. <jsp:getProberty> zur Abfrage eines Wertes einer Java Bean). Es be-

steht aber die Möglichkeit weitere Aktionselemente zu definieren, die in sogenannten Tag-Bibliotheken zusammengefasst sind. Die Aktionselemente wie sie in der JSP aus Abbildung 20 zum Einsatz kommen, gehören zur JSP Standard Tag Library (JSTL), welche durch die *taglib* Direktive in die JSP eingebunden ist. In der JSTL sind Aktionselemente definiert, die Kernfunktionen wie Schleifen und bedingte Verzweigungen kapseln. In Abbildung 20 haben die JSLT-Elemente den Präfix *c* und kommen zum Einsatz, um in Abhänigkeit des Wertes der Eigenschaft *Person.sex* den Inhalt des Antwort-Dokuments festzulegen. Im *test* Attribut des *c:when* Elements kommt ein EL-Ausdruck zum Einsatz. Diese können verwendet werden, um zur Laufzeit auf Java Bean Objekte zu zugreifen und arithmetische Operationen auszuführen. Ein EL-Ausdruck kann sowohl innerhalb eines statischen Teils einer JSP, als auch innerhalb von Aktionselementen vorkommen. Der EL-Ausdruck *${Person.sex == 'male'}* liest den Wert der Eigenschaft *sex* der Java Bean *Person* und vergleicht diesen mit dem String *male*. Die Auswertung des Ausdrucks (true oder false) liefert den Wert des *test* Attributs im *c:when* Element, wodurch das *span* Element mit der entsprechenden Anrede ausgewählt wird. In der unteren Hälfte der Abbildung 20 ist ein HTML-Dokument zu sehen, welches als mögliche Antwort auf eine Anfrage nach der JSP an den Browser gesendet werden könnte. Das *html* und das *body* Element, die den statischen Teil der JSP darstellen, bleiben unverändert erhalten, während der Inhalt des *span* Elements in Abhängigkeit des Zustandes der Java Bean *Person* zum Zeitpunkt der Anfrage dynamisch erzeugt wurde. Das *span* Element kann also bei weiteren Anfragen einen anderen Inhalt haben.
Die Skriptelemente enthalten gewöhnlichen Java Code, der zum Zeitpunkt der Anfrage der JSP zur Ausführung gebracht wird. Die JSP in Abbildung 20 enthält keine Skriptelemente. [BERG][J2EETut02][JSP]

4.2.4 Java Beans

Unter Java Beans versteht man Java Klassen, die bestimmte Design Konventionen einhalten. Damit JSPs auf Daten, die durch eine Java Klasse gekapselt sind, zugreifen können, müssen diese Klassen die Eigenschaften einer Java Bean mitbringen. Die Daten, die durch eine Java Bean gekapselt sind, werden im folgenden als eine Eigenschaft bezeichnet, um eine Abgrenzung zu den Klassenattributen einer Java Bean herzustellen.
Eine Eigenschaft kann nur lesbar, nur schreibbar oder les- und schreibbar sein. Der Wert einer Eigenschaft kann einem einfachen Datentyp wie int oder einer Java Klasse entsprechen. Der lesende und schreibende Zugriff auf die Eigenschaften erfolgt durch spezielle Zugriffsmethoden in der Form *getEigenschaftsname()* und *setEigenschaftsname()*, wobei die erste Methode den Wert der Eigenschaft zurückliefert und mit der zweiten Methode der Wert der Eigenschaft gesetzt wird. Der Name der sogenannten Getter und Setter Methoden setzen sich zusammen aus dem Wort get oder set und dem Namen der Eigenschaft, der mit einem großen Buchstaben beginnt. Der Name der Eigenschaft selbst beginnt mit einem kleinen Buchstaben. Eine Eigenschaft einer JavaBean kann beispielsweise durch ein Klassenattribut repräsentiert sein, sodass durch eine entsprechenden Get-Methode einfach der Wert des Klassenattributs zurückgegeben wird. Grundsätzlich ist eine Eigenschaft aber durch die Anwesenheit einer Zugriffsmethode definiert.
Der Zugriff auf eine Eigenschaft einer Bean aus einer JSP heraus, kann wie in Abbildung 20 gezeigt, durch einen EL-Ausdruck erfolgen. In dem EL-Ausdruck *${Person.name}* ist *Person* ein Kontext abhängiger Name einer Bean und *name* ist der Name einer Eigenschaft dieser Bean, die durch eine *getName()* Methode innerhalb der Bean definert sein muss, damit der EL-Ausdruck den Wert der Eigenschaft lesen kann.
Neben einem parameterlosen Konstruktor und den Getter und Setter Methoden, darf eine Java Bean beliebige andere Methoden besitzen. [BERG][J2EETut03]

4.2.5 XML Verarbeitung

Um die Inhalte eines XML-Dokuments für Anwendungen verfügbar zu machen, werden Parser eingesetzt. Hinsichtlich der Art und Weise mit der XML-Parser die Daten eines XML-Dokuments zur Verarbeitung bereitstellen, lassen sich zwei grundlegende Konzepte unterscheiden. Beim ereignisbasiertem Parsen werden in Abhängigkeit bestimmter Ereignisse während dem Parse-Vorgang Methoden aufgerufen, die entsprechend dem Ereignis bestimmte Aktionen ausführen können. In der zweiten Variante wird das gesamte XML-Dokument in Form seiner hierarchischen Baumstruktur in den Arbeitsspeicher geladen, sodass nach dem Parsen sämtliche Bestandteile des Dokuments für die weitere Verarbeitung zur Verfügung stehen.
Die Simple API for XML (SAX) definiert die Klassen und Schnittstellen für die ereignisbasierte Verarbeitung von XML-Dokumenten mit Java. XML-Parser, die diese Schnittstellen implementieren werden daher als SAX-Parser bezeichnet. Ein SAX-Parser ruft beim Eintreffen eines Ereignisses, zum Beispiel der Beginn eines Elements, eine entsprechende Methode auf, sodass der Parsevorgang erst fortgesetzt wird, nachdem die Methode ausgeführt wurde. Die Reihenfolge mit der die Ereignisse auftreten, entspricht der Reihenfolge mit der die einzelnen Komponenten eines XML-Dokuments in seiner textuellen Darstellung vorkommen. Der Inhalt eines XML-Elements (Text, Kindelemente, ...) wird also so verarbeitet, wie er zwischen dem Start- und Ende-Tag des Elements vorliegt. Die Methoden, die beim Auftreten eines Ereignisses ausgeführt werden, sind in Java Klassen definiert, die als ContentHandler bezeichnet werden, da sie ein gleichnamiges Interface implementieren müssen. Ein ContentHandler muss vor dem Parsen beim Parser registriert werden, damit die Methoden zur Verfügung stehen. Ein Beispiel für eine Methode, wie sie durch den ContentHandler zu implementieren ist, ist die Methode *startElement(String namespaceURI, String localName, String qName, Attributes atts)*. Sie wird ausgeführt, wenn der Parser auf den Anfang eines Elements trifft. Innerhalb diese Methode können dann die anwendungsspezifischen Aktionen festgelegt werden, die bei diesem Ereignis durchgeführt werden sollen. Im Beispiel der *startElement()* Methode stehen dazu einige Parameter bereit, wie eine Liste der Attribute und den Namen des Elements.
Die sequentielle Verarbeitung von XML-Dokumenten durch einen SAX-Parser ist schnell und sehr speicherschonend, weist jedoch den Nachteil auf, dass auf die Inhalte des Dokuments nur zum Zeitpunkt eines entsprechenden Ereignisses zugegriffen werden kann. Werden bestimmte Inhalte nachträglich noch benötigt, müssen diese für späterere Zugriffe gespeichert werden.
Die zweite grundlegende Möglichkeit XML-Dokument mit Java zu Verarbeiten, besteht darin, das gesamte XML-Dokument als Document Object Model (DOM) in den Arbeitsspeicher zu laden, um einen wahlfreien Zugriff auf alle Teile des Dokuments zu haben. DOM ist eine durch das W3C spezifizierte Schnittstelle für den Zugriff auf XML-Dokumente. Im DOM Strukturmodell wird ein XML-Dokument als ein hierarchisch strukturierter Baum von *Node* Objekten abgebildet, wobei es *Node* Objekte verschiedener Typen gibt (z.B. *Element Node*, *Attribut Node*, *Text Node*, ...). Für jedes *Node* Objekt sind die zulässigen *Child-Nodes* und die Schnittstellen (Methoden) für den Zugriff auf *Nodes* definiert. XML-Parser, die als Ergebnis des Parse-Prozesses einen *Document-Node* (dem DOM Modell entsprechend), der das gesamte XML-Dokument enthält, liefern, werden als DOM-Parser bezeichnet.
Nach dem Parsen eines XML-Dokuments durch einen DOM-Parser, steht das gesamte Dokument als *Document-Node* in Form eines Java Objekts im Speicher bereit, so dass mit den Java Implemetierungen (DOM-API) der DOM Schnittstellen auf beliebige Inhalte des Dokuments zugegriffen werden kann. Dieser komfortable Zugriff auf XML-Dokumente, geht auf Kosten eines hohen Speicherbedarfs bei großen Dokumenten. [BRET][DOM]

4.3 Web-Design

Da clientseitig ein Web-Browser zum Einsatz kommt, um mit der Anwendung zu kommunizieren, sind die Benutzeroberflächen, wie bereits erwähnt, als HTML-Seiten konzipiert. Um das Design der Oberflächen unabhängig von dem Inhalt der HTML-Dokumente zu gestalten, kommen Cascading Style Sheets (CSS) zum Einsatz, die in separaten Dateien vorliegen. Die Bedienung der Oberflächen wird teilweise durch die clientseitige Skriptsprache JavaScript realisiert. Der JavaScript-Code liegt ebenfalls in separaten Dateien vor.

4.3.1 HTML / XHTML

HTML ist eine Auszeichnungssprache zur Strukturierung der Inhalte von Dokumenten. Dabei wird der Text eines Dokuments mit Auszeichnungen in Form von Tags versehen, wodurch einzelnen Textteilen eine bestimmte Bedeutung zugewiesen werden kann, und das gesamte Dokument eine Struktur erhält. Die HTML-Spezifikation des W3Cs definiert diverse Tags und deren Bedeutung, wie sie von verarbeitenden Programmen (z.B. Web-Browser) zu interpretieren sind.
Wie HTML ist die Extensible HyperText Markup Language (XHTML) eine Auszeichnungssprache zur Strukturierung von Dokumenten, die die gleichen Tags verwendet wie sie der HTML-Standard definiert. Der Unterschied zu HTML besteht darin, dass XHTML-Dokumente den Syntaxregeln von XML entsprechen, also gleichzeitig valide XML-Dokumente sind. Da dies eine Einschränkung zu HTML darstellt, können Browser, die HTML-Dokumente anzeigen können, auch XHTML-Dokumente interpretieren. Die Benutzeroberflächen der Anwendung sind XHTML-Dokumente. [HTML][XHTML]

4.3.2 Cascading Style Sheets

Cascading Style Sheets (CSS) ist eine Stylesheet-Sprache für strukturierte Dokumente und wird hauptsächlich in Zusammenhang mit HTML eingesetzt. Durch CSS wird festgelegt, wie die Inhalte eines Dokuments (hier also die HTML-Elemente) dargestellt werden soll, so dass eine HTML-Seite in verschiedenen Browsern immer das gleiche Erscheinungsbild hat. Außerdem wird durch den Einsatz von CSS eine Trennung zwischen dem Inhalt eines HTML-Dokuments und der Beschreibung der Darstellung des Dokuments erreicht. Somit kann das Design einer HTML-Seite leicht verändert werden, bzw. es können verschiedene Darstellungen für das gleiche Dokument definiert werden, ohne das Dokument selbst zu verändern. CSS wird vom W3C spezifiziert und wird von allen gebräuchlichen Web-Browsern ausreichend unterstützt.[CSS]

4.3.3 JavaScript

JavaScript ist eine objektbasierte Programmiersprache, die hauptsächlich eingesetzt wird, um HTML-Dokumente, die als solche ein rein statisches Erscheinungsbild aufweisen, mit erweiterter Funktionalität und Anzeigeeffekten auszustatten. Dazu wird der JavaScript-Code in das HTML-Dokument eingebettet und clientseitig von einem browser-internen Interpreter ausgeführt. Mit JavaScript können beispielsweise in Abhängigkeit von Anwenderaktionen die Inhalte und deren Erscheinungsform von HTML-Dokumenten manipuliert werden. Die grundlegenden Sprachelemente sind im ECMAScript Standard definiert, die von allen JavaScript-fähigen Browsern unterstützt werden. Neuere Web-Browser unterstützen DOM, sodass durch JavaScript auf die HTML-Elemente in Form von DOM-Objekten zugegriffen werden kann. [BEBO][ECMA]

5 Aufbau der Anwendung

Um ein möglichst flexibles Programmdesign bereitzustellen, das nachträgliche Änderungen und Erweiterungen erleichtert, ist die Anwendung komponentenweise aufgebaut. Gleichzeitig kommen Technologien zum Einsatz (siehe Kapitel 4), die sich im Bereich der Softwareentwicklung etabliert haben und somit eine breite Unterstützung hinsichtlich unterstützender (freier) Software und Dokumentation gewährleistet ist. In den folgenden Kapiteln werden der Aufbau der Anwendung und deren einzelnen Komponenten näher beschrieben.

5.1 Design Modell

Der Aufbau der Anwendung liegt dem Model-View-Controller (MVC) Modell zugrunde. Dieses Programmiermodell sieht eine Gliederung eines Softwaresystems in die drei Funktionseinheiten Datenhaltung (Model), Präsentation (View) und Programmsteuerung (Controller) vor. [BERG]
Unter der Einheit Datenhaltung sind neben den Anwendungsdaten, auch die Operationen auf diesen, sowie die gesamte Geschäftslogik der Anwendung zu sehen. Dieser Funktionsbereich wird in der Anwendung durch XML-Dateien, Java Beans und anderen Java Klassen abgebildet. Der Bereich Präsentation enthält alle Komponenten, die die Anwendungsdaten (bzw. Teile davon) dem Benutzer zugänglich machen. Die Präsentation ist durch Java Server Pages realisiert. Die Programmsteuerung verkörpert die Komponenten, die in Abhänigkeit von Benutzereingaben und dem Zustand der Daten entscheidet, welche Aktionen auszuführen sind. In der Anwendung ist die Steuerung durch Servlets repräsentiert. In Abbildung 21 sind die Komponenten der Anwendung unter Berücksichtigung des MVC Modells dargestellt.
Alle Requests der Clients werden durch die Kontrolleinheiten, also den Servlets, entgegengenommen. Diese bilden damit den Zugang des Clients zur Anwendung. Ein Request kann einerseits eine reine Aufforderung zum Senden einer Web-Seite sein, wie beispielsweise die Suchmaske zum Suchen nach Datensätzen, und kann andererseits eine Anfrage zum Abrufen oder Manipulieren von Daten sein. Die Servlets haben damit die Aufgabe die Anfragen durch den Client zu bewerten und die Generierung einer entsprechenden Antwort zu veranlassen. Um die Clientanfrage auszuwerten, verwenden die Servlets Methoden der Java Beans, die die Daten der Anwendung abbilden und die Operationen auf diesen definieren. Die Resultate der ausgeführten Methoden liefert den Servlets die nötige Information, um den weiteren Programmablauf festzulegen, d.h. die Anfrage an eine der Auswertung entsprechende JSP weiterzuleiten. Die JSPs erzeugen nun die Antwort an den Client in Form einer XHTML-Seite, die die Sicht auf den Teil der Anwendungsdaten frei gibt, wie sie durch den Client angefordert wurde. Der Zugriff auf die Daten durch die JSPs erfolgt ebenfalls durch Methoden der Java Beans. Neben der Sicht auf die Daten, sind durch die JSPs die Zugriffsmöglichkeiten des Clients auf die Daten definiert. Da die Benutzerschnittstellen, wie zum Beispiel die Suchmaske, festlegen, welche Parameter vom Benutzer verwendet werden können und durch die Servlets ausgewertet werden, sind die Möglichkeiten der Datenmanipulation des Benutzers, durch die JSPs vorgegeben.
Die Daten der Anwendung sind zur Laufzeit durch die Java Beans bereitgestellt, die die Schnittstelle der Daten zu den Funktionseinheiten View und Controller darstellen. Die Schnittstellen sind durch die Methoden der Beans definiert. Während die JSPs durch die Getter-Methoden der Beans ausschließlich lesend auf die Daten zugreifen, verwenden die Servlets zur Verarbeitung eines Requests auch die Methoden der Beans, die eine Manipulation der Anwendungsdaten mit sich bringen. Für die Ausführung der Methoden, die von den JSPs und den Servlets eingesetzt werden, verwenden die Beans weitere Java Klassen. Die persistenten Daten wie die Datensätze und die Meta-Daten der Datensätze sind in Form von XML-Dateien gespeichert.
Durch den Einsatz dieses Programmiermodells wird die Anwendung flexibel und überschaubar, sodass

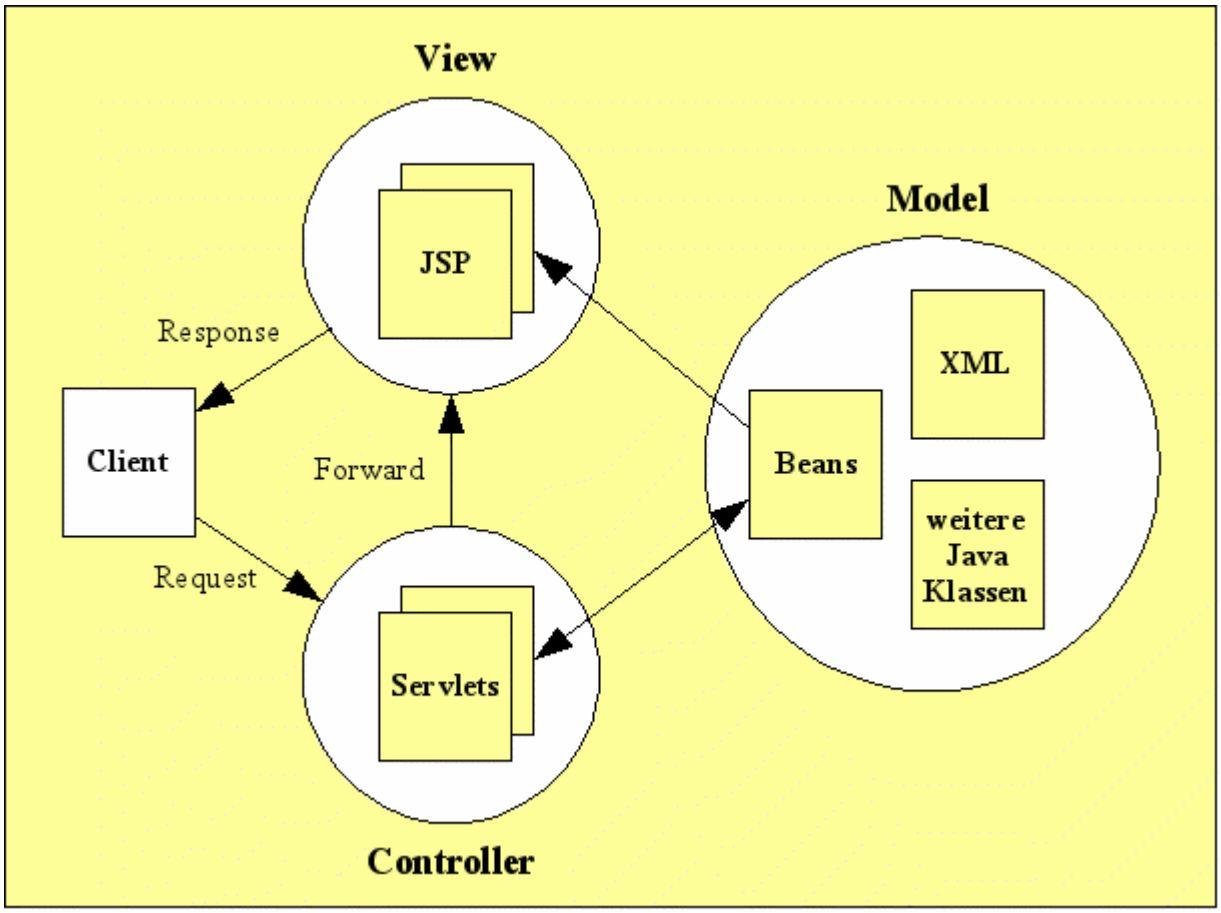

Abbildung 21: Programmiermodell der Anwendung

Änderungen und Erweiterungen leichter durchzuführen sind. Die Komponenten der Präsentation können, unter Berücksichtigung der Schnittstellen, ungeachtet der Anwendungslogik verändert werden, um dem Benutzer eine andere oder zusätzliche Sicht auf die Daten zu ermöglichen. Außerdem kann ein (beliebig) anderes Format anstatt XHTML für die Benutzeroberfächen verwendet werden ohne die Anwendungslogik zu beeinflussen. Dadurch, dass die JSPs in ihrer Funktion als Präsentationskomponenten (fast) keinen Java-Code enthalten, sind diese übersichtlich und entsprechen in ihrer Erscheinung im Wesentlichen einem gewöhnlichen XHTML-Dokument.
Bei Erweiterungen der Anwendungslogik sind durch Java Beans gegebenenfalls entsprechende Schnittstellen bereitzustellen, damit die Erweiterungen durch die Präsentations- und Controller-Komponenten eingesetzt werden können.

5.1.1 View

Der View gliedert sich entsprechend der Benutzerinteraktion mit der Anwendung in die folgenden JSPs:

- *Form.jsp* - Diese JSP ist für die Erzeugung der Suchmaske zuständig. In der Suchmaske kann der Benutzer unabhängig voneinander für die einzelnen Meta-Daten Werte eingegeben, die dann von der Anwendungslogik verwendet werden, um die Datensätze bereitzustellen, die entsprechende Meta-Daten besitzen. Die Suchmaske ist der Startpunkt zum Auschecken von Datensätzen.
- *Hits.jsp* - Die JSP liefert eine XHTML-Seite an den Client, die eine Liste aller Datensätze mit den

erwünschten Eigenschaften enthält. Aus dieser Liste kann der Anwender nun diejenigen Datensätze auswählen, die er auschecken möchte.

- *Download.jsp* - Auf dieser Seite kann der Anwender das Dateiformat auswählen, in dem die ausgesuchten Datensätze zum Download bereitgestellt werden sollen.
- *CheckInUpload.jsp* - Diese JSP stellt eine Maske bereit, die es dem Anwender ermöglicht einen Datei-Upload durchzuführen, um einen neuen Datensatz in die bestehende Sammlung zu integrieren. Sie ist die Startseite zum Einchecken von Datensätzen.
- *CheckInConfirm.jsp* - Die JSP erzeugt eine XHTML-Seite, die entweder eine erfolgreiche Integration eines neuen Datensatzes bestätigt oder eine Fehlermeldung bezüglich des Eincheckens anzeigt.
- *Error.jsp* - Der Inhalt diese Seite enthält den Hinweis, dass während der Programmausführung ein Fehler (Exception) aufgetreten ist. Die eigentliche Fehlermeldung wird aber in ein Logfile geschrieben und so vor dem Benutzer verborgen.

Das Erscheinungsbild im Web-Browser, der durch die JSPs erzeugten XHTML-Dokumente, ist durch CSS definiert. Die CSS liegen für jede JSP getrennt in separaten gleichnamigen Dateien (Form.css, Hits.css, Download.css, CheckInUpload.css, CheckInConfirm.css und Error.jsp) vor. Die CSS-Dateien sind über das *link* Element in die XHTML-Dokumente eingebunden und haben die Form: *<link rel="stylesheet" href="dateiURI.css" type="text/css"/>*.
Desweiteren kommt in der XHTML-Seite, die die Treffer der Suchanfrage anzeigt (Hits.jsp), JavaScript zum Einsatz. Der JavaScript-Quelltext liegt ebenfalls in separaten Dateien vor, die durch *script* Elemente in das Dokument eingebunden werden: *<script type="text/javascript" language="JavaScript" src="dateiURI.js"/>*.
In Konsequenz des MVC enthalten die JSPs keine Anwendungslogik, sodass nur Aktionselemente der JSP Standard Tag Library zum Einsatz kommen, um in Abhängigkeit relevanter Anwendungsdaten den dynamischen Teil der XHTML-Dokumente zu erzeugen. Für den ausschließlich lesenden Zugriff auf die Daten werden ohne Ausnahme EL-Ausdrücke verwendet.

5.1.2 Controller

Die Servlets der Anwendung sind durch URLs adressiert, sodass diese durch entsprechende Requests durch einen Web-Browser zur Ausführung gebracht werden können. Die Servlets lassen sich in drei Gruppen teilen.
Die erste Gruppe enthält die Servlets, die einen Request bearbeiten, indem sie diesen lediglich zu einer JSP weiterleiten. Diese Servlets können sowohl einen GET-, als auch einen POST-Request verarbeiten, und bilden die Einstiegspunkte für den Benutzer zu der Anwendung, da sie die Anfrage auf die JSPs *Form.jsp* und *CheckInUpload.jsp* weiterleiten. Die Servlets werden durch Links von allen JSPs referenziert. Im einzelnen sind das:

- *GetForm* - Leitet die Anfrage zur *Form.jsp* weiter, wodurch dem Anwender die Suchmaske zugestellt wird.
- *GetCheckInUpload* - Leitet die Anfrage zur *CheckInUpload.jsp* weiter, wodurch dem Anwender die Maske zum Einchecken von Datensätzen übermittelt wird.

Die zweite Gruppe von Servlets sind diejenigen, die während dem Vorgang des Ein- oder Auscheckens aufgerufen werden. Die Requests an diese Servlets werden ausschließlich durch das Abschicken von HTML-Formulardaten durch den Anwender ausgelöst. Da diese Servlets Parameter entgegennehmen müssen und

ein Aufruf nur im Kontext eines Ein-, bzw. Auscheck-Vorgangs zweckmäßig ist, unterstützen sie nur die POST-Methode des HTTP-Protokolls. Folgend die Servlets, die beim Auschecken zum Einsatz kommen:

- *ValidateForm* - Veranlasst die Überprüfung der Benutzereingaben in der Suchmaske auf Plausibilität und leitet den Request entsprechend dem Ergebnis der Überprüfung weiter.
- *SearchDataset* - Veranlasst die Durchführung einer Suchanfrage nach Datensätzen mit erwünschten Eigenschaften.
- *GetDownloadPage* - Leitet die Anfrage nach Auswahl der Datensätze weiter an die Download JSP.
- *Download* - Veranlasst die Transformation der Datensätze in das entsprechende Dateiformat und stellt die Datensätze als Zip-Datei zum Download zur Verfügung.

Der Vorgang des Eincheckens wird durch die nachstehenden Servlets gesteuert:

- *ValidateUpload* - Veranlasst die Validierung der Upload-Datei, die den einzucheckenden Datensatz enthält, und die Überprüfung der übrigen Benutzereingaben aus der Maske zum Einchecken von Datensätzen.
- *CreateNewDataset* - Veranlasst die Integration eines neuen Datensatzes in die bestehende Sammlung.

Die dritte Gruppe besteht nur aus dem Servlet *ErrorHandle*. Treten bei der Programmausführung Fehler auf, wird das Servlet ErrorHandle aufgerufen, welches Informationen über das Exception-Objekt in ein Logfile schreibt und den Aufruf an die JSP *Error.jsp* weiterleitet.

5.1.3 Model

Die persistenten Daten der Anwendung werden in Form von XML-Dateien abgespeichert. Es existieren zwei verschiedene XML-Schemata zur Definition von XML-Dateien. Das eine Schema definiert ein XML-Dokument, welches zur Speicherung der Meta-Daten verwendet wird und das andere Schema definiert das XML-Speicherformt für die Datensätze. Die Meta-Daten aller vorhandenen Datensätze sind innerhalb einer XML-Datei gespeichert, während jeder Datensatz durch eine separate XML-Datei gespeichert wird. Die Operationen auf den Anwendungsdaten sowie die gesamte Anwendungslogik verteilt sich auf mehrere Java Klassen. Die Klassen, die als Java Beans realisiert sind, unterscheiden sich von den übrigen Klassen (neben den Eigenschaften, die sie als Java Bean mitbringen) darin, dass sie von den Servlets instanziert werden. Damit haben die Servlets als steuernde Komponenten die Möglichkeit in Abhängigkeit der Benutzereingaben über die Methoden der Java Beans die Daten zu manipulieren. Die Methoden der Java Beans stellen somit eine Schnittstelle zwischen den Servlets und den Anwendungsdaten dar. Eine Manipulation der Daten ist aus Sicht der Servlets immer nur in dem Rahmen möglich, wie es durch die öffentlichen Methoden der Beans definiert ist. In Hinblick auf die JSPs geben die (Getter-)Methoden der Beans die Sicht auf die Daten frei, die für die jeweiligen JSPs relevant sind.
Die Java Beans zur Unterstützung des Auscheckens lauten wie folgt:

- *Form.java* - Die Bean implementiert die öffentliche Methode *validate()*, mit der die Benutzereingaben aus der Suchmaske hinsichtlich ihrer Plausibilität untersucht werden können.
- *DatasetQuery.java* - Die Bean enthält Methoden, die das Durchführen einer Suchanfrage in Abhängigkeit der Benutzereingaben ermöglichen.

- *MetaData.java* - Die Methoden dieser Bean ermöglichen es die Meta-Daten eines Datensatzes zu kapseln, um diese in einer JSP anzuzeigen.
- *Transform.java* - Die Bean implementiert Methoden, um Datensätze, bevor sie ausgecheckt werden, in das gewünschte Datei-Format zu transformieren.

Die folgenden Java Beans definieren Methoden zum Einchecken von Datensätzen:

- *CheckInUpload.java* - Die Bean wird verwendet, um die Benutzereingaben der Eincheck-Maske und das Dateiformat der Upload-Datei zu validieren.
- *DatasetCreator.java* - Die Bean implemeniert alle Methoden die beim Integrieren eines neuen Datensatzes in die bestehende Sammlung benötigt werden.

Um den erforderlichen Funktionsumfang ihrer Methoden zu realisieren, verwenden die Java Beans Methoden weitere Java Klassen.
Das Transformieren der gespeicherten Datensätze in eines der unterstützten Dateiformate wird durch eine Reihe von Klassen realisiert, die von der *Transform.java* Bean verwendet werden. Es handelt sich dabei fast ausschließlich um ContentHandler-Klassen, die es ermöglichen während einem Parse-Vorgang der Speicherdatei eines Datensatzes, den Datensatz in eine neue Datei eines anderen Formats zu überführen. Für jedes Zielformat existiert eine spezielle ContentHandler-Klasse. Die nachstehenden Java Klassen realisieren die Transformation von Datensätzen im anwendungsspezifischen Speicherformat in ein anderes Dateiformat:

- *ArffTransformer.java* - Eine ContentHandler Klasse, die beim Parsen eines Datensatzes , den Datensatz in eine Datei im ARFF-Format schreibt.
- *SparseArffTransformer.java* - Eine ContentHandler Klasse, die beim Parsen eines Datensatzes, den Datensatz in eine Datei im sparse ARFF-Format schreibt.
- *XrffTransformer.java* - Eine ContentHandler Klasse, die beim Parsen eines Datensatzes, den Datensatz in eine Datei im XRFF-Format schreibt.
- *SparseXrffTransformer.java* - Eine ContentHandler Klasse, die beim Parsen eines Datensatzes, den Datensatz in eine Datei im sparse XRFF-Format schreibt.
- *C45DataTransformer.java* - Eine ContentHandler Klasse, die beim Parsen eines Datensatzes, den Datensatz in eine *.data Datei im C4.5-Format schreibt.
- *C45NamesTransformer.java* - Eine ContentHandler Klasse, die beim Parsen eines Datensatzes, den Datensatz in eine *.names Datei im C4.5-Format schreibt.
- *C50DataTransformer.java* - Eine ContentHandler Klasse, die beim Parsen eines Datensatzes, den Datensatz in eine *.data Datei im C5.0-Format schreibt.
- *C50NamesTransformer.java* - Eine ContentHandler Klasse, die beim Parsen eines Datensatzes, den Datensatz in eine *.names Datei im C5.0-Format schreibt.
- *CSVTransformaer.java* - Eine ContentHandler Klasse, die beim Parsen eines Datensatzes, den Datensatz in eine Datei im CSV-Format schreibt.
- *TimeFormatChecker.java* - Die Java Klasse stellt Methoden bereit, die ein Attribut vom Typ *date* eines Datensatzes in einen äquivalenten Datumtyp des C5.0-Formats umwandelt.

Die Java Klassen, zum Lesen und Schreiben der XML-Speicherdateien lassen sich zu einer weiteren Gruppe zusammenfassen. Im einzelnen sind das:

- *DatasetDBReader.java* - Diese Klasse stellt Methoden bereit, um bestimmte Informationen eines Datensatzes aus der XML-Datei, die die Meta-Daten der Datensätze enthält, auszulesen.
- *DatasetDBWriter.java* - Die Klasse enthält Methoden, um die Meta-Daten Datei um einen weiteren Eintrag, also die Meta-Daten eines neues Datensatzes, zu ergänzen.
- *DatasetReader.java* - Die Klasse enthält Methoden, um Teile eines Datensatzes aus der entsprechenden XML-Datei zu lesen.
- *DiscreteElementHandler.java* - Die Klasse wird von der DatasetReader Klasse als ContentHandler beim Parsen eines Datensatzes verwendet, um die diskreten Werte eines numerischen Klassenattributs auszulesen.
- *AttributeDeclarationHandler.java* - Die Klasse wird von der DatasetReader Klasse als ContentHandler beim Parsen eines Datensatzes verwendet, um die Attributdeklarationen eines Datensatzes zu lesen.
- *BreakException.java* - Die Klasse implementiert eine Exception und wird einigen ContentHandler Klassen geworfen, um ein Parse-Vorgang vorzeitig abzubrechen.

Die restlichen Java Klassen lauten wie folgt:

- *SortByIndex.java* - Die Klasse definiert eine Vergleichsfunktion, die verwendet werden kann um die Objekte einer Liste zu sortieren. In der Anwendung wird sie verwendet, um eine Liste von XML-Elementen entsprechend des Wertes des *index* Attributs der Elemente zu sortieren.
- *SortByName.java* - Die Klasse definiert eine Vergleichsfunktion, die verwendet werden kann um die Objekte einer Liste zu sortieren. In der Anwendung wird sie verwendet, um eine Liste von XML-Elementen entsprechend des Wertes des *name* Attributs der Elemente zu sortieren.
- *DiscreteChecker.java* - Die Klasse stellt Methoden zur Verfügung, um den Werten eines numerischen Klassenattributs einem der Intervalle zu zuordnen, die den gesamten Zahlenraum des Klassenattributs abbilden.

5.1.4 Datenteilung im MVC Modell

Durch den gegliederten Aufbau der Anwendung in die Funktionsbereiche des MVC Modells, sind Mechanismen notwendig, die eine Datenteilung (d.h. verschiedenen Komponenten arbeiten auf denselben Datenobjekte) zwischen den Komponenten verschiedener Funktionsbereiche ermöglichen. Bezüglich der Anwendung sind die folgenden Problemstellungen relevant:

- die Parameter (Benutzereingaben) aus den Formularfeldern der XHTML-Seiten müssen für das verarbeitende Servlet und den verwendeten Java Beans verfügbar gemacht werden
- bestimmte Eigenschaften von Java Bean Objekten, die durch Servlets erzeugt wurden, müssen für JSPs zugänglich sein
- bei der Verarbeitung einer Anfrage durch mehrere Servlets müssen Daten von einem zum anderen Servlet weitergereicht werden

- Daten müssen über mehrere Requests hinweg verfügbar bleiben

Die Verfügbarkeit von Daten wird im Web-Container über die sogenannte Scope-Objekte geregelt. Es werden die vier verschiedene Scope Objekte unterschieden, wie sie in Abbildung 22 als abstraktes Konzept gezeigt sind.

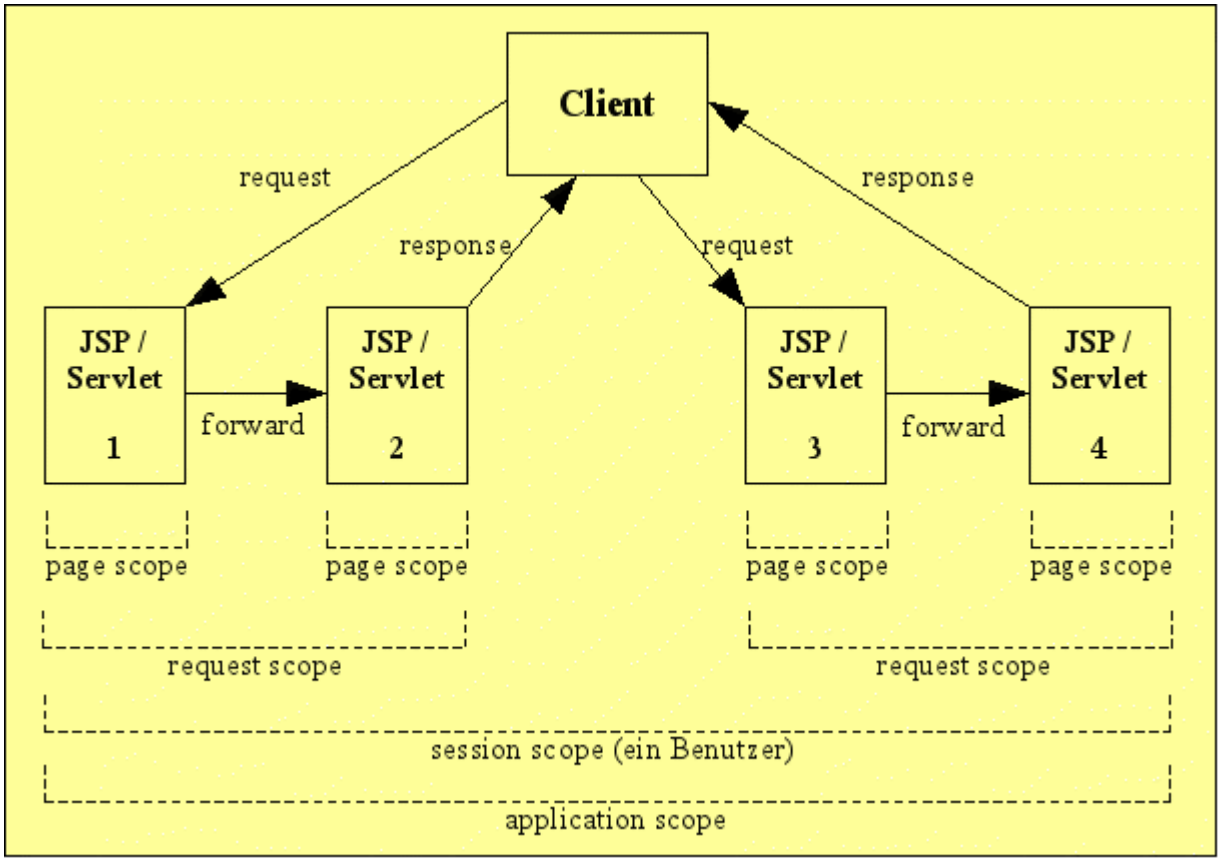

Abbildung 22: Scope-Objekte

Die Datenobjekte, die sich im Page Scope befinden, sind nur innerhalb der JSP bzw. des Servlets verfügbar, in denen sie erzeugt wurden, sodass diese nicht weiter verwendet werden können, wenn eine Anfrage zur Verarbeitung an ein anderes Servlet weitergereicht werden soll. Damit Daten für mehrere Komponenten innerhalb eines Request verfügbar sind, müssen sich diese im Request Scope befinden. In dem Beispielszenario aus Abbildung 22 bearbeiten beispielsweise die Servlets 1und 2 einen Request gemeinsam. Das Weiterleiten (forward) des Client-Requests durch das Servlet 1 an das Servlet 2 (oder JSP 2), stoppt die Ausführung des ersten Servlets und startet die Ausführung des Servlets 2, womit alle Daten, die sich im Page Scope des Servlet 1 befinden, verloren gehen. Diejenigen Daten, die von dem zweiten Servlet weiter verwendet werden sollen, müssen vor der Weiterleitung in den Request Scope gestellt werden, damit diese vom Servlet 2 verwendet werden können. Ein Request Scope und die daran gebundenen Datenobjekte existieren für das Servlet, das den Request entgegennimmt, bis zu dem Servlet (oder JSP), welches die Anwort an den Client sendet. Für eine erneute Anfrage durch den Client stehen die Datenobjekte vorangegangener Request Scopes nicht mehr zur Verfügung. Wie in Abbildung 22 dargestellt, spielt sich der zweite Request (Servlet 3 und 4) in einem separaten Scope ab.
In einigen Fällen besteht die Notwendigkeit Datenobjekte über mehrere Requests hinweg verfügbar zu halten. Datenobjekte im Session Scope sind für alle Requests, die durch den gleichen Browser ausgelöst

wurden, verfügbar (z.B. für Einkaufskorb im On-line Shop), während die Daten im Application Scope über alle Requests hinweg allen Benutzern einer Anwendungs verfügbar sind.
Realisiert sind die verschiedenen Scopes durch Java Objekte, welche in den JSPs und den Servlets verwendet werden können. Mit den *getAttribute()* und *setAttribute()* Methoden dieser Objekte, können Daten in den Scope gestellt werden, den das Objekt repräsentiert, bzw. es kann auf die Datenobjekte, die sich im jeweiligen Scope befinden, zugegriffen werden. Im Rahmen der Anwendung wird die Verfügbarkeit der Daten ausschließlich durch das Request Objekt gewährleistet.
Abbildung 23 zeigt beispielhaft die gemeinsame Verwendung von Datenobjekten durch verschiedene Komponenten der Anwendung.

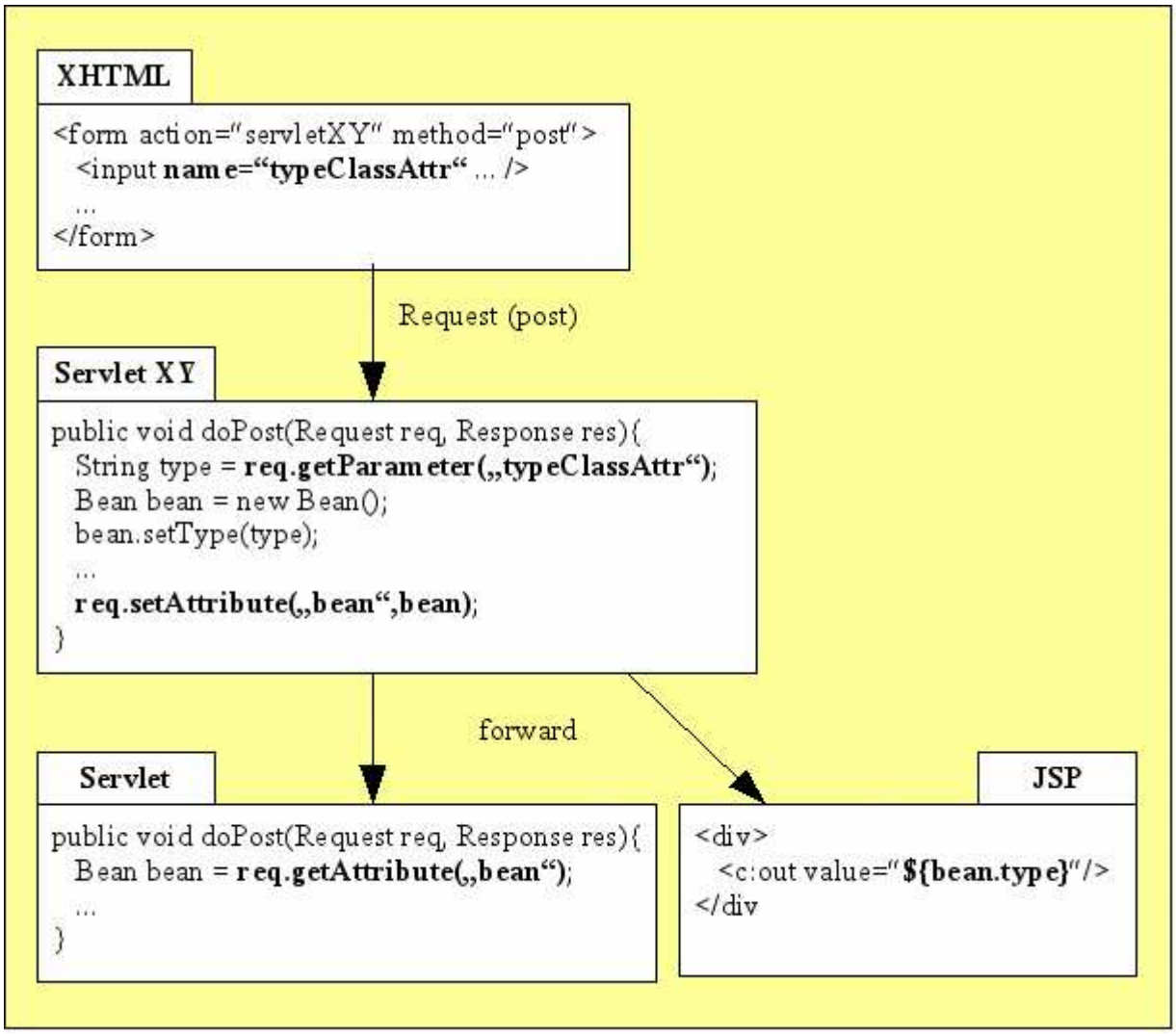

Abbildung 23: Datenteilung durch verschiedene Komponenten

Ausgehend von der XHTML-Seite aus Abbildung 23, die dem Anwender als Antwort auf eine vorangegangene Anfrage zugesendet wurde, stellt der Anwender durch das Abschicken des Formulars eine Anfrage an der Server. Durch die Attribute *action* und *methode* im *form* Element, wird zum einen das Servlet bezeichnet, welches die Anfrage bearbeitet und die HTTP-Methode festgelegt, mit der die Anfrage erfolgt. Im Beispiel wird also ein POST-Request an das Servlet *servletXY* gestellt, wobei mit dem Request die Parameter der Formularfelder gesendet werden. Die Parameter sind im Beispiel die Benutzereingaben des

input Elements mit dem Namen *typeClassAttr*. Entsprechend dem Request wird serverseitig die *doPost()* Methode des Servlets *servletXY* ausgeführt. Einer der Parameter der *doPost()* Methode ist ein Request Objekt, das den Request des Clients mit sämtlichen Parameteren enthält und damit den Request Scope darstellt. Das Request Objekt stellt verschiedene Methoden bereit, um auf die Parameter des Requests zu zugreifen. Die *getParameter()* Methode beispielweise liefert die Parameter als String zurück, wobei die Request Parameter über ihren Namen angesprochen werden.
Werden zur Verarbeitung der Parameter in der Anwendnung Java Beans verwendet, müssen die Parameter für diese verfügbar gemacht werden. Dazu werden von den Servlets Java Bean Objekte erzeugt, welchen anschließend durch ihre Setter-Methoden die Parameter übergeben werden. Im Beispiel aus Abbildung 23 wird das Bean Objekt *bean* erzeugt, dem durch die *setType()* Methode der Parameter *typeClassAttr* übergeben wird. Das Bean Objekt befindet sich somit zunächst im Page Scope. Wird das Bean Objekt zur Bearbeitung des Request noch von weiteren Servlets oder JSPs benötigt, muss es vor der Weiterleitung in den Request Scope gestellt werden. Dafür wird die *setAttribute()* Methode verwendet, wodurch das Bean Objekt als Attribut des Request Objekts gespeichert wird. Durch die Methode wird auch der Name für das Attribut festgelegt, der im Beispiel ebenfalls *bean* ist. Da nach der Weiterleitung (forward) das Request Objekt erhalten bleibt, können die Servlets oder JSPs die den Request entgegennehmen, auf die Attribute des Request Objekts zugreifen, womit ihnen alle Daten im Request Scope zugänglich sind. In Abbildung 23 ist exemplarisch eine Weiterleitung an ein Servlet und an eine JSP dargestellt. Im Falle des Servlets kann auf die Daten im Request Scope durch die *getAttribute()* Methode zugegriffen werden, während in einer JSP mit einem EL-Ausdruck alle verfügbaren Daten direkt über ihren Namen ansprechbar sind. [BERG][J2EETut04]

5.2 Programm-Ablauf-Szenarien

Aus Sicht des Benutzers stellt die Anwendung zwei Dienste bereit: das Auschecken und das Einchecken von Datensätzen. Der interne Programmablauf und das Zusammenspiel der Anwendungskomponenten beim Verwenden dieser Dienste durch den Benutzer wird folgend beschrieben.

5.2.1 Auschecken von Datensätzen

Das Auschecken von Datensätzen erfolgt schrittweise durch eine Interaktion des Benutzers mit der Anwendung. In Abbildung 24 sind die clientseitigen Aktionen einerseits, und der daraus resultierende Programmablauf mit seinen wesentlichen Komponenten auf der Serverseite andererseits, dargestellt. Links sind die Aktionen des Benutzers abgebildet und auf der rechten Seite der Abbildung sind die einzelnen Komponenten der Anwendung zu sehen.
Der Vorgang des Auschecken von Datensätzen wird durch den Client eingeleitet, indem dieser die Suchmaske vom Server anfordert. Dieser Request wird von dem Servlet *GetForm* entgegengenommen und ohne weitere Verarbeitung an die JSP *Form* weitergeleitet. Die JSP Form generiert das XHTML-Dokument, das die Suchmaske darstellt und als Antwort an den Client gesendet wird. Auf der Clientseite kann der Benutzer nun in der Suchmaske Angaben zu den Eigenschaften der gesuchten Datensätze machen und eine Suchanfrage starten, indem er einen Request aus der Suchmaske heraus auslöst (also das HTML-Formular abschickt). Bei dem Request handelt es sich um einen Post-Request, der alle Benutzereingaben der Suchmaske enthält und durch das Servlet *ValidateForm* bearbeitet wird. Dieses Servlet erzeugt zunächst ein Objekt der Bean-Klasse *Form* und übergibt die Parameter des Request an das Bean-Objekt unter Verwendung der Setter-Methoden der *Form* Bean. Anschließend veranlasst das Servlet die Validierung der Request-Parameter durch Aufruf einer entsprechenden Methode des Bean-Objekts.

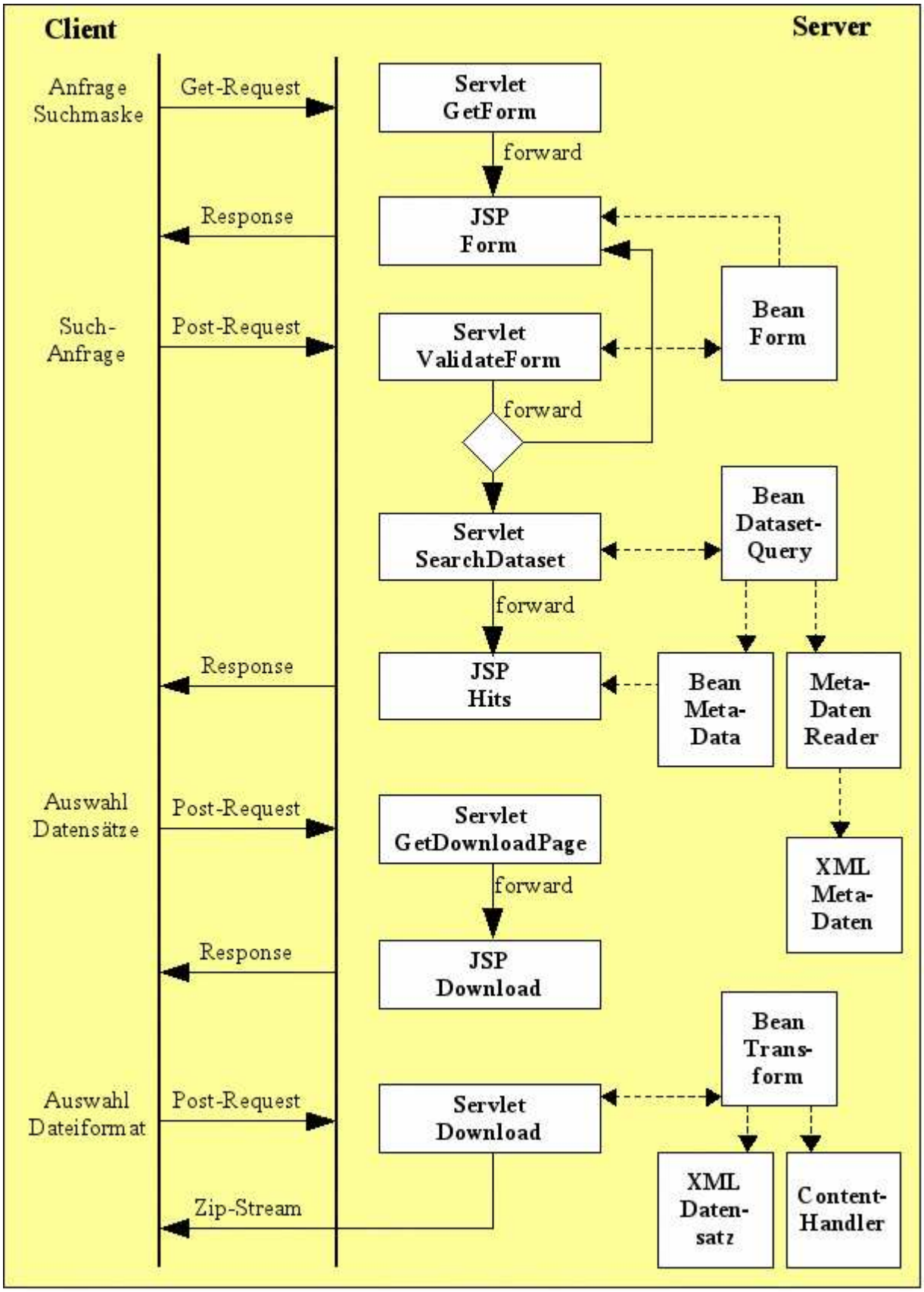

Abbildung 24: Programmablauf beim Auschecken

Die Methode überprüft die Parameter und erzeugt gegebenenfalls eine Fehlermeldung, wenn eine Benutzereingabe nicht im erwarteten Wertebereich liegt (z.B.: Parameter, der eine Zahl sein muss, enthält Buchstaben). Das Vorhandensein von fehlerhaften Benutzereingaben zeigt das Bean-Objekt durch eine Klassenvariable an, welche von dem Servlet *ValidateForm* nach der Validierung ausgewertet wird, um den weiteren Programmablauf zu steuern. Sind die Benutzereingaben fehlerhaft wird der Request zurück an die JSP *Form* geleitet, die erneut die Suchmaske generiert, wobei diesmal die Fehlermeldungen des Bean-Objekts in die XHTML-Seite integriert werden. Somit kann der Benutzer die fehlerbehafteten Angaben korrigieren und eine neue Suchanfrage starten. In dem Fall, dass die Eingaben korrekt sind, wird der Request an das Servlet *SearchDataset* weitergeleitet. In beiden Fällen wird das Bean-Objekt durch das *ValidateForm* Servlet vor der Weiterleitung in den Request Scope gestellt.
Das Servlet *SearchDataset* hat die Aufgabe eine Datensatzsuche, auf Basis der im *Form* Bean-Objekt gespeicherten Benutzereingaben, zu initiieren. Dazu erzeugt das Servlet ein Objekt der Bean-Klasse *DatasetQuery*, übergibt diesem das *Form* Bean-Objekt aus dem Request Scope und ruft die Methode *exeQuery()* des *DatasetQuery* Objekts auf, die die Suchanfrage durchführt. Der Kern dieser Methode besteht in der Erzeugung eines XPath-Ausdrucks, der die Datensätze aus der XML-Datei für die Meta-Daten auswählt. Der eigentliche Zugriff auf die XML-Datei erfolgt mit Hilfe einer *Reader* Klasse, die diverse Methoden für den lesenden Zugriff auf die Meta-Daten Datei enthält. Für die Durchführung einer Suchanfrage existiert eine Methode, die einen XPath-Ausdruck auf die Meta-Daten Datei anwendet und eine Liste von XML-Elementen zurückgibt, die durch den XPath-Ausdruck ausgewählt werden. Nachdem die ausgewählten XML-Elemente, die die Eigenschaften der durch die Suchanfrage gefundenen Datensätze abbilden, an die aufrufende Stelle zurückgeliefert worden sind, werden die Inhalte der XML-Elemente jeweils in einem separaten *Meta-Data* Bean-Objekt gekapselt. Anschließend stellt das *SearchDataset* Servlet die *DatasetQuery* Bean und damit auch alle *Meta-Data* Beans in den Request Scope und leitet den Request weiter an die JSP *Hits*. Die JSP erzeugt durch Zugriff auf die *Meta-Data* Bean eine XHTML-Seite, die alle gefundenen Datensätze mit deren Eigenschaften auflistet. Die XHTML-Seite wird dann an den Client gesendet, um ihm die Suchtreffer anzuzeigen, aus denen der Benutzer diejenigen Datensätze auswählen kann, die er gerne auschecken möchte.
Auf der Clientseite stellt der Benutzer eine erneute Anfrage in Form eines Post-Requests, indem er das Formular mit den ausgesuchten Datensätzen abschickt. Das Servlet *GetDownloadPage* nimmt die Anfrage entgegen und leitet sie weiter an die JSP *Download*. Diese JSP erzeugt eine XHTML-Seite, auf der der Benutzer das Dateiformat, in dem die Datensätze zum Download bereitgestellt werden sollen, auswählen kann. Außerdem wird dem XHTML-Dokument die Namen der Datensätze, die im Request enthalten sind, hinzugefügt. Die Datensatznamen werden, für den Benutzer nicht sichtbar, in dem Formular zur Auswahl des Dateiformats eingefügt, damit diese als Parameter des folgenden Requests erneut an den Server gesendet werden. Auf diese Weise werden die Namen der auszucheckenden Datensätze von einem Request Scope in den nächsten Request Scope übernommen.
Nach der Auswahl des Dateiformats und dem Abschicken des Formulars durch den Benutzer, werden also neben dem Dateiformat auch die Namen der Datensätze als Parameter mit dem Post-Request an den Server geschickt. Dort wird der Request von dem Servlet *Download* verarbeitet. Bevor die Datensätze zum Download bereitgestellt werden können, müssen diese in das gewünschte Dateiformat transformiert werden. Dazu erzeugt das *Download* Servlet ein Objekt der Bean-Klasse *Transform* und übergibt diesem das Dateiformat. Danach veranlasst das Servlet für jeden Datensatz die Transformation in das angegebene Dateiformat unter Verwendung des *Transform* Objekts. Dieses wählt dem Dateiformat entsprechend ein ContentHandler aus und registriert diesen bei einem SAX-Parser, der die XML-Datei des Datensatzes, im Anschluß parst. Die durch den ContentHandler erzeugte Datei mit dem Datensatz im Zielformat wird an das Servlet *Download* geliefert. Nach Abschluss der Transformation aller Datensätze, erzeugt das

Servlet ein Zip-Stream, der sämtliche ausgewählten Datensätze im angeforderten Format enthält, sodass der Benutzer die Datensätze entgegennehmen kann.

5.2.2 Einchecken von Datensätzen

Der Ablauf beim Einchecken ist in Abbildung 25 analog der Darstellung der Abfolge beim Auschecken aus Abbildung 24 gezeigt.
Zum Einchecken von Datensätzen fordert der Benutzer auf der Clientseite zunächst, die Oberfläche zum Integrieren neuer Datensätze beim Server an. Den Request nimmt das *GetCheckInUpload* Servlet entgegen und leitet diesen weiter zur JSP *CheckInUpload*. Die von der JSP erzeugte XHTML-Seite enthält ein Formular, welches ein Eingabefeld (input Element) besitzt, das dem Benutzer ermöglicht eine Datei von seinem lokalen Rechner zusammen mit dem Formlur an den Server zu übertragen. Diese Datei enthält den einzucheckenden Datensatz und muss dem ARFF- oder dem sparse ARFF-Format entsprechen. Durch das Abschicken des Formulars durch den Client, wird ein Post-Request ausgelöst, der durch das Servlet *ValidateUpload* verarbeitet wird. Das Servlet erzeugt einen Input-Stream für den Zugriff auf die im Formular angegebenen Upload-Datei und speichert diese vorläufig für die weitere Verarbeitung ab. Außerdem erzeugt das *ValidateUpload* Servlet ein Objekt der Bean-Klasse *CheckInUpload*, übergibt diesem sowohl die Upload-Datei, als auch alle anderen Parameter des Requests. Die Aufgabe der *CheckInUpload* Bean besteht darin, sämtliche Benutzereingaben zu validieren und gegebenenfalls entsprechende Fehlermeldungen zu erzeugen. Dazu zählt die Überprüfung der Upload-Datei auf Konformität mit dem ARFF-Format. Außerdem ermittelt das Bean-Objekt während dem Validierungsvorgang die Meta-Daten des Datensatzes, der in der Upload-Datei gespeichert ist, und legt diese in die dafür vorgesehende Klassenvariablen ab. Die notwendigen Zugriffe auf die Meta-Daten XML-Datei beim Validieren, erfolgt durch ein Objekt der speziell dafür vorgesehenen *Reader* Klasse. Nach Abschluss des Validierungsprozesses leitet das *ValidateUpload* Servlet den Request entweder an das Servlet *CreateNewDataset* weiter, oder zurück an die *CheckInUpload* JSP. Im ersten Fall waren alle Benutzereingaben korrekt und die Meta-Daten des Datensatzes konnten ermittelt werden, während im zweiten Fall der Benutzer zur Korrektur seine Eingaben aufgefordert werden muss, indem die Oberfläche zum Einchecken mit Einbindung der Fehlermeldungen erneut zum Client gesendet wird.
Das Servlet *CreateNewDataset* erzeugt ein Objekt der Bean-Klasse *DatasetCreator* und übergibt diesem, das im Request Scope befindliche *CheckInUpload* Bean-Objekt, welches neben den Meta-Daten die Referenz auf die Upload-Datei kapselt. Anschließend veranlasst das Servlet die *DatasetCreator* Bean den Datensatz der Upload-Datei in das anwendungsspezifische Speicherformat zu transferieren und in der Meta-Daten Datei für den neuen Datensatz einen entsprechenden Eintrag zu machen. Handelt es sich bei dem einzucheckenden Datensatz um einen Testdatensatz, überprüft die *DatasetCreator* Bean, ob dieser mit dem zugrunde liegenden Datensatz kompatibel ist. Das erfordert einen lesenden Zugriff auf einen bereits vorhandenen Datensatz, den die Bean unter Verwendung der Methoden einer speziellen Reader Klasse durchführt. Sind der Testdatensatz und der dazugehörige Datensatz nicht kompatibel, so erzeugt die *DatasetCreator* Bean Fehlermeldungen und legt diese in entsprechende Klassenvariablen ab. Das Lesen und Schreiben der Meta-Daten Datei durch die *DatasetCreator* Bean erfolgt jeweils über Objekte spezieller Reader- bzw. Writer-Klassen. Das Schreiben des Datensatzes im anwendungsspezifischen Speicherformat wird durch die *DatasetCreator* Bean selbst durchgeführt.
Nach der erfolgreichen Integration des neuen Datensatzes bzw. nach einer Festellung von Inkompatibilität des einzucheckenden Testdatensatzes, wird der Request von dem Servlet *CreateNewDataset* an die JSP *CheckInConfirm* weitergeleitet.

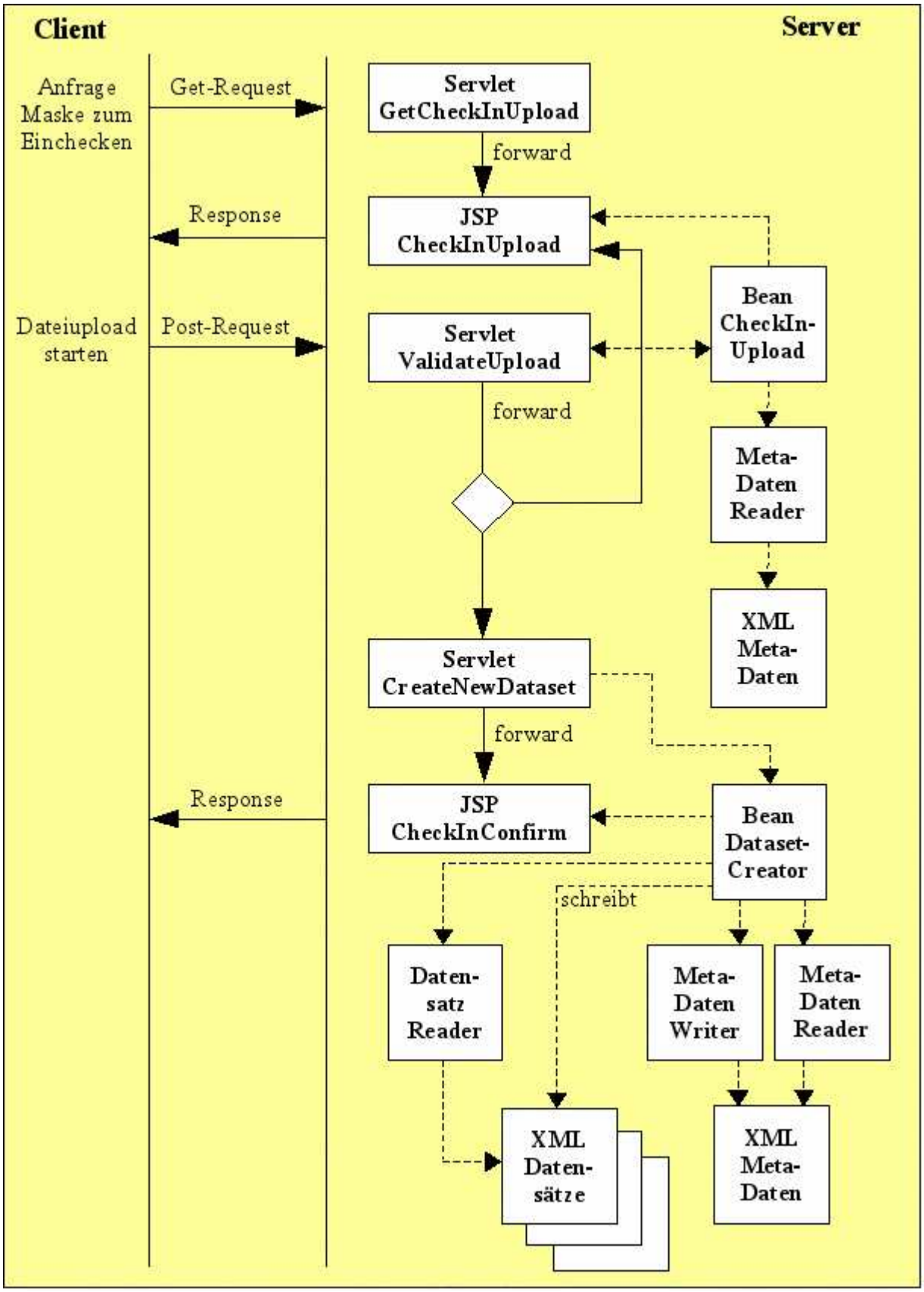

Abbildung 25: Programmablauf beim Einchecken

Diese erzeugt die Antwort XHTML-Seite, die entweder eine Bestätigung über das erfolgreiche Einchecken oder die Fehlermeldungen der *DatasetCreator* Bean bezüglich der Inkompatibilität, enthält.

6 Implementierung der Anwendung

In den folgenden Kapiteln wird die Implementierung der wesentlichen Komponenten der Web-Anwendung beschrieben. Ein Übersicht der Java Klassen der Anwendung ist in Abbildung 26 in Form eines Klassendiagramms zu sehen.
Die Klassen verteilen sich auf die drei Pakete servlets, beans und helpers. Im Paket helpers sind alle Klassen zusammen gefasst, die die Java Beans verwenden, um die Anwendungslogik zu realisieren, jedoch nicht über die Eigenschaften einer Java Bean verfügen. Die Pfeile mit durchgehender Linie beschreiben eine "benutzt" Beziehung, d.h. ein Objekt einer Klasse benutzt eine andere Klasse, wenn sie ein Objekt der anderen Klasse erzeugt, um deren Methoden zu nutzen. Ein Pfeil mit gestrichelter Linie zwischen zwei Klassen zeigt an, dass die Klasse, von der der Pfeil ausgeht, ein Klasseattribut vom Typ der anderen Klasse besitzt.

6.1 Verwendete Software

Die Basis aller Java Klassen der Web-Anwendung bilden die APIs der Java SE [J2SE] in der Version 1.5. Darauf aufbauend kommen die folgenden Java APIs zum Einsatz:

- Die Servlets der Anwendung erweitern die Java Klasse *javax.servlet.http.HttpServlet* der Servlet API in der Version 2.5, die Teil der Java EE ist. [JAVAX]
- Bei der Verarbeitung von XML-Dateien und der Erzeugung von XML-Strukturen werden in der Anwendung Klassen der JDOM API 1.0 verwendet. JDOM ist eine Java Repräsentation eines XML-Dokuments. Die API enthält Klassen, für das Lesen, Schreiben und Manipulieren von XML-Dateien, wobei die Bestandteile eines XML-Dokuments als Java Objekte abgebildet werden. Unter anderem stellt die API Klassen und Methoden bereit, um Knoten eines XML-Dokuments mit Hilfe von XPath-Ausdrücken zu selektieren. [JDOM]
- In der Anwendung werden SAX- und DOM-Parser des Apache Projekts Xerces zum Parsen von XML-Dokumenten in der Version 2.9 verwendet. [XERC]
- Das Einchecken von Datensätzen im Arff-Format wird durch Klassen der Weka-Software in der Version 3.5.6 unterstützt. [WEKA]
- Der Datei-Upload beim Einchecken von Datensätzen wird mit Hilfe einer Datei-Upload und einer IO API des Apache Projekts Commons realisiert. Die Upload API stellt Klassen und Methoden bereit, mit denen eingegehende Upload-Requests geparst werden können, um die einzelnen Bestandteile des Requests nach Bedarf zu verarbeiten (Upload-Dateien des Requests als Stream verarbeiten). In der Anwendung wird die Upload API in der Version 1.2 benutzt. Beim Datei-Upload kommt außerdem eine Methode der IO API (Version 1.3) des Apache Projekts Commons zum Einsatz. [FILE][IO]
- Beim Schreiben der XML-Dateien für neu eingecheckte Datensätze, bzw. das Erzeugen von Dateien im XRFF-Format erfolgt mit Hilfe der DataWriter Klasse der Megginson Technologies XMLWriter API. Mit dieser Klasse kann ein XML-Dokument von einem SAX Ereignis Strom geschrieben werden. In der Anwendung kommt die Klasse in der Version 0.2. [MEGG]

Die JSPs, die in der Anwendung zum Generieren von XHTML-Dokumenten verwendet werden, um diese an den Cient als Antwort auf eine Anfrage zu senden, bauen auf den JSP APIs in der Version 2.1 auf, welche den Einsatz von EL-Ausdrücken unterstützen. In den JSPs der Anwendung kommen Aktionselemente der JSTL Version 1.2 zum Einsatz. [JSP21][JSTL]

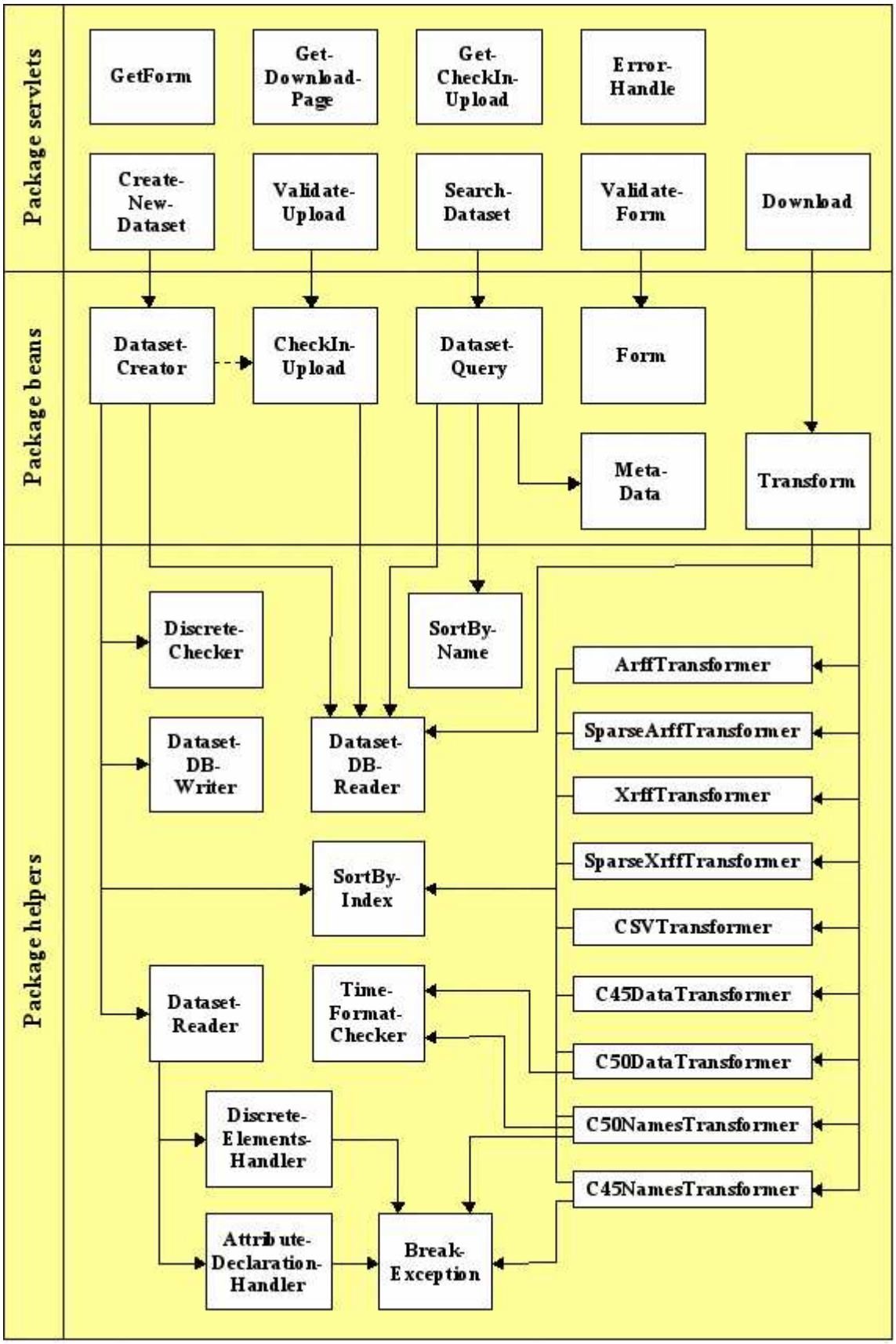

Abbildung 26: Klassendiagramm der Web-Anwendung

Die durch die JSP erzeugten Benutzeroberflächen der Anwendung sind valide XHTML-Dokumente (Version 1.0), deren Darstellung im Browser durch Cascading Style Sheets des Level 2 beschrieben ist. Außerdem kommt die clientseitige Sprache JavaScript zum Einsatz, um die HTML-Elemente einer Oberfläche zum Zweck der Bedienbarkeit zu manipulieren. Der Zugriff auf die Elemente erfolgt über die DOM Level 2 Schnittstellen. [XHTML][CSS][DOM2][JSCR]
Für die korrekte Anzeige und die vollständige Nutzung der Bedienelemente der Oberflächen, wird also ein JavaScript-fähiger Web-Browser mit ausreichender Unterstützung des DOM Level 2 und der CSS Level 2 Spezifikation benötigt.
Als Laufzeitumgebung der Anwendung wurde der Apache Tomcat Server in der Version 6.0 verwendet, der einen Web-Container für die Ausführung von JSPs und Servlets enthält. Der Web-Container des Servers in dieser Version implementiert die Servlet Spezifikation 2.5 und die JSP Spezifikation 2.1. [TOM]

6.2 Speicherformate

Die dauerhaft zu speichernden Daten der Anwendung, also die Datensätze und die Meta-Daten der Datensätze, werden in XML-basierten Formaten gespeichert. Jeder Datensatz (und Testdatensatz) ist in einer separaten XML-Datei gespeichert. Die Meta-Daten aller Datensätze sind hingegen in einer XML-Datei abgelegt. Es existieren also zwei verschiedenen Speicherformate, deren Struktur und Elemente jeweils durch ein XML-Schemata definiert sind.

6.2.1 Datensätze

Das XML-Format zur Speicherung der Datensätze ist in Abbildung 27 exemplarisch dargestellt.
Sämtliche Typdefinitionen und Elementdeklarationen, denen eine anwendungsspezifische Datensatzdatei zugrunde liegt, sind an den Namensraum *http://www.thomaswanderer.de/datasetSchema* geknüpft und sind durch das XML-Schema *datasetSchema.xsd* definiert.
Das Wurzelelement einer Datensatzdatei ist das *dataset* Element, das in seinem *xsi:schemaLocation* Attribut den Verweis auf das XML-Schema enthält, auf das sich die Datensatzdatei bezieht. Die Kindelemente des *dataset* Elements sind die Elemente *docu*, *header*, und *data*. Während beliebig viele *docu* Elemente, zur Aufnahme von Kommentaren und Dokumentation zum Datensatz, auftreten können, muss ein *dataset* Element genau ein *header* und ein *data* Element besitzen. Das *header* Element dient zur Aufnahme der Beschreibung des Datensatzes, während das *data* Element die eigentlichen Daten, also die Instanzen enthält. Das *header* Element besitzt die zwei Elemente *relation* und *attributes*. Im *relation* Element befindet sich der Name des Datensatzes als String und das *attributes* Element ist ein Container für sämtliche Attributdeklarationen des Datensatzes, welches beliebig viele *attribute* Elemente besitzen kann. Ein Attribut eines Datensatzes wird jeweils durch ein *attribute* Element beschrieben. Im zugrunde liegenden XML-Schema ist ein *attribute* Element als abstrakter Typ definiert, der in Abhängigkeit des darzustellenden Attributtyps des Datensatzes erweitert ist. In der Datensatzdatei aus Abbildung 27 ist für jeden der sieben Attributtypen, wie sie in Kapitel 3.1.1 (Abbildung 11) unterschieden werden, ein Beispiel für ein entsprechendes *attribute* Element gegeben.
Zunächst besitzt jedes *attribut* Element, unabhängig vom darzustellenden Attributtyp, die Attribute *name*, *index*, *class* und *weight*. Das *name* Attribute gibt den Namen eines Datensatzattributs an und ist vom Typ ID[11], d.h. der Wert eines *name* Attributs eines *attribute* Elements ist im gesamten Dokument einmalig.

[11] Der Typ ID ist einer der im XML-Schema Standard definierten Datentypen für Attribute. Der Wert eines Attributs vom Typ ID darf im gesamten Instanzdokument nur einmal vorkommen, damit dieses valide ist.

```
<dataset xmlns="http://www.thomaswanderer.de/datasetSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.thomaswanderer.de/datasetSchema ../../xsd/datasetSchema.xsd">
  <docu> Dokumentation, Kommentare bezüglich des Datensatzes</docu>
  <header>
    <relation>Name des Datensatzes</relation>
    <attributes>
      <attribute name="attr1" index="0" type="string" numValues="4" xsi:type="stringAttribute"/>
      <attribute name="attr2" index="1" type="nominal" weight="0.9" xsi:type="nominalAttribute">
        <labels numLabels="2">
          <label>ja</label>
          <label>nein</label>
        </labels>
      </attribute>
      <attribute name="attr3" index="2" class="yes" type="numeric" xsi:type="numericAttribute">
        <discretize>
          <discrete>'(-inf-38.3]'</discrete>
          <discrete>'(38.3-75.7]'</discrete>
          ...
        </discretize>
      </attribute>
      <attribute name="attr4" index=3" type="relational" xsi:type="relationalAttribute">
        <attributes>
          <attribute name="attr40" index="0" type="real" xsi:type="numericAttribute">
          <attribute name="attr41" index="1" type="integer" xsi:type="numericAttribute">
        </attributes>
      </attribute>
      <attribute name="attr5" index="4" type="date" numValues="7" xsi:type="dateAttribute">
        <dateFormat>yyyy-MM-dd</dateFormat>
      </attribute>
      <attribute name="attr6" index="5" type="c50Label" xsi:type="c50LabelAttribute"/>
      <attribute name="attr7" index="6" type="formula" xsi:type="formulaAttribute">
        <formula>attr3 + 10</formlua>
      </attribute>
    </attributes>
  </header>
  <data>
    <instance>
      <value ref="attr1">auto</value>
      <value ref="attr2">ja</value>
      <value ref="attr3">40.6</value>
      <value ref="attr3" discrete="true">'(38.3-75.7]'</value>
      <value ref="attr4">
        <instance>
          <value ref="attr40">4.5</value>
          <value ref="attr41">23</value>
        </instance>
      </value>
      <value ref="attr5">2008-01-12</value>
      <value ref="attr6">6376xy</value>
    </instance>
  </data>
</dataset>
```

Abbildung 27: XML-Speicherformat für Datensätze

Der Index eines Datensatzattributs wird durch das *index* Attribut festgelegt, woraus sich die Reihenfolge der Attributwerte in den Instanzen des Datensatzes ergibt. Die Kennzeichnung eines Datensatzattributs

als Klassenattribut, erfolgt durch das *class* Attribute, das den Wert *yes* oder *no* haben kann. Da der Defaultwert dieses Attributs *no* ist, bedeutet die Abwesentheit des Attributs, das der Wert *no* ist und somit das betreffende Attribut nicht das Klassenattribut ist. Für die Gewichtung eines Datensatzattributs steht das Attribut *weight* zur Verfügung, das Werte von 0 bis 1 annehmen kann. Der Defaultwert ist 1.0. Das *xsi:type* Attribut eines *attribut* Elements gibt dessen Typ an. Die Bezeichnung der verschiedenen Typen lehnt sich an den darzustellenden Datensatzattributtyp an. Das erste *attribut* Element der Datensatzdatei aus Abbildung 27 stellt beispielsweise ein Datensatzattribut vom Typ string dar, was durch sein *type* Attribut mit dem Wert *string* ausgedrückt ist. Der Typ des *attribut* Elements zur Darstellung von Datensatzattributen des Typs string, trägt die Bezeichnung *stringAttribute*. In analoger Weise erfolgt die Bezeichnung aller anderer Elementtypen.Neben einem *type* Attribut mit dem Wert *string* besitzt ein *attribut* Element vom Typ *stringAttribute* ein *numValues* Attribut. Dieses enthält die Anzahl der verschiedenen vorkommenden Werte des Datensatzattributs in den Instanzen.
Ein *attribute* Element zur Beschreibung nomineller Attribute besitzt ein *type* Attribut mit dem Wert *nominal* und das Kindelement *labels*, welches die zwei Attribute *numValues* und *ordered* besitzt. Das *numValues* Attribut gibt die Anzahl der verschiedenen möglichen Werte des nominellen Datensatzattributes an und das *ordered* Attribut zeigt an, ob die Werte des nominellen Attributs hierarchisch geordnet sind. Da der Default-Wert des *ordered* Attributs *no* ist, ist dieses Attribute im *labels* Element aus dem Beispiel in Abbildung 27 nicht explizit angegeben. Für die Darstellung der möglichen Werte, die das nominelle Datensatzattribut annehmen kann, besitzt das *labels* Element zwei *label* Elemente. Da die Gewichtung des nominellen Attributs aus dem Beispiel von dem Standardwert 1 abweicht, ist die Gewichtung durch ein *weight* Attribut explizit angegeben.
Anschließend folgt im Beispiel die Deklaration eines numerischen Attributs, die durch ein *attribute* Element vom Typ *numericAttribute* repräsentiert ist. Solche *attribute* Elemente müssen ein *type* Attribut mit dem Wert *numeric*, *real* oder *integer* haben. Da das beschriebene Attribut das Klassenattribut des Datensatzes ist, enthält das *attribute* Element ein *class* Attribut mit dem Wert *yes*. Zur Speicherung von diskreten Werten für die Abbildung des Wertebereichs numerischer Attribute (für die Darstellung des Datensatzes im C5.0- und C4.5-Format), darf ein *attribute* Element vom Typ *numericAttribute* ein *discretize* Element besitzen, welches wiederum für jeden diskreten Wert ein *discrete* Element als Kind hat. Der Wert eines *discrete* Elements ist ein String, der einen bestimmten Zahlenintervall repräsentiert. *Attribute* Elemente vom Typ *relationalAttribute* haben ein *type* Attribut mit dem Wert *relational* und beschreiben relationale Datensatzattribute. Außerdem besitzen diese wiederum ein *attributes* Element als Container für die Attribute des relationalen Attributs. Die *attribute* Elemente dieses *attributes* Elements entsprechen ebenfalls wieder einem für die *attribute* Elemente vorgesehenen Typ. Im Beispiel aus Abbildung 27 sind das zwei *attribute* Elemente vom Typ *numericAttribute*.
Zur Beschreibung von Datensatzattributen des Typs date, steht das *attribute* Element vom Typ *dateAttribute* zur Verfügung mit dem Wert *date* im *type* Attribute. Dieses *attribute* Element besitzt auch ein *numValues* Attribute, das die Anzahl aller Werte des Datensatzattributs in den Instanzen beinhaltet. Das Kindelement *dateFormat* nimmt das Datumsformat auf, in dem die Attributwerte in den Instanzen vorkommen. Der String, der das Datumsformat abbildet, entspricht den Regeln der Java Klasse SimpleDateFormat zur Formatierung von Zeitangaben. Das Attribute namens *attr5* aus Abbildung 27 ist ein Beispiel für ein *attribute* Element vom Typ *dateAttribute*.
Attribute des C5.0 spezifischen Typs label werden durch *attribute* Elemente des Typs *c50LabelAttribute* beschrieben, welche zusätzlich zu den allgemeinen Attributen nur ein *type* Attribute mit dem Wert *c50Label* besitzen. Der ebenfalls C5.0 spezifische Attributtyp Formel wird durch *attribute* Elemente vom Typ *formulaAttribute* unterstützt. Neben dem Attribut *type* mit dem Wert *formula*, besitzt ein solches *attribute* Element ein *formula* Kindelement, das zur Aufnahme der Formel, durch die das Datensatzat-

tribut definiert ist, dient. Die Attributbeschreibungen für die Attribute namens *attr6* und *attr7* aus der Datensatzdatei in Abbildung 27 zeigen jeweils ein Beispiel für die C5.0 spezifischen Attributtypen.
Im Anschluss an die Attributbeschreibungen folgt das *data* Element, das die Instanzen des Datensatzes beinhaltet. Ein *data* Element kann beliebig viele *instance* Elemente besitzen, wobei jedes *instance* Element eine Instanz des Datensatzes repräsentiert. Ein *instance* Element besitzt wie ein *attribute* Element ein *weight* Attribute mit dem Default-Wert 1.0, um einzelnen Instanzen zu gewichten und eine Menge von *value* Elementen. Die *value* Elemente beinhalten die Attributwerte der Instanzen. Jedes *value* Element besitzt ein *ref* Attribut vom Typ IDREF[12], sodass der Inhalt eines *value* Elements, also ein Wert einer Instanz, eindeutig einem Datensatzattribut zugewiesen werden kann. Durch diese Referenz und dem Vorhandensein des *index* Attributs im *attribute* Element, ist die Reihenfolge der Attributwerte in den Instanzen des Datensatzes festgelegt, unabhängig von der Reihenfolge mit der die *attribute* und *value* Elemente in der Datensatzdatei auftreten. Mit Ausnahme von relationalen Datensatzattributen oder numerischen Klassenattributen, existiert für jedes *attribute* Element ein *value* Element innerhalb eines *instance* Elements, das den Wert des entsprechenden Datensatzattributs abbildet. Zur Darstellung der Werte von relationalen Attributen, besitzt ein *value* Element ein *instance* Element, welches wiederum eine Menge von *value* Elementen enthält, die in analoger Weise die Werte der Attribute des relationalen Attributs darstellen. Die Werte numerischer Klassenattribute werden auf zwei *value* Elemente abgebildet, wobei das eine den tatsächlichen Wert des Attributs enthält und das andere den Zahlenintervall repräsentiert, in den der tatsächliche Wert fällt. Zur Unterscheidung der beiden Elemente besitzt das *value* Element, das den Zahlenintervall repräsentiert, ein *discrete* Attribute mit dem Wert *true*.
Der Datensatz aus Abbildung 27 besitzt nur eine Instanz, die für jedes Attribut einen Wert besitzt, wobei die Werte von Attributen des Typs *formula*, wie auch im C5.0-Format, nicht in den Instanzen aufgeführt werden. Fehlende Werte in den Instanzen, werden in diesem Speicherformat durch leere *value* Elemente angezeigt. Die Abwesentheit eines *value* Elements bedeutet, dass das dazugehörige Datensatzattribut in der entsprechenden Instanz den Wert 0 hat.

6.2.2 Meta-Daten

Das XML-Format zur Speicherung der Meta-Daten der Datensätze ist durch das XML-Schema *datasetDBSchema.xsd* definiert, welches alle darin definierten Elemente und Attribute an den Namensraum *http://www.thomaswanderer.de/datasetDBSchema* bindet. Folglich verweist das *xsi:schemaLocation* Attribut des *datasetDB* Elements der Meta-Daten Datei aus Abbildung 28 auf das XML-Schema für den erwähnten Namensraum.
Das Wurzelelement datasetDB der Meta-Daten Datei besitzt für jeden vorhandenen Datensatz ein dataset Element, welches sämtliche Meta-Daten eines Datensatzes aufnimmt. Die Kindelemente eines *dataset* Elements lauten: *name*, *descritption*, *uri*, *attributes*, *classAttribute*, *instances*, *missingValues* und *testSet*. Das *name* Element enthält den Namen des Datensatzes und das *description* Element enthält eine kurze Beschreibung des Datenssatzes, sofern eine solche vorhanden ist. Existiert keine Beschreibung bleibt das *description* Element leer. Das Element *uri* nimmt den Namen der XML-Datei auf, in der der betreffende Datensatz gespeichert ist. Der Name der Datei, wie auch der Name des Datensatzes, ist innerhalb der gesamten Anwendung einmalig, sodass eine eindeutige Zuordung zwischen Datensatz und dazugehöriger Datensatzdatei möglich ist. Das *attributes* Element besitzt vier Kindelemente, in denen die Eigenschaften bezüglich der Attribute des Datensatzes abgelegt sind: Das Element *numAll* beinhaltet die Anzahl aller Attribute des Datensatzes, das Element *numNumeric* beinhaltet die Anzahl der numerischen Attribute, das Element *numNominal* enthält die Anzahl der nominellen Attribute und das Element *numBoolean*

[12] Der Typ IDREF ist einer der im XML-Schema Standard definierten Datentypen für Attribute. Der Wert eines Attributs vom Typ IDREF muss dem Wert eines Attributs vom Typ ID im gleichen Dokument entsprechen.

```
<datasetDB xmlns="http://www.thomaswanderer.de/datasetDBSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.thomaswanderer.de/datasetDBSchema ../xsd/datasetDBSchema.xsd">

  <dataset>
    <name>beispiel</name>
    <description>kurze Beschreibung des Datensatzes</description>
    <uri>beispiel.xml</uri>
    <attributes>
      <numAll>9</numAll>
      <numNumeric>8</numNumeric>
      <numNominal>1</numNominal>
      <numBoolean>0</numBoolean>
    </attributes>
    <classAttribute>
      <type>nominal</type>
      <numValues>5</numValues>
      <classDist>56.8</classDist>
    </classAttribute>
    <instances>4177</instances>
    <missingValues>10.1</missingValues>
    <testSet numTestSets="2">
      <uri>testset1-beispiel.xml</uri>
      <uri>testset2-beispiel.xml</uri>
    </testSet>
  </dataset>
  ...

</datasetDB>
```

Abbildung 28: XML-Datei zur Speicherung der Meta-Daten

enthält die Anzahl der booleschen Attribute des Datensatzes.
Die Eigenschaften hinsichtlich des Klassenattributs werden durch die drei Kindelemente *type*, *numValues* und *classDist* des Elements *classAttribute* repräsentiert. Das Element *type* speichert den Typ des Klassenattributs. Mögliche Inhalte dieses Elements sind: *boolean*, *nominal*, *ordinal*, *integer*, *real*, *numeric*, *string*, *date*, *relational* und *none*. Obwohl das maßgebende XML-Schema hier die Werte *real* und *integer* zulässt, werden beim Einchecken nur der Datentyp *numeric* bei numerischen Datensatzattributen unterstützt, sodass die Datentypen *real* und *integer* in der Meta-Daten Datei in der momentanen Form nicht vorkommen. Die Inhalte der Elemente *numValues* und *classDist* haben nur für nominale Klassenattribute eine Bedeutung und sind daher in allen anderen Fällen 0. Das Element *numValues* enthält die Anzahl der verschiedenen möglichen Werte eines nominellen Klassenattributs, also die Anzahl der Klassen des Datensatzes, während das *classDist* die Häufigkeit der grössten Klasse des Datensatzes in Prozent abbildet.
Der Inhalt des *instances* Element stellt die Anzahl der Instanzen des Datensatzes dar, und das *missingValues* Element gibt an, wieviele Werte in den Instanzen fehlen. Dabei handelt es sich um einen Prozentsatz auf Basis aller möglichen Werte in den Instanzen.
Das *testSet* Element enthält Verweise auf eventuell vorhandene Testdatensätze. Es besitzt ein Attribut namens *numTestSet*, das die Anzahl aller vorhandenen Testdatensätze anzeigt und es besitzt für jeden Testdatensatz ein *uri* Element. Die *uri* Elemente enthalten die Namen der XML-Dateien, in denen die Testdatensätze gespeichert sind. Der Name einer Testdatensatzdatei hat die allgemeine Form: *testset<n>*-

<Name Datensatzdatei>. Der Name beginnt mit *testset*, um auszudrücken, dass die XML-Datei einen Testdatensatz abbildet, und wird von der fortlaufenden Zahl *n* gefolgt, um beim Vorhandensein von mehreren Testdatensätzen, diese voneinander unterscheiden zu können. Getrennt von einem Minuszeichen folgt danach der Name der Datensatzdatei, auf die sich der Testdatensatz bezieht. Im Beispiel aus Abbildung 28 sind für den gezeigten Datensatz zwei Testdatensätze vorhanden, die in XML-Dateien mit einem entsprechenden Namen abgelegt sind.

6.2.3 Zugriff auf die Datensätze

Ein Zugriff auf die Datensatzdateien erfolgt, wie in Kapitel 5.2.1 und 5.2.2 bereits erwähnt, in den folgenden drei Situationen:

- lesender Zugriff durch ein Transform Bean-Objekt beim Auschecken
- lesender Zugriff durch ein DatasetCreator Bean-Objekt mit Hilfe einer Reader-Klasse beim Einchecken
- schreibender Zugriff, d.h. Erzeugen einer neuen Datensatzdatei durch ein DatasetCreator Bean-Objekt beim Einchecken

Der lesende Zugriff auf eine Datensatzdatei wird im Folgenden anhand der *DatasetReader* Klasse beschrieben, deren grundlegende Vorgehensweise sich beim Lesen eines Datensatzes durch eine *Transform* Bean wiederfindet. Das Erzeugen einer Datensatzdatei im anwendungsspezifischen Speicherformat wird in einem späteren Kapitel erläutert.
Die *DatasetReader* Klasse implementiert Methoden, die bestimmte Elemente aus der XML-Datensatzdatei auslesen und in Form einer Liste an die aufrufende Stelle zurückliefern. Dazu werden geeignete ContentHandler Klassen verwendet, die während dem Parsen der Datensatzdatei bestimmte Elemente extrahieren.
In Abbildung 29 ist ein Teil des Quelltextes der *DatasetReader* Klasse mit der Methode *getAttributeDeclaration()* abgebildet. Diese Methode liefert alle *attribute* Elemente einer Datensatzdatei zurück.
Die Klasse *DatasetReader* besitzt drei Klassenvariablen. Die Variable *reader*, die den SAX-Parser aufnimmt, der zum Parsen der Datensatzdatei verwendet wird, die Variable *handler*, der je nach Bedarf ein geeigneter ContentHandler zugewiesen wird und die Variable *ns*, die den Namensraum, an den die Elemente einer Datensatzdatei geknüpft sind, abbildet. Durch den Aufruf des Konstruktors, also beim Erzeugen eines Objekts vom Typ *DatasetReader*, wird der Apache SAX-Parser zum Lesen der XML-Dateien festgelegt.
Die *getAttributeDeclaration()* Methode übernimmt einen String, der den absoluten URI der zu lesenden Datensatzdatei darstellt. Als Container für die auszulesenden *attribute* Elemente erzeugt die *getAttributeDeclaration()* Methode zunächst ein Objekt der Klasse Element der JDOM API mit dem Namen *root*. Dieses repräsentiert ein XML-Element als Java Objekt, an das die *attribute* Elemente als Kindelemente angefügt werden. Danach wird ein ContentHandler Objekt vom Typ *AttributeDeclarationHandler* erzeugt, dessen Konstruktor das *root* Objekt als Parameter übernimmt. Nachdem der ContentHandler beim SAX-Parser durch den Aufruf der Methode *this.reader.setContentHandler(this.handler)* registriert wurde, wird der Parsevorgang durch *this.reader.parse(datasetURI)* gestartet. Der SAX-Parser verarbeitet damit die Datensatzdatei, die mit dem URI *datasetURI* bezeichnet wird und ruft beim Auftreten bestimmter Ereignisse die Methoden des registrierten ContentHandlers auf. Der ContentHandler seinerseits extrahiert die *attribute* Elemente und fügt sie dem übernommenen *root* Objekt hinzu. Die Wesentlichen Auszüge des ContentHandlers sind in Abbildung 30 zu sehen.
Die *AttributeDeclarationHandler* Klasse besitzt die zwei Klassenvariablen *root* und *currentElement*. Die Variable *root* ist eine Referenz auf das *root* Objekt, welches von der *getAttributeDeclaration()* Methode

```
public class DatasetReader{

 private XMLReader reader;
 private ContentHandler handler;
 private Namespace ns = Namespace.getNamespace(„http://www.thomaswanderer.de/datasetSchema“);

 public DatasetReader() {
  this.reader = XMLReaderFactory.createXMLReader(„org.apache.xerces.parsers.SAXParser“);
 }

 public LinkedList getAttributeDeclaration(String datasetURI){

  LinkedList attrDecl;
  Element root = new Element(„root“);
  this.handler = new AttributeDeclarationHandler(root);
  this.reader.setContentHandler(this.handler);

  try{
   this.reader.parse(datasetURI);
  }
  catch(BreakException e){}

  attrDecl = new LinkedList(root.getChild(„attributes“,this.ns).getChildren(„attribute“,this.ns));
  return attrDecl;
 }
 ...
}
```

Abbildung 29: Quelltext der DatasetReader Klasse

der *DatasetReader* Klasse erzeugt wurde, und die *currentElement* Variable ist ein Zeiger auf das gerade fokusierte Element Objekt. Beim Erzeugen des ContentHandlers wird der Zeiger auf das durch den Konstruktor übernommene *root* Objekt gesetzt.
Trifft der SAX-Parser auf ein Start-Tag eines XML-Elements der Datensatzdatei, wird die Methode *startElement()* des ContentHandlers ausgeführt. Diese ruft in Abhängigkeit der angetroffenen Elemente weitere Methoden auf. Trifft der Parser auf ein *attribute* Element (*if(localName.equals("attribute"))*) wird die Methode *createNewElement()* und für jedes vorhandene Attribut des Elements die Methode *addAttribute()* aufgerufen. In der *startElement()* Methode aus Abbildung 30 ist der Auffruf der *addAttribute()* Methode exemplarisch für das *name* Attribute gezeigt. Die *createNewElement()* Methode erzeugt nun ein neues Element Objekt, das ein Abbild des beim Parsen angetroffenen *attribute* Elements darstellt und fügt dieses Objekt dem gerade aktuellen Element Objekt (*currentElement*) als Kindelement mit der Methode *addContent()* hinzu. Anschließend wird der Zeiger auf das neu angelegte Kindelement gesetzt. Durch die folgenden Aufrufe der *addAttribute()* Methode werden dem neuen Element Objekt die entsprechenden Attribute hinzugefügt. In analoger Weise werden für alle weiteren relevanten XML-Elemente der Datensatzdatei Element Objekte erzeugt und an das gerade aktuelle Element Objekt angehängt.

```
public class AttributeDeclarationHandler{

  private Element root;
  private Element currentElement;

  public AttributeDeclarationHandler(Element root){
    this.root = root;
    this.currentElement = this.root;
  }

  public void startElement(String namespaceURI, String localName, String qName, Attributes attrs){
    if(localName.equals(„attribute“)){
      this.createNewElement(localName, namespaceURI);
      this.addAttribute(„name“, attrs.getValue(„“,“name“));
      ...
    }
    if(localName.equals(„labels“) || localName.equals(„label“) || ...){
      this.createNewElement(localName, namespaceURI);
    }
    if(localName.equals(„data“)){
      this.stopParsing();
    }
  }

  public void endElement(String namespaceURI, String localName, String qName){
    if(localName.equals(„attribute“) || localName.equals(„labels“) || ...){
      this.oneStepBack();
    }
  }

  private void createNew Element(String elementName, String namespaceURI){
    Element e = new Element(elementName,namespaceURI);
    this.currentElement.addContent(e);
    this.currentElement = e;
  }

  private void oneStepBack(){
    this.currentElement = this.currentElement.getParentElement();
  }

  private void addAttribute(String name, String value){
    this.currentElement.setAttribute(name,value);
  }

  private void stopParsing(){
    throw new BreakException();
  }

}
```

Abbildung 30: Quelltext der AttributeDeclarationHandler Klasse

Wenn der Parser ein Ende-Tag in der XML-Datei vorfindet, ruft dieser die *endElement()* Methode auf, welche wiederum für alle XML-Elemente, für die ein neues Element Objekt erzeugt wurde, die Methode *oneStepBack()* aufruft. Die Methode *oneStepBack()* hat die Aufgabe den Zeiger nach dem Anbinden eines neuen Elements (bzw. eines "Teilsbaums"), wieder zurück auf das Eltern-Element Objekt zu setzen, um

die Struktur der XML-Elemente richtig auf die Element Objekte abzubilden. Nach dem Parsen bildet das *root* Objekt den Teilbaum der XML-Datensatzdatei als Java Objekt ab, der alle *attribute* Elemente einschließlich deren Kindelemente enthält.
Um das Parsen zu verkürzen wird beim Antreffen des *data* Elements die Methode *stopParsing()* aufgerufen, die eine *BreakException* wirft, womit der Parservorgang abgebrochen wird.
Die Methode *getAttributeDeclaration()* der *DatasetReader* Klasse fängt die *BreakException* ab und erzeugt abschließend eine Liste, die mit den Element Objekten, die die *attribute* Elemente der XML-Datei darstellen, gefüllt wird. Diese Liste wird von der Methode *getAttributeDeclaration()* an die aufrufende Stelle zurückgegeben.
Jede Methode der *DatasetReader* Klasse verwendet einen spezifischen ContentHandler, der beim Parsen den jeweiligen Anforderungen entsprechende Aktionen ausführt.

6.2.4 Zugriff auf die Meta-Daten

Der lesende Zugriff auf die Meta-Daten Datei erfolgt über die Klasse *DatasetDBReader*, und für schreibende Zugriffe auf die Meta-Daten steht die Klasse *DatasetDBWriter* zur Verfügung. Beide Klassen besitzen die gleichen Klassenvariablen und den gleichen Konstruktor. Beim Erzeugen eines Objekts einer der beiden Klassen sorgt jeweils der Konstruktor dafür, das die Meta-Daten Datei mit einem DOM-Parser geparst wird, sodass der Inhalt der gesamten XML-Datei im Speicher steht. Während die *DatasetDBReader* Klasse Methoden implementiert, die lesend auf das im Speicher befindliche Dokument zugreift, kann mit den Methoden der *DatasetDBWriter* Klasse das Dokument verändert werden. Beispielhaft für beide Klassen sind die Klassenvariablen und der Konstruktor der *DatasetDBReader* Klasse in Abbildung 31 dargestellt.

```
public class DatasetDBReader{

  private String xmlDBUri;
  private Document datasetDB;
  private Element rootDB;
  private Namespace nsDB;

  public DatasetDBReader (String uri) throws SAXException, IOException{
    this.xmlDBUri = uri;
    DOMParser parser = new DOMParser();
    parser.parse(this.xmlDBUri);
    org.w3c.dom.Document domDoc = parser.getDocument();
    DOMBuilder builder = new DOMBuilder();
    this.datasetDB = builder.build(domDoc);
    this.rootDB =  this.datasetDB.getRootElement();
    this.nsDB = rootDB.getNamespace();
  }
  ...
}
```

Abbildung 31: Konstruktor der DatasetDBReader Klasse

Jede der Klassen besitzt die vier Klassenvariablen:

- *xmlDBUri* - Die Variable nimmt den URI der zu verarbeitenden Meta-Datendatei als String auf.
- *datasetDB* - Die Variable enthält ein Objekt vom Typ Document, welches das gesamte XML-Dokument nach dem Parsen repräsentiert.

- *rootDB* - Die Variable enthält das Wurzelelement des geparsten Dokuments, also das datasetDB Element der Meta-Daten Datei, als Element Objekt.
- *nsDB* - Die Variable dient zur Aufnahme des Namensraums des Wurzelelements, welcher der Namensraum aller Elemente einer Meta-Daten Datei ist.

Beim Ausführen des Konstruktors wird zuerst der Inhalt des Parameters *uri* der Klassenvariable *xmlDBUri* zugewiesen. Dieser String muss ein absoluter URI der Meta-Daten Datei sein, so dass die Meta-Daten Datei, nachdem ein DOM-Parser erzeugt wurde, geparst werden kann. Nach dem Parsen der Datei durch den Xerces DOM-Parser, liegt der Inhalt der Datei zunächst als ein Objekt der Klasse *org.w3c.dom.Document* vor. Dieses Objekt wird daher anschliessend mit Hilfe der *DOMBuilder* Klasse der JDOM API in ein JDOM *Document* Objekt umgewandelt und in der Klassenvariable *datasetDB* abgelegt (*this.datasetDB = builder.build(domDoc)*). Entsprechend werden die übrigen Klassenvariablen gesetzt.
Nachdem ein *DatasetDBReader* oder *DatasetDBWriter* Objekt erzeugt wurden und somit deren Konstruktor ausgeführt wurde, steht der Inhalt der Meta-Daten Datei in den Klassenvariablen des jeweiligen Objekts für die weitere Verarbeitung zur Verfügung.
Abbildung 32 zeigt zwei beispielhafte Methoden der *DatasetDBReader* Klasse für den lesenden Zugriff auf die Meta-Daten Datei.

```
public class DatasetDBReader{
  ...
  public boolean nameExists(String name){
    boolean ex=false;
    LinkedList datasetElements = new LinkedList(this.rootDB.getChildren());
    Element current;
    String content;
    for(int i=0;i< datasetElements.size();i++){
      current = (Element) datasetElements.get(i);
      content = current.getChild("name",this.nsDB).getTextTrim();
      if(content.equals(name)){
        ex=true;
        i=datasetElements.size();
      }
    }
    return ex;
  }

  public LinkedList xQuery(String xpath, String name, int num, ...){
    LinkedList result = new LinkedList();
    XPath search = XPath.newInstance(xpath);
    search.addNamespace("x",this.nsDB.getURI());
    search.setVariable("NameDataset",name);
    search.setVariable("NumNumericAttr",num);
    ...
    result = new LinkedList(search.selectNodes(this.datasetDB));
    return result;
  }
  ...
}
```

Abbildung 32: Methoden der DatasetDBReader Klasse

Mit der Methode *nameExists()* kann überprüft werden, ob ein Datensatz mit einem bestimmten Namen

in der Datensatzsammlung bereits existiert. Die Methode liefert *false*, wenn noch kein Datensatz existiert, dessen Namen dem übergebenen Parameter *name* entspricht. Sonst liefert die Methode *true*.
Zuerst erzeugt die Methode eine Liste aller Kindelemente des Wurzelelements der Meta-Daten Datei. Diese Liste enthält alle dataset Elemente der Meta-Daten Datei. Anschließend wird innerhalb der for-Schleife über die Elemente dieser Liste iteriert, wobei der Inhalt jedes *name* Elements (*content = current.getChild("name",this.nsDB).getTextTrim()*) mit dem übergebenen String *name* verglichen wird (*content.equals(name)*). Sind die beiden Strings gleich, wird der Rückgabewert auf *true* gesetzt und der Schleifeniterator wird auf einen Wert gesetzt, das die Schleife abbricht. Andere Methoden dieser Klasse arbeiten nach einer ähnlichen Vorgehensweise.
Die Methode *xQuery()* aus Abbildung 32 ist in der Lage, einen übergebenen XPath-Ausdruck auf die Meta-Daten Datei anzuwenden, um bestimmte *dataset* Elemente zu selektieren und an die aufrufende Stelle zurückzugeben. Der übergebene String *xpath* repräsentiert den anzuwendenden XPath-Ausdruck, der folgende Form haben muss: */x:datasetDB/x:dataset[... and x:attributes/x:numNumeric = $NumNumericAttr and ...]*.
Der XPath-Ausdruck bis zur eckigen Klammer selektiert alle *dataset* Elemente der Meta-Daten Datei und bildet den statischen Teil des XPath-Ausdrucks. Die Prädikate in den eckigen Klammer können je nach Anfrage variieren. Die Methode *xQuery()* erwartet, das sämtliche Elementbezeichnungen mit dem Präfix *x* versehen sind, damit die Elemente einem Namensraum zugeordnet werden können. Die Prädikate sind durch AND Operatoren miteinander verknüpft. Ein einzelnes Prädikat besteht i.d.R. aus einem Vergleich des Inhaltes eines bestimmten Elements mit dem Wert einer noch nicht initialisierten Variable. In dem Beispielausdruck wird der Inhalt eines *numNumeric* Elements mit dem Wert der Variable *$NumNumericAttr* vergliechen.
Zuerst erzeugt die *xQuery()* Methode aus Abbildung 32 ein XPath Objekt der JDOM API für den gegebenen XPath-Ausdruck. Anschließend wird dem Objekt der Namensraum der Elemente der Meta-Daten Datei übergeben und an das Präfix *x* gebunden. Danach werden die Werte der Variablen des XPath-Ausdrucks durch die Methode *setVariable()* gesetzt. Die Werte werden, wie auch der XPath-Ausdruck, als Parameter beim Aufruf der *xQuery()* Methode übergeben und stellen Benutzereingaben dar. Im Beispiel aus Abbildung 32 werden beispielhaft die Variablen *$NameDataset* und *$NumNumericAttr* (Name des Datensatzes und Anzahl der numerischen Attribute des Datensatzes) gesetzt. Abschließend wertet das XPath Objekt den XPath-Ausdruck auf der Meta-Daten Datei durch die Methode *selectNodes()* aus und liefert somit eine Liste von *dataset* Elementen, die die gesuchten Datensätze repräsentieren. Die Liste mit den *dataset* Elementen ist der Rückgabewert der *xQuery()* Methode. Diese Methode wird verwendet, um die Suche nach Datensätzen durch den Anwender umzusetzen.
Der schreibende Zugriff auf die Meta-Daten Datei durch Methoden der DatasetDBWriter Klasse, besteht aus zwei Schritten.

1. Erzeugung von Element oder Attribut Objekten und Anfügen dieser Objekte an das im Speicher stehende XML-Dokument

2. Zurückschreiben des veränderten XML-Dokuments zur persistenten Speicherung

Der Ausschnitt der Methode *writeDBEntry()* der *DatasetDBWriter* Klasse aus Abbildung 33 verdeutlicht diese beiden Schritte.
Die Methode *writeDBEntry()* fügt der Meta-Daten Datei ein *dataset* Element hinzu, um die Meta-Daten eines neu integrierten Datensatzes zu speichern. Dazu erzeugt die Methode einen XML-Teilbaum mit einem *dataset* Element an der Wurzel und fügt die Meta-Daten den entsprechenden Elementen hinzu. In Abbildung 33 ist dies exemplarisch nur für das Kindelement *name* gezeigt.

```
public class DatasetDBWriter{
  ...
  public void writeDBEntry(String na, ...){
   Element dataset = new Element("dataset",this.nsDB);
   Element name = new Element("name",this.nsDB);
   name.addContent(na);
   dataset.addContent(name);
   ...
   this.rootDB.addContent(dataset);
   this.writeBack();
  }

  private void writeBack() {
   Format outFormat = Format.getPrettyFormat();
   outFormat.setIndent("    ");
   outFormat.setOmitEncoding(false);
   XMLOutputter outputter = new XMLOutputter(outFormat);
   outputter.output(this.datasetDB, new FileOutputStream(new File(this.xmlDBUri)));
  }
  ...
}
```

Abbildung 33: Ausschnitt aus der DatasetDBWriter Klasse

Zuerst erzeugt die Methode *writeDBEntry()* ein *dataset* Element im entsprechenden Namensraum in Form eines Element Objekts, wie auch ein *name* Element. Dem *name* Element wird anschließend als Inhalt der Wert eines übergebenen Parameters mit der Methode *addContent()* hinzugefügt. Der Parameter enthält in diesem Fall den Namen des Datensatzes. Das *name* Element wird dann dem *dataset* Element als Kindelement zugefügt. Auf gleiche Weise werden alle anderen Elemente erzeugt und an ihrer vorgesehenen Stelle im Teilbaum hinzugefügt. Zum Schluß wird das Wurzelelement datasetDB des im Speicher befindlichen XML-Dokument, um das neue dataset Element ergänzt. Um das veränderte XML-Dokument persistent zu speichern wird die Klassenmethode *writeBack()* aufgerufen. Diese verwendet die *XMLOutputter* Klasse der JDOM API, um das *Document* Objekt in die Meta-Daten Datei zu schreiben.

6.3 Realisierung der Datensatzsuche

Nachdem die, durch den Benutzer in der Suchmaske angegebenen Suchparameter, validiert wurden, kann eine Suchanfrage durchgeführt werden. Die Durchführung einer Suchanfrage und die anschließende Aufbereitung des Ergebnisses der Anfrage gliedert sich in drei Schritte:

- Aufbereitung der Suchparameter
- XPath-Ausdruck formulieren und auf Meta-Daten Datei anwenden
- Resultat der Anfrage in eine weiterverwendbare Speicherstruktur (für die Einbindung in eine JSP) ablegen

Im Wesentlichen stellt die Bean Klasse *DatasetQuery* die notwendige Funktionalität zur Durchführung einer Suchanfrage bereit. Eingeleitet wird eine Suchanfrage durch das Servlet *SearchDataset*, dessen *doPost()* Methode ausschnittsweise in Abbildung 34 zu sehen ist.
Beim Ausführen der *doPost()* Methode des Servlets wird zuerst ein Objekt der Klasse *DatasetQuery* erzeugt. Dem *DatasetQuery* Objekt wird anschließend durch seine Setter-Methode *setApplPath()* der

```
public class SearchDataset extends HttpServlet{
  ...
  public void doPost(HttpServletRequest request, HttpServletResponse response){

    DatasetQuery query = new DatasetQuery();

    ServletContext application = getServletContext();
    query.setApplPath(application.getRealPath("/"));

    Form form = (Form)request.getAttribute("form");
    query.setQueryParam(form);

    try{
      query.exeQuery();
    }
    catch(Exception e){...}

    request.setAttribute("query", query);
    RequestDispatcher rd = request.getRequestDispatcher("/jsp/Hits.jsp");
    rd.forward(request, response);

  }
...
}
```

Abbildung 34: doPost() Methode des SearchDataset Servlets

absolute URI des home Verzeichnisses der Anwendung übergeben. Das *DatasetQuery* Objekt benötigt den URI zur Lokalisierung der Meta-Daten Datei. Ermittelt wird der URI mit Hilfe der Methode *getRealPath()* des *ServletContext* Objekts, das den Application Scope der Anwendung darstellt. Da die Methode *getRealPath("/")* einen absoluten URI auf Basis des vorhandenen Filesystems des Servers für den übergebenen anwendungsbezogenen Pfad (hier: Wurzelverzeichnis der Anwendung) ermittelt, kann das *DatasetQuery* Objekt unabhängig von der zugrundeliegenden Plattform, die Meta-Daten Datei lokalisieren. Anschließend wird die *setQueryParam()* Methode des *DatasetQuery* Objekts mit dem Parameter *form* aufgerufen. Dieser Parameter ist ein Bean-Objekt vom Typ *Form*, welches sich im Request Scope befindet und sämtliche Suchparameter des Benutzers aus der Suchmaske enthält. Nachdem die Methode *exeQuery()* aufgerufen wurde, die die eigentliche Suchanfrage ausführt, wird das *DatasetQuery* Objekt in den Request Scope gestellt und der Request mit Hilfe des *RequestDispatcher* Objekts an die *Hits* JSP weitergeleitet, wo die Treffer der Suchanfrage angezeigt werden.
In der Abbildung 35 sind unter anderem Teile der Methoden *setQueryParam()* und *exeQuery()* abgebildet. Die *setQueryParam()* Methode hat die Aufgabe, die Suchparameter des Benutzers, die in dem übergebenen *Form* Bean-Objekt gekapselt sind, aufzubereiten. Da alle Suchparameter als Strings vorliegen, müssen die Suchparameter, die eine Zahl darstellen, in einen geeigneten Datentyp umgewandelt werden. Zu jedem zahlenmäßigen Suchparameter existiert ein Operator, der die Werte "equal", "less than" oder "greater than" haben kann. Diese Werte müssen durch ein entsprechendes Symbol (=, < oder >) ersetzt werden.
In Abbildung 35 ist die Verarbeitung der drei Suchparameter *nameDataset*, *opNumericAttr* und *numNumericAttr* durch die *setQueryParam()* Methode gezeigt. Der Suchparameter *nameDataset* kann unverändert als String in eine eigene Klassenvariable übernommen werden, während der Inhalt des Suchparameters *opNumericAttr*, der sich auf die Anzahl der numerischen Attribute des Datensatzes bezieht,

```
public class DatasetQuery{
  ...
  public void setQueryParam(Form form){
    this.nameDataset = form.getNameDataset();
    this.opNumericAttr = this.convertOperator(form.getOpNumericAttr());
    if(form.getNumNumericAttr().equals("")){
      this.numNumericAttr = -1;
    }
    else{
      this.numNumericAttr = Integer.parseInt(form.getNumNumericAttr());
    }
  }

  public void exeQuery(){
    DatasetDBReader reader = new DatasetDBReader(this.applPath+"data"+this.sep+"datasetDB.xml");
    this.queryResult = reader.xQuery(this.createXPathString(),this.nameDataset,this.numNumericAttr, ...);
    this.hits = this.queryResult.size();
    this.addData(queryResult);
  }

  private String createXPathString(){
    String xpath="/x:datasetDB/x:dataset[";
    if(!this.nameDataset.equals("")){
      xpath = xpath + "contains(x:name,$NameDataset) and ";
    }
    if(this.numNumericAttr != -1){
      xpath = xpath + "x:attributes/x:numNumeric"+this.opNumericAttr+"$NumNumericAttr and ";
    }
    ...
    return xpath;
  }
...
}
```

Abbildung 35: Auszüge aus der DatasetQuery Bean

zuerst in ein entsprechendes Symbol umzuwandeln ist, bevor dieser in eine Klassenvariable abgelegt wird. Die Umwandlung der Operatoren übernimmt die Klassenmethode *convertOperator()*. Der Suchparameter *numNumericAttr*, der eine Anzahl darstellt, wird in den Datentyp *int* transferiert. Beinhaltet ein Suchparameter des *Form* Objekts einen leeren String, bedeutet das, dass der Benutzer an dieser Stelle keine Eingabe gemacht hat und somit der Parameter bei der Suche unberücksichtigt bleibt. In diesen Fällen wird der Suchparameter auf den Wert -1 gesetzt, wie anhand der if-Abfrage der *setQueryParam()* Methode zu sehen ist. Alle anderen Suchparameter werden auf die gleiche Weise verarbeitet, bevor sie in die eigenen Klassenvariablen abgelegt werden.
Danach kann die Methode *exeQuery()* ausgeführt werden, die die bereits beschriebene *xQuery()* Methode der *DatasetDBReader* Klasse verwendet, um eine Suchanfrage durchzuführen. Dazu erzeugt sie zunächst ein Objekt der Reader Klasse und übergibt dabei einen String, der den absoluten URI der Meta-Daten Datei enthält. Der String setzt sich zusammen aus dem Wurzelverzeichnis der Anwendung (*appPath*) zuzüglich dem anwendungsbezogenen Pfad (*./data/datasetDB.xml*) der Meta-Daten Datei. Vor dem Ausführen der *xQuery()* Methode, wird durch die *createXPathString()* Methode, der XPath-Ausdruck erzeugt, der durch die *xQuery()* Methode ausgewertet werden soll.

In Abbildung 35 ist der Aufbau des XPath-Ausdruck beispielhaft für zwei Suchparameter dargestellt. Die Basis des XPath-Ausdrucks ist der String */x:datasetDB/x:dataset[]*. Mit diesem Ausdruck werden alle *dataset* Elemente der Meta-Daten Datei ausgewählt. In Abhängigkeit der Suchparameter des Benutzers wird dieser Basisausdruck um bestimmte Prädikate ergänzt, sodass aus der Menge aller *dataset* Elemente, diejenigen gefiltert werden, die den Benutzerangaben entsprechen. Werden durch den Benutzer keine Angaben gemacht, bleibt die eckige Klammer leer und alle Datensätze werden ausgewählt. Wie in Abbildung 35 zu sehen ist, wird durch eine if-Abfrage für jeden Suchparameter geprüft, ob dieser einen Wert enthält. Liegt beispielsweise ein Datensatzname als Suchparameter vor, wird der Basisausdruck um das Prädikat *contains(x:name, $NameDataset) and* ergänzt. Die Variable *$NameDataset* repräsentiert die Benutzereingabe für den Datensatznamen und dient als Vergleichsgröße für den Inhalte des *name* Elements der Meta-Daten Datei. Die hier verwendete XPath-Methode *contains()* liefert true, wenn der Inhalt der Variable *$NameDataset* im Textknoten des *name* Elements enthalten ist. D.h. der Suchparameter *auto* für den Namen des Datensatzes würde auch einen Datensatz namens *automobile* auswählen. Bei numerischen Suchparametern wird der XPath-Ausdruck um ein Prädikat ergänzt, welches den Inhalt eines Elements der Meta-Daten Datei mit dem entsprechenden Suchparameter unter Verwendung des ausgwählten Operators, vergleicht. Im Beispiel aus Abbildung 35 ist dies für den Parameter *numNumericAttr* gezeigt. Der erste Teil des Prädikats wählt das Element der Datensatz Datei aus (*x:attributes/x:numNumeric*), das den gesuchten Wert darstellt. Dann folgt der Vergleichsoperator, der bereits in seiner symbolischen Form in der entsprechenden Klassenvariable vorliegt (*this.opNumericAttr*), und der letzte Teil bildet die Vergleichsvariable *$NumNumericAttr*, die die Benutzereingabe darstellt.
Für die restlichen Suchparameter wird der XPath-Ausdruck in analoger Weise ergänzt, wobei die einzelnen Prädikate durch den *AND* Operator miteinander verknüpft werden, sodass nur die *dataset* Elemente ausgewählt werden, die alle Suchkriterien erfüllen. Nachdem der XPath-Ausdruck generiert wurde, kann die *xQuery()* Methode aufgerufen werden, an die neben dem XPath-Ausdruck sämtliche Suchparameter des Benutzer übergeben wird. Die *xQuery()* Methode liefert eine List von *dataset* Elementen in Form von Element Objekten der JDOM API zurück. Um die Meta-Daten der gefundenen Datensätze in der *Hits* JSP anzeigen zu können, werden die Inhalte der *dataset* Elemente durch Objekte der *MetaData* Klasse gekapselt, was durch die Methode *addData()* durchgeführt wird. Die Eigenschaften der *MetaData* Klasse bilden sämtliche Meta-Daten ab. Die Klasse besitzt ausschließlich Getter- und Setter- Methoden.

6.4 Transformation / Download der Datensätze

Nachdem der Benutzer die Datensätze und das Dateiformat ausgewählt hat, können die Datensätze in das gewünschte Dateiformat transformiert werden, um sie anschließend als Zipstream zum Download bereitzustellen. Die Transformation wird durch das Servlet *Download* eingeleitet, welches auch den Zipstream erzeugt. Die wesentlichen Bestandteile der *doPost()* Methode dieses Servlets sind in Abbildung 36 gezeigt. Durch die Ausführung der *doPost()* Methode wird ein Objekt der Bean-Klasse *Transform* erzeugt, welche die Methoden besitzt, um einen Datensatz im anwendungsspezifischen Speicherformat auf eines der möglichen Dateiformate abzubilden. Dem *Transform* Objekt wird das vom Benutzer gewählte Dateiformat durch die *setFormat()* Methode übergeben. Außer dem Dateiformat werden mit dem Request die Namen der Datensätze übermittelt, die für die weitere Verwendung in dem String-Array *datasets* abgelegt werden. Des weiteren wird die Liste *downloadFiles* erzeugt, die sämtliche transformierte Datensatzdateien aufnimmt, bevor der Zipstream erzeugt wird.
Nach dem Initialisieren der benötigten Objekte, wird mit einer for-Schleife über die Liste der Datensatznamen iteriert. Für jeden Datensatz werden zunächst der Datensatzname, der Dateiname, der den Datensatz enthält, und die Anzahl der vorhandenen Testdatensätze an das *Transform* Objekt übergeben.

```
public class Download extends HttpServlet{
  ...
  public void doPost(HttpServletRequest request, HttpServletResponse response){

    Transform transform = new Transform();
    transform.setFormat(request.getParameter("format"));

    String[] datasets = request.getParameterValues("datasets");
    LinkedList downloadFiles = new LinkedList();

    for(int i=0;i<datasets.length;i++){
      transform.setName(datasets[i]);
      transform.setUri(dbReader.getDatasetUri(datasets[i]));
      transform.setNumTestSet(String.valueOf(dbReader.getNumTestSet(datasets[i])));
      classType = dbReader.getClassAttributeType(datasets[i]);

      if(transform.getFormat().equals("c45") || transform.getFormat().equals("c50")){
        if(!classType.equals("none") && !classType.equals("string") && ...){
          dummyFile = new File("dum");
          dummyFile = transform.createNamesFile();
          downloadFiles.add(dummyFile);
          for(int j=0;j<=Integer.parseInt(transform.getNumTestSet());j++){
            dummyFile = new File("dum");
            dummyFile = transform.createCDataFile(j);
            downloadFiles.add(dummyFile);
          }
        }
      }
      else{
          for(int j=0;j<=Integer.parseInt(transform.getNumTestSet());j++){
            dummyFile = new File("dum");
            dummyFile = transform.transformFile(j);
            downloadFiles.add(dummyFile);
          }
      }
    }
    if(!downloadFiles.isEmpty()){
      this.createZipStream(downloadFiles,response,transform.getFormat());
    }
    else{
      //erzeuge Datei mit Hinweis + füge diese in Liste downloadFile
       this.createZipStream(downloadFiles,response,transform.getFormat());
    }
  }
...
}
```

Abbildung 36: doPost() Methode des Servlets Download

Der Dateiname und die Anzahl der Testdatensätze werden mit Hilfe eines *DatasetDBReader* Objekts aus der Meta-Daten Datei ausgelesen. Zusätzlich wird der Typ des Klassenattributs ausgelesen und in der Variable *classType* zwischengespeichert.
Danach wird in Abhängigkeit des gewählten Dateiformats und dem Typ des Klassenattributs die Transformation des gerade fokusierten Datensatzes durchgeführt. Die *Transform* Klasse stellt drei verschiedene Methoden für die Datensatztransformation zur Verfügung:

- *createNamesFile()* - Veranlasst die Erzeugung einer *.names Datei im C4.5- oder C5.0-Format für einen gegebenen Datensatz
- *createCDataFile()* - Veranlasst die Erzeugung einer *.data Datei im C4.5- oder C5.0-Format für einen gegebenen Datensatz
- *transformFile()* - Veranlasst die Erzeugung einer Datensatzdatei im ARFF-, sparse ARFF- , XRFF-, sparse XRFF- oder im CSV-Format für einen gegebenen Datensatz

Für die Formate C4.5 und C5.0 wird vor der Transformation geprüft, ob der Typ des Klassenattributs geeignet ist, um eine Transformation in eins der genannten Formate durchzuführen. Ist das Klassenattribut vom Typ string, date oder relational oder besitzt der Datensatz kein Klassenattribut, wird keine Transformation durchgeführt. In allen anderen Fällen wird für den Datensatz mit der *createNamesFile()* Methode eine *.names Datei erzeugt. Außerdem wird für den Datensatz und allen vorhandenen Testdatensätzen jeweils eine *.data Datei mit der Methode *createCDataFile()* erzeugt. Die *createCDataFile()* Methode übernimmt dabei als Parameter den Iterator, der über die Anzahl der Testdatensätze iteriert, um den zu transformierenden Datensatz zu identifizierten. Der Wert 0 bezeichnet dabei den eigentlichen Datensatz und die restlichen Werte bezeichnen die Testdatensätze in der Reihenfolge wie sie in der Meta-Daten Datei aufgeführt sind. Die Transformation in eins der übrigen Dateiformate erfolgt analog mit der Methode *transformFile()*.
Nach einer Transformation werden die neu erzeugten Datensatzdateien in der Liste *downloadFiles* abgelegt. Die Liste wird an die Methode *createZipStream()* übergeben, die alle in der Liste enthaltenen Dateien in einen Zipstream packt und dem Client als Antwort auf seine Anfrage zu sendet. Sollte die Liste *downloadFiles* leer sein, wird eine Textdatei mit dem Hinweis, dass keine Transformationen durchgeführt werden konnten, erzeugt und der Liste downloadFiles zugefügt. Danach wird ebenfalls die *createZipStream()* Methode aufgerufen. Die Liste kann in den Fällen leer sein, in denen der Benutzer das Dateiformat C4.5 oder C5.0 gewählt hat und die Klassenattribute aller zu transformierenden Datensätze ein Typ haben, der eine Transformation nicht zulässt.
Die Dauer, die für eine Transformation benötigt wird, also die Dauer der Ausführung einer der drei Transformations-Methoden, wird gemessen und für jeden Datensatz in einer separaten Liste zwischengespeichert. Aus Gründen der Übersichtlichkeit ist dies in Abbildung 36 nicht aufgeführt. Nach Abschluss aller Transformationen werden im finally-Block der doPost() Methode, die Messwerte ausgewertet. Die Datensatzdateien, deren Transformationen länger als 10 Sekunden gedauert hat, bleiben serverseitig in einem speziell dafür vorgesehenen Verzeichnis persistent gespeichert. Dadurch kann für eine wiederholte Anfrage auf eine erneute Transformation verzichtet und direkt auf die gespeicherte Datei zugegriffen werden, um die Bearbeitungszeit der Clientanfragen zu beschleunigen.
Stellvertretend für die drei Transformations-Methoden der Transform Klasse ist in Abbildung 37 die Methode *transformFile()* gezeigt.
Die *transformFile()* Methode erzeugt zunächst ein File Objekt namens result, das die neu zu erzeugende Datensatzdatei aufnimmt. Der URI der neuen Datei wird unter Verwendung der Methode *createFileName()* gebildet, welche den Dateinamen erzeugt. Der Dateiname enthält neben dem Namen des Datensatzes Informationen über das Dateiformat und die Information, ob es sich bei der Datei um einen Testdatensatz handelt. Ein beispielhafter Dateiname für ein Testdatensatz im sparse ARFF-Format lautet: *iris_testdata-1_sparse.arff* .
In der anschließenden if-Abfrage wird überprüft, ob die zu erzeugende Datensatzdatei bereits existiert, womit sich eine Transformation erübrigt und das *File* Objekt *result* direkt an die aufrufende Stelle zurückgegeben werden kann. Existiert die Datei noch nicht, wird das *PrintWriter* Objekt erzeugt, das

```
public class Transform{
  ...
  public File transformFile(int fileNum){
    File result = new File(this.applPath+"data"+this.sep+"download"+this.sep+this.createFileName(fileNum));
    if(!result.exists()){
      PrintWriter writer = new PrintWriter(result);
      if(fileNum==0){
        datasetUri = this.applPath+"data"+this.sep+"xml"+this.sep+this.uri;
      }
      else{
        DatasetDBReader dbReader = new DatasetDBReader(this.applPath+"data"+this.sep+"datasetDB.xml");
        datasetUri = this.applPath+"data"+this.sep+"xml"+this.sep+dbReader.getTestSetUri(...);
      }
      if(this.getFormat().equals("arff")){
        this.handler = new ArffTransformer(writer);
      }
      ...
      this.reader.setContentHandler(this.handler);
      this.reader.parse(datasetUri);
    }
    return result;
  }
...
}
```

Abbildung 37: Methode transformFile() der Bean Transform

die neue Datei schreibt, und der URI der Datensatzdatei ermittelt. Bei einem Testdatensatz muss dazu der Dateiname aus der Meta-Daten Datei ausgelesen werden.
Der eigentliche Transformationsvorgang ist durch ein SAX-Parser mit einem entsprechenden Content-Handler realisiert. Für jedes Dateiformat steht eine separate ContentHandler Klasse zur Verfügung, die beim Parsen der Datensatzdatei den Datensatz in die Zieldatei *result* schreibt. Für die Formate C4.5 und C5.0 existieren für die *.names und *.data (bzw. *.test) Dateien jeweils separate ContentHandler Klassen. In Abhängigkeit des gewählten Dateiformats wird ein Objekt des entsprechenden ContentHandlers erzeugt. In der t*ransformFile()* Methode aus Abbildung 37 ist dies beispielhaft für das ARFF-Format gezeigt. Dem ContentHandler wird das *PrintWriter* Objekt übergeben, das zeilenweise in die Zieldatei schreibt. Nachdem das ContentHandler Objekt beim SAX-Parser registriert wurde, wird das Parsen der Datensatzdatei gestartet (*this.reader.parse(datasetUri)*).
Die Funktionsweisen der verschiedenen ContentHandler sind in ihren wesentlichen Zügen identisch. Wie auch die in Kapitel 6.2.3 beschriebene *AttributeDeclarationHandler* Klasse besitzen die ContentHandler Klassen für die Transformation von Datensätzen ein Element Objekt *current*, das das gerade aktuelle Element der Datensatzdatei beim Parsen repräsentiert. Immer dann, wenn der Parser auf ein Element von Interesse trifft, wird ein neues Element Objekt erzeugt, an das *current* Element als Kindelement angehängt und die Referenz des *current* Elements auf das Kindelement gesetzt, sodass dieses das neue *current* Element ist. Auf diese Weise können beliebige Teile der geparsten Datei in Form von Element Objekten, die eine XML-Baumstruktur bilden, zwischengespeichert werden. Zum Aufbau dieser Struktur stellen die ContentHandler Klassen die Methoden *addElement()*, *addAttribute()* und *oneStepBack()*, wie sie bereits in Kapitel 6.2.3 beschrieben sind, zur Verfügung. Zum Hinzufügen von Textknoten existieren ebenfalls Methoden.
Die Vorgehensweise der ContentHandler bei der Transformation von Datensätzen gliedert sich in die

folgenden Schritte:

1. den Inhalt eines *docu* Elements als Kommentar in die Zieldatei schreiben
2. Extrahieren der *attribute* Elemente (einschließlich deren gesamter Inhalt) der Datensatzdatei und diese in eine geeignete Datenstruktur für die weitere Verwendung zwischenspeichern
3. nach dem Parsen des Headers der Datensatzdatei, die Attributdeklaration der Zieldatei schreiben
4. beim Parsen des Datenteils jedes einzelne *instance* Element und dessen Inhalt extrahieren, zwischenspeichern und eine entsprechende Instanz in die Zieldatei schreiben

Trifft der Parser auf ein *docu* Element, wird ein Element Objekt erzeugt, dem der Inhalt des *docu* Elements als String hinzugefügt wird, zuzüglich dem formatspezifischen Zeichen für Kommentare (für das ARFF-Format beispielsweise: *% Inhalte_des_docu_Elements*). Außerdem wird das *current* Element wie oben beschrieben versetzt. Beim Antreffen des Ende-Tags eines *docu* Elements, wird der Inhalt des *current* Elements in die Zieldatei geschrieben, die Methode *oneStepBack()* aufgerufen und das Kindelement vom *current* Element entfernt. Auf die gleiche Weise wird mit allen *docu* Elementen verfahren bis alle Kommentarzeilen geschrieben sind.
Beim darauffolgenden Parsen des Headers der Datensatzdatei werden alle *attribute* Elemente, wie in Kapitel 6.2.3 beschrieben, extrahiert. Beim Antreffen des Ende-Tags des *attributes* Elements wird das Schreiben der Attributdeklaration der Zieldatei eingeleitet. In Abbildung 38 ist der betreffende Ausschnitt der ArffTransformer Klasse exemplarisch für alle ContentHandler gezeigt.
Trifft der Parser auf das Ende-Tag eines *attributes* Elements, muss geprüft werden, ob das Ende-Tag Teil eines relationalen *attribute* Elements ist, oder ob es sich dabei um das schließende Tag des *attributes* Elements handelt, das alle *attribute* Elemente der Datensatzdatei enthält. Im zweiten Fall wird, nachdem die Methode oneStepBack() ausgeführt wurde, eine Liste names attributeElements erzeugt, in die sämtliche attribute Element Objekte des current Elements abgelegt werden. Diese Liste wird anschließend mit Hilfe der *Collections* Klasse (Java SE) sortiert. Als Vergleichsfunktion dient die Methode *compare()* der *SortByIndex* Klasse, wodurch die Elemente der Liste bezüglich ihres *index* Attributs aufsteigend sortiert werden. Da im ARFF-Format Attribute des Typs *c50Label* und *formula* bedeutungslos sind, werden diese aus der *attributElements* Liste entfernt. Das Entfernen der Elemente wird durch die Methode *removeFormulaC50Attribute()* ausgeführt. Danach wird die Methode *writeAttributeDeclaration()* aufgerufen, um die Attributdeklaration der Zieldatei zu schreiben.
Die *writeAttributeDeclaration()* iteriert über die sortierte Liste von attribute Elementen und schreibt in Abhängigkeit des Attributtypes eine Attributdeklaration in die Zieldatei. In Abbildung 38 ist das in der *writeAttributeDeclaration()* Methode für die Attributetypen *string* und *relational* gezeigt. Für den Typ *string* setzt sich die zu schreibende Zeile zusammen aus dem Schlüsselwort *@attribute*, dem Inhalt des *name* Attributs und dem Inhalt des *type* Attributs des betreffenden *attribute* Elements. Im Falle eines relationalen Attributs wird, nachdem die einleitende Zeile der Attributdeklaration geschrieben wurde, eine Liste namens *relAttr* erzeugt, die die *attribute* Elemente des relationalen *attribute* Elements enthält. Diese Liste wird ebenfalls, wie oben beschrieben, sortiert, worauf ein rekursiver Aufruf der Methode *writeAttributeDeclaration()* erfolgt. Die als Parameter übergebene Liste *relAttr* wird dann in gleicher Weise verarbeitet. Nach der Rückkehr aus der Rekursion wird noch das Schlüsselwort *@end* in die Zieldatei geschrieben und die Verarbeitung der restlichen *attribute* Elemente fortgesetzt.
Nachdem die Attributdeklaration geschrieben wurde, wird in der *endElement()* Methode das *attributes* Element vom *current* Element entfernt und der Parse-Vorgang setzt sich fort.
Das Schreiben einer Instanz der Zieldatei erfolgt immer dann, wenn ein schließendes Tag eines *instance* Elements beim Parsen angetroffen wird. Analog zum Schreiben der Attributdeklaration, wird dazu eine

```
public class ArffTransformer implements ContentHandler{
  ...
 public void endElement(String namespaceURI, String localName, String qName) {
  ...
  if(localName.equals("attributes")){
   if(this.current.getParentElement().getName().equals("attribute")){
    this.oneStepBack();
   }
   else{
    this.oneStepBack();
    this.attributeElements = new LinkedList(this.current.getChild("attributes").getChildren("attribute"));
    Collections.sort(this.attributeElements,new SortByIndex());
    this.attributeElements = this.removeFormulaC50Attributs(this.attributeElements);
    this.writeAttributeDeclaration(this.attributeElements);
    this.current.removeChild("attributes");
   }
  }
 }

 private void writeAttributeDeclaration(LinkedList attrDecl){
  Element currentAttr;
  for(int i=0;i<attrDecl.size();i++){
   currentAttr = (Element)attrDecl.get(i);
   if(currentAttr.getAttributeValue("type").equals("string")){
    this.writer.println("@attribute "+currentAttr.getAttributeValue("name")+" "
    +currentAttr.getAttributeValue("type"));
   }
   if(currentAttr.getAttributeValue("type").equals("relational")){
    this.writer.println("@attribute "+currentAttr.getAttributeValue("name")+" relational");
    LinkedList relAttr = new LinkedList(currentAttr.getChild("attributes").getChildren("attribute"));
    Collections.sort(relAttr,new SortByIndex());
    this.writeAttributeDeclaration(relAttr);
    this.writer.println("@end "+currentAttr.getAttributeValue("name"));
   }
  }
 }
...
}
```

Abbildung 38: Ausschnitt aus der ArffTransformer Klasse

Liste mit den *value* Elementen einer Instanz erzeugt, bevor die Methode *writeInstance()* aufgerufen wird. Diese Methode übernimmt zwei Parameter: die Liste mit den *attribute* Elementen und die Liste mit den *value* Elementen. Das Schreiben der Instanzen ist am Beispiel der *C50DataTransformer* Klasse in Abbildung 39 zu sehen.

Da der letzte Wert einer Instanz im C5.0-Format der Wert des Klassenattributs ist, wird in der *writeInstance()* Methode der *C50DataTransformer* Klasse zunächst das Klassenattribut in einer separaten Variable abgelegt (*this.setClassAttribute(attr)*) und von der Liste mit den *attribute* Elementen entfernt. In der folgenden for-Schleife wird über die Liste der *attribute* Elemente iteriert. Das gerade durch den Iterator bezeichnete *attribute* Element wird in der Variable *currentAttr*, und der in der Instanz dazugehörige Wert des *value* Elements in der Variable *currentValue,* zwischengespeichert. Für die Ermittlung des Wertes des value Elements, wird die Methode *getValueElement()* eingesetzt. Diese ermittelt den Wert über die ID/IDREF-Referenz zwischen dem *ref* Attribut eines *value* Elements und dem *name* Attribut

```
public class C50DataTransformer implements ContentHandler{
  ...
 private void writeInstance(LinkedList attr,LinkedList values){
  this.setClassAttribute(attr);
  attr.remove(this.classAttr);
  for(int i=0;i<attr.size();i++){
   currentAttr = (Element)attr.get(i);
   currentValue = this.getValueElement(currentAttr.getAttributeValue("name"),values);
   if(currentValue==null){
    line = line + "0,";
   }
   else{
    if(currentValue.getText().equals("")){
     line = line + "?,";
    }
    else{
     if(currentAttr.getAttributeValue("type").equals("string")){
      line = line + this.quoteEscapeText(currentValue.getText()) + ",";
     }
     if(currentAttr.getAttributeValue("type").equals("date")){
      line = line + TimeFormatChecker.converteDate(currentAttr.getChildText("dateFormat"),
      currentValue.getText()) + ",";
     }
    }else
   }else
  }for-schleife
  // Wert des Klassenattributs an line anhängen
  this.writer.println(line);
 }

...
}
```

Abbildung 39: writeInstance() Methode der C50DataTransformer Klasse

eines *attribute* Elements, wozu sie den Wert des *name* Attributes und die Liste der *value* Elemente als Parameter übernimmt.
Der vor der Schleife noch leere String *line* hat die Aufgabe eine Instanz aufzunehmen, die nach Vervollständigung des Strings in die Zieldatei geschrieben wird. In Abhängigkeit der Werte der *value* Elemente und dem Attributtyp, den die *attribute* Elemente darstellen, wird sukzessiv der String *line* aufgebaut. In Bezug auf ein *value* Element sind drei Fälle zu unterscheiden:

1. Für das aktuelle *attribute* Element existiert kein *value* Element in der Instanz. Das bedeutet, dass das Datensatzattribut in dieser Instanz den Wert 0 hat und dem String *line* wird somit eine *0* hinzugefügt.

2. Das *value* Element ist leer, wodurch ausgedrückt ist, dass der Wert an dieser Stelle unbekannt ist. Der String *line* wird somit um das Zeichen *?* ergänzt.

3. Das *value* Element enthält einen Wert, um den der String *line* ergänzt wird. Hinsichtlich des dazugehörigen Attributtyps ist der Wert noch zubearbeiten, bevor er dem String *line* hinzugefügt wird.

Für den dritten Fall ist in der Abbildung 39 ein Beispiel für den Attributtyp *string* und den Attributtyp

date gegeben. Stellt das *value* Element den Wert eines Attributs vom Typ *string* dar, müssen formatspezifische Sonderzeichen des Wertes maskiert werden. Im C5.0-Format sind das beispielsweise Zeichen wie der Doppelpunkt, einfacher Punkt, Fragezeichen und einige andere. Derartige Zeichen werden durch ein Backslash maskiert und anschließend wird der gesamte Wert in einfache Anführungsstriche gesetzt, sodass der Wert durch die verarbeitende Software als String-Wert interpretiert wird. Das Bearbeiten solcher Werte erfolgt durch die Methode *quoteEscapeText()*.
Attributwerte des Typs *date* müssen auf eines der drei verschiedenen Datumsformate des C5.0-Formats abgebildet werden. Dazu wird die Methode *convertDate()* der Klasse *TimeFormatChecker* eingesetzt. Diese Methode übernimmt als Parameter den Inhalt des *dateFormat* Elements, das das Datumsformat beschreibt und den Wert aus dem *value* Element. Nachdem die Methode *convertDate()* den Wert auf eins der drei möglichen Datumsformate abgebildet hat, wird das konvertierte Datum an den String *line* angehängt. Wenn alle Attributwerte in der for-Schleife verarbeitet wurden, wird abschließend der Wert des Klassenattributs in gleicher Weise an den String *line* angefügt. Der String *line*, der nun die vollständige Instanz enthält, wird in das Zieldokument geschrieben.

6.5 Einchecken von Datensätzen

Der gesamte Vorgang des Eincheckens gliedert sich in die Teilschritte Datei-Upload, Validieren der Upload-Datei und Ermittlung der Meta-Daten des Datensatzes der Upload-Datei und das persistente Speichern des Datensatzes im anwendungsspezifische Speicherformat. Diese Teilschritte verteilen sich auf das Servlet *ValidateUpload* und auf die zwei Beans *CheckIn* und *DatasetCreator*. Das Servlet *ValidateUpload*, das den Datei-Upload realisiert und die *CheckIn* Klasse verwendet, um die Validierung der Upload-Datei durchzuführen, ist in ihren wesentlichen Bestandteilen in Abbildung 40 zu sehen.
Beim Ausführen der *doPost()* Methode wird zuerst ein Objekt der *ServletFileUpload* Klasse und der *checkIn* Klasse erzeugt. Die erst genannte Klasse gehört zur Apache Upload API. Mit dieser Klasse können Request verarbeitet werden, die neben Parameter aus Formularfeldern auch Parameter in Form von Upload-Dateien enthalten. Um zwischen den verschiedenartigen Parametern unterscheiden zu können, wird ein Iterator mit Hilfe des *ServletFileUpload* Objekts erzeugt, der das *request* Objekt parst und eine Liste der Request-Parameter bereitstellt (*FileItemIterator = upload.getItemIterator(request)*). In der darauffolgenden while-Schleife wird über diese Liste iteriert, wobei für jeden Parameter der Liste ein *InputStream* Objekt bereitgestellt wird, um auf den Parameter zu zugreifen. Für jeden Parameter (*item* Objekt) wird durch die if-Abfrage überprüft, ob es sich dabei um eine Upload-Datei oder um einen Eintrag eines Formularfeldes handelt. Im ersten Fall wird der Input-Stream gelesen und in eine temoräre Datei abgelegt. Im anderen Fall wird der Input-Stream als String an das *checkIn* Objekt übergeben. Wenn alle Parameter des Request entgegengenommen wurden, wird die Upload-Datei an das *checkIn* Objekt übergeben und die Methode *validate()* aufgerufen, um die Validierung der Parameter durchzuführen. Die Upload-Datei, die den einzucheckenden Datensatz enthält muss sich im ARFF- oder sparse ARFF-Format befinden. Dieses Format liefert, mit Ausnahme der Information, welches Attribut das Klasssenattribut ist, alle Informationen, die benötigt werden, um einen Datensatz auf alle anderen unterstützten Dateiformate abzubilden. Neben der Datensatzdatei sind weitere Angaben durch den Benutzer für das Integrieren eines Datensatzes erforderlich. Im einzelnen sind das:

- Name des Datensatzes
- optionale kurze Beschreibung des Datensatzes
- der Index des Klassenattributs
- eine Angabe, ob das Klassenattribut vom Typ ordinal ist

```
public class ValidateUpload extends HttpServlet{
  ...
  public void doPost(HttpServletRequest request, HttpServletResponse response){
   ServletFileUpload upload = new ServletFileUpload();
   CheckInUpload checkIn = new CheckInUpload();
   try{
    FileItemIterator iter = upload.getItemIterator(request);
    while (iter.hasNext()) {
     FileItemStream item = iter.next();
     InputStream uploadStream = item.openStream();
     if (!item.isFormField()){
      uploadStream lesen und in temporäre Datei schreiben
      ...
      checkIn.setUploadFile(file);
      checkIn.validate();
     }
     else{
      //Formularparameter an checkIn Objekt übergeben
     }
    }
   }
   catch(ParseException e){
    checkIn.setFileError(e.getMessage());
    RequestDispatcher rd = request.getRequestDispatcher("/jsp/CheckInUpload.jsp");
    rd.forward(request, response);
   }
   catch(Exception e){
    request.setAttribute("error",e);
    RequestDispatcher rd = request.getRequestDispatcher("/error");
    rd.forward(request, response);
   }
  }
...
}
```

Abbildung 40: doPost() Methode des Servlet ValidateUpload

- eine Angabe, ob es sich bei dem Datensatz um ein Testdatensatz handelt

Die Validierung der Parameter durch das *checkIn* Objekt, umfasst folgende Punkte:

- Überprüfung, ob der Name und die Kurzbeschreibung ausschließlich zulässig Zeichen enthalten
- Überprüfung, ob bereits ein Datensatz existiert, der den gleichen Namen hat, wie der des einzucheckenden Datensatzes
- handelt es sich bei dem einzucheckenden Datensatz um einen Testdatensatz, wird geprüft, ob ein gleichnamiger Datensatz bereits existiert
- Überprüfung, ob der angegebene Index eine positive Ganzzahl darstellt
- Überprüfung, ob die Upload-Datei dem ARFF-Format entspricht

Es sind zwei Vorgehensweisen bei der Erkennung von fehlerhaften Parametern zu unterscheiden. Bei den ersten vier, der oben aufgeführten Überprüfungen wird eine Fehlermeldung in eine dafür vorgesehene Klassenvariable abgelegt und der Request wird durch das *ValidateUpload* Servlet zurück an die

CheckInUpload JSP geleitet, die den Inhalt der entsprechenden Klassenvariable der *CheckIn* Bean anzeigt. Bei der Überprüfung des Dateiformats wird die *Instances* Klasse der Weka API eingesetzt. Diese parst die ARFF-Datei und wirft eine *IOException*, wenn die Datei nicht gelesen werden kann, weil das ARFF-Format nicht eingehalten wurde. Diese *IOExceptions* werden innerhalb der *validate()* Methode der *CheckIn* Klasse abgefangen und es werden stattdessen *ParseExceptions* geworfen. Diese können nun im Servlet *ValidateUpload* abgefangen und von anderen *IOExceptions* unterschieden werden. In dem entsprechenden catch-Block des *ValidateUpload* Servlets aus Abbildung 40 wird schließlich die Fehlermeldung der *ParseException* in eine Klassenvariable des *checkIn* Objekts abgelegt, damit diese nach der Request-Weiterleitung in der *CheckInUpload* JSP angezeigt werden können.
Ausschnitte aus der *validate()* Methode der *CheckIn* Klasse sind in Abbildung 41 gezeigt.

```
public class CheckInUpload{
 ...
 public void validate() throws FileNotFoundException, ParseException, IOException, SAXException{
  this.validateName();
  ...
  try{
   FileReader reader = new FileReader(this.uploadFile);
   Instances header = new Instances(reader);
   ...
   header.setClassIndex(this.classIndex);
   if(header.classAttribute().isNumeric()){
    this.classAttrType="numeric";
   }
   ...
   this.instances = header.numInstances();
   this.uri = this.name.replace(" ","") + ".xml";
   this.missingValues = this.calculateMissingValues(header);
   ...
  }
  catch(IOException e){
   throw new ParseException(e.getMessage(),0);
  }
 }
...
}
```

Abbildung 41: validate() Methode der CheckIn Klasse

Für die Überprüfungen, die nicht das Dateiformt betreffen, werden durch die *validate()* Methode separate Methoden aufgerufen. In Abbildung 41 ist dies am Beispiel der Methode *validateName()* gezeigt, die sämtliche Überprüfungen hinsichtlich des Datensatznamens durchführt. In dem folgenden try-Block wird ein *FileReader* Objekt erzeugt, das dem Konstruktor der *Instances* Klasse übergeben wird. Das *Instances* Objekt liest nun von diesem Reader die ARFF-Datei und wirft gegebenenfalls *IOExceptions*, welche in dem catch-Block abgefangen werden. Um solche *IOException* Objekte von anderen *IOException* Objekten im Servlet *ValidateUpload* unterscheiden zu können, wird vom catch-Block ein *ParseExecption* Objekt geworfen, dem die Fehlermeldung des *IOException* Objekts übergeben wird.
Des weitern werden im try-Block die Meta-Daten des Datensatzes durch Unterstützung der Methoden der *Instances* Klasse ermittelt, sofern keine *IOExceptions* geworfen werden. In Abbildung 41 ist dies exemplarisch für einige Meta-Daten aufgeführt. Nachdem das Klassenattribut im *Instances* Objekt gesetzt ist (*header.setClassIndex(this.classIndex)*), kann der Typ des Klassenattributs ermittelt werden. Im Beispiel

ist das für den Typ numeric gezeigt. Als weitere Beispiele sind die Ermittlungen der Anzahl der Instanzen, der URI und des Prozentsatzes der fehlenden Werte gezeigt. Alle Meta-Daten werden in Klassenvariablen der *checkIn* Klasse gespeichert, sodass ein *CheckIn* Objekt nach einer erfolgreichen Validierung neben dem Datensatz im ARFF-Format, sämtliche Meta-Daten zu diesem Datensatz kapselt.
Das persistente Abspeichern des Datensatzes stellt eine Transformation vom ARFF-Format in das anwendungsspezifische Speicherformat dar und wird durch Methoden der *DatasetCreator* Klasse durchgeführt. Durch die Ausführung der *doPost()* Methode des *CreateNewDataset* Servlet, wird ein Objekt der Bean-Klasse *DatasetCreator* erzeugt. Diesem wird das im Request Scope stehende *CheckIn* Objekt übergeben und anschließend wird die Methode *createDataset()* aufgerufen, die das Schreiben der Datensatzdatei einleitet. Ausgehend von der *createDataset()* Methode werden sukzessiv weitere Methoden aufgerufen, die jeweils einen bestimmte Teil der Datensatzdatei schreiben. Für (fast) jedes XML-Element des anwendungsspezifischen Speicherformats, existiert eine separate Methode. Die Abbildung 42 zeigt einen Ausschnitt der *DatasetCreator* Klasse, der die Vorgehensweise beim Schreiben der Datensatzdatei veranschaulicht.

```
public class DatasetCreator{
  ...
  public void createDataset(){
   if(this.checkIn.getDataOrTest().equals("data")){
     this.createData();
     DatasetDBWriter dbWriter = new DatasetDBWriter(this.applPath+"data"+this.sep+"datasetDB.xml");
     dbWriter.writeDBEntry(...);
   }
   else{
     this.createTestdata();
     ...
   }
   private void createData(){
     DataWriter datasetWriter = new DataWriter(out);
     Instances head = this.createWekaInstances();
     if(this.classAttr.isNumeric()){
       discreteHead = this.createDiscreteWekaInstances(head,this.checkIn.getClassIndex()+1);
     }
     this.createStartDatasetElement(datasetWriter);
     this.createDocuElements(datasetWriter);
     this.createHeaderElement(datasetWriter,head,discreteHead);
     this.createDataElement(datasetWriter,head,discreteHead);
     this.createEndDatasetElement(datasetWriter);
   }
   private void createHeaderElement(datasetWriter, Instances head, Instances discreteHead) {
     datasetWriter.startElement(this.datasetNsUri,"header");
     datasetWriter.dataElement(this.datasetNsUri,"relation",head.relationName());
     this.createAttributesElements(datasetWriter,head,discreteHead);
     datasetWriter.endElement(this.datasetNsUri,"header");
   }
...
}
```

Abbildung 42: Ausschnitt aus der DatasetCreator Klasse

Die *createDataset()* Methode überprüft, ob es sich bei dem einzucheckenden Datensatz um Trainings- oder Testdaten handelt und ruft dann entweder die Methode *createData()* oder die Methode *createTestdata()* auf. In Abbildung 42 ist beispielhaft die Methode *createData()* zu sehen. Diese erzeugt ein Objekt

der Klasse *DataWriter*, welches die XML-Elemente in die Datensatzdatei schreibt. Außerdem wird ein *Instances* Objekt durch die *createWekaInstances()* Methode erzeugt, das den Zugriff auf die Inhalte der ARFF-Datei ermöglicht. In dem Fall, dass das Klassenattribut numerisch ist, wird ein zweites *Instances* Objekt erzeugt, das ebenfalls die ARFF-Datei repräsentiert, jedoch sind in diesem Objekt die Werte des numerischen Klassenattributs auf eine Menge von diskreten Werten abgebildet. Die Methode *createDiscreteWekaInstances()* verwendet dazu die Weka Klasse *weka.filters.unsupervised.attribute.Discretize*, die den Wertebereich des numerischen Klassenattributs in den Instanzen auf zehn gleichmäßige Intervalle abbildet und somit das numerische Attribut als ein nominelles Attribut darstellt. Wie bereits erwähnt, werden die diskreten Werte benötigt, um einen solchen Datensatz in das C4.5- oder C5.0-Format umzuwandeln.
Anschließend werden in der Reihenfolge wie die Elemente in der Datensatzdatei erscheinen, die Methoden aufgerufen, die die jeweiligen Elemente in die Zieldatei schreiben. Wie im Beispiel zu sehen, existieren zum Schreiben des *dataset* Elements und seine Kindelemente jeweils eine separate Methode, wobei sich das Schreiben des *dataset* Elements auf zwei Methoden verteilt. Jede Methode übernimmt als Parameter das Writer Objekt und je nach Bedarf die *Instances* Objekte. In Abbildung 42 ist die Methode *createHeaderElement()* zum Schreiben des *header* Elements der Datensatzdatei stellvertretend abgebildet. Diese Methode schreibt nun das Start-Tag des *header* Elements und das *relation* Element, und ruft, bevor es das schließende Tag des *header* Elements schreibt, die Methode *createAttributsElements()* auf. Diese Methode wiederum ruft weitere Methoden auf, die in Abhängigkeit des Attributtyps die *attribute* Elemente in die Zieldatei schreiben und gegebenenfalls weitere Methoden zum Schreiben von Elementen aufrufen. Auf diese Weise wird die Zieldatei zeilenweise geschrieben. Wenn die *createData()* Methode ausgeführt wurde und somit die Datensatzdatei vollständig geschrieben wurde, wird durch die *createDataset()* Methode ein neuer Eintrag in die Meta-Daten Datei für den neuen Datensatz vorgenommen.
Das Schreiben eines Testdatensatzes erfolgt in gleicher Weise und wird weitgehend durch die selben Methoden abgewickelt. Im Falle eines Testdatensatzes muss jedoch vor dem Schreiben der Datei überprüft werden, ob der einzucheckende Testdatensatz kompatibel zu dem zugrunde liegenden Datensatz ist. Dazu steht die Methode *isCompatible()* zur Verfügung. Diese vergleicht die Attribute beider Datensätze bezüglich deren Namen, der Reihenfolge, der Anzahl und des Datentyps. Liegen dies bezüglich Abweichungen zwischen den Datensätzen vor, erzeugt die Methode entsprechende Fehlermeldungen, die in der Check-In Oberfläche angezeigt werden.

6.6 Behandlung von Exceptions

Hinsichtlich dem Auftreten von Exceptions während der Laufzeit der Anwendung lassen sich vier Fälle unterscheiden:

1. *BreakException*, um einen Parse-Vorgang vorzeitig abzubrechen
2. *ParseException* beim Validieren einer Upload-Datei
3. Sonstige Exceptions, die von Bean-Objekten oder von Objekten des helpers Pakets ausgelöst werden und durch die Servlets behandelt werden
4. Exceptions, die nicht von Servlets abgefangen und behandelt werden

Die beiden Exceptions aus Punkt 1. und 2. werden gezielt geworfen, um den Programmablauf zu beeinflussen und wurden in den vorangegangenen Kapiteln bereits beschrieben. Alle anderen Exceptions werden, ausgehend von der auslösenden Methode, solange weitergegeben, bis die Exceptions schließlich

in einem Servlet abgefangen und bearbeitet werden. Die betreffenden Servlets verarbeiten die abgefangenen Exceptions, indem sie den ursprünglichen Request an das Servlet *ErrorHandle* weiterleiten. Dieses Servlet schreibt die Fehlermeldung des Exception Objekts in eine Log-Datei und leitet den Request wiederum weiter an die *Error* JSP, die dem Benutzer mitteilt, dass ein Fehler bei der Programmausführung aufgetreten ist. Diejenigen Exceptions, die nicht durch die Servlets behandelt werden können, werden vom Web-Server behandelt. Durch den sogenannten Web Deployment Descriptor (siehe Kapitel 6.8) ist das Servlet *ErrorHandle* als "Error-Page" definiert, die vom Web-Server zur Behandlung von Exceptions aufgerufen wird. In Abbildung 43 ist der wesentliche Quelltext des Servlet *ErrorHandle* dargestellt.

```
public class ErrorHandle extends HttpServlet{
  ...
  public void doPost(HttpServletRequest request, HttpServletResponse response){
    Date date = new Date();
    Throwable ex;

    if(request.getAttribute("error") == null){
      ex = (Throwable)request.getAttribute("javax.servlet.error.exception");
    }
    else{
      ex = (Throwable)request.getAttribute("error");
    }

    File logFile = new File(URI-Log-File);
    PrintStream output = new PrintStream(new FileOutputStream(logFile));
    output.println(date.toString());
    ex.printStackTrace(output);
    output.close();

  }
  ...
}
```

Abbildung 43: doPost() Methode des ErrorHandle Servlets

In der *doPost()* Methode des Servlets wird zuerst ein Datumsobjekt erzeugt, das das aktuelle Datum mit Uhrzeit enthält, und die Variable *ex* deklariert, die das Exception Objekt aufnehmen wird. Die Klasse *Throwable* ist die Superklasse aller Exception Klassen. In der folgenden if-Abfrage wird geprüft, ob die Exception durch den Web-Server behandelt wurde, oder ob die Exception durch ein Servlet abgefangen wund in der Request Scope gestellt wurde. Im erste Fall existiert kein Attribut namens *error* im Request Scope, jedoch das Attribut *javax.servlet.error.exception*, das durch den Server bereitgestellt wird, wenn dieser eine Exception behandeln muss. Nachdem das Exception Objekt der Variable *ex* zugewiesen wurde, wird die Log-Datei mit dem aktuellen Datum und dem gesamten Stacktrace des Exception Objekts beschrieben.

6.7 Implementierung der Benutzeroberflächen

Die Benutzeroberfächen präsentieren sich dem Anwender als XHTML-Seiten, die durch JSPs dynamisch erzeugt werden.

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
 <head>
  <title>ML-Datasets check-out form</title>
  <link rel="stylesheet" href="../css/Form.css" type="text/css"></link>
 </head>
 <body>
 ...
  <form action="/validate" method="post">
   <fieldset class="attr">
    <legend class="leg">attributes</legend>
    <span class="attrNum">number of numeric attributes:</span>
    <select name="opNumericAttr" class="opNumAttr">
     <c:choose>
      <c:when test="${form.opNumericAttr == 'equal' or form.opNumericAttr == null}">
       <option selected="selected" class="op">equal</option>
       <option class="op">less than</option>
       <option class="op">greater than</option>
      </c:when>
      <c:when test="${form.opNumericAttr == 'less than'}">
       <option>equal</option>
       <option selected="selected">less than</option>
       <option>greater than</option>
      </c:when>
      <c:otherwise>
       <option>equal</option>
       <option>less than</option>
       <option selected="selected">greater than</option>
      </c:otherwise>
     </c:choose>
    </select>
    <input class="inNumAttr" type="text" name="numNumericAttr"
     value="<c:out value="${form.numNumericAttr}"/>"/>
    <span class="errorNumAttr"><c:out value="${form.numNumericAttrError}"/></span><br/>

    ...
    </fieldset>
    <input class="sub" type="submit" value="request"/>
   </form>
  ...
 </body>
</html>
```

Abbildung 44: Ausschnitt aus der Form JSP

Der Kern dieser XHTML-Seiten besteht in einem form-Element, welches die Eingabefelder besitzt, über die der Benutzer Daten an die Anwendung senden kann. In Abbildung 44 ist ein Ausschnitt der *Form* JSP gezeigt, die die Datensatz-Suchmaske erzeugt.
In der Abbildung sind die JSP Elemente fett gedruck. Der statische Teil der JSP ist ein valides XHTML-Dokument in der Version 1.0. Das *head* Element des XHTML-Dokuments enthält neben einem Titel ein *link* Element, welches auf die zu verwendenden CSS verweist. Das *body* Element enthält das Formular (*form* Element), das die eigentliche Suchmaske mit ihren Eingabe- und Auswahlfelder darstellt. Das

action Attribut im *form* Element, bezeichnet das Servlet, das den Request entgegennimmt, der durch Absenden dieses Formulars ausgelöst wird. Außerdem wird durch das *method* Attribut die HTTP-Methode festgelegt, mit der der Request erfolgt. Innerhalb des *form* Elements befinden sich die HTML-Elemente, die die Formularfelder repräsentieren. In Abbildung 44 sind die Eingabefelder für die Anzahl der numerischen Attribute des Datensatzes abgebildet. Diese bestehen aus einem *select* und einem *input* Element. Das *select* Element stellt drei Optionen zur Auswahl, die durch *option* Elemente dargestellt werden. Der Benutzer kann hier zwischen *equal*, *less than* und *greater than* auswählen. Beim Abrufen dieser Seite ist eins der Optionsfelder durch das *selected* Attribut bereits vorselektiert. Welche Option das ist, wird durch JSP Aktionselemente festgelegt. Durch die EL-Ausdrücke in den *c:when* Aktionselementen wird die Eigenschaft *opNumericAttr* des im Request Scope befindlichen Bean Objekts *form* untersucht, um den Inhalt des *c:when* Elements in das XHTML-Dokument zu übernehmen, dessen EL-Ausdruck *true* liefert. Ist der Wert der Eigenschaft beispielsweise *equal* oder wurde die Eigenschaft noch nicht initialisiert (*opNumericAttr == null*), wird die Option *equal* vorselektiert, da der entsprechende Block von *options* Elemente in das XHTML-Dokument geschrieben wird. Alle anderen Felder der Suchmaske sind ebenfalls mit den aktuellen Werten der entsprechenden Eigenschaften der *form* Bean belegt, damit der Benutzer bei einer fehlerhaften Eingabe und der anschließenden Zurückleitung der Anfrage an die Form JSP, nicht alle Eingabefelder erneut ausfüllen muss. Bei *input* Elementen vom Typ *text*, erfolgt die Vorbelegung durch das *value* Attribut. Wie in Abbildung 44 am Beispiel des *input* Element namens *numNumericAttr* zu sehen ist, ergibt sich der Wert des *value* Attributs durch das Aktionselement *c:out*, das den Wert der Eigenschaft *numNumericAttr* liefert. Liegt eine Fehlermeldung für dieses Eingabefeld vor, d.h. besitzt die Eigenschaft *numNumericAttrError* der *form* Bean einen Wert, so wird diese im anschließenden *span* Element angezeigt, so dass der Benutzer die Einagbe korrigieren kann.
Das *input* Element mit dem Wert *submit* im *type* Attribut stellt sich im Web-Browser als ein Bestätigungsknopf dar, dessen Betätigung den Request auslöst, der die Benutzereingaben an den Server sendet. Dieser Request wird durch das Servlet *ValidateForm* verarbeitet, das die übertragenen Parameter über ihre Namen (durch *name* Attribute der betreffenden HTML-Elementen festgelegt) ansprechen kann.
Das Erscheiningsbild der Oberflächen im Web-Browser ist durch CSS definiert. Des weiteren wird die Oberfläche, die die Treffer anzeigt, durch JavaScript unterstützt. In Abbildung 45 ist ein Ausschnitt der *Hits* JSP angezeigt. Außerdem enthält die Abbildung eine JavaScript Funktion, die die durch die JSP generierte XHTML-Seite unterstützt, sowie einen Ausschnitt der verwendeten CSS.
Durch das *link* und *script* Element werden die Dateien, die die JavaScript Funktionen und die CSS enthalten, in die XHTML-Seite eingebunden. Um alle Treffer einer Suchanfrage anzuzeigen, wird in der forEach-Schleife über die Liste alle *Meta-Data* Objekte iteriert. Für jeden Datensatz wird ein *span* Element erzeugt, mit einem Attribut namens *onClick*, das den Namen der JavaScript Funktion *showFeatures()* enthält. Der Inhalt des Elements selbst ist der Name des Datensatzes. Außerdem wird für jeden Datensatz ein *div* Element angelegt, das die Meta-Daten des jeweiligen Datensatzes darstellt. In Abbildung 45 ist das für die Anzahl der numerischen, der nominellen und der booleschen Attribute angedeutet. Dieses *div* Element besitzt neben dem Attribut *class* mit dem initialen Wert *attrFeatures* ein Attribut namens *id*, welches den Namen des Datensatzes beinhaltet.
Die Darstellung des *div* Elements, das die Meta-Daten beinhaltet, ist durch die Stylesheets aus Abbildung 45 (unten) beschrieben. Der Ausdruck *div.attrFeatures* bezeichnet alle *div* Elemente, die ein *class* Attribut mit dem Wert *attrFeatures* besitzen. Der Auszug des Stylesheets zeigt neben der Definition der Größe (*width*, *height*) und der Farbe (*background-color*) des *div* Elements, die Definition der Sichtbarkeit des Elements im Browser. Da die Eigenschaft *visibility* mit dem Wert *hidden* belegt ist, wird dieses Element im Browser nicht angezeigt.

Hits JSP:

```
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
 <head>
  <title>ML-Datasets hits</title>
  <link rel="stylesheet" href="../css/Hits.css" type="text/css"></link>
  <script type="text/javascript" language="JavaScript" src="../javaScript/hide.js"></script>
 </head>
 <body>
  <c:forEach items="${query.datasets}" var="current">
   <span class="dataset" onClick="showFeatures(this);">
    <c:out value="${current.nameDataset}"/>
   </span>
   <div class="attrFeatures" id="${current.nameDataset}">
    <span>numeric: <c:out value="${current.numNumericAttr}"/></span>
    <span>nominal: <c:out value="${current.numNominalAttr}"/></span>
    <span>boolean: <c:out value="${current.numBooleanAttr}"/></span>
   </div>
  </c:foreach>
 </body>
</html>
```

hide.js:

```
function showFeatures(obj){
 if(document.getElementById(obj.firstChild.nodeValue).getAttribute("class") == "attrFeatures"){
  document.getElementById(obj.firstChild.nodeValue).setAttribute("class","attrFeaturesVisible");
 }
 else{
  document.getElementById(obj.firstChild.nodeValue).setAttribute("class","attrFeatures");
 }
}
```

Hits.css:

```
div.attrFeatures{
  ...
  width: 700px;
  height: 150px;
  background-color: #e1e1e1;
  visibility: hidden;
  }
div.attrFeaturesVisible{
  ...
  width: 700px;
  height: 150px;
  background-color: #e1e1e1;
  visibility: visible;
  }
```

Abbildung 45: Hits JSP mit JavaScript Funktion und CSS

Beim Abruf dieser Seite wird dem Benutzer also zunächst nur eine Liste der Namen aller Datensätze angezeigt.
Durch das *onClick* Attribut ist das *span* Element zur Darstellung des Datensatznamens mit einem sogenannten EventHandler ausgestattet, sodass das Ereignis "Mausklick auf das span Element" den Aufruf

der JavaScript Funktion *showFeatures()* auslöst. Diese Methode manipuliert das auf das *span* Element folgende *div* Element, indem es dessen *class* Attributwert verändert. Ist der gerade aktuelle Wert *attrFeatures*, wird der Wert auf *attrFeaturesVisible* gesetzt, und umgekehrt. Durch das Ändern des Attributwerts, ändert sich auch das Stylesheet, das auf das *div* Element passt. Wie in Abbildung 45 zu sehen ist, unterscheidet sich das Stylesheet *div.attrFeaturesVisible* von dem ersten Stylesheet in der Eigenschaft *visibility*. Da hier der Wert *visible* ist, werden die *div* Elemente mit dem *class* Attributwert *attrFeaturesVisible* im Browser angezeigt. Durch ein wiederholtes Klicken auf ein *span* Element, also auf den Namen eines Datensatzes, wird das dazugehörige *div* Element, und somit die Meta-Daten des Datensatzes, ein- bzw. ausgeblendet.
Die Methode *showFeatures()* wird mit dem Parameter *this* aufgerufen, welches ein DOM-Objekt ist, das das Element darstellt, in dem sich das *onClick* Attribute befindet (also das *span* Element). In Abbildung 45 (Mitte) ist die Methode abgebildet. Durch die if-Abfrage wird der momentane Wert des *class* Attributs im *div* Element überprüft. Der Zugriff auf das *div* Element erfolgt über die DOM-Schnittstelle *getElementById()*. Da der Wert des *id* Attributs des *div* Elements dem Datensatznamen entspricht, wie auch der Inhalt des *span* Elements, und ein Datensatzname innerhalb der Anwendung einmalig ist, besteht eine eindeutige Referenz zwischen dem *span* und dem *div* Element. Auf den Wert des *span* Elements wird über die Knoteneigenschaft *firstChild* zugegriffen, die hier den Textknoten bezeichnet. Den Wert des *class* Attributs liefert die Schnittstelle *getAttribut()*. Das *class* Attribut des *div* Elements wird in Abhängigkeit des eigenen Werts auf den jeweils anderen Wert (*attrFeaturesVisible* oder *attrFeature*s) gesetzt.

6.8 Dateien und Verzeichnisstruktur der Anwendung

Abschließend wird die Verzeichnisstruktur der Anwendung beschrieben, die in Abhängigkeit des verwendeten Applikationsservers bestimmte Merkmale aufweisen muss. Alle serverbezogenen Angaben in diesem Kapitel beziehen sich auf den Apache Tomcat Server in der Version 6.0.
In Abbildung 46 ist die Verzeichnisstruktur der Anwendung dargestellt.
Das Wurzelverzeichnis der Anwendung ist das Verzeichis *datasets*, das alle Dateien und Verzeichnisse enthält, die für die Anwendung benötigt werden. Der Pfad zu diesem Verzeichnis muss dem Server über eine Konfigurationsdatei mitgeteilt werden. Im Falle des Tomcat Servers wird die Datei *server.xml* im *./conf* Verzeichnis der Tomcat Installation, um einen Eintrag der Form *<Context path="" docBase="../datasets" reloadable="true"/>* ergänzt. In dem unter *docBase* angegebenen Ordner erwartet der Tomcat Server ein Verzeichnis names *WEB-INF*, das neben dem sogenannten Web Deployment Descriptor (WDD) weitere Unterordner mit den verwendeten Java Klassen enthält.
Der WDD ist eine XML-Datei namens *web.xml*, über die die Anwendung konfiguriert werden kann. Dazu zählt die Festlegung, unter welcher URL ein Servlet erreichbar sein soll, wie es in Abbildung 47 anhand eines Beispiels gezeigt ist.
Das Binden eines Servlets an einen URL besteht aus zwei Einträgen in der Datei *web.xml*. Zuerst wird dem Servlet im *<servlet>* Element ein eindeutiger Name zugeordnet, wobei das Servlet über seinen vollständigen Klassenpfad bezeichnet werden muss (*servlets.GetForm*). Anschließend wird das Servlet im *<servlet-mapping>* Element an einen beliebigen URL gebunden. Im Beispiel ist das der URL *start*. Das Servlet ist nun unter der Internetadresse *http://hostName:8080/start* abrufbar. Wie in Kapitel 6.6 erwähnt, kann durch den WDD eine "Error-Page" festgelegt werden. Durch das *<error-page>* Element wird angegeben, unter welcher URL die "Error-Page" erreichbar ist und beim Auftreten welcher Exception diese aufgerufen werden soll. Da die Klasse *Throwable* die Superklasse aller Exception Klassen ist, wird die "Error-Page" hier bei jeder Exception aufgerufen.
Des weiteren müssen sich die verwendeten Java Klassen der Anwendung in den Unterverzeichnissen *clas-*

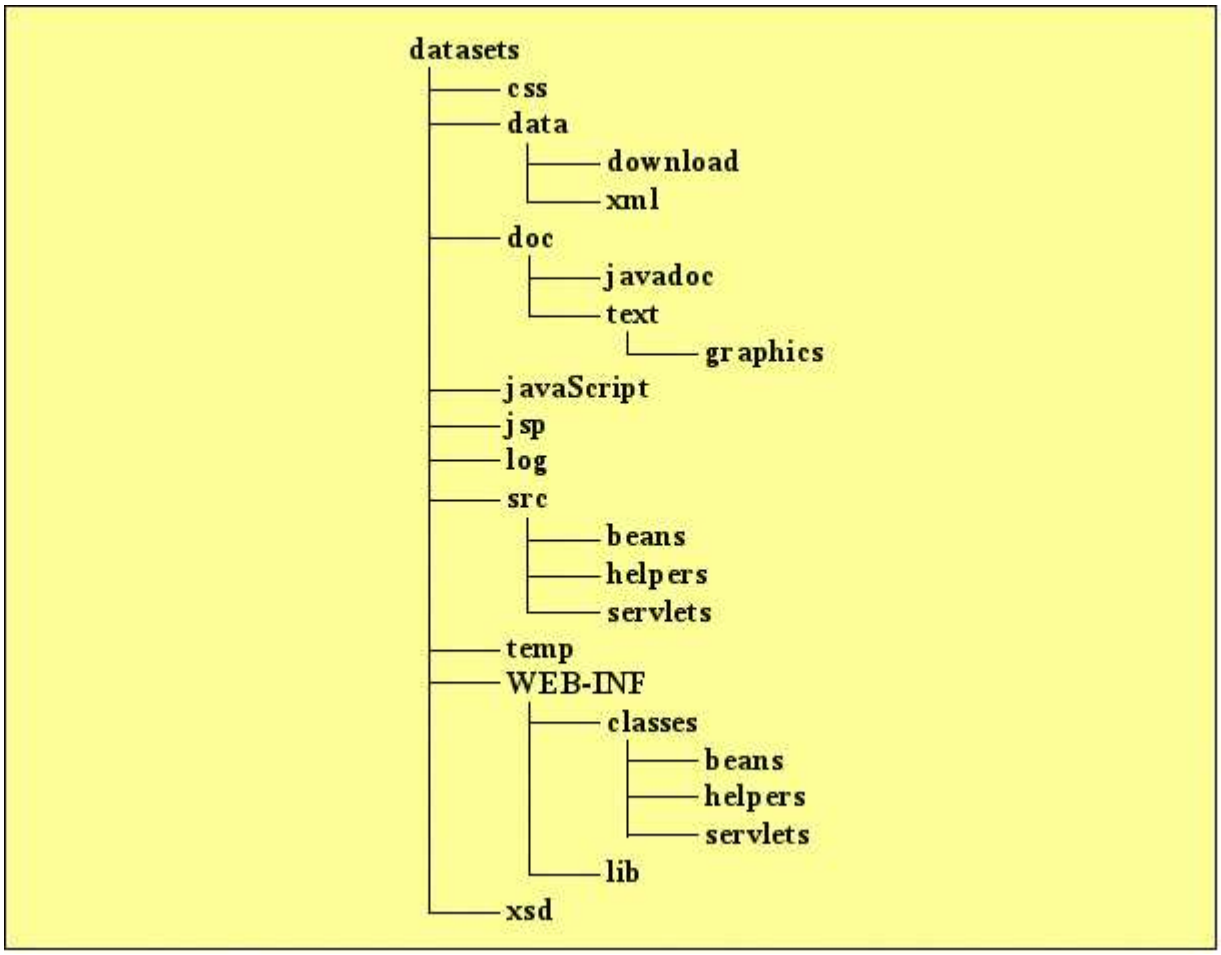

Abbildung 46: Verzeichnisstruktur der Anwendung

ses und *lib* des *WEB-INF* Verzeichnisses befinden. Das *classes* Verzeichnis ist für alle **.class* Dateien vorgesehen, die sich entsprechend ihrer Paketzugehörigkeit auf gleichnamige Ordner verteilen. Im Verzeichnis *lib* befinden sich für die Anwendung zusätzliche notwendige Java Bibliotheken in Form von **. jar* Dateien. Für die Anwendung dieser Arbeit sind das:

- *xercesImpl.jar* - SAX- und DOM-Parser des Apache Projekts Xerces
- *jdom.jar, jaxen-core.jar, saxpath.jar, jaxen-jdom.jar* - JDOM API einschließlich der XPath-Bibliotheken
- *xml-writer.jar* - Megginson XMLWriter API
- *jstl.jar, standard.jar* - APIs der JSP Standard Tag Library
- *weka.jar* - Weka APIs
- *commons-fileupload-1.2.jar, commons-io-1.3.2.jar* - FileUpload und IO API des Apache Projekts Commons

Alle anderen Dateien und Verzeichnisse im Wurzelverzeichnis *datasets* sind serverunabhängig und beziehen sich nur auf die Anwendung. Das Verzeichnis *jsp* enthält alle JSP Dateien und die in den Benutzeroberflächen verwendeten CSS- und JavaScript Dateien befinden sich in den Verzeichnissen *css* und *javaScript*. Die Dateien mit dem Quelltext der Java Klassen der Anwendung verteilen sich auf die Unterordner *beans*,

```
...
<servlet>
  <servlet-name>getForm</servlet-name>
  <servlet-class>servlets.GetForm</servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>getForm</servlet-name>
  <url-pattern>/start</url-pattern>
</servlet-mapping>
...
<error-page>
  <exception-type>java.lang.Throwable</exception-type>
  <location>/error</location>
</error-page>
```

Abbildung 47: Web Deployment Descriptor der Anwendung

servlets und *helpers* des Verzeichnisses *src*, wobei jeder Ordner die Dateien eines Java-Pakets zusammenfasst. In dem Verzeichnis *data* sind die Datensätze abgespeichert. Das Unterverzeichnis xml enthält alle Datensätze im anwendungsspezifischen XML-Format, während im Verzeichnis download die Datensätze abgelegt werden, dessen Transformation länger als 10 Sekunden dauert. In diesem Verzeichnis kann ein Datensatz also mehrfach, aber in verschiedenen Formaten vorliegen. Die Meta-Daten Datei befindet sich direkt im Ordner data. Zum Zwischenspeichern von Dateien, wie beispielsweise beim Upload einer Datensatzdatei, steht das Verzeichnis *temp* zur Verfügung und das Verzeichnis *log* nimmt die Log-Datei auf, in die die Fehlermeldung beim Auftreten einer Exception geschrieben wird. Diese Datei wird immer wieder überschrieben, sodass sie nur die jeweils letzte Fehlermeldung enthält. Im Verzeichnis *xsd* sind die XML-Schemata, die die XML-Formate für die Meta-Daten Datei und die Datensatzdatein definieren. Die Dokumentation der Anwendung befindet sich im Verzeichnis *doc*, dessen Unterverzeichnis *javadoc* die Dokumentation der Javaklassen im HTML-Format enthält. Die Datei *index.html* eigent sich als Ausgangspunkt der Dokumentation. Im Unterverzeichnis *text* befindet sich die schriftliche Ausarbeitung der Arbeit.

A Abkürzungen

API	Application Programming Interfaces
ARFF	Attribute-Relation File Format
CSS	Cascading Style Sheets
CSV	Comma Separated Value
DOM	Document Object Model
DTD	Document Type Definition
EIS	Enterprise Information System
EJB	Enterprise Java Beans
EL	Expression Language
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
Java EE	Java Plattform Enterprise Edition
Java SE	Java Plattform Standard Edition
JSTL	JSP Standard Tag Library
MVC	Model-View-Controller
SAX	Simple API for XML
URI	Uniform Resource Identifiers
URL	Uniform Resource Locator
W3C	World Wide Web Consortium
WDD	Web Deployment Descriptor
WWW	World Wide Webs
XHTML	Extensible HyperText Markup Language
XML	eXtensible Markup Language
XPath	XML Path Sprache
XRFF	eXtensible attribute-Relation File Format
XSLT	Extensible Stylesheet Language Transformations

B Quellen

[ARFF] The University of Waikato
Weka Software
http://weka.sourceforge.net/wekadoc/index.php/en:ARFF_%283.5.6%29

[BEBO] Bergmann, Olaf / Bormann, Carsten (2005): AJAX Frische Ansätze für das Web-Design, Berlin: TEIA Lehrbuch Verlag

[BERG] Bergsten, Hans (2003): JavaServer Pages, Sebastopol: O'Reilly Media

[BRET] McLaughlin, Brett (2001): Java & XML, Sebastopol: O'Reilly & Associates

[BONG] Bongers, Frank (2004): XSLT 2.0, Bonn: Galileo Press

[C45] Rulequest Research
C4.5 Software Release 8
http://www.rulequest.com/see5-comparison.html

[C50] Rulequest Research
C5.0 Software Release 2.05
http://www.rulequest.com/see5-unix.html#Data

[CSS] W3C Recommendation 12 May 1998
Cascading Style Sheets, level2
http://www.w3.org/TR/REC-CSS2/

[CSV] RFC - Common Format and MIME Type for CSV Files
http://tools.ietf.org/html/rfc4180

[DATA] W3C - Recommendation 23 January 2007
XQuery 1.0 and XPath 2.0 Data Model (XDM)
http://www.w3.org/TR/xpath-datamodel/

[DOM] W3C
Document Object Model (DOM)
http://www.w3.org/DOM/

[DOM2] W3C
Document Object Model (DOM) Level 2 HTML Specification
http://www.w3.org/DOM-Level-2-HTML/

[ECMA] ECMA International
Standard ECMA-262, ECMAScript Language Specification
http://www.ecma-international.org/publications/standards/Ecma-262.htm

[FILE] Apache Commons
FileUpload APIs
http://commons.apache.org/fileupload/

[HOLZ] Holzner, Steven (2001): XML Insider, München: Mark+Technik Verlag

[HTML] W3C - Recommendation 24 December 1999
HTML 4.01 Specification
http://www.w3.org/TR/html4/

[HUNT] Hunter, Jason (2001): Java Servlet Programming, Sebastopol: O'Reilly Media

[IO] Apache Commons
IO APIs
http://commons.apache.org/io/

[J2EETut01] Sun Microsystems
The Java EE 5 Tutorial
http://java.sun.com/javaee/5/docs/tutorial/doc/bnaay.html#bnabl

[J2EETut02] Sun Microsystems
The Java EE 5 Tutorial
http://java.sun.com/javaee/5/docs/tutorial/doc/bnagx.html

[J2EETut03] Sun Microsystems
The Java EE 5 Tutorial
http://java.sun.com/javaee/5/docs/tutorial/doc/bnair.html

[J2EETut04] Sun Microsystems
The Java EE 5 Tutorial
http://java.sun.com/javaee/5/docs/tutorial/doc/bnafo.html

[J2SE] Sun Microsystems
Java 2 Platform, Standard Edition
http://java.sun.com/javase/

[JAVA] Sun Mircosystems
The Java Tutorials
http://java.sun.com/docs/books/tutorial/getStarted/intro/definition.html

[JAVAX] Sun Microsystems
Java Servlet Technology
http://java.sun.com/products/servlet/index.jsp

[JDOM] JDOM Project
http://www.jdom.org/index.html

[JSCR] Mozilla Developer Center
Core JavaScript 1.5
http://developer.mozilla.org/en/docs/Core_JavaScript_1.5_Reference

[JSP] Seeboerger-Weichselbaum, Michael (2003): JSP mit Tomcat, München: Mark+Technik Verlag

[JSP21] Sun Microsystems
JavaServer Pages Technology
http://java.sun.com/products/jsp/

[JSTL] Sun Microsystems
JavaServer Pages Standard Tag Library
http://java.sun.com/products/jsp/jstl/

[MEGG] Megginson Technologies
XMLWriter
http://www.megginson.com/downloads

[MIT] Mitchell, Tom M. (1997): Machine Learning, Singapore: McCraw-Hill Book Co

[ML] Prof. Dr. Johannes Fürnkranz: Vorlesungsfolien Maschinelles Lernen - Symbolische Ansätze (WS 05/06), TU-Darmstadt

[RFC] RFC - Offizielle Webseite
http://www.rfc-editor.org/

[TOM] Apache Tomcat
http://tomcat.apache.org/

[URI] RFC - Uniform Resource Identifier (URI): Generic Syntax
http://www.rfc-editor.org/cgi-bin/rfcdoctype.pl?loc=RFC&letsgo=3986&type=http&file_format=txt

[WEB] Prof. Dr. Johannes Fürnkranz: Vorlesungsfolien Web Mining - Data Mining im Internet (SS 06), TU-Darmstadt

[WEKA] The University of Waikato
Weka Software
http://www.cs.waikato.ac.nz/ml/weka/

[WITT] Witten, Ian H. / Frank, Eibe (2005): Data Mining - Practical Machine Learning Tools and Techniques, San Francisco: Elsevier

[XERC] Apache XML Project
Xerces2 Java Parser
http://xerces.apache.org/xerces2-j/

[XHTML] W3C - Recommendation 26 January 2000, revised 1 August 2002
XHTML 1.0 The Extensible HyperText Markup Language (Second Edition)
http://www.w3.org/TR/xhtml1/

[XML] W3C - Recommendation 16 August 2006
Extensible Markup Language (XML) 1.0 (Fourth Edition)
http://www.w3.org/TR/xml/

[XMLname] W3C - Recommendation 16 August 2006
Namespaces in XML 1.0 (Second Edition)
http://www.w3.org/TR/REC-xml-names/

[XMLsch] W3C - Recommendation 28 October 2004
XML Schema Part 0: Primer Second Edition
http://www.w3.org/TR/xmlschema-0/

[XPath] W3C - Recommendation 23 January 2007
XML Path Language (XPath) 2.0
http://www.w3.org/TR/xpath20/

[XRFF] The University of Waikato
Weka Software
http://weka.sourceforge.net/wekadoc/index.php/en:XRFF_%283.5.6%29

C Liste der Datensätze

Folgend sind alle eingecheckten Datensätze der Anwendung einschließlich ihrer Meta-Daten alphabetisch sortiert aufgeführt. Die Quelle eines Datensatzes ist durch ein hochgestelltes Zeichen angegeben. Diese Quellen wurden verwendet:

- A - UCI-Repository, http://mlearn.ics.uci.edu/databases/
- B - Frequent Itemset Mining Dataset Repository, http://fimi.cs.helsinki.fi/data
- C - Weka (StatLib Datasets Archive), http://www.cs.waikato.ac.nz/ml/weka/index_datasets.html
- D - Weka (agricultural datasets), http://www.cs.waikato.ac.nz/ml/weka/index_datasets.html
- E - Weka (Arie Ben David), http://www.cs.waikato.ac.nz/ml/weka/index_datasets.html
- F - Weka(Georg Forman), http://www.cs.waikato.ac.nz/ml/weka/index_datasets.html
- G - Weka (UCI), http://www.cs.waikato.ac.nz/ml/weka/index_datasets.html

2dplanes[G]
ATTRIBUTES: numeric: 11 nominal: 0 boolean: 0 all: 11
CLASS ATTRIBUTE: type: numeric number of values: 0 major class: 0.0 %
FURTHER INFO: number of instances: 40768 missing values: 0.0 % number of testsets: 0

abalone[A]
ATTRIBUTES: numeric: 8 nominal: 1 boolean: 0 all: 9
CLASS ATTRIBUTE: type: numeric number of values: 0 major class: 0.0 %
FURTHER INFO: number of instances: 4177 missing values: 0.0 % number of testsets: 0

adult[A]
ATTRIBUTES: numeric: 6 nominal: 9 boolean: 2 all: 15
CLASS ATTRIBUTE: type: boolean number of values: 2 major class: 75.9 %
FURTHER INFO: number of instances: 32561 missing values: 0.9 % number of testsets: 1

ailerons[G]
ATTRIBUTES: numeric: 41 nominal: 0 boolean: 0 all: 41
CLASS ATTRIBUTE: type: numeric number of values: 0 major class: 0.0 %
FURTHER INFO: number of instances: 13750 missing values: 0.0 % number of testsets: 0

analcatdata-aids[C]
ATTRIBUTES: numeric: 2 nominal: 3 boolean: 1 all: 5
CLASS ATTRIBUTE: type: boolean number of values: 2 major class: 50.0 %
FURTHER INFO: number of instances: 50 missing values: 0.0 % number of testsets: 0

analcatdata-asbestos[C]
ATTRIBUTES: numeric: 1 nominal: 3 boolean: 2 all: 4
CLASS ATTRIBUTE: type: boolean number of values: 2 major class: 55.4 %
FURTHER INFO: number of instances: 83 missing values: 0.0 % number of testsets: 0

analcatdata-assessment[C]
ATTRIBUTES: numeric: 0 nominal: 16 boolean: 3 all: 16

CLASS ATTRIBUTE: type: none number of values: 0 major class: 0.0 %
FURTHER INFO: number of instances: 14 missing values: 2.2 % number of testsets: 0

analcatdata-authorship[C]

ATTRIBUTES: numeric: 70 nominal: 1 boolean: 0 all: 71
CLASS ATTRIBUTE: type: nominal number of values: 4 major class: 37.7 %
FURTHER INFO: number of instances: 841 missing values: 0.0 % number of testsets: 0

analcatdata-bankruptcy[C]

ATTRIBUTES: numeric: 5 nominal: 2 boolean: 1 all: 7
CLASS ATTRIBUTE: type: boolean number of values: 2 major class: 50.0 %
FURTHER INFO: number of instances: 50 missing values: 0.0 % number of testsets: 0

analcatdata-benford[C]

ATTRIBUTES: numeric: 0 nominal: 7 boolean: 0 all: 7
CLASS ATTRIBUTE: type: nominal number of values: 9 major class: 11.1 %
FURTHER INFO: number of instances: 9 missing values: 0.0 % number of testsets: 0

analcatdata-birthday[C]

ATTRIBUTES: numeric: 1 nominal: 3 boolean: 0 all: 4
CLASS ATTRIBUTE: type: none number of values: 0 major class: 0.0 %
FURTHER INFO: number of instances: 365 missing values: 2.1 % number of testsets: 0

analcatdata-bondrate[C]

ATTRIBUTES: numeric: 4 nominal: 8 boolean: 1 all: 12
CLASS ATTRIBUTE: type: nominal number of values: 5 major class: 57.9 %
FURTHER INFO: number of instances: 57 missing values: 0.1 % number of testsets: 0

analcatdata-boxing1[C]

ATTRIBUTES: numeric: 0 nominal: 4 boolean: 2 all: 4
CLASS ATTRIBUTE: type: boolean number of values: 2 major class: 65.0 %
FURTHER INFO: number of instances: 120 missing values: 0.0 % number of testsets: 0

analcatdata-boxing2[C]

ATTRIBUTES: numeric: 0 nominal: 4 boolean: 2 all: 4
CLASS ATTRIBUTE: type: boolean number of values: 2 major class: 53.8 %
FURTHER INFO: number of instances: 132 missing values: 0.0 % number of testsets: 0

analcatdata-braziltourism[C]

ATTRIBUTES: numeric: 4 nominal: 5 boolean: 1 all: 9
CLASS ATTRIBUTE: type: nominal number of values: 7 major class: 77.2 %
FURTHER INFO: number of instances: 412 missing values: 2.6 % number of testsets: 0

analcatdata-broadway[C]

ATTRIBUTES: numeric: 3 nominal: 7 boolean: 1 all: 10
CLASS ATTRIBUTE: type: nominal number of values: 5 major class: 71.6 %
FURTHER INFO: number of instances: 95 missing values: 0.9 % number of testsets: 0

analcatdata-broadwaymult[C]

ATTRIBUTES: numeric: 3 nominal: 5 boolean: 1 all: 8
CLASS ATTRIBUTE: type: nominal number of values: 7 major class: 41.4 %
FURTHER INFO: number of instances: 285 missing values: 1.2 % number of testsets: 0

analcatdata-cancerrate[C]

ATTRIBUTES: numeric: 2 nominal: 2 boolean: 1 all: 4
CLASS ATTRIBUTE: type: nominal number of values: 5 major class: 25.0 %
FURTHER INFO: number of instances: 8 missing values: 0.0 % number of testsets: 0

analcatdata-chall101[C]

ATTRIBUTES: numeric: 1 nominal: 2 boolean: 1 all: 3

CLASS ATTRIBUTE: type: boolean number of values: 2 major class: 93.5 %

FURTHER INFO: number of instances: 138 missing values: 0.0 % number of testsets: 0

analcatdata-chall2[C]

ATTRIBUTES: numeric: 1 nominal: 4 boolean: 0 all: 5

CLASS ATTRIBUTE: type: nominal number of values: 3 major class: 47.8 %

FURTHER INFO: number of instances: 23 missing values: 0.0 % number of testsets: 0

analcatdata-challenger[C]

ATTRIBUTES: numeric: 1 nominal: 5 boolean: 1 all: 6

CLASS ATTRIBUTE: type: nominal number of values: 3 major class: 69.6 %

FURTHER INFO: number of instances: 23 missing values: 0.0 % number of testsets: 0

analcatdata-creditscore[C]

ATTRIBUTES: numeric: 3 nominal: 4 boolean: 3 all: 7

CLASS ATTRIBUTE: type: boolean number of values: 2 major class: 73.0 %

FURTHER INFO: number of instances: 100 missing values: 0.0 % number of testsets: 0

analcatdata-currency[C]

ATTRIBUTES: numeric: 1 nominal: 3 boolean: 0 all: 4

CLASS ATTRIBUTE: type: nominal number of values: 7 major class: 29.0 %

FURTHER INFO: number of instances: 31 missing values: 0.0 % number of testsets: 0

analcatdata-cyyoung8092[C]

ATTRIBUTES: numeric: 7 nominal: 4 boolean: 3 all: 11

CLASS ATTRIBUTE: type: boolean number of values: 2 major class: 75.3 %

FURTHER INFO: number of instances: 97 missing values: 0.0 % number of testsets: 0

analcatdata-cyyoung9302[C]

ATTRIBUTES: numeric: 6 nominal: 5 boolean: 3 all: 11

CLASS ATTRIBUTE: type: boolean number of values: 2 major class: 79.3 %

FURTHER INFO: number of instances: 92 missing values: 0.0 % number of testsets: 0

analcatdata-devils[C]

ATTRIBUTES: numeric: 0 nominal: 2 boolean: 0 all: 2

CLASS ATTRIBUTE: type: none number of values: 0 major class: 0.0 %

FURTHER INFO: number of instances: 82 missing values: 0.0 % number of testsets: 0

analcatdata-dmft[C]

ATTRIBUTES: numeric: 0 nominal: 5 boolean: 1 all: 5

CLASS ATTRIBUTE: type: nominal number of values: 6 major class: 19.4 %

FURTHER INFO: number of instances: 797 missing values: 0.0 % number of testsets: 0

analcatdata-donner[C]

ATTRIBUTES: numeric: 0 nominal: 4 boolean: 2 all: 4

CLASS ATTRIBUTE: type: boolean number of values: 2 major class: 50.0 %

FURTHER INFO: number of instances: 28 missing values: 0.0 % number of testsets: 0

analcatdata-draft[C]

ATTRIBUTES: numeric: 3 nominal: 3 boolean: 0 all: 6

CLASS ATTRIBUTE: type: none number of values: 0 major class: 0.0 %

FURTHER INFO: number of instances: 366 missing values: 0.1 % number of testsets: 0

analcatdata-esr[C]

ATTRIBUTES: numeric: 2 nominal: 1 boolean: 1 all: 3

CLASS ATTRIBUTE: type: boolean number of values: 2 major class: 81.3 %
FURTHER INFO: number of instances: 32 missing values: 0.0 % number of testsets: 0

analcatdata-famufsu[C]

ATTRIBUTES: numeric: 1 nominal: 3 boolean: 1 all: 4
CLASS ATTRIBUTE: type: boolean number of values: 2 major class: 50.0 %
FURTHER INFO: number of instances: 14 missing values: 0.0 % number of testsets: 0

analcatdata-fraud[C]

ATTRIBUTES: numeric: 0 nominal: 12 boolean: 11 all: 12
CLASS ATTRIBUTE: type: boolean number of values: 2 major class: 69.0 %
FURTHER INFO: number of instances: 42 missing values: 0.0 % number of testsets: 0

analcatdata-germangss[C]

ATTRIBUTES: numeric: 1 nominal: 5 boolean: 2 all: 6
CLASS ATTRIBUTE: type: ordinal number of values: 4 major class: 25.0 %
FURTHER INFO: number of instances: 400 missing values: 0.0 % number of testsets: 0

analcatdata-halloffame[C]

ATTRIBUTES: numeric: 15 nominal: 3 boolean: 0 all: 18
CLASS ATTRIBUTE: type: nominal number of values: 3 major class: 90.7 %
FURTHER INFO: number of instances: 1340 missing values: 0.1 % number of testsets: 0

analcatdata-happiness[C]

ATTRIBUTES: numeric: 1 nominal: 3 boolean: 0 all: 4
CLASS ATTRIBUTE: type: ordinal number of values: 3 major class: 33.3 %
FURTHER INFO: number of instances: 60 missing values: 0.0 % number of testsets: 0

analcatdata-homerun[C]

ATTRIBUTES: numeric: 13 nominal: 15 boolean: 8 all: 28
CLASS ATTRIBUTE: type: none number of values: 0 major class: 0.0 %
FURTHER INFO: number of instances: 163 missing values: 0.2 % number of testsets: 0

analcatdata-hurricanes[C]

ATTRIBUTES: numeric: 1 nominal: 1 boolean: 0 all: 2
CLASS ATTRIBUTE: type: nominal number of values: 8 major class: 29.8 %
FURTHER INFO: number of instances: 57 missing values: 0.0 % number of testsets: 0

analcatdata-japansolvent[C]

ATTRIBUTES: numeric: 8 nominal: 2 boolean: 1 all: 10
CLASS ATTRIBUTE: type: boolean number of values: 2 major class: 51.9 %
FURTHER INFO: number of instances: 52 missing values: 0.0 % number of testsets: 0

analcatdata-lawsuit[C]

ATTRIBUTES: numeric: 3 nominal: 2 boolean: 2 all: 5
CLASS ATTRIBUTE: type: boolean number of values: 2 major class: 92.8 %
FURTHER INFO: number of instances: 264 missing values: 0.0 % number of testsets: 0

analcatdata-mapleleafs[C]

ATTRIBUTES: numeric: 0 nominal: 2 boolean: 1 all: 2
CLASS ATTRIBUTE: type: ordinal number of values: 3 major class: 51.2 %
FURTHER INFO: number of instances: 84 missing values: 0.0 % number of testsets: 0

analcatdata-marketing[C]

ATTRIBUTES: numeric: 0 nominal: 33 boolean: 0 all: 33
CLASS ATTRIBUTE: type: none number of values: 0 major class: 0.0 %
FURTHER INFO: number of instances: 364 missing values: 0.8 % number of testsets: 0

analcatdata-reviewer[C]

ATTRIBUTES: numeric: 0 nominal: 9 boolean: 0 all: 9

CLASS ATTRIBUTE: type: none number of values: 0 major class: 0.0 %

FURTHER INFO: number of instances: 379 missing values: 41.6 % number of testsets: 0

analcatdata-votesurvey[C]

ATTRIBUTES: numeric: 3 nominal: 2 boolean: 1 all: 5

CLASS ATTRIBUTE: type: nominal number of values: 4 major class: 37.5 %

FURTHER INFO: number of instances: 48 missing values: 0.0 % number of testsets: 0

annealing[A]

ATTRIBUTES: numeric: 6 nominal: 33 boolean: 19 all: 39

CLASS ATTRIBUTE: type: nominal number of values: 6 major class: 76.2 %

FURTHER INFO: number of instances: 798 missing values: 63.3 % number of testsets: 1

arrhythmia[A]

ATTRIBUTES: numeric: 206 nominal: 2 boolean: 1 all: 280

CLASS ATTRIBUTE: type: nominal number of values: 16 major class: 54.2 %

FURTHER INFO: number of instances: 452 missing values: 0.3 % number of testsets: 0

audiology[A]

ATTRIBUTES: numeric: 0 nominal: 70 boolean: 61 all: 71

CLASS ATTRIBUTE: type: nominal number of values: 24 major class: 24.0 %

FURTHER INFO: number of instances: 200 missing values: 2.0 % number of testsets: 1

auto-mpg[A]

ATTRIBUTES: numeric: 5 nominal: 3 boolean: 0 all: 9

CLASS ATTRIBUTE: type: numeric number of values: 0 major class: 0.0 %

FURTHER INFO: number of instances: 398 missing values: 0.2 % number of testsets: 0

auto-mpg-original[A]

ATTRIBUTES: numeric: 5 nominal: 3 boolean: 0 all: 9

CLASS ATTRIBUTE: type: numeric number of values: 0 major class: 0.0 %

FURTHER INFO: number of instances: 406 missing values: 0.4 % number of testsets: 0

automobile[A]

ATTRIBUTES: numeric: 15 nominal: 11 boolean: 4 all: 26

CLASS ATTRIBUTE: type: numeric number of values: 0 major class: 0.0 %

FURTHER INFO: number of instances: 205 missing values: 1.1 % number of testsets: 0

backache[C]

ATTRIBUTES: numeric: 6 nominal: 27 boolean: 23 all: 33

CLASS ATTRIBUTE: type: boolean number of values: 2 major class: 86.1 %

FURTHER INFO: number of instances: 180 missing values: 0.0 % number of testsets: 0

badges[A]

ATTRIBUTES: numeric: 0 nominal: 1 boolean: 1 all: 2

CLASS ATTRIBUTE: type: boolean number of values: 2 major class: 71.4 %

FURTHER INFO: number of instances: 294 missing values: 0.0 % number of testsets: 0

balance[A]

ATTRIBUTES: numeric: 0 nominal: 5 boolean: 0 all: 5

CLASS ATTRIBUTE: type: nominal number of values: 3 major class: 46.1 %

FURTHER INFO: number of instances: 625 missing values: 0.0 % number of testsets: 0

ballon01[A]

ATTRIBUTES: numeric: 0 nominal: 5 boolean: 5 all: 5

CLASS ATTRIBUTE: type: boolean number of values: 2 major class: 60.0 %
FURTHER INFO: number of instances: 20 missing values: 0.0 % number of testsets: 0

ballon02[A]

ATTRIBUTES: numeric: 0 nominal: 5 boolean: 5 all: 5
CLASS ATTRIBUTE: type: boolean number of values: 2 major class: 60.0 %
FURTHER INFO: number of instances: 20 missing values: 0.0 % number of testsets: 0

ballon03[A]

ATTRIBUTES: numeric: 0 nominal: 5 boolean: 5 all: 5
CLASS ATTRIBUTE: type: boolean number of values: 2 major class: 56.3 %
FURTHER INFO: number of instances: 16 missing values: 0.0 % number of testsets: 0

ballon04[A]

ATTRIBUTES: numeric: 0 nominal: 5 boolean: 5 all: 5
CLASS ATTRIBUTE: type: boolean number of values: 2 major class: 60.0 %
FURTHER INFO: number of instances: 20 missing values: 0.0 % number of testsets: 0

bank32nh[G]

ATTRIBUTES: numeric: 33 nominal: 0 boolean: 0 all: 33
CLASS ATTRIBUTE: type: numeric number of values: 0 major class: 0.0 %
FURTHER INFO: number of instances: 8192 missing values: 0.0 % number of testsets: 0

bank8fm[G]

ATTRIBUTES: numeric: 9 nominal: 0 boolean: 0 all: 9
CLASS ATTRIBUTE: type: numeric number of values: 0 major class: 0.0 %
FURTHER INFO: number of instances: 8192 missing values: 0.0 % number of testsets: 0

biomed[C]

ATTRIBUTES: numeric: 7 nominal: 2 boolean: 1 all: 9
CLASS ATTRIBUTE: type: boolean number of values: 2 major class: 64.1 %
FURTHER INFO: number of instances: 209 missing values: 0.8 % number of testsets: 0

boston housing[A]

ATTRIBUTES: numeric: 14 nominal: 0 boolean: 0 all: 14
CLASS ATTRIBUTE: type: numeric number of values: 0 major class: 0.0 %
FURTHER INFO: number of instances: 506 missing values: 0.0 % number of testsets: 0

cal_housing[G]

ATTRIBUTES: numeric: 9 nominal: 0 boolean: 0 all: 9
CLASS ATTRIBUTE: type: numeric number of values: 0 major class: 0.0 %
FURTHER INFO: number of instances: 20640 missing values: 0.0 % number of testsets: 0

calit2-01[A]

ATTRIBUTES: numeric: 1 nominal: 1 boolean: 1 all: 4
CLASS ATTRIBUTE: type: none number of values: 0 major class: 0.0 %
FURTHER INFO: number of instances: 10080 missing values: 0.0 % number of testsets: 0

calit2-02[A]

ATTRIBUTES: numeric: 0 nominal: 0 boolean: 0 all: 4
CLASS ATTRIBUTE: type: none number of values: 0 major class: 0.0 %
FURTHER INFO: number of instances: 30 missing values: 0.0 % number of testsets: 0

car evaluation[A]

ATTRIBUTES: numeric: 0 nominal: 7 boolean: 0 all: 7
CLASS ATTRIBUTE: type: nominal number of values: 4 major class: 70.0 %
FURTHER INFO: number of instances: 1728 missing values: 0.0 % number of testsets: 0

cars[C]

ATTRIBUTES: numeric: 6 nominal: 2 boolean: 0 all: 8

CLASS ATTRIBUTE: type: nominal number of values: 3 major class: 62.6 %

FURTHER INFO: number of instances: 406 missing values: 0.4 % number of testsets: 0

cars-with-names[C]

ATTRIBUTES: numeric: 6 nominal: 3 boolean: 0 all: 9

CLASS ATTRIBUTE: type: nominal number of values: 3 major class: 62.6 %

FURTHER INFO: number of instances: 406 missing values: 0.4 % number of testsets: 0

challenger o-ring erosion[A]

ATTRIBUTES: numeric: 5 nominal: 0 boolean: 0 all: 5

CLASS ATTRIBUTE: type: numeric number of values: 0 major class: 0.0 %

FURTHER INFO: number of instances: 23 missing values: 0.0 % number of testsets: 0

challenger o-ring erosion blowby[A]

ATTRIBUTES: numeric: 5 nominal: 0 boolean: 0 all: 5

CLASS ATTRIBUTE: type: numeric number of values: 0 major class: 0.0 %

FURTHER INFO: number of instances: 23 missing values: 0.0 % number of testsets: 0

chess[B]

ATTRIBUTES: numeric: 37 nominal: 0 boolean: 0 all: 37

CLASS ATTRIBUTE: type: none number of values: 0 major class: 0.0 %

FURTHER INFO: number of instances: 3196 missing values: 0.0 % number of testsets: 0

chess-kr-vs-k[A]

ATTRIBUTES: numeric: 0 nominal: 7 boolean: 0 all: 7

CLASS ATTRIBUTE: type: ordinal number of values: 18 major class: 16.2 %

FURTHER INFO: number of instances: 28056 missing values: 0.0 % number of testsets: 0

chess-kr-vs-kp[A]

ATTRIBUTES: numeric: 0 nominal: 37 boolean: 35 all: 37

CLASS ATTRIBUTE: type: boolean number of values: 2 major class: 52.2 %

FURTHER INFO: number of instances: 3196 missing values: 0.0 % number of testsets: 0

cjs[C]

ATTRIBUTES: numeric: 32 nominal: 3 boolean: 0 all: 35

CLASS ATTRIBUTE: type: nominal number of values: 6 major class: 24.3 %

FURTHER INFO: number of instances: 2796 missing values: 69.6 % number of testsets: 0

cloud[C]

ATTRIBUTES: numeric: 6 nominal: 2 boolean: 1 all: 8

CLASS ATTRIBUTE: type: nominal number of values: 4 major class: 29.6 %

FURTHER INFO: number of instances: 108 missing values: 0.0 % number of testsets: 0

clouds 1[A]

ATTRIBUTES: numeric: 10 nominal: 0 boolean: 0 all: 10

CLASS ATTRIBUTE: type: none number of values: 0 major class: 0.0 %

FURTHER INFO: number of instances: 1024 missing values: 0.0 % number of testsets: 0

clouds 2[A]

ATTRIBUTES: numeric: 10 nominal: 0 boolean: 0 all: 10

CLASS ATTRIBUTE: type: none number of values: 0 major class: 0.0 %

FURTHER INFO: number of instances: 1024 missing values: 0.0 % number of testsets: 0

collins[C]

ATTRIBUTES: numeric: 20 nominal: 4 boolean: 0 all: 24

CLASS ATTRIBUTE: type: nominal number of values: 15 major class: 16.0 %
FURTHER INFO: number of instances: 500 missing values: 0.0 % number of testsets: 0

confidence[C]

ATTRIBUTES: numeric: 3 nominal: 1 boolean: 0 all: 4
CLASS ATTRIBUTE: type: nominal number of values: 6 major class: 16.7 %
FURTHER INFO: number of instances: 72 missing values: 0.0 % number of testsets: 0

congressional voting[A]

ATTRIBUTES: numeric: 0 nominal: 17 boolean: 17 all: 17
CLASS ATTRIBUTE: type: boolean number of values: 2 major class: 61.4 %
FURTHER INFO: number of instances: 435 missing values: 5.3 % number of testsets: 0

connect[B]

ATTRIBUTES: numeric: 43 nominal: 0 boolean: 0 all: 43
CLASS ATTRIBUTE: type: none number of values: 0 major class: 0.0 %
FURTHER INFO: number of instances: 67557 missing values: 0.0 % number of testsets: 0

connect-4 opening[A]

ATTRIBUTES: numeric: 0 nominal: 43 boolean: 0 all: 43
CLASS ATTRIBUTE: type: nominal number of values: 3 major class: 65.8 %
FURTHER INFO: number of instances: 67557 missing values: 0.0 % number of testsets: 0

contact lenses[A]

ATTRIBUTES: numeric: 0 nominal: 5 boolean: 3 all: 5
CLASS ATTRIBUTE: type: nominal number of values: 3 major class: 62.5 %
FURTHER INFO: number of instances: 24 missing values: 0.0 % number of testsets: 0

contraceptive method choice[A]

ATTRIBUTES: numeric: 2 nominal: 8 boolean: 3 all: 10
CLASS ATTRIBUTE: type: nominal number of values: 3 major class: 42.7 %
FURTHER INFO: number of instances: 1473 missing values: 0.0 % number of testsets: 0

covertype[A]

ATTRIBUTES: numeric: 10 nominal: 45 boolean: 44 all: 55
CLASS ATTRIBUTE: type: nominal number of values: 7 major class: 48.8 %
FURTHER INFO: number of instances: 581012 missing values: 0.0 % number of testsets: 0

cpu performance[A]

ATTRIBUTES: numeric: 8 nominal: 1 boolean: 0 all: 10
CLASS ATTRIBUTE: type: numeric number of values: 0 major class: 0.0 %
FURTHER INFO: number of instances: 209 missing values: 0.0 % number of testsets: 0

cpu_act[G]

ATTRIBUTES: numeric: 22 nominal: 0 boolean: 0 all: 22
CLASS ATTRIBUTE: type: none number of values: 0 major class: 0.0 %
FURTHER INFO: number of instances: 8192 missing values: 0.0 % number of testsets: 0

cpu_small[G]

ATTRIBUTES: numeric: 13 nominal: 0 boolean: 0 all: 13
CLASS ATTRIBUTE: type: none number of values: 0 major class: 0.0 %
FURTHER INFO: number of instances: 8192 missing values: 0.0 % number of testsets: 0

credit approval[A]

ATTRIBUTES: numeric: 6 nominal: 10 boolean: 5 all: 16
CLASS ATTRIBUTE: type: boolean number of values: 2 major class: 55.5 %
FURTHER INFO: number of instances: 690 missing values: 0.6 % number of testsets: 0

cylinder bands[A]

ATTRIBUTES: numeric: 21 nominal: 17 boolean: 6 all: 40

CLASS ATTRIBUTE: type: boolean number of values: 2 major class: 57.8 %

FURTHER INFO: number of instances: 540 missing values: 4.6 % number of testsets: 0

delta_ailerons[G]

ATTRIBUTES: numeric: 6 nominal: 0 boolean: 0 all: 6

CLASS ATTRIBUTE: type: none number of values: 0 major class: 0.0 %

FURTHER INFO: number of instances: 7129 missing values: 0.0 % number of testsets: 0

delta_elevators[G]

ATTRIBUTES: numeric: 7 nominal: 0 boolean: 0 all: 7

CLASS ATTRIBUTE: type: none number of values: 0 major class: 0.0 %

FURTHER INFO: number of instances: 9517 missing values: 0.0 % number of testsets: 0

dermatology[A]

ATTRIBUTES: numeric: 1 nominal: 34 boolean: 1 all: 35

CLASS ATTRIBUTE: type: nominal number of values: 6 major class: 30.6 %

FURTHER INFO: number of instances: 366 missing values: 0.1 % number of testsets: 0

diabetes_numeric[A]

ATTRIBUTES: numeric: 3 nominal: 0 boolean: 0 all: 3

CLASS ATTRIBUTE: type: none number of values: 0 major class: 0.0 %

FURTHER INFO: number of instances: 43 missing values: 0.0 % number of testsets: 0

dodgers loop sensor 1[A]

ATTRIBUTES: numeric: 1 nominal: 0 boolean: 0 all: 3

CLASS ATTRIBUTE: type: none number of values: 0 major class: 0.0 %

FURTHER INFO: number of instances: 50400 missing values: 0.0 % number of testsets: 0

dodgers loop sensor 2[A]

ATTRIBUTES: numeric: 1 nominal: 0 boolean: 0 all: 6

CLASS ATTRIBUTE: type: none number of values: 0 major class: 0.0 %

FURTHER INFO: number of instances: 81 missing values: 0.0 % number of testsets: 0

echocardiogram[A]

ATTRIBUTES: numeric: 9 nominal: 3 boolean: 3 all: 13

CLASS ATTRIBUTE: type: boolean number of values: 2 major class: 37.9 %

FURTHER INFO: number of instances: 132 missing values: 7.7 % number of testsets: 0

elevators[G]

ATTRIBUTES: numeric: 19 nominal: 0 boolean: 0 all: 19

CLASS ATTRIBUTE: type: numeric number of values: 0 major class: 0.0 %

FURTHER INFO: number of instances: 16599 missing values: 0.0 % number of testsets: 0

era[E]

ATTRIBUTES: numeric: 5 nominal: 0 boolean: 0 all: 5

CLASS ATTRIBUTE: type: numeric number of values: 0 major class: 0.0 %

FURTHER INFO: number of instances: 1000 missing values: 0.0 % number of testsets: 0

esl[E]

ATTRIBUTES: numeric: 5 nominal: 0 boolean: 0 all: 5

CLASS ATTRIBUTE: type: numeric number of values: 0 major class: 0.0 %

FURTHER INFO: number of instances: 488 missing values: 0.0 % number of testsets: 0

eucalyptus[D]

ATTRIBUTES: numeric: 14 nominal: 6 boolean: 0 all: 20

CLASS ATTRIBUTE: type: ordinal number of values: 5 major class: 29.1 %
FURTHER INFO: number of instances: 736 missing values: 3.0 % number of testsets: 0

fbis[F]

ATTRIBUTES: numeric: 2000 nominal: 1 boolean: 0 all: 2001
CLASS ATTRIBUTE: type: nominal number of values: 17 major class: 20.5 %
FURTHER INFO: number of instances: 2463 missing values: 0.0 % number of testsets: 0

fl2000[C]

ATTRIBUTES: numeric: 14 nominal: 3 boolean: 1 all: 17
CLASS ATTRIBUTE: type: nominal number of values: 5 major class: 61.2 %
FURTHER INFO: number of instances: 67 missing values: 0.0 % number of testsets: 0

flags[A]

ATTRIBUTES: numeric: 10 nominal: 16 boolean: 12 all: 30
CLASS ATTRIBUTE: type: none number of values: 0 major class: 0.0 %
FURTHER INFO: number of instances: 194 missing values: 0.0 % number of testsets: 0

fried[G]

ATTRIBUTES: numeric: 11 nominal: 0 boolean: 0 all: 11
CLASS ATTRIBUTE: type: numeric number of values: 0 major class: 0.0 %
FURTHER INFO: number of instances: 40768 missing values: 0.0 % number of testsets: 0

glass identification[A]

ATTRIBUTES: numeric: 10 nominal: 1 boolean: 0 all: 11
CLASS ATTRIBUTE: type: nominal number of values: 7 major class: 35.5 %
FURTHER INFO: number of instances: 214 missing values: 0.0 % number of testsets: 0

grub-damage[D]

ATTRIBUTES: numeric: 2 nominal: 7 boolean: 0 all: 9
CLASS ATTRIBUTE: type: ordinal number of values: 4 major class: 31.6 %
FURTHER INFO: number of instances: 155 missing values: 0.0 % number of testsets: 0

habermans survival[A]

ATTRIBUTES: numeric: 3 nominal: 1 boolean: 1 all: 4
CLASS ATTRIBUTE: type: boolean number of values: 2 major class: 73.5 %
FURTHER INFO: number of instances: 306 missing values: 0.0 % number of testsets: 0

hayes-roth[A]

ATTRIBUTES: numeric: 1 nominal: 5 boolean: 0 all: 6
CLASS ATTRIBUTE: type: nominal number of values: 3 major class: 38.6 %
FURTHER INFO: number of instances: 132 missing values: 0.0 % number of testsets: 0

heart disease cleveland[A]

ATTRIBUTES: numeric: 13 nominal: 1 boolean: 0 all: 14
CLASS ATTRIBUTE: type: nominal number of values: 5 major class: 54.1 %
FURTHER INFO: number of instances: 303 missing values: 0.1 % number of testsets: 0

heart disease hungarian[A]

ATTRIBUTES: numeric: 13 nominal: 1 boolean: 0 all: 14
CLASS ATTRIBUTE: type: nominal number of values: 5 major class: 54.1 %
FURTHER INFO: number of instances: 303 missing values: 0.1 % number of testsets: 0

heart disease long beach[A]

ATTRIBUTES: numeric: 13 nominal: 1 boolean: 0 all: 14
CLASS ATTRIBUTE: type: nominal number of values: 5 major class: 28.0 %
FURTHER INFO: number of instances: 200 missing values: 24.9 % number of testsets: 0

heart disease switzerland[A]

ATTRIBUTES: numeric: 13 nominal: 1 boolean: 0 all: 14

CLASS ATTRIBUTE: type: nominal number of values: 5 major class: 39.0 %

FURTHER INFO: number of instances: 123 missing values: 15.9 % number of testsets: 0

hepatitis[A]

ATTRIBUTES: numeric: 6 nominal: 14 boolean: 14 all: 20

CLASS ATTRIBUTE: type: boolean number of values: 2 major class: 79.4 %

FURTHER INFO: number of instances: 155 missing values: 5.4 % number of testsets: 0

horse colic[A]

ATTRIBUTES: numeric: 11 nominal: 17 boolean: 4 all: 28

CLASS ATTRIBUTE: type: boolean number of values: 2 major class: 60.0 %

FURTHER INFO: number of instances: 300 missing values: 19.1 % number of testsets: 1

house_16h[G]

ATTRIBUTES: numeric: 17 nominal: 0 boolean: 0 all: 17

CLASS ATTRIBUTE: type: numeric number of values: 0 major class: 0.0 %

FURTHER INFO: number of instances: 22784 missing values: 0.0 % number of testsets: 0

house_8l[G]

ATTRIBUTES: numeric: 9 nominal: 0 boolean: 0 all: 9

CLASS ATTRIBUTE: type: numeric number of values: 0 major class: 0.0 %

FURTHER INFO: number of instances: 22784 missing values: 0.0 % number of testsets: 0

hypothyroid[A]

ATTRIBUTES: numeric: 7 nominal: 19 boolean: 19 all: 26

CLASS ATTRIBUTE: type: boolean number of values: 2 major class: 95.2 %

FURTHER INFO: number of instances: 3163 missing values: 6.5 % number of testsets: 0

icu flowsheet[A]

ATTRIBUTES: numeric: 1 nominal: 0 boolean: 0 all: 4

CLASS ATTRIBUTE: type: none number of values: 0 major class: 0.0 %

FURTHER INFO: number of instances: 39 missing values: 0.0 % number of testsets: 0

icu lab[A]

ATTRIBUTES: numeric: 1 nominal: 0 boolean: 0 all: 4

CLASS ATTRIBUTE: type: none number of values: 0 major class: 0.0 %

FURTHER INFO: number of instances: 81 missing values: 0.0 % number of testsets: 0

icu monitor[A]

ATTRIBUTES: numeric: 2 nominal: 0 boolean: 0 all: 3

CLASS ATTRIBUTE: type: none number of values: 0 major class: 0.0 %

FURTHER INFO: number of instances: 7931 missing values: 0.0 % number of testsets: 0

image segmentation[A]

ATTRIBUTES: numeric: 18 nominal: 2 boolean: 0 all: 20

CLASS ATTRIBUTE: type: nominal number of values: 7 major class: 14.3 %

FURTHER INFO: number of instances: 210 missing values: 0.0 % number of testsets: 1

internet advertisement[A]

ATTRIBUTES: numeric: 3 nominal: 1556 boolean: 1556 all: 1559

CLASS ATTRIBUTE: type: boolean number of values: 2 major class: 86.0 %

FURTHER INFO: number of instances: 3279 missing values: 0.1 % number of testsets: 0

ionoshere[A]

ATTRIBUTES: numeric: 34 nominal: 1 boolean: 1 all: 35

CLASS ATTRIBUTE: type: boolean number of values: 2 major class: 64.1 %
FURTHER INFO: number of instances: 351 missing values: 0.0 % number of testsets: 0

iris[A]

ATTRIBUTES: numeric: 4 nominal: 1 boolean: 0 all: 5
CLASS ATTRIBUTE: type: nominal number of values: 3 major class: 33.3 %
FURTHER INFO: number of instances: 150 missing values: 0.0 % number of testsets: 0

irish[C]

ATTRIBUTES: numeric: 2 nominal: 4 boolean: 2 all: 6
CLASS ATTRIBUTE: type: boolean number of values: 2 major class: 55.6 %
FURTHER INFO: number of instances: 500 missing values: 1.1 % number of testsets: 0

isolet spoken letters[C]

ATTRIBUTES: numeric: 617 nominal: 1 boolean: 0 all: 618
CLASS ATTRIBUTE: type: nominal number of values: 26 major class: 3.8 %
FURTHER INFO: number of instances: 6238 missing values: 0.0 % number of testsets: 1

kin8nm[G]

ATTRIBUTES: numeric: 9 nominal: 0 boolean: 0 all: 9
CLASS ATTRIBUTE: type: numeric number of values: 0 major class: 0.0 %
FURTHER INFO: number of instances: 8192 missing values: 0.0 % number of testsets: 0

labor[A]

ATTRIBUTES: numeric: 8 nominal: 9 boolean: 4 all: 17
CLASS ATTRIBUTE: type: boolean number of values: 2 major class: 64.9 %
FURTHER INFO: number of instances: 57 missing values: 33.6 % number of testsets: 0

letter image recognition[A]

ATTRIBUTES: numeric: 16 nominal: 1 boolean: 0 all: 17
CLASS ATTRIBUTE: type: nominal number of values: 26 major class: 4.1 %
FURTHER INFO: number of instances: 20000 missing values: 0.0 % number of testsets: 0

lev[E]

ATTRIBUTES: numeric: 5 nominal: 0 boolean: 0 all: 5
CLASS ATTRIBUTE: type: numeric number of values: 0 major class: 0.0 %
FURTHER INFO: number of instances: 1000 missing values: 0.0 % number of testsets: 0

liver disorders[A]

ATTRIBUTES: numeric: 6 nominal: 1 boolean: 1 all: 7
CLASS ATTRIBUTE: type: boolean number of values: 2 major class: 58.0 %
FURTHER INFO: number of instances: 345 missing values: 0.0 % number of testsets: 0

low resolution spectrometer[A]

ATTRIBUTES: numeric: 102 nominal: 0 boolean: 0 all: 103
CLASS ATTRIBUTE: type: numeric number of values: 0 major class: 0.0 %
FURTHER INFO: number of instances: 531 missing values: 0.0 % number of testsets: 0

lung cancer[A]

ATTRIBUTES: numeric: 0 nominal: 57 boolean: 0 all: 57
CLASS ATTRIBUTE: type: nominal number of values: 3 major class: 40.6 %
FURTHER INFO: number of instances: 32 missing values: 0.3 % number of testsets: 0

lupus[C]

ATTRIBUTES: numeric: 3 nominal: 1 boolean: 1 all: 4
CLASS ATTRIBUTE: type: boolean number of values: 2 major class: 59.8 %
FURTHER INFO: number of instances: 87 missing values: 0.0 % number of testsets: 0

lymphography[A]

ATTRIBUTES: numeric: 3 nominal: 16 boolean: 9 all: 19

CLASS ATTRIBUTE: type: nominal number of values: 4 major class: 54.7 %

FURTHER INFO: number of instances: 148 missing values: 0.0 % number of testsets: 0

machine_cpu[G]

ATTRIBUTES: numeric: 7 nominal: 0 boolean: 0 all: 7

CLASS ATTRIBUTE: type: numeric number of values: 0 major class: 0.0 %

FURTHER INFO: number of instances: 209 missing values: 0.0 % number of testsets: 0

magic gamma telescope[A]

ATTRIBUTES: numeric: 10 nominal: 1 boolean: 1 all: 11

CLASS ATTRIBUTE: type: boolean number of values: 2 major class: 64.8 %

FURTHER INFO: number of instances: 19020 missing values: 0.0 % number of testsets: 0

meta-data[A]

ATTRIBUTES: numeric: 19 nominal: 1 boolean: 1 all: 22

CLASS ATTRIBUTE: type: numeric number of values: 0 major class: 0.0 %

FURTHER INFO: number of instances: 528 missing values: 4.3 % number of testsets: 0

molecular biology promoters[A]

ATTRIBUTES: numeric: 0 nominal: 58 boolean: 1 all: 59

CLASS ATTRIBUTE: type: boolean number of values: 2 major class: 50.0 %

FURTHER INFO: number of instances: 106 missing values: 0.0 % number of testsets: 0

molecular biology splice[A]

ATTRIBUTES: numeric: 0 nominal: 61 boolean: 0 all: 62

CLASS ATTRIBUTE: type: nominal number of values: 3 major class: 51.9 %

FURTHER INFO: number of instances: 3190 missing values: 0.0 % number of testsets: 0

monks problems 1[A]

ATTRIBUTES: numeric: 0 nominal: 7 boolean: 3 all: 8

CLASS ATTRIBUTE: type: boolean number of values: 2 major class: 50.0 %

FURTHER INFO: number of instances: 124 missing values: 0.0 % number of testsets: 1

monks problems 2[A]

ATTRIBUTES: numeric: 0 nominal: 7 boolean: 3 all: 8

CLASS ATTRIBUTE: type: boolean number of values: 2 major class: 62.1 %

FURTHER INFO: number of instances: 169 missing values: 0.0 % number of testsets: 1

monks problems 3[A]

ATTRIBUTES: numeric: 0 nominal: 7 boolean: 3 all: 8

CLASS ATTRIBUTE: type: boolean number of values: 2 major class: 50.8 %

FURTHER INFO: number of instances: 122 missing values: 0.0 % number of testsets: 1

multi feature digit fac[A]

ATTRIBUTES: numeric: 216 nominal: 1 boolean: 0 all: 217

CLASS ATTRIBUTE: type: nominal number of values: 10 major class: 10.1 %

FURTHER INFO: number of instances: 2000 missing values: 0.0 % number of testsets: 0

multi feature digit fou[A]

ATTRIBUTES: numeric: 76 nominal: 1 boolean: 0 all: 77

CLASS ATTRIBUTE: type: nominal number of values: 10 major class: 10.1 %

FURTHER INFO: number of instances: 2000 missing values: 0.0 % number of testsets: 0

multi feature digit kar[A]

ATTRIBUTES: numeric: 64 nominal: 1 boolean: 0 all: 65

CLASS ATTRIBUTE: type: nominal number of values: 10 major class: 10.1 %
FURTHER INFO: number of instances: 2000 missing values: 0.0 % number of testsets: 0

multi feature digit mor[A]

ATTRIBUTES: numeric: 6 nominal: 1 boolean: 0 all: 7
CLASS ATTRIBUTE: type: nominal number of values: 10 major class: 10.1 %
FURTHER INFO: number of instances: 2000 missing values: 0.0 % number of testsets: 0

multi feature digit pix[A]

ATTRIBUTES: numeric: 240 nominal: 1 boolean: 0 all: 241
CLASS ATTRIBUTE: type: nominal number of values: 10 major class: 10.1 %
FURTHER INFO: number of instances: 2000 missing values: 0.0 % number of testsets: 0

multi feature digit zer[A]

ATTRIBUTES: numeric: 47 nominal: 1 boolean: 0 all: 48
CLASS ATTRIBUTE: type: nominal number of values: 10 major class: 10.1 %
FURTHER INFO: number of instances: 2000 missing values: 0.0 % number of testsets: 0

mushroom[B]

ATTRIBUTES: numeric: 23 nominal: 0 boolean: 0 all: 23
CLASS ATTRIBUTE: type: none number of values: 0 major class: 0.0 %
FURTHER INFO: number of instances: 8124 missing values: 0.0 % number of testsets: 0

mushrooms[A]

ATTRIBUTES: numeric: 0 nominal: 23 boolean: 5 all: 23
CLASS ATTRIBUTE: type: boolean number of values: 2 major class: 51.8 %
FURTHER INFO: number of instances: 8124 missing values: 1.3 % number of testsets: 0

musk01[A]

ATTRIBUTES: numeric: 166 nominal: 3 boolean: 1 all: 169
CLASS ATTRIBUTE: type: boolean number of values: 2 major class: 56.5 %
FURTHER INFO: number of instances: 476 missing values: 0.0 % number of testsets: 0

musk02[A]

ATTRIBUTES: NUMERIC: 166 nominal: 3 boolean: 1 all: 169
CLASS ATTRIBUTE: type: boolean number of values: 2 major class: 84.6 %
FURTHER INFO: number of instances: 6598 missing values: 0.0 % number of testsets: 0

mv[G]

ATTRIBUTES: numeric: 8 nominal: 3 boolean: 2 all: 11
CLASS ATTRIBUTE: type: numeric number of values: 0 major class: 0.0 %
FURTHER INFO: number of instances: 40768 missing values: 0.0 % number of testsets: 0

nursery[A]

ATTRIBUTES: numeric: 0 nominal: 9 boolean: 1 all: 9
CLASS ATTRIBUTE: type: nominal number of values: 5 major class: 33.3 %
FURTHER INFO: number of instances: 12960 missing values: 0.0 % number of testsets: 0

oh0[F]

ATTRIBUTES: numeric: 3182 nominal: 1 boolean: 0 all: 3183
CLASS ATTRIBUTE: type: nominal number of values: 10 major class: 19.3 %
FURTHER INFO: number of instances: 1003 missing values: 0.0 % number of testsets: 0

oh10[F]

ATTRIBUTES: numeric: 3238 nominal: 1 boolean: 0 all: 3239
CLASS ATTRIBUTE: type: nominal number of values: 10 major class: 15.7 %
FURTHER INFO: number of instances: 1050 missing values: 0.0 % number of testsets: 0

oh15[F]

ATTRIBUTES: numeric: 3100 nominal: 1 boolean: 0 all: 3101

CLASS ATTRIBUTE: type: nominal number of values: 10 major class: 17.2 %

FURTHER INFO: number of instances: 913 missing values: 0.0 % number of testsets: 0

oh5[F]

ATTRIBUTES: numeric: 3012 nominal: 1 boolean: 0 all: 3013

CLASS ATTRIBUTE: type: nominal number of values: 10 major class: 16.2 %

FURTHER INFO: number of instances: 918 missing values: 0.0 % number of testsets: 0

page blocks[A]

ATTRIBUTES: numeric: 10 nominal: 1 boolean: 0 all: 11

CLASS ATTRIBUTE: type: nominal number of values: 5 major class: 89.8 %

FURTHER INFO: number of instances: 5473 missing values: 0.0 % number of testsets: 0

pasture-production[D]

ATTRIBUTES: numeric: 21 nominal: 2 boolean: 0 all: 23

CLASS ATTRIBUTE: type: ordinal number of values: 3 major class: 33.3 %

FURTHER INFO: number of instances: 36 missing values: 0.0 % number of testsets: 0

pima indians diabetes[A]

ATTRIBUTES: numeric: 8 nominal: 1 boolean: 1 all: 9

CLASS ATTRIBUTE: type: boolean number of values: 2 major class: 65.1 %

FURTHER INFO: number of instances: 768 missing values: 0.0 % number of testsets: 0

pittsburgh bridges 01[A]

ATTRIBUTES: numeric: 3 nominal: 9 boolean: 2 all: 13

CLASS ATTRIBUTE: type: none number of values: 0 major class: 0.0 %

FURTHER INFO: number of instances: 108 missing values: 5.5 % number of testsets: 0

pittsburgh bridges 02[A]

ATTRIBUTES: numeric: 1 nominal: 11 boolean: 2 all: 13

CLASS ATTRIBUTE: type: none number of values: 0 major class: 0.0 %

FURTHER INFO: number of instances: 108 missing values: 5.5 % number of testsets: 0

poker[A]

ATTRIBUTES: numeric: 0 nominal: 11 boolean: 0 all: 11

CLASS ATTRIBUTE: type: nominal number of values: 10 major class: 50.0 %

FURTHER INFO: number of instances: 25010 missing values: 0.0 % number of testsets: 1

pol[G]

ATTRIBUTES: numeric: 49 nominal: 0 boolean: 0 all: 49

CLASS ATTRIBUTE: type: none number of values: 0 major class: 0.0 %

FURTHER INFO: number of instances: 15000 missing values: 0.0 % number of testsets: 0

postoperative patient[A]

ATTRIBUTES: numeric: 1 nominal: 8 boolean: 0 all: 9

CLASS ATTRIBUTE: type: nominal number of values: 3 major class: 71.1 %

FURTHER INFO: number of instances: 90 missing values: 0.4 % number of testsets: 0

prnn-crab[C]

ATTRIBUTES: numeric: 6 nominal: 2 boolean: 2 all: 8

CLASS ATTRIBUTE: type: boolean number of values: 2 major class: 50.0 %

FURTHER INFO: number of instances: 200 missing values: 0.0 % number of testsets: 0

prnn-cushings[C]

ATTRIBUTES: numeric: 2 nominal: 2 boolean: 0 all: 4

CLASS ATTRIBUTE: type: nominal number of values: 4 major class: 44.4 %
FURTHER INFO: number of instances: 27 missing values: 0.0 % number of testsets: 0

prnn-fglass[C]

ATTRIBUTES: numeric: 9 nominal: 1 boolean: 0 all: 10
CLASS ATTRIBUTE: type: nominal number of values: 6 major class: 35.5 %
FURTHER INFO: number of instances: 214 missing values: 0.0 % number of testsets: 0

prnn-synth[C]

ATTRIBUTES: numeric: 2 nominal: 1 boolean: 1 all: 3
CLASS ATTRIBUTE: type: boolean number of values: 2 major class: 50.0 %
FURTHER INFO: number of instances: 250 missing values: 0.0 % number of testsets: 0

prnn-virus3[C]

ATTRIBUTES: numeric: 4 nominal: 14 boolean: 0 all: 18
CLASS ATTRIBUTE: type: none number of values: 0 major class: 0.0 %
FURTHER INFO: number of instances: 38 missing values: 0.0 % number of testsets: 0

prnn-viruses[C]

ATTRIBUTES: numeric: 10 nominal: 8 boolean: 0 all: 18
CLASS ATTRIBUTE: type: none number of values: 0 major class: 0.0 %
FURTHER INFO: number of instances: 61 missing values: 0.0 % number of testsets: 0

profb[C]

ATTRIBUTES: numeric: 5 nominal: 5 boolean: 1 all: 10
CLASS ATTRIBUTE: type: boolean number of values: 2 major class: 66.7 %
FURTHER INFO: number of instances: 672 missing values: 17.9 % number of testsets: 0

protein localization sites[A]

ATTRIBUTES: numeric: 7 nominal: 1 boolean: 0 all: 9
CLASS ATTRIBUTE: type: nominal number of values: 8 major class: 42.6 %
FURTHER INFO: number of instances: 336 missing values: 0.0 % number of testsets: 0

puma32h[G]

ATTRIBUTES: numeric: 33 nominal: 0 boolean: 0 all: 33
CLASS ATTRIBUTE: type: numeric number of values: 0 major class: 0.0 %
FURTHER INFO: number of instances: 8192 missing values: 0.0 % number of testsets: 0

puma8nh[G]

ATTRIBUTES: numeric: 9 nominal: 0 boolean: 0 all: 9
CLASS ATTRIBUTE: type: numeric number of values: 0 major class: 0.0 %
FURTHER INFO: number of instances: 8192 missing values: 0.0 % number of testsets: 0

pumsb[B]

ATTRIBUTES: numeric: 74 nominal: 0 boolean: 0 all: 74
CLASS ATTRIBUTE: type: none number of values: 0 major class: 0.0 %
FURTHER INFO: number of instances: 49046 missing values: 0.0 % number of testsets: 0

pyrimidines01 boolean[A]

ATTRIBUTES: numeric: 54 nominal: 1 boolean: 1 all: 55
CLASS ATTRIBUTE: type: boolean number of values: 2 major class: 50.0 %
FURTHER INFO: number of instances: 1762 missing values: 0.0 % number of testsets: 1

pyrimidines01 real[A]

ATTRIBUTES: numeric: 28 nominal: 0 boolean: 0 all: 28
CLASS ATTRIBUTE: type: numeric number of values: 0 major class: 0.0 %
FURTHER INFO: number of instances: 44 missing values: 0.0 % number of testsets: 1

pyrimidines02 boolean[A]

ATTRIBUTES: numeric: 54 nominal: 1 boolean: 1 all: 55

CLASS ATTRIBUTE: type: boolean number of values: 2 major class: 50.0 %

FURTHER INFO: number of instances: 1778 missing values: 0.0 % number of testsets: 1

pyrimidines02 real[A]

ATTRIBUTES: numeric: 28 nominal: 0 boolean: 0 all: 28

CLASS ATTRIBUTE: type: none number of values: 0 major class: 0.0 %

FURTHER INFO: number of instances: 44 missing values: 0.0 % number of testsets: 1

pyrimidines03 boolean[A]

ATTRIBUTES: numeric: 54 nominal: 1 boolean: 1 all: 55

CLASS ATTRIBUTE: type: boolean number of values: 2 major class: 50.0 %

FURTHER INFO: number of instances: 1774 missing values: 0.0 % number of testsets: 1

pyrimidines03 real[A]

ATTRIBUTES: numeric: 28 nominal: 0 boolean: 0 all: 28

CLASS ATTRIBUTE: type: none number of values: 0 major class: 0.0 %

FURTHER INFO: number of instances: 44 missing values: 0.0 % number of testsets: 1

pyrimidines04 boolean[A]

ATTRIBUTES: numeric: 54 nominal: 1 boolean: 1 all: 55

CLASS ATTRIBUTE: type: boolean number of values: 2 major class: 50.0 %

FURTHER INFO: number of instances: 1776 missing values: 0.0 % number of testsets: 1

pyrimidines04 real[A]

ATTRIBUTES: numeric: 28 nominal: 0 boolean: 0 all: 28

CLASS ATTRIBUTE: type: numeric number of values: 0 major class: 0.0 %

FURTHER INFO: number of instances: 44 missing values: 0.0 % number of testsets: 1

pyrimidines05 boolean[A]

ATTRIBUTES: numeric: 54 nominal: 1 boolean: 1 all: 55

CLASS ATTRIBUTE: type: boolean number of values: 2 major class: 50.0 %

FURTHER INFO: number of instances: 1784 missing values: 0.0 % number of testsets: 1

pyrimidines05 real[A]

ATTRIBUTES: numeric: 28 nominal: 0 boolean: 0 all: 28

CLASS ATTRIBUTE: type: numeric number of values: 0 major class: 0.0 %

FURTHER INFO: number of instances: 44 missing values: 0.0 % number of testsets: 1

re0[F]

ATTRIBUTES: numeric: 2886 nominal: 1 boolean: 0 all: 2887

CLASS ATTRIBUTE: type: nominal number of values: 13 major class: 40.4 %

FURTHER INFO: number of instances: 1504 missing values: 0.0 % number of testsets: 0

re1[F]

ATTRIBUTES: numeric: 3758 nominal: 1 boolean: 0 all: 3759

CLASS ATTRIBUTE: type: nominal number of values: 25 major class: 22.4 %

FURTHER INFO: number of instances: 1657 missing values: 0.0 % number of testsets: 0

recognition digits optical[A]

ATTRIBUTES: numeric: 64 nominal: 1 boolean: 0 all: 65

CLASS ATTRIBUTE: type: nominal number of values: 10 major class: 10.2 %

FURTHER INFO: number of instances: 3823 missing values: 0.0 % number of testsets: 1

recognition digits pen-based[A]

ATTRIBUTES: numeric: 16 nominal: 1 boolean: 0 all: 17

CLASS ATTRIBUTE: type: nominal number of values: 10 major class: 10.4 %
FURTHER INFO: number of instances: 7494 missing values: 0.0 % number of testsets: 1

schizo[C]

ATTRIBUTES: numeric: 12 nominal: 3 boolean: 2 all: 15
CLASS ATTRIBUTE: type: boolean number of values: 2 major class: 52.1 %
FURTHER INFO: number of instances: 340 missing values: 16.4 % number of testsets: 0

servo[A]

ATTRIBUTES: numeric: 1 nominal: 4 boolean: 0 all: 5
CLASS ATTRIBUTE: type: numeric number of values: 0 major class: 0.0 %
FURTHER INFO: number of instances: 167 missing values: 0.0 % number of testsets: 0

shuttle landing[A]

ATTRIBUTES: numeric: 0 nominal: 7 boolean: 2 all: 7
CLASS ATTRIBUTE: type: boolean number of values: 2 major class: 60.0 %
FURTHER INFO: number of instances: 15 missing values: 0.0 % number of testsets: 0

solar flare 1[A]

ATTRIBUTES: numeric: 3 nominal: 10 boolean: 5 all: 13
CLASS ATTRIBUTE: type: none number of values: 0 major class: 0.0 %
FURTHER INFO: number of instances: 323 missing values: 0.0 % number of testsets: 0

solar flare 2[A]

ATTRIBUTES: numeric: 3 nominal: 10 boolean: 5 all: 13
CLASS ATTRIBUTE: type: none number of values: 0 major class: 0.0 %
FURTHER INFO: number of instances: 1066 missing values: 0.0 % number of testsets: 0

sonar[A]

ATTRIBUTES: numeric: 60 nominal: 1 boolean: 1 all: 61
CLASS ATTRIBUTE: type: boolean number of values: 2 major class: 53.4 %
FURTHER INFO: number of instances: 208 missing values: 0.0 % number of testsets: 0

soybean large[A]

ATTRIBUTES: numeric: 0 nominal: 36 boolean: 16 all: 36
CLASS ATTRIBUTE: type: nominal number of values: 19 major class: 13.0 %
FURTHER INFO: number of instances: 307 missing values: 6.4 % number of testsets: 1

soybean small[A]

ATTRIBUTES: numeric: 0 nominal: 36 boolean: 16 all: 36
CLASS ATTRIBUTE: type: none number of values: 0 major class: 0.0 %
FURTHER INFO: number of instances: 47 missing values: 0.0 % number of testsets: 0

spambase[A]

ATTRIBUTES: numeric: 57 nominal: 1 boolean: 1 all: 58
CLASS ATTRIBUTE: type: boolean number of values: 2 major class: 60.6 %
FURTHER INFO: number of instances: 4601 missing values: 0.0 % number of testsets: 0

spec-t heart[A]

ATTRIBUTES: numeric: 0 nominal: 23 boolean: 23 all: 23
CLASS ATTRIBUTE: type: boolean number of values: 2 major class: 50.0 %
FURTHER INFO: number of instances: 80 missing values: 0.0 % number of testsets: 1

spec-tf heart[A]

ATTRIBUTES: numeric: 44 nominal: 1 boolean: 1 all: 45
CLASS ATTRIBUTE: type: boolean number of values: 2 major class: 50.0 %
FURTHER INFO: number of instances: 80 missing values: 0.0 % number of testsets: 1

sponge[A]

ATTRIBUTES: numeric: 0 nominal: 46 boolean: 15 all: 46

CLASS ATTRIBUTE: type: nominal number of values: 76 major class: 1.3 %

FURTHER INFO: number of instances: 76 missing values: 0.6 % number of testsets: 0

squash-stored[D]

ATTRIBUTES: numeric: 21 nominal: 4 boolean: 0 all: 25

CLASS ATTRIBUTE: type: ordinal number of values: 3 major class: 44.2 %

FURTHER INFO: number of instances: 52 missing values: 0.5 % number of testsets: 0

squash-unstored[D]

ATTRIBUTES: numeric: 20 nominal: 4 boolean: 0 all: 24

CLASS ATTRIBUTE: type: nominal number of values: 3 major class: 46.2 %

FURTHER INFO: number of instances: 52 missing values: 3.1 % number of testsets: 0

statlog australian credit[A]

ATTRIBUTES: numeric: 6 nominal: 9 boolean: 5 all: 15

CLASS ATTRIBUTE: type: boolean number of values: 2 major class: 55.5 %

FURTHER INFO: number of instances: 690 missing values: 0.0 % number of testsets: 0

statlog german credit[A]

ATTRIBUTES: numeric: 7 nominal: 14 boolean: 3 all: 21

CLASS ATTRIBUTE: type: boolean number of values: 2 major class: 70.0 %

FURTHER INFO: number of instances: 1000 missing values: 0.0 % number of testsets: 0

statlog german credit numeric[A]

ATTRIBUTES: numeric: 24 nominal: 1 boolean: 1 all: 25

CLASS ATTRIBUTE: type: boolean number of values: 2 major class: 70.0 %

FURTHER INFO: number of instances: 1000 missing values: 0.0 % number of testsets: 0

statlog heart[A]

ATTRIBUTES: numeric: 7 nominal: 7 boolean: 3 all: 14

CLASS ATTRIBUTE: type: boolean number of values: 2 major class: 55.6 %

FURTHER INFO: number of instances: 270 missing values: 0.0 % number of testsets: 0

statlog image segmentation[A]

ATTRIBUTES: numeric: 18 nominal: 2 boolean: 0 all: 20

CLASS ATTRIBUTE: type: nominal number of values: 7 major class: 14.3 %

FURTHER INFO: number of instances: 2310 missing values: 0.0 % number of testsets: 0

statlog satellite[A]

ATTRIBUTES: numeric: 36 nominal: 1 boolean: 0 all: 37

CLASS ATTRIBUTE: type: nominal number of values: 7 major class: 24.2 %

FURTHER INFO: number of instances: 4435 missing values: 0.0 % number of testsets: 1

statlog shuttle[A]

ATTRIBUTES: numeric: 9 nominal: 1 boolean: 0 all: 10

CLASS ATTRIBUTE: type: nominal number of values: 7 major class: 78.4 %

FURTHER INFO: number of instances: 43500 missing values: 0.0 % number of testsets: 1

statlog vehicle xaa[A]

ATTRIBUTES: numeric: 18 nominal: 1 boolean: 0 all: 19

CLASS ATTRIBUTE: type: nominal number of values: 4 major class: 29.8 %

FURTHER INFO: number of instances: 94 missing values: 0.0 % number of testsets: 0

statlog vehicle xab[A]

ATTRIBUTES: numeric: 18 nominal: 1 boolean: 0 all: 19

CLASS ATTRIBUTE: type: nominal number of values: 4 major class: 27.7 %
FURTHER INFO: number of instances: 94 missing values: 0.0 % number of testsets: 0

statlog vehicle xac[A]

ATTRIBUTES: numeric: 18 nominal: 1 boolean: 0 all: 19
CLASS ATTRIBUTE: type: nominal number of values: 4 major class: 28.7 %
FURTHER INFO: number of instances: 94 missing values: 0.0 % number of testsets: 0

statlog vehicle xad[A]

ATTRIBUTES: numeric: 18 nominal: 1 boolean: 0 all: 19
CLASS ATTRIBUTE: type: nominal number of values: 4 major class: 30.9 %
FURTHER INFO: number of instances: 94 missing values: 0.0 % number of testsets: 0

statlog vehicle xae[A]

ATTRIBUTES: numeric: 18 nominal: 1 boolean: 0 all: 19
CLASS ATTRIBUTE: type: nominal number of values: 4 major class: 30.9 %
FURTHER INFO: number of instances: 94 missing values: 0.0 % number of testsets: 0

statlog vehicle xaf[A]

ATTRIBUTES: numeric: 18 nominal: 1 boolean: 0 all: 19
CLASS ATTRIBUTE: type: nominal number of values: 4 major class: 31.9 %
FURTHER INFO: number of instances: 94 missing values: 0.0 % number of testsets: 0

statlog vehicle xag[A]

ATTRIBUTES: numeric: 18 nominal: 1 boolean: 0 all: 19
CLASS ATTRIBUTE: type: nominal number of values: 4 major class: 29.8 %
FURTHER INFO: number of instances: 94 missing values: 0.0 % number of testsets: 0

statlog vehicle xah[A]

ATTRIBUTES: numeric: 18 nominal: 1 boolean: 0 all: 19
CLASS ATTRIBUTE: type: nominal number of values: 4 major class: 27.7 %
FURTHER INFO: number of instances: 94 missing values: 0.0 % number of testsets: 0

statlog vehicle xai[A]

ATTRIBUTES: numeric: 18 nominal: 1 boolean: 0 all: 19
CLASS ATTRIBUTE: type: nominal number of values: 4 major class: 29.8 %
FURTHER INFO: number of instances: 94 missing values: 0.0 % number of testsets: 0

stock[G]

ATTRIBUTES: numeric: 10 nominal: 0 boolean: 0 all: 10
CLASS ATTRIBUTE: type: none number of values: 0 major class: 0.0 %
FURTHER INFO: number of instances: 950 missing values: 0.0 % number of testsets: 0

swd[E]

ATTRIBUTES: numeric: 11 nominal: 0 boolean: 0 all: 11
CLASS ATTRIBUTE: type: numeric number of values: 0 major class: 0.0 %
FURTHER INFO: number of instances: 1000 missing values: 0.0 % number of testsets: 0

teaching assistant[A]

ATTRIBUTES: numeric: 1 nominal: 5 boolean: 2 all: 6
CLASS ATTRIBUTE: type: ordinal number of values: 3 major class: 34.4 %
FURTHER INFO: number of instances: 151 missing values: 0.0 % number of testsets: 0

thyroid 87[A]

ATTRIBUTES: numeric: 7 nominal: 22 boolean: 21 all: 30
CLASS ATTRIBUTE: type: string number of values: 0 major class: 0.0 %
FURTHER INFO: number of instances: 9172 missing values: 5.3 % number of testsets: 0

thyroid allbp[A]

ATTRIBUTES: numeric: 7 nominal: 22 boolean: 21 all: 30

CLASS ATTRIBUTE: type: string number of values: 0 major class: 0.0 %

FURTHER INFO: number of instances: 2800 missing values: 5.4 % number of testsets: 1

thyroid allhyp[A]

ATTRIBUTES: numeric: 7 nominal: 22 boolean: 21 all: 30

CLASS ATTRIBUTE: type: string number of values: 0 major class: 0.0 %

FURTHER INFO: number of instances: 2800 missing values: 5.4 % number of testsets: 1

thyroid allhyper[A]

ATTRIBUTES: numeric: 7 nominal: 22 boolean: 21 all: 30

CLASS ATTRIBUTE: type: string number of values: 0 major class: 0.0 %

FURTHER INFO: number of instances: 2800 missing values: 5.4 % number of testsets: 1

thyroid allrep[A]

ATTRIBUTES: numeric: 7 nominal: 22 boolean: 21 all: 30

CLASS ATTRIBUTE: type: string number of values: 0 major class: 0.0 %

FURTHER INFO: number of instances: 2800 missing values: 5.4 % number of testsets: 1

thyroid ann[A]

ATTRIBUTES: numeric: 6 nominal: 16 boolean: 15 all: 22

CLASS ATTRIBUTE: type: nominal number of values: 3 major class: 92.5 %

FURTHER INFO: number of instances: 3772 missing values: 0.0 % number of testsets: 1

thyroid dis[A]

ATTRIBUTES: numeric: 7 nominal: 22 boolean: 21 all: 30

CLASS ATTRIBUTE: type: string number of values: 0 major class: 0.0 %

FURTHER INFO: number of instances: 2800 missing values: 5.4 % number of testsets: 1

thyroid new[A]

ATTRIBUTES: numeric: 5 nominal: 1 boolean: 0 all: 6

CLASS ATTRIBUTE: type: nominal number of values: 3 major class: 69.8 %

FURTHER INFO: number of instances: 215 missing values: 0.0 % number of testsets: 0

thyroid sick[A]

ATTRIBUTES: numeric: 7 nominal: 22 boolean: 21 all: 30

CLASS ATTRIBUTE: type: string number of values: 0 major class: 0.0 %

FURTHER INFO: number of instances: 2800 missing values: 5.4 % number of testsets: 1

thyroid sick-euthyroid[A]

ATTRIBUTES: numeric: 7 nominal: 19 boolean: 19 all: 26

CLASS ATTRIBUTE: type: boolean number of values: 2 major class: 90.7 %

FURTHER INFO: number of instances: 3163 missing values: 6.5 % number of testsets: 0

tic tac toe[A]

ATTRIBUTES: numeric: 0 nominal: 10 boolean: 1 all: 10

CLASS ATTRIBUTE: type: boolean number of values: 2 major class: 65.3 %

FURTHER INFO: number of instances: 958 missing values: 0.0 % number of testsets: 0

tr11[F]

ATTRIBUTES: numeric: 6429 nominal: 1 boolean: 0 all: 6430

CLASS ATTRIBUTE: type: nominal number of values: 9 major class: 31.9 %

FURTHER INFO: number of instances: 414 missing values: 0.0 % number of testsets: 0

tr12[F]

ATTRIBUTES: numeric: 5804 nominal: 1 boolean: 0 all: 5805

CLASS ATTRIBUTE: type: nominal number of values: 8 major class: 29.7 %
FURTHER INFO: number of instances: 313 missing values: 0.0 % number of testsets: 0

tr21$^{\mathbf{F}}$

ATTRIBUTES: numeric: 7902 nominal: 1 boolean: 0 all: 7903
CLASS ATTRIBUTE: type: nominal number of values: 6 major class: 68.8 %
FURTHER INFO: number of instances: 336 missing values: 0.0 % number of testsets: 0

tr23$^{\mathbf{F}}$

ATTRIBUTES: numeric: 5832 nominal: 1 boolean: 0 all: 5833
CLASS ATTRIBUTE: type: nominal number of values: 6 major class: 44.6 %
FURTHER INFO: number of instances: 204 missing values: 0.0 % number of testsets: 0

tr31$^{\mathbf{F}}$

ATTRIBUTES: numeric: 10128 nominal: 1 boolean: 0 all: 10129
CLASS ATTRIBUTE: type: nominal number of values: 7 major class: 38.0 %
FURTHER INFO: number of instances: 927 missing values: 0.0 % number of testsets: 0

tr41$^{\mathbf{F}}$

ATTRIBUTES: numeric: 7454 nominal: 1 boolean: 0 all: 7455
CLASS ATTRIBUTE: type: nominal number of values: 10 major class: 27.7 %
FURTHER INFO: number of instances: 878 missing values: 0.0 % number of testsets: 0

tr45$^{\mathbf{F}}$

ATTRIBUTES: numeric: 8261 nominal: 1 boolean: 0 all: 8262
CLASS ATTRIBUTE: type: nominal number of values: 10 major class: 23.2 %
FURTHER INFO: number of instances: 690 missing values: 0.0 % number of testsets: 0

trains$^{\mathbf{A}}$

ATTRIBUTES: numeric: 0 nominal: 33 boolean: 15 all: 33
CLASS ATTRIBUTE: type: boolean number of values: 2 major class: 50.0 %
FURTHER INFO: number of instances: 10 missing values: 0.0 % number of testsets: 0

triazines01 boolean$^{\mathbf{A}}$

ATTRIBUTES: numeric: 120 nominal: 1 boolean: 1 all: 121
CLASS ATTRIBUTE: type: boolean number of values: 2 major class: 50.0 %
FURTHER INFO: number of instances: 23650 missing values: 0.0 % number of testsets: 1

triazines01 real$^{\mathbf{A}}$

ATTRIBUTES: numeric: 61 nominal: 0 boolean: 0 all: 61
CLASS ATTRIBUTE: type: numeric number of values: 0 major class: 0.0 %
FURTHER INFO: number of instances: 155 missing values: 0.0 % number of testsets: 1

triazines02 boolean$^{\mathbf{A}}$

ATTRIBUTES: numeric: 120 nominal: 1 boolean: 1 all: 121
CLASS ATTRIBUTE: type: boolean number of values: 2 major class: 50.0 %
FURTHER INFO: number of instances: 23662 missing values: 0.0 % number of testsets: 1

triazines02 real$^{\mathbf{A}}$

ATTRIBUTES: numeric: 61 nominal: 0 boolean: 0 all: 61
CLASS ATTRIBUTE: type: numeric number of values: 0 major class: 0.0 %
FURTHER INFO: number of instances: 155 missing values: 0.0 % number of testsets: 1

triazines03 boolean$^{\mathbf{A}}$

ATTRIBUTES: numeric: 120 nominal: 1 boolean: 1 all: 121
CLASS ATTRIBUTE: type: boolean number of values: 2 major class: 50.0 %
FURTHER INFO: number of instances: 23696 missing values: 0.0 % number of testsets: 1

triazines03 real[A]

ATTRIBUTES: numeric: 61 nominal: 0 boolean: 0 all: 61

CLASS ATTRIBUTE: type: numeric number of values: 0 major class: 0.0 %

FURTHER INFO: number of instances: 155 missing values: 0.0 % number of testsets: 1

triazines04 boolean[A]

ATTRIBUTES: numeric: 120 nominal: 1 boolean: 1 all: 121

CLASS ATTRIBUTE: type: boolean number of values: 2 major class: 50.0 %

FURTHER INFO: number of instances: 23660 missing values: 0.0 % number of testsets: 1

triazines04 real[A]

ATTRIBUTES: Nnumeric: 61 nominal: 0 boolean: 0 all: 61

CLASS ATTRIBUTE: type: numeric number of values: 0 major class: 0.0 %

FURTHER INFO: number of instances: 155 missing values: 0.0 % number of testsets: 1

triazines05 boolean[A]

ATTRIBUTES: numeric: 120 nominal: 1 boolean: 1 all: 121

CLASS ATTRIBUTE: type: boolean number of values: 2 major class: 50.0 %

FURTHER INFO: number of instances: 23698 missing values: 0.0 % number of testsets: 1

triazines05 real[A]

ATTRIBUTES: numeric: 61 nominal: 0 boolean: 0 all: 61

CLASS ATTRIBUTE: type: numeric number of values: 0 major class: 0.0 %

FURTHER INFO: number of instances: 155 missing values: 0.0 % number of testsets: 1

triazines06 boolean[A]

ATTRIBUTES: numeric: 120 nominal: 1 boolean: 1 all: 121

CLASS ATTRIBUTE: type: boolean number of values: 2 major class: 50.0 %

FURTHER INFO: number of instances: 23672 missing values: 0.0 % number of testsets: 1

triazines06 real[A]

ATTRIBUTES: numeric: 61 nominal: 0 boolean: 0 all: 61

CLASS ATTRIBUTE: type: numeric number of values: 0 major class: 0.0 %

FURTHER INFO: number of instances: 155 missing values: 0.0 % number of testsets: 1

vowel[A]

ATTRIBUTES: numeric: 10 nominal: 4 boolean: 2 all: 14

CLASS ATTRIBUTE: type: nominal number of values: 11 major class: 9.1 %

FURTHER INFO: number of instances: 990 missing values: 0.0 % number of testsets: 0

wap[F]

ATTRIBUTES: numeric: 8460 nominal: 1 boolean: 0 all: 8461

CLASS ATTRIBUTE: type: nominal number of values: 20 major class: 21.9 %

FURTHER INFO: number of instances: 1560 missing values: 0.0 % number of testsets: 0

water treatement plant[A]

ATTRIBUTES: numeric: 38 nominal: 0 boolean: 0 all: 39

CLASS ATTRIBUTE: type: none number of values: 0 major class: 0.0 %

FURTHER INFO: number of instances: 527 missing values: 2.9 % number of testsets: 0

waveform with noise[A]

ATTRIBUTES: numeric: 40 nominal: 1 boolean: 0 all: 41

CLASS ATTRIBUTE: type: nominal number of values: 3 major class: 33.8 %

FURTHER INFO: number of instances: 5000 missing values: 0.0 % number of testsets: 0

waveform without noise[A]

ATTRIBUTES: numeric: 21 nominal: 1 boolean: 0 all: 22

CLASS ATTRIBUTE: type: nominal number of values: 3 major class: 33.9 %
FURTHER INFO: number of instances: 5000 missing values: 0.0 % number of testsets: 0

white-clover[D]

ATTRIBUTES: numeric: 27 nominal: 5 boolean: 0 all: 32
CLASS ATTRIBUTE: type: nominal number of values: 4 major class: 60.3 %
FURTHER INFO: number of instances: 63 missing values: 0.0 % number of testsets: 0

wine recognition[A]

ATTRIBUTES: numeric: 13 nominal: 1 boolean: 0 all: 14
CLASS ATTRIBUTE: type: nominal number of values: 3 major class: 39.9 %
FURTHER INFO: number of instances: 178 missing values: 0.0 % number of testsets: 0

wisconsin breast cancer[A]

ATTRIBUTES: numeric: 10 nominal: 1 boolean: 1 all: 11
CLASS ATTRIBUTE: type: boolean number of values: 2 major class: 65.5 %
FURTHER INFO: number of instances: 699 missing values: 0.2 % number of testsets: 0

wisconsin diagnostic breast cancer[A]

ATTRIBUTES: numeric: 31 nominal: 1 boolean: 1 all: 32
CLASS ATTRIBUTE: type: boolean number of values: 2 major class: 62.7 %
FURTHER INFO: number of instances: 569 missing values: 0.0 % number of testsets: 0

wisconsin prognostic breast cancer[A]

ATTRIBUTES: numeric: 34 nominal: 1 boolean: 1 all: 35
CLASS ATTRIBUTE: type: boolean number of values: 2 major class: 76.3 %
FURTHER INFO: number of instances: 198 missing values: 0.1 % number of testsets: 0

wseries[C]

ATTRIBUTES: numeric: 1 nominal: 8 boolean: 1 all: 9
CLASS ATTRIBUTE: type: none number of values: 0 major class: 0.0 %
FURTHER INFO: number of instances: 90 missing values: 23.6 % number of testsets: 0

yeast[A]

ATTRIBUTES: numeric: 8 nominal: 1 boolean: 0 all: 10
CLASS ATTRIBUTE: type: nominal number of values: 10 major class: 31.2 %
FURTHER INFO: number of instances: 1484 missing values: 0.0 % number of testsets: 0

zoo[A]

ATTRIBUTES: numeric: 0 nominal: 17 boolean: 15 all: 18
CLASS ATTRIBUTE: type: nominal number of values: 7 major class: 40.6 %
FURTHER INFO: number of instances: 101 missing values: 0.0 %