Universität Ulm
Fakultät für Ingenieurwissenschaften und Informatik
Institut für Datenbanken und Informationssysteme

DaimlerChrysler

Konzeption und Umsetzung eines Workflow Management Systems zur Entwicklung und Qualitätssicherung von Erkennungssystemen

Diplomarbeit zur Erlangung des akademischen Grades Diplom-Informatiker

Diplomand:	Johann Specht
Betreuer:	Prof. Dr. Peter Dadam
Zweitkorrektor:	Dr. Stefanie Rinderle-Ma
Abgabedatum:	11. Mai 2007

Danksagung

Ich möchte mich bei allen bedanken, die mich beim Erstellen meiner Diplomarbeit unterstützt haben. Vor allem bei meinen Betreuern Matthias Oberländer und Prof. Dr. Peter Dadam. Aber auch bei Tilo Schwarz für seine hilfreichen Tipps und bei Ulrich Kreher für seine Unterstützung im Umgang mit ADEPT. Und ich will mich bei meiner Schwester Tamara Specht dafür bedanken, dass sie fleißig Rechtschreibfehler in dieser Arbeit korrigiert hat.

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Motivation

Dem Thema Sicherheit im Straßenverkehr wird von der Automobilbranche eine immer höhere Bedeutung beigemessen. Vor allem wird viel in Forschung und Entwicklung von Technologien investiert, die helfen sollen die Zahl der Unfälle zu reduzieren und die Straßen sicherer zu machen. Und wie die Unfallstatistiken zeigen ([Statis]), helfen diese Investitionen tatsächlich, Menschenleben zu retten.

Die Informationstechnik eröffnet hier neue Möglichkeiten. Eine davon sind die sogenannten "Erkennungssysteme", die mit Hilfe von Kameras und intelligenter Software den Fahrzeugen die Fähigkeit zu "sehen" vermitteln. Damit kann ein Fahrzeug einen Fußgänger auf der Fahrbahn erkennen, oder Alarm schlagen, falls es von der Straße abkommt.

An solchen Systemen wird bei DaimlerChrysler ([Daimler]) intensiv geforscht. Dabei werden große Datenmengen erzeugt und verarbeitet: Bisher wurden etwa 1500 Bild-Sequenzen mit jeweils etwa 1200 Einzelaufnahmen aufgenommen, dazu kommen etwa 1600 Label-Dokumente, welche Markierungen (Label) für unterschiedliche Objekte enthalten (etwa 930000). All diese Daten müssen effizient verwaltet und verarbeitet werden.

Von zentraler Bedeutung sind die Arbeitsabläufe oder auch der Workflow. Es müssen immer wieder die gleichen Aufgaben erledigt werden, wobei nichts vergessen werden darf und die Qualität der Ergebnisse am Ende stimmen muss. Von

Fall zu Fall kann es aber auch Unterschiede geben, weshalb Flexibilität bei den Arbeitsabläufen gefragt ist. Zu berücksichtigen ist auch, dass es Prozesse gibt, die ohne Interaktion mit dem Benutzer auskommen und solche, die mit dem Benutzer interagieren müssen. Es wird also ein System benötigt, welches automatisch Aufgaben erledigen kann und dem Benutzer hilft seine Arbeit zu erledigen.

1.2 Aufbau der Arbeit

Im Kapitel 2 wird zuerst die Aufgabenstellung vorgestellt. Anschließend werden im Kapitel 3 mögliche Lösungswege besprochen und diskutiert, allerdings ohne auf bestimmte technische Umsetzungen einzugehen. Kapitel 4 beschäftigt sich vor allem mit der technischen Seite und stellt entsprechende Lösungen vor. Kapitel 5 geht auf weniger technische Fragestellungen ein.

1.3 Software

Entwickelt wurde unter SuSE Linux 9.3 ([SuSELi]) mit Python 2.4.4 [Python], GCC 3.3.5 ([GNUGCC]), Java 1.3.1 ([Java13]), ADEPT 1 ([ADEPT]), Apache 2.2.4 ([Apache]), mod_python ([modpy]), MochiKit ([MochiKit]), SCons 0.96.1 ([Scons]), SWIG 1.3.21 ([SWIG]). Die Diagramme wurden mit Dia 0.96.1 ([Dia]) unter Linux erstellt.

Kapitel 2

Aufgabenstellung

Ein Erkennungssystem ist eine Software, die auf Bildern Objekte oder Menschen erkennen kann. Es besteht aus einer ausführbaren Datei und einer oder mehreren Konfigurationsdateien. Wichtig dabei ist, dass eine Änderung der ausführbaren Datei oder der Konfiguration ein neues Erkennungssystem ergibt, da die Änderung die Eigenschaften des Erkennungssystems beeinflusst. Die Erkennungssysteme, die bei DaimlerChrysler entwickelt werden, markieren erkannte Objekte, indem sie Rechtecke um diese Objekte legen. Es können aber auch offene oder geschlossene Polygone und Punkte sein.

Das System muss die Möglichkeit bieten, die Erkennungssysteme zu verwalten, also neue Erkennungssysteme zu definieren, vorhandene zu bearbeiten oder zu löschen.

Um Erkennungssysteme zu entwickeln, braucht man vor allem folgende drei Dinge:

- Algorithmen und Verfahren
- Daten
- Tests

Dazu kommen einige Abläufe, die immerzu wiederholt werden müssen, wobei die Qualität der Ergebnisse sichergestellt werden muss. In diesem Kapitel werden die Daten, die Arbeitsabläufe und einige andere Anforderungen beschrieben.

2.1 Daten

Den mit Abstand größten Teil der Daten machen die Bild-Sequenzen und die Label-Dokumente aus. Der Datenbestand hat bereits eine beachtliche Größe erreicht und wird in Zukunft noch weiter wachsen. Es ist deswegen wichtig, die Daten effizient zu verwalten und flexible und schnelle Abfragen anzubieten.

2.1.1 Bild-Sequenzen

Die Bild-Sequenzen bestehen aus Infrarot-Aufnahmen, die mit speziell ausgerüsteten Fahrzeugen aufgenommen werden. Meist besteht eine Bild-Sequenz aus mehreren Kanälen (channel), wobei ein Kanal einer bestimmten Infrarot-Kamera entspricht. Die Bild-Sequenzen werden in Verzeichnissen abgelegt, die das Aufnah-

Abbildung 2.1: Beispiel einer Infrarotaufnahme bei Nacht.

medatum als Namen haben. Jede Sequenz bekommt ihr eigenes Unterverzeichnis,

welches das Aufnahmedatum und die Aufnahmezeit im Namen trägt, siehe Abbildung 2.2. Auch die Kanäle werden in eigenen Unterverzeichnissen gespeichert.

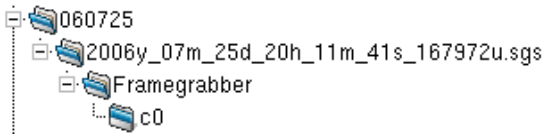

Abbildung 2.2: Speicherung der Bild-Sequenzen.

Im Beispiel liegen im Verzeichnis "060725" die Sequenzen, die am 25.07.2006 aufgenommen wurden. Hier ist es nur eine einzige:
"2006y_07m_25d_20h_11m_41s_167972u.sgs", die einen Kanal mit dem Namen "Framegrabber/c0" enthält.

Die einzelnen Aufnahmen (Frames) werden als 16-Bit Graustufenbilder im Tiff-Format [TIFF] im jeweiligen Kanal-Verzeichnis gespeichert.

Benutzt werden die Bild-Sequenzen für zweierlei:

- Erkennungssysteme trainieren
- Erkennungssysteme testen

Die genauen Abläufe beim Testen der Erkennungssysteme werden weiter unten beschrieben. Das Trainieren der Erkennungssysteme ist hier nicht weiter von Bedeutung, nur soviel: es dient dazu, die Konfigurationen für Erkennungssysteme zu erstellen.

Die Sequenzen, Kanäle und einzelnen Frames müssen mit Tags [1] und Kommentaren versehen werden können.

Inzwischen wurden etwa 1400 Sequenzen aufgenommen, die durchschnittlich je 1200 Frames in mehreren Kanälen enthalten. Ein Frame ist zwischen 30KByte und 600KByte groß. Bei einer durchschnittlichen Frame-Größe von 300KByte (0.3MByte) ergibt sich eine Datenmenge von: $1400 * 1200 * 0.3 = 504000$MByte.

[1] Oder auch Etikett, Marke. Dient der Auszeichnung mit zusätzlichen Informationen.

2.1.2 Label-Dokumente

Ein Label-Dokument ist eine Textdatei, die eine Python-Datenstruktur enthält. Diese Datenstruktur wiederum enthält zu jedem Frame die zugehörigen Markierungen (Label) von Objekten. Ein Label-Dokument bezieht sich immer auf eine Bild-Sequenz und einen Kanal. Ein Label ist immer einem Frame zugeordnet und ist ein geometrisches Objekt, meist ein Rechteck oder auch ein Punkt oder ein geschlossenes/offenes Polygon. Die geometrische Form eines Labels wird als Typ mitgespeichert, zum Beispiel "rect" für Rechteck oder "point" für Punkt. Jedes Label bekommt eine Bezeichnung (track_id), die sich aus dem Namen des markierten Objektes und einer fortlaufenden Nummer zusammen setzt (zum Beispiel: "Car_1" für ein PKW, oder "Pedestrian_3" für einen Fußgänger). Die Punkte eines Labels werden in zwei Listen gespeichert, eine für die X-Koordinaten und eine für die Y-Koordinaten.

Abbildung 2.3: Label Beispiele: Rechtecke, Punkte und offene Polygone.

Neben den Labeln werden noch weitere Informationen in einem Label-Dokument gespeichert:

- Name der Bild-Sequenz und Kanal
- Name des Bearbeiters
- Erstellungsdatum
- die verwendete Konfiguration für die Anwendung Lava, die zum Markieren von Objekten eingesetzt wird (siehe 2.2.2)

Label-Dokumente enthalten auch Tags und Kommentare für das Dokument selbst, für die Bild-Sequenz/Kanal, für einzelne Frames und Label. Diese Tags und Kommentare müssen ebenfalls gespeichert und verwaltet werden.

Es wird zwischen Soll- und Ist-Daten unterschieden. Erstere sind Label-Dokumente, die beim manuellen Labeln (Markieren) entstanden sind. Sie werden benutzt, um die Güte eines Erkennungssystems zu prüfen. Dazu werden die Soll-Daten mit den Ist-Daten, die von den Erkennungssystemen ausgegeben werden, verglichen.

Bis jetzt wurden ungefähr 1600 Label-Dokumente (Soll-Daten) erstellt, die zusammen etwa 930000 einzelne Label enthalten. Bei geschätzten 46 Bytes/Label in der Datenbank ergibt das $930000 * 46 = 40,8$MByte.

Es muss sicher gestellt werden, dass die Label-Daten schnell und flexibel abgefragt werden können. Unter anderem müssen geometrische Anfragen möglich sein wie: gib alle rechteckigen Label zurück, die eine Fläche von mindestens 100 Pixeln haben.

2.2 Arbeitsabläufe

Die zweite wichtige Aufgabenstellung neben der Verwaltung der Daten, sind die Arbeitsabläufe. Diese müssen nach Möglichkeit automatisiert werden und wo es nicht geht, muss das System den Benutzer unterstützen. Zu beachten ist dabei, dass sich die Arbeitsabläufe durchaus ändern können und zwar sowohl temporär (also während der Ausführung) als auch dauerhaft. Das System muss also eine Möglichkeit bieten die Arbeitsabläufe mit wenig Aufwand zu ändern.

2.2.1 Bild-Sequenzen

Wie oben bereits erwähnt wurde, werden die Bild-Sequenzen mit speziell ausgerüsteten Fahrzeugen aufgenommen und auf der Festplatte des Fahrzeug-Rechners zwischengespeichert. Danach werden die Sequenzen auf den NFS-Server der Abteilung übertragen. Wichtig dabei ist, dass die Sequenzen nach vorgegebenem Muster abgelegt werden. In der Regel bedeutet die Aufnahme neuer Bild-Sequenzen, dass neue Soll-Daten erstellt werden müssen (siehe Kapitel 2.2.2).

Das System muss die Bild-Sequenzen verwalten und die Benutzer beim Import neuer Sequenzen unterstützen, indem diese automatisch an die richtige Stelle kopiert werden.

2.2.2 Markieren

Vor den eigentlichen Arbeitsabläufen soll hier noch kurz eine Anwendung namens “Lava” vorgestellt werden, die dazu benutzt wird, Soll-Daten zu erstellen (siehe Abbildung 2.4). Lava ist in Python geschrieben und läuft unter Windows und Linux. Es kann Label-Dokumente lesen und schreiben und Bild-Sequenzen mit 16-Bit Graustufen Bildern laden und anzeigen. Lava wird über eine Konfigurationsdatei konfiguriert und kann so auf die Label-Aufgabe angepasst werden (indem zum Beispiel nur die benötigten Zeichenwerkzeuge angeboten werden oder das richtige Konvertierungsverfahren von 16Bit zu 8Bit Graustufenbildern ausgewählt wird).

Auf der Abbildung 2.4 sieht man ein Beispiel für ein Label vom Typ “Rechteck” mit der track_id [2] “Pedestrian_1”.

Die Aufnahme neuer Bild-Sequenzen löst normalerweise die Erstellung neuer Soll-Daten für diese Sequenzen aus. Dazu erstellt ein Betreuer für jede Bild-Sequenz/Kanal-Kombination, für die Soll-Daten benötigt werden, ein Label-Dokument (Label-Auftrag, entspricht dem Label-Dokument aus Kapitel 2.1.2 aber noch ohne Label), welches die benötigten Informationen (Sequenz, Kanal, Speicherort der Sequenz und die Lava-Konfiguration) enthält. Der Name dieser Datei besteht aus dem Namen der Sequenz, dem Namen des Kanals, dem Namen

[2] Eine, pro Objekt, eindeutige Bezeichnung einer Markierung in einem Label-Dokument. Setzt sich aus dem Namen des markierten Objektes und einer fortlaufenden Nummer zusammen.

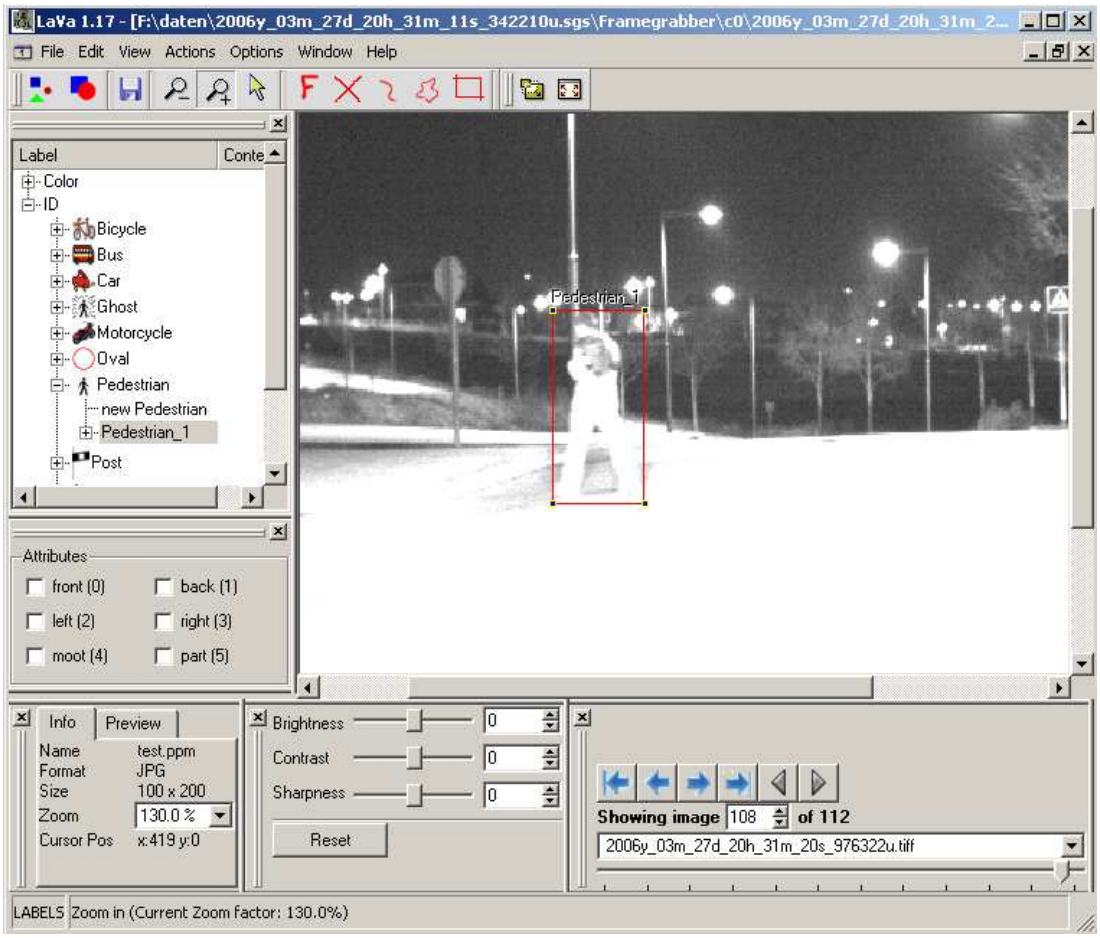

Abbildung 2.4: Lava unter Windows.

des Objekts, das markiert werden soll, und einer fortlaufenden Versionsnummer: "2005y_07m_26d_21h_14m_19s_360587u_IndigoCar;0.py". Diese Datei wird dann dem Bearbeiter (Labeler) zur Verfügung gestellt. Wobei zu beachten ist, dass es auch externe Bearbeiter gibt (Studenten, Schüler). Zusätzlich benötigt der Bearbeiter die richtige Lava-Konfiguration.

Der Bearbeiter kann dann das Label-Dokument mit Lava öffnen und bearbeiten. Das geänderte Label-Dokument schickt der Bearbeiter entweder per EMail an den Betreuer, oder legt es auf dem NFS-Server ab. Als nächstes überprüft der Betreuer das Ergebnis und schickt es zurück an den Bearbeiter, falls es nicht zufriedenstellend ist. Dabei kann er im Label-Dokument einen Kommentar mit der Beschreibung des Problems unterbringen. Sobald das Ergebnis die Qualitätskontrolle passiert hat, wird es auf dem NFS-Server abgelegt und steht dann für die Tests der Erkennungssysteme zur Verfügung.

Das System muss die Betreuer beim Anlegen neuer Label-Aufträge unterstützen und diese automatisch den richtigen Bearbeitern zur Verfügung stellen. Die Bearbeiter müssen die Möglichkeit haben, die Ergebnisse an den Betreuer zu schicken, wobei die Qualitätskontrolle sichergestellt werden muss. Danach können die Label-Dokumente als Soll-Daten abgespeichert werden.

2.2.3 Testläufe

Um zu überprüfen wie gut ein Erkennungssystem bestimmte Objekte erkennen kann, startet man einen Testlauf. Ein Testlauf wird definiert durch ein Erkennungssystem, Bild-Sequenzen und einen Rechner, auf dem das Erkennungssystem ausgeführt wird. Es werden einige Arbeitsplatzrechner benutzt, die unter Linux laufen und einen Zugang über SSH [OpenSSH] bieten. Ein Testlauf dauert meist sehr lange, weswegen man ihn im Hintergrund mit niedriger Priorität laufen lässt. Die Dauer eines Testlaufs hängt vom Erkennungssystem und der Zahl und der Größe der Bild-Sequenzen ab. Die Berechnungszeit pro Bild liegt zwischen Echtzeit und 5 Sekunden. Bei einer Bild-Sequenz mit 1200 Bildern (48 Sekunden bei 25 Bildern Pro Sekunde) und 5 Sekunden pro Bild ergiben sich 100 Minuten für die Berechnung. Ablauf:

- Einen Rechner aussuchen, auf dem noch keine Berechnung läuft

- Erkennungssystem auf diesen Rechner kopieren (zum Beispiel mit scp [OpenSSH]), die Bild-Sequenz kann direkt vom NFS-Server gelesen werden
- Auf dem Rechner direkt oder über SSH einloggen
- Das Erkennungssystem starten
- Sobald die Berechnung abgeschlossen wurde, das Ergebnis abholen

Aus den Abweichungen der Soll- und Ist-Daten können Diagramme erstellt werden, die die Abweichungen visualisieren.

Das System muss die Verwaltung der Testläufe unterstützen und diese automatisch ausführen. Es muss also möglich sein, einen neuen Testlauf zu definieren, indem man ein Erkennungssystem, Bild-Sequenzen und Maschinen angibt. Danach muss das System die Testläufe automatisch ausführen. Dabei muss es möglich sein, den Testläufen Prioritäten zuzuordnen, damit zuerst höher priorisierte Testläufe ausgeführt werden.

2.3 Benutzerverwaltung

Es wurden bereits zwei Typen von Benutzern erwähnt: Betreuer und Bearbeiter (Labeler). Diese haben unterschiedliche Aufgaben und Rechte, was eine Benutzerverwaltung notwendig macht. Das System sollte Aufgaben an "Rollen" knüpfen und auch "Fähigkeiten" kennen, die angeben, was der Benutzer kann. Folgende Rollen sind denkbar:

- Verwalter von Sequenzen
- Verwalter von Label-Dokumenten
- Verwalter von Label-Aufträgen
- Labeler [3]
- Verwalter von Erkennungssystemen
- Verwalter von Testläufen

[3] Benutzer, die auf Infrarotaufnahmen Objekte markieren.

Mögliche Fähigkeiten:

- Sequenzen importieren, löschen
- Label-Dokumente erzeugen/löschen
- Label-Dokumente modifizieren
- Erkennungssysteme anlegen, bearbeiten, löschen
- Testläufe anlegen, abbrechen, löschen

2.3.1 Zugriffsrechte

Um absichtliches oder unabsichtliches Löschen oder Verändern der Daten zu verhindern, muss das System den Zugriff auf die Daten kontrollieren und anhand der Zugriffsrechte zulassen oder ablehnen. Ein Labeler darf zum Beispiel keine Bild-Sequenzen löschen oder lesend/schreibend auf die Erkennungssysteme zugreifen.

2.4 Benutzerschnittstelle

Von großer Bedeutung ist das Benutzerinterface, da die Benutzer, insbesondere die Labeler, damit arbeiten müssen, und die Programmierschnittstelle, mit der auf das System zugegriffen werden kann.

2.4.1 Web-Interface

Das Benutzerinterface soll Web-basiert sein, da dies einige Vorteile mit sich bringt:

- keine Installation auf dem Client nötig
- plattformunabhängig
- von überall zugreifbar
- zentrale Administration und Pflege

Wichtig ist hier, dass das Web-Interface mit allen gängigen Browsern funktioniert, also Internet Explorer, Firefox, Opera. Auch die Sicherheit darf nicht zu kurz kommen, das System muss also zuerst den Benutzer zum Login auffordern und ihn danach nur auf die Seiten lassen, auf die er zugreifen darf.

Normale GUI-Anwendungen sollten nur in Ausnahmefällen benutzt werden (zum Beispiel wird Lava weiterhin zum Labeln benutzt).

2.4.2 Programmierschnittstelle

Großer Wert wird auch auf die Möglichkeit gelegt, das System möglichst vollständig und einfach über eine Programmierschnittstelle (API - Application Programming Interface) ansprechen zu können. Da Python die beforzugte Programmiersprache ist (siehe Kapitel 2.4.3), muss mindestens eine Python-API angeboten werden.

2.4.3 Sonstiges

Das System muss unter Linux laufen. Bevorzugte Programmiersprache ist Python. Es existiert ein lokales Netzwerk mit NFS-Freigaben, die Home-Verzeichnisse der Benutzer werden ebenfalls über NFS gemountet.

Kapitel 3

Analyse

Nachdem die Aufgabenstellung bereits vorgestellt wurde, wird in diesem Kapitel untersucht, welche Möglichkeiten für die Lösung der Aufgaben zur Verfügung stehen und was ihre Vor- und Nachteile sind. Es wird hier allerdings nicht auf konkrete technische Lösungen eingegangen (dies wird in den Kapiteln 4 und 5 gemacht), sondern allgemeine Fragen erörtert, zum Beispiel: wie können die Daten verwaltet werden oder wie können die Arbeitsabläufe (Prozesse) vom System unterstützt werden?

Wie man im vorigen Kapitel sehen konnte, gibt es vor allem zwei große Herausforderungen: Datenverwaltung und Workflow. Aus diesem Grund werden im Kapitel 3.1 zuerst die Datenverwaltung und im Kapitel 3.2 dann die Prozess-Abläufe diskutiert. Anschliessend werden noch die Benutzerverwaltung und das Thema Sicherheit erörtert.

3.1 Daten

Die Daten können in zwei große Bereiche unterteilt werden:

- Konkrete Daten
  - Bild-Sequenzen
  - Label-Dokumente

- Verwaltungsdaten
  - Benutzerdaten
  - Erkennungssysteme
  - Testläufe
  - Label-Aufträge

Die konkreten Daten werden bereits im Dateisystem abgelegt. Die Verwaltungsdaten existieren entweder nicht explizit (Testläufe), werden nicht zentral gespeichert (Erkennungssysteme) oder sind nur implizit vorhanden (Benutzerdaten, UNIX-Benutzer, Windows-Benutzer).

Für die Verwaltung der Daten gibt es im Wesentlichen zwei Möglichkeiten:

- Dateisystem
- Datenbankmanagementsystem (DBMS)

3.1.1 Anforderungen

Folgende Anforderungen werden an die Datenverwaltung gestellt:

- effiziente Verwaltung großer Datenmengen
- flexible Abfragen der Daten
- geometrische Datentypen

Die Menge der Daten sollte einen möglichst geringen Einfluss auf die Abfrage-Geschwindigkeit haben. Die Abfragen selbst können sehr komplex sein und sind nicht alle im Voraus zu bestimmen. Beispiel für eine Abfrage: Gebe alle Label zurück, die rechteckig und rot sind, deren Fläche mindestens 100 Pixel beträgt und die für die Bild-Sequenz “xyz” erstellt wurden.

3.1.2 Dateisysteme

Jedes Betriebssystem bietet für die Verwaltung der Daten mindestens ein Dateisystem an. Solch ein Dateisystem muss universell sein, da es alle möglichen Daten speichern können muss (Textdateien, Bilder, Maschinencode usw.). Diese Universalität bedeutet allerdings, dass viele Daten nicht optimal in einem Dateisystem verwaltet werden können. Vor allem sind die Möglichkeiten für die Suche, Sicherstellung der Integrität der Daten und flexible Abfragen sehr eingeschränkt. Allerdings braucht man all das nicht immer, so dass viele Daten tatsächlich im Dateisystem verbleiben können und damit für alle Anwendungen ohne Umwege verfügbar bleiben.

3.1.3 Datenbankmanagementsysteme

Wesentlich mehr Möglichkeiten bieten die Datenbankmanagementsysteme (DBMS): schnelle Suche, Sicherstellung der Integrität, sehr flexible Abfrage-Möglichkeiten usw. Ein weiterer großer Vorteil von DBMS ist die Möglichkeit, einen Teil der Anwendungslogik zentral in der Datenbank so abzulegen (Trigger, Stored Procedures, referenzielle Integrität), dass diese Logik nicht von Anwendungen oder Benutzern "vergessen" werden kann.

Viele Datenbankmanagementsysteme können auch mit geometrischen Daten umgehen (PostgreSQL, Oracle). Damit wäre auch die dritte Anforderung abgedeckt (siehe 3.1.1).

Allerdings macht es nicht bei allen Arten von Daten Sinn, diese in einer Datenbank zu speichern, dazu gehören vor allem binäre Daten: ausführbare Programme, Bilder, Videos, Musik usw. Solche Daten sind aus Sicht des DBMS einfach Datenblöcke, die nicht sinnvoll abgefragt werden können. Daten dieser Art können im Dateisystem verbleiben, wobei bestimmte Informationen (Pfad, Tags, Kommentare usw.) in die Datenbank geschrieben werden können.

3.1.4 Konkrete Daten

Die Bild-Sequenzen bestehen einerseits aus jeder Menge binärer Daten (Tiff-Bilder), andererseits gibt es auch weitere Informationen, die verwaltet werden

müssen (Tags, Kommentare). Aus diesem Grund scheint es hier sinnvoll zu sein, die Daten auf Dateisystem und Datenbank zu verteilen: Die Bilder selbst verbleiben im Dateisystem, in die Datenbank wandern Daten wie Tags, Kommentare und Pfad (Speicherort im Dateisystem). Das ermöglicht zum einen flexible Abfragen nach Tags und Kommentaren, zum anderen wird die Datenbank nicht mit Binärdaten vollgestopft.

Die Label-Dokumente enthalten keinerlei binäre Daten und können somit vollständig in der Datenbank gespeichert werden. So können sämtliche Attribute eines Label-Dokumentes leicht abgefragt werden.

3.1.5 Verwaltungsdaten

Für die Implementierung der Benutzerverwaltung gibt es drei Möglichkeiten:

- eigene Implementierung
- vorhandene nutzen (zum Beispiel UNIX-Benutzerverwaltung)
- Workflow Management System, sollte eines eingesetzt werden, bringt bereits eigene Benutzerverwaltung mit (siehe Kapitel 4.5)

Im ersten Fall können die Benutzerdaten in der Datenbank gespeichert werden. Sollte der WfMS-Server bereits eine eigene Benutzerverwaltung enthalten, dann ist die Speicherung der Benutzerdaten seine Aufgabe, ebenso beim Einsatz der UNIX-Benutzerverwaltung.

Die Erkennungssysteme ähneln in ihrer Beschaffenheit den Bild-Sequenzen: Es gibt sowohl binäre Daten als auch solche, die sinnvoll in einer Datenbank gespeichert werden können.

Die Testläufe wiederum enthalten keine Binärdaten und können vollständig in der Datenbank verwaltet werden. Das Gleiche gilt auch für die Label-Aufträge.

3.2 Prozesse

Die Prozess-Abläufe können grob in zwei Gruppen eingeteilt werden, mit viel Benutzerinteraktion (Labeln) und mit wenig oder gar keiner Benutzerinteraktion (Testläufe). Benutzerinteraktion erfordert zusätzlichen Aufwand, da einige Aktivitäten eines Prozesses den Benutzer involvieren müssen.

3.2.1 Anforderungen

Bei der Umsetzung der Prozess-Abläufe muss auf Folgendes geachtet werden:

- einfache Definition und Änderung der Prozesse
- Abweichungen während der Ausführung

Die Prozesse dürfen also nicht statisch sein, sondern leicht änderbar und anpassbar. Damit die Änderungen schnell und mit wenig Aufwand vorgenommen werden können, sollten die Prozesse nicht fest in den Code eingebaut (also implizit), sondern von diesem klar getrennt werden. Zusätzlich sollten die Prozesse explizit modelliert werden, zum Beispiel grafisch oder textuell. Damit wäre es möglich, die Prozesse jederzeit zu ändern, ohne den Code anzufassen.

Die Möglichkeit, Prozesse während der Ausführung ändern zu können, ist erwünscht, weil es oft vorkommt, dass irgend ein Schritt ausgelassen, oder ein zusätzlicher hinzugenommen wird. Zum Beispiel könnte unter Zeitdruck die Qualitätskontrolle beim Labeln ausgelassen, oder aber bei besonders kritischen Daten eine weitere Person als Prüfer hinzugenommen werden.

3.2.2 Datenbankmanagementsystem

Eine Möglichkeit Prozesse zu verwalten und auszuführen besteht darin, die Prozess-Instanzen in einer Datenbank-Tabelle zu speichern und zu verwalten. In dieser Tabelle können der Status und der nächste Bearbeiter verzeichnet werden. Mit Hilfe von Triggern und Stored Procedures können sogar viele Aktivitäten automatisch ausgeführt werden.

Diese Lösung bietet sich für das Labeln an, das in Form von Label-Aufträgen abgewickelt werden kann. Dazu legt ein Betreuer einen Label-Auftrag an und bestimmt den Bearbeiter. Neben anderen Daten enthält ein Label-Auftrag auch einen Status. Sobald der Bearbeiter den Label-Auftrag abruft, wird der Status automatisch in “in Bearbeitung” geändert (Trigger). Schickt er das Ergebnis ein, ändert sich der Status in “nicht geprüft” und nach erfolgreich bestandener Qualitätsprüfung in ”Fertig”.

Dieses Verfahren kann zwar recht einfach implementiert werden, erfüllt aber die oben genannten Anforderungen nicht: Der Ablauf ist fest in die Datenbank “eingebaut” und kann nicht so einfach geändert werden.

3.2.3 Workflow Management System

Einen anderen Weg gehen die so genannten “Workflow Management Systeme” (WfMS). Diese bestehen aus einer “Workflow Execution Engine”, welche Prozess-Graphen ausführt, und bieten eventuell weitere Komponenten (wie Benutzerverwaltung) an. Das WfMS bestimmt die Aktivitäten, die als nächste ausgeführt werden können (Arbeitsliste), startet externe Programme, versorgt diese mit Eingabeparametern und leitet der Ausgabeparameter weiter, bestimmt die möglichen Bearbeiter usw.

Die Prozesse werden also explizit in Prozess-Graphen modelliert und können somit jederzeit geändert werden ohne, dass der Code angepasst werden muss. So gesehen stellen die Prozess-Graphen die Anwendungslogik dar und die den Aktivitäten zugeordneten Programme erledigen die eigentliche Arbeit.

Man kann sagen, dass die Workflow Management Systeme für den Workflow das sind, was die Datenbankmanagementsysteme für die Datenverwaltung sind: Statt mühsam immer wieder die gleiche Arbeit zu erledigen wird diese einmal im WfMS-Server gemacht. Danach kann man sich auf die eigentliche Aufgabe, die Erstellung der Prozess-Graphen, konzentrieren.

Damit wäre die erste Anforderung (siehe Kapitel 3.2.1) erfüllt, die zweite kann von einigen Workflow Management Systemen erfüllt werden (wie MQSeriesWorkflow [MqWork], Staffware [Staffware] und ADEPT [ADEPT]), wobei ADEPT bei dieser Anforderung wohl den größten Funktionsumfang bietet.

3.3 Benutzerverwaltung

Es werden viele Benutzer mit dem System arbeiten und dabei verschiedene Aufgaben erledigen müssen, wobei diese Aufgaben auch die Zugriffsrechte und die zur Verfügung stehenden Werkzeuge definieren. Ein Labeler sollte also keine Testläufe verwalten dürfen und das Web-Interface sollte dem Benutzer nur die für ihn relevanten Seiten anzeigen.

Naheliegend erscheint eine Lösung, bei der jedem Benutzer Rollen und Fähigkeiten zugewiesen werden können, die seine Zugriffsrechte und Werkzeuge festlegen. Dabei werden die Fähigkeiten den Rollen untergeordnet, können aber auch einzeln zugewiesen werden. Beispiel: Rolle "Labeler" enthält die Fähigkeit "Labeln". Ein Benutzer, dem diese Rolle zugewiesen wurde, kann für ihn bestimmte Label-Aufträge exportieren und später das Ergebnis importieren.

Neben den Rollen/Fähigkeiten sollen für einen Benutzer noch andere Daten angegeben werden können: Passwort, E-Mail, Betreuer.

3.3.1 Lösungen

Im Kapitel 3.1.5 wurden bereits drei Möglichkeiten erwähnt, um die Benutzerverwaltung zu implementieren. Die einfachste ist die Nutzung einer vom WfMS-Server angebotenen Lösung (falls dieser eine solche mitbringt).

Die schwierigste Lösung dürfte eine eigene Implementierung sein. Für die Speicherung der Benutzerdaten kann auf die Datenbank zurückgegriffen werden, wobei der Zugriff auf die entsprechenden Relationen nur dem System gestattet werden darf. Die Passwörter sollten nicht direkt, sondern als Hash gespeichert werden, damit sie bei einem Einbruch nicht einfach ausgelesen werden können.

Noch eine denkbare Lösung ist die Nutzung der UNIX-Benutzerverwaltung. Diese kennt allerdings nur Benutzer und Gruppen, also keine Rollen und Fähigkeiten und ist somit weniger interessant.

Bei den ersten beiden Lösungen ergibt sich noch ein Problem: falls ein Benutzer Kommandozeilenwerkzeuge oder direkt die API in eigenen Scripten nutzt, weiß das System erst einmal nicht, welcher Benutzer es ist. Entweder müsste sich der Benutzer zuerst authentifizieren, was aber unpraktisch ist, wenn ein Script im

Hintergrund laufen soll. Oder aber das System erfragt den UNIX-Benutzernamen des Benutzers und bildet diesen auf den System-Benutzer ab. Um das Ganze zu vereinfachen, könnte man die beiden Namensräume auch vereinheitlichen, indem grundsätzlich die UNIX-Namen benutzt werden.

3.4 Sicherheit

Da man den Benutzern nicht immer trauen kann, sollte das System auch Zugriffsschutz bieten. Wobei zwischen dem Zugriff auf die Daten und der Nutzung der Schnittstellen (API, Web-Interface) unterschieden werden muss.

3.4.1 Daten

Die Entscheidung, ob ein Zugriff gewährt wird oder nicht, kann anhand der Rollen und Fähigkeiten des Benutzers getroffen werden. Beispiel: Ein Benutzer mit der Rolle “Labeler” bekommt lesenden Zugriff auf alle Label-Aufträge, die für ihn bestimmt sind, ein Benutzer mit der Rolle “LabelJobManager” kann dagegen neue Label-Aufträge anlegen und die von ihm angelegten Aufträge bearbeiten (also schreiben).

Diese Lösung ist zwar einfach, aber recht grob. Man kann zum Beispiel nicht sagen: Dieser Benutzer hat zwar keinen Zugriff auf die Daten der Testläufe, aber den Testlauf “xyz” soll er lesen können. Ein Ausweg wäre eine zusätzliche Rechteverwaltung in Form einer Datenbank-Tabelle, in der zu einem Objekt (eine Tabelle, Datei usw.) Benutzer und deren Zugriffsrechte (lesen, schreiben, ausführen) angegeben werden können. Bei einem Zugriff würde das System neben den Rollen und Fähigkeiten auch diese Tabelle abfragen.

3.4.2 API, Web-Interface

Der Zugriff auf die einzelnen Seiten des Web-Interface kann mit Hilfe der Rollen und Fähigkeiten gesteuert werden. Zum einen kann man nur auf Seiten verlinken, die dem jeweiligen Benutzer zur Verfügung stehen, zum anderen müssen aber auch

bei jedem Seitenaufruf die Zugriffsrechte geprüft werden, da die Adresse (URL) auch direkt vom Benutzer angegeben werden kann.

Bei der Python-API verhält es sich ähnlich: Der Zugriff auf Funktionen kann mit Hilfe der Rollen/Fähigkeiten gesteuert werden, wobei auch feinere Steuerung denkbar wäre. Die Lösung ist dabei die gleiche wie bei den Daten: Zu jeder Funktion kann gespeichert werden, welcher Benutzer diese ausführen kann (zusätzlich zu den Rollen/Fähigkeiten). Zu bedenken ist natürlich, dass das Abfragen der Datenbank-Tabelle mit den Zugriffsrechten bei jedem Aufruf einer Funktion die Performance drastisch senken kann.

Kapitel 4

Technische Umsetzung

Im kapitel 3 wurden allgemeine Fragestellungen diskutiert und mögliche Lösungswege erörtert. In diesem Kapitel werden nun die entsprechenden technischen Möglichkeiten beschrieben, diskutiert und die gewählte und umgesetzte Lösung vorgestellt.

4.1 Architektur

Bevor es weiter gehen kann, muss noch kurz die Architektur des Gesamtsystems vorgestellt werden, da sich aus dieser einige zusätzliche Fragestellungen ergeben, die im Kapitel 2 nicht erwähnt wurden, aber ebenfalls diskutiert werden müssen. Die Begründung für diese Architektur wird an dieser Stelle allerdings noch nicht gegeben (siehe dazu Kapitel 4.2.1).

Die Abbildung 4.1 zeigt einen groben Überblick. Man kann dem Bild entnehmen, dass das System aus insgesamt vier Servern besteht (diese werden in eigenen Kapiteln beschrieben):

- HTTP-Server (Kapitel 4.3)
- DBMS-Server (Kapitel 4.4)
- WfMS-Server (Kapitel 4.5)
- Glue-Server (Kapitel 4.7)

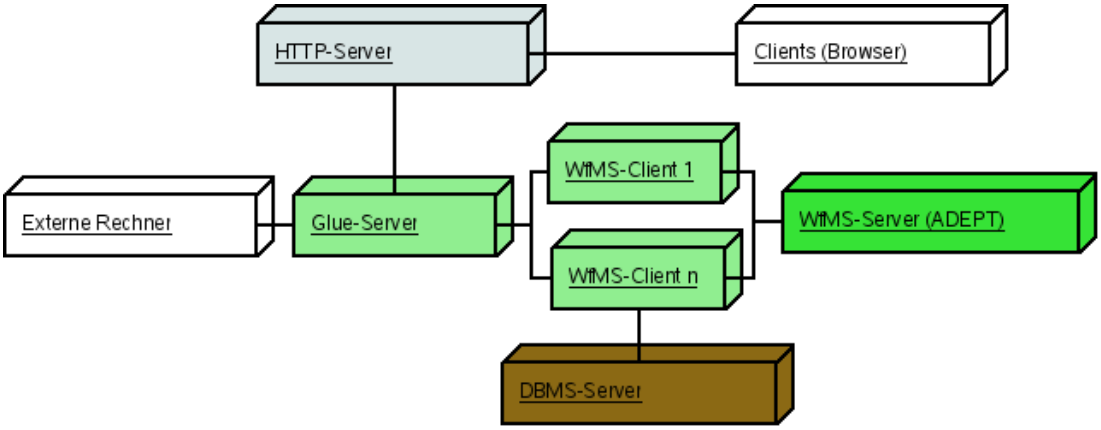

Abbildung 4.1: Architektur des Systems

Wozu die ersten drei Server benötigt werden, sollte klar sein: Der HTTP-Server wird für das Web-Interface benötigt, der DBMS-Server für die Speicherung der Daten und der WfMS-Server kontrolliert den Arbeitsablauf. Da sich die Notwendigkeit für den Glue-Server aus der Kommunikation Python $\longleftrightarrow$ Java ergibt, wird hier auf Kapitel 4.2.1 verwiesen.

Alle vier Server können auf eigenen Maschinen laufen, um so die Last zu verteilen. Eine mögliche Vereinfachung der Architektur wäre die Zusammenlegung des Webservers mit dem Glue-Server. Damit würde die netzwerkbasierte Kommunikation zwischen den beiden entfallen, was der Performance zugute kommen würde. Dabei könnte auch der Apache HTTP-Server entfallen, da Python schon einen einfachen HTTP-Server bietet. Aber zumindest eine Trennung in zwei Prozesse macht Sinn, damit ein Absturz des Webservers nicht auch den Rest mit sich zieht.

4.2 Kommunikation

Eine der zentralen Komponenten des Systems ist der Workflow-Server ADEPT 1 (siehe Kapitel 4.5). Dieser ist in Java geschrieben und bietet nur eine Java-API an. Da der Rest des Systems aber in Python geschrieben ist, stellt sich die Frage, ob die WfMS-Clients in Java oder in Python geschrieben werden sollen. Da Python bevorzugt eingesetzt werden soll, wurde nach Möglichkeiten gesucht, die Clients in Python schreiben zu können.

Ein in Python geschriebener WfMS-Client müsste auf die Java-API von ADEPT zugreifen können. Diese wiederum kommuniziert über RMI (Remote Method Invocation [RMI]) mit dem ADEPT-Server. Leider existiert keine einfache Methode, eine Java-API in Python zu benutzen, da sowohl Python- als auch Java-Anwendungen in eigenen virtuellen Maschinen und somit in getrennten Adressräumen laufen. JNI (Java Native interface [JNI]) schafft hier auch keine Abhilfe, da man damit zwar eine Bibliothek schreiben kann, die von einer Java-Anwendung benutzt werden kann, es bleibt aber die Frage wie diese Bibliothek mit einer Python-Anwendung kommunizieren soll.

Mit erheblich mehr Aufwand sind zwei mögliche Lösungen für dieses Problem verbunden:

- Jython ([Jithon])
- JPype ([JPype])

Jython ist eine Python-Implementierung in Java und hat somit auch Zugriff auf die Java-Bibliotheken. Jython implementiert den Sprachumfang von Python 2.1 und bietet auch Teile der Standardbibliothek von Python 2.1. Allerdings ist Jython deutlich langsamer als das normale Python (CPython) und kann keine CPython-Bibliotheken nutzen. Außerdem würde sich beim Einsatz von Jython die Frage stellen, wie normale Python-Anwendungen mit Jython-Anwendungen kommunizieren sollen. Man könnte also die WfMS-Clients in Jython schreiben, das Problem der Kommunikation wäre dann aber immer noch nicht gelöst.

JPype bietet einen anderen Ansatz als Jython: Es wird das normale CPython benutzt, JPype bietet aber die Möglichkeit, eine JVM zu starten und im Python-Programm die Java-Bibliotheken zu nutzen. Dies sieht dann so aus (aus [JPypeEx]):

Listing 4.1: Beispiel für JPype

```
from jpype import *
startJVM("d:/tools/j2sdk/jre/bin/client/jvm.dll", "-ea")
java.lang.System.out.println("hello world")
shutdownJVM()
```

Die Kommunikation mit der JVM erledigt JPype über JNI. Man kann also mit CPython arbeiten und trotzdem auf Java-Bibliotheken zugreifen. Diesen Vorteil erkauft man allerdings mit schlechterer Performance, da Konvertierungen der Daten von Java nach Python bzw. umgekehrt notwendig sind und JNI zusätzlichen Aufwand verursacht.

Da die beiden vorgestellten Lösungen mit Nachteilen behaftet sind, wurde entschieden, die WfMS-Clients in Java zu schreiben und sie mit dem Glue-Server kommunizieren zu lassen. Der Vorteil ist offensichtlich: Die Clients werden ganz normal in Java geschrieben, ohne, dass irgendwelche mehr oder weniger ausgereiften Lösungen benötigt werden, die Python und Java verbinden. Allerdings laufen die Clients bei dieser Lösung als eigenständige Prozesse, so dass ein Weg gefunden werden muss, diese mit dem Rest des Systems kommunizieren zu lassen.

Für die Kommunikation zwischen Prozessen gibt es grundsätzlich zwei Möglichkeiten:

- IPC - Inter-Prozess Communication
- netzwerkbasierte Kommunikation

Die netzwerkbasierte Kommunikation hat den Vorteil, dass sie auch zwischen einzelnen Rechnern funktioniert, was aber zusätzlichen Aufwand verursacht und so die Geschwindigkeit reduziert. Dagegen verzichtet IPC auf die Fähigkeit zwischen einzelnen Rechnern zu vermitteln und kann so sehr hohe Leistung bieten. Da der Glue-Server und die WfMS-Clients auf der gleichen Maschine laufen können und IPC Performance-Vorteile verspricht, fiel die Wahl auf IPC.

4.2.1 UNIX Message Queues

In der UNIX-Welt gibt es mehrere Lösungen für Inter-Prozess Communication:

- Pipes
- Named Pipes
- Shared Memory

- UNIX Message queues

Pipes (siehe [Hero99, Seite 717]) sind in diesem Kontext schwierig zu benutzen, da hier der Vater-Prozess vor dem *fork*[1]-Aufruf die Pipe einrichten muss und die beiden Seiten virtuelle Maschinen einsetzen. Wenn also ein Python-Script eine Pipe einrichtet und dann mit *fork* und anschließendem *exec*[2]-Aufruf eine Java-Anwendung startet dürfte es für letztere schwierig werden auf die Pipe zuzugreifen.

Named Pipes (siehe [Hero99, Seite 744]) wären einfacher als Pipes zu benutzen, da hier nur der Name der Datei für die Named Pipe bekannt sein muss. Allerdings müsste man für jeden WfMS-Client eine Named Pipe einrichten und die Implementierung eines Kommunikationsprotokolls mit Pipes oder Named Pipes ist nicht gerade einfach (Datei lesen/schreiben Semantik, Parsen des Datenstroms).

Shared Memory (siehe [Hero99, Seite 780]) würde wohl die beste Performance bieten, da die Kommunikation direkt über den Arbeitsspeicher abläuft. Hier trifft man aber auf das gleiche Problem wie bei den Pipes: Beide Kommunikationspartner laufen in eigenen virtuellen Maschinen, woraus folgt, dass Shared Memory nicht so einfach benutzt werden kann.

Bleiben noch die UNIX Message Queues (siehe [Hero99, Seite 756]), welche für Nachrichtenaustausch zwischen Prozessen benutzt werden können und auf hohen Durchsatz ausgelegt sind. UNIX Message Queues werden von den meisten UNIX-Systemen angeboten. Die Schnittstelle ist überschaubar (es sind 4 Systemaufrufe) und die Funktionsweise einfach: Eine Message Queue ist eine Art verkettete Liste, an die neue Einträge angehängt und aus der vorhandene Einträge entnommen werden können. Auch ist es möglich einen Empfänger zu adressieren, indem man eine Integerzahl beim Versand und Empfang angibt. Beim Abruf der Nachrichten wird immer die älteste Nachricht mit der angegebenen Adresse zurückgegeben. Es ist nicht nötig, eine Verbindung zwischen den Prozessen aufzubauen, und Versand und Empfang können sowohl synchron (blockierend) als auch asynchron geschehen. Wegen dieser Vorteile werden die Message Queues für die Kommunikation benutzt.

[1] UNIX Systemaufruf *fork*, siehe [Hero99, Seite 486].
[2] UNIX Systemaufruf *exec*, siehe [Hero99, Seite 520].

Damit die WfMS-Clients mit dem Rest des Systems kommunizieren können, wird eine Weitere Komponente benötigt, welche über UNIX Message Queues mit den Clients und über ein netzwerktransparentes Protokoll mit den anderen Komponenten kommuniziert. Diese Komponente ist der Glue-Server (Kapitel 4.7).

Da weder Python noch Java Message Queues unterstützen, wurde eine eigene Bibliothek mit C++ geschrieben, die mit Hilfe der Systemaufrufe *msgget*, *msgsnd*, *msgrcv* und *msgctl* (in "sys/msg.h") eine Kommunikationsschicht zwischen Glue-Server und WfMS-Clinets bildet. Dabei wurde darauf geachtet, dass diese Bibliothek leicht durch eine andere ersetzt werden kann, falls man sich in Zukunft für ein anderes Verfahren entscheiden sollte. Der Code liegt im Verzeichnis "GlueServer/messaging".

Die Schnittstelle wird in der abstrakten Klasse "AbstractChannel" definiert (in "abstract_channel.h"). Sie besteht nur aus drei Methoden:

Listing 4.2: Methode "deleteChannel"

```
bool deleteChannel(void) - Ressourcen wieder freigeben,
```

Listing 4.3: Methode "sendMessage"

```
bool sendMessage(char* data, long msgType, bool block)
    Nachricht versenden
    data - Daten, die versendet werden sollen
    msgType - die Adresse
    block - blockieren/nicht blockieren
    Rückgabe: true bei Erfolg, false sonst
```

Listing 4.4: Methode "receiveMessage"

```
char* receiveMessage(long msgType, bool block)
    Nachricht abholen
    msgType - die Adresse
    block - blockieren/nicht blockieren
    Rückgabe: Zeiger auf die Daten, NULL sonst
```

Jede Implementierung, die diese Schnittstelle implementiert, sollte problemlos benutzt werden können.

Die vorhandene Implementierung (Klasse “MessageQueueChannel”) liegt in den Dateien “mque_channel.h” und “mque_channel.cpp”, die zu “mque_channel.o” kompiliert werden (dazu wird SCons benutzt). Um die Bibliothek in Python und Java benutzen zu können, wird zusätzlich mit SWIG je ein Interface erzeugt:

- “GlueServer/messaging/python/_mque_channel.so” für Python
- “GlueServer/messaging/java/mque_channel.so” für Java

Listing 4.5 zeigt, wie das “mque_channel”-Modul in einem Python-Script importiert werden kann.

Listing 4.5: Import des “mque_channel”-Modules

```
from messaging.python.mque_channel import *
```

Damit die WfMS-Clients die Bibliothek nutzen können, genügt es den Inhalt des “GlueServer/messaging/java” Verzeichnises in das Verzeichnis “client” zu kopieren.

Eine Instanz der Klasse “MessageQueueChannel” kann in genau eine Message Queue schreiben bzw. lesen (siehe Abbildung 4.2). Der Konstruktor bekommt als Parameter einen Message Queue Schlüssel, der eine Message Queue identifiziert, maximale Größe einer Nachricht und ein Verzeichnis für die Nachrichten (siehe unten).

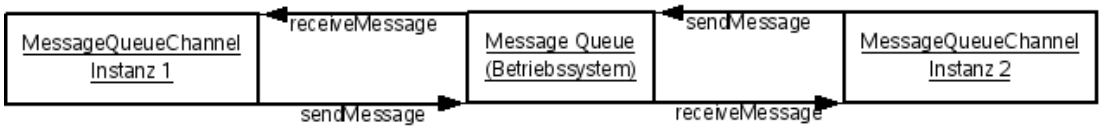

Abbildung 4.2: Kommunikation über eine Message Queue

Das Format der Nachrichten wurde folgendermaßen festgelegt: Die ersten 8 Bytes sind für den Header reserviert, danach kommt die eigentliche Nachricht. Der Header besteht aus einer achtstelligen Hexadezimalzahl (Buchstaben A-F groß geschrieben), die einen fortlaufenden Zähler darstellt. Mit dieser Zahl kann die Reihenfolge der Nachrichten bestimmt werden.

Da eine Message Queue nicht beliebig viele Daten aufnehmen kann, werden größere Nachrichten nicht direkt in die Message Queue geschrieben. Stattdessen wird

der Inhalt der Nachricht in eine Datei geschrieben und nur der Header versendet. Vorher wird allerdings noch das oberste Bit im Header gesetzt, damit der Empfänger merkt, dass die Nachricht in einer Datei liegt. Der zweite Parameter des Konstruktors legt fest, ab welcher Größe eine Nachricht nicht direkt versendet wird.

Der dritte Parameter des Konstruktors bestimmt, in welchem Verzeichnis die Dateien mit dem Inhalt der Nachrichten abgelegt werden. Der Name dieser Dateien hat folgenden Aufbau:

```
Schlüssel_Adresse_Zähler.msg
Beispiel: 10000_1_10.msg
```

Der Empfänger kann also leicht herausfinden, welche Datei er lesen muss, da der Zähler im Header steht. Nachdem der Empfänger die Datei gelesen hat, löscht er diese.

Für die Kommunikation mit den WfMS-Clients richtet der Glue-Server drei Message Queues ein:

- eine für die Richtung Glue-Server → Client
- eine für die Richtung Client → Glue-Server
- eine für unerwartete Nachrichten vom Client (z.B. wenn ein unerwarteter Fehler auftritt)

Zusätzlich werden drei weitere Message Queues eingerichtet, die dazu dienen, mit einem speziellen WfMS-Client (ADEPTManager, zuständig für die Benutzerverwaltung, siehe Kapitel 4.6) zu kommunizieren. Beim Beenden des Glue-Servers werden sämtliche Message Queues wieder freigegeben (nicht abgeholte Nachrichten gehen verloren!).

Vor dem Starten eines neuen WfMS-Clients erzeugt der Glue-Server eine Adresse für diesen (es wird einfach eine Integer-Zahl hochgezählt).

Die Parameter (Message Queue Schlüssel, maximale Nachrichtengröße und das Verzeichnis für die Nachrichten) können in der Konfigurationsdatei “glue_server.cfg“ eingestellt werden.

Die Kommunikation über Message Queues hat bis jetzt gut funktioniert. In Zukunft sollte aber die Fehlerbehandlung ausgebaut werden, da die Nachrichten solange in der Warteschlange verbleiben, bis sie entweder abgeholt werden oder die Warteschlange gelöscht wird. Nach einem Absturz und einem erneutem Start kann es deswegen zu Inkonsistenzen kommen.

4.2.2 XML-RPC

Nachdem begründet wurde, warum der Glue-Server benötigt wird, soll in diesem Kapitel diskutiert werden, wie dieser mit dem HTTP-Server kommuniziert (genauer: wie dieser mit den Python-Scripten kommuniziert, die im Kontext des HTTP-Servers laufen, siehe Kapitel 4.3).

Zuerst muss aber noch eine Design-Frage geklärt werden: Sollen der HTTP-Server und der Glue-Server auf getrennten Maschinen laufen können oder sollen sie zusammengelegt werden? Im letzteren Fall wäre das System-Design einfacher, da die Python-Scripte im Kontext des HTTP-Servers direkt über Message Queues mit den WfMS-Clients kommunizieren und so den Glue-Server überflüssig machen würden. Dagegen muss beim Verteilen der beiden Server zusätzlicher Aufwand bei der Kommunikation betrieben werden.

An dieser Stelle soll noch etwas vorweg genommen werden: der Glue-Server dient nicht nur der Kommunikation mit den WfMS-Clients, sondern verwaltet diese auch (Starten, Beenden). Außerdem bildet er die Schnittstelle für die Python-API (die ja auch über Rechner-Grenzen hinweg funktionieren muss).

Aus den oben genannten Gründen fiel die Entscheidung zugunsten der Verteilung der beiden Komponenten. Es bleibt also noch zu klären, wie die Kommunikation ablaufen soll.

Wegen der Verteilung kommen nur Kommunikations-Mechanismen in Frage, die netzwerktransparent sind. Hier stehen einige Möglichkeiten zur Verfügung: von low-level Mechanismen, wie Sockets, bis hin zu high-level Mechanismen, wie RPC ([RPC]) oder SOAP ([SOAP]). Da die Entwicklung mit low-level Protokollen aufwendig und fehlerträchtig ist, wurden nur high-level Protokolle betrachtet. Großer Wert wurde dabei auf gute Python-Integration gelegt. Schwergewichte wie SOAP

wurden weggelassen, da sie für diese Aufgabe überdimensioniert scheinen. In die engere Wahl fielen so nur Twisted Spread ([twissp]) und XML-RPC ([XMLRPC]).

Twisted ist eine Python-Bibliothek für Netzwerkprogrammierung und bietet mit Twisted Spread auch einen RPC-Mechanismus. Beim Einsatz von Twisted Spread gab es jedoch einige Probleme. So klappte die Verbindung zum Glue-Server oft nicht, wenn man das Web-Interface nutzte, wobei die Ursache des Problems selbst nach langer Suche nicht gefunden werden konnte (es hat vermutlich mit Multithreading zu tun, von dem Twisted massiven Gebrauch macht). Dazu kommt noch eine merkwürdige Einschränkung von Twisted Spread: Es kann maximal 600KByte große Daten (Strings) auf einmal übertragen. Es ist zwar möglich, diese Einschränkung zu umgehen, die Lösung ist aber recht aufwendig und nicht transparent (neben dem Server muss auch der Client angepasst werden).

Als weitaus unproblematischer erwies sich die XML-RPC Bibliothek, die mit Python mitgeliefert wird. XML-RPC nutzt HTTP als Protokoll, wobei Python dafür auch schon einen einfachen Server mitliefert ([PyHTTP]). Die Daten werden in XML verpackt und verschickt. Unterstützt werden sämtliche Datentypen von Python: Zahlen, Strings, Listen, aber auch Datum und binäre Daten. Instanzen von Klassen (außer date) können leider nicht verschickt werden, was aber keine große Einschränkung ist, da die Schnittstelle einfach gehalten ist.

Ein Nachteil der XML-RPC Implementierung von Python ist die Tatsache, dass es nicht multithreaded ist und beim Verarbeiten einer Anfrage der Server keine weiteren Anfragen annimmt. Es existiert allerdings eine einfache Lösung für dieses Problem: Statt der Klasse SimpleXMLRPCServer (die einen einfachen XML-RPC Server implementiert) wird eine abgeleitete Klasse benutzt (siehe Listing 4.6, aus [ThXMLRPC]).

Listing 4.6: Klasse AsyncXMLRPCServer

```
class AsyncXMLRPCServer(SocketServer.ThreadingMixIn,
                        SimpleXMLRPCServer):
    pass
```

Damit wird erreicht, dass der Server jede Anfrage im eigenen Thread abarbeitet. In [ThXMLRPC] wird allerdings behauptet:
"This implementation will probably have some scalability issues and it can't compete with the feature-set which for example Twisted has."

Das heißt, diese Lösung skaliert vermutlich nicht gut, was aber erst zum Problem wird, wenn die Last groß wird, was aber zur Zeit nicht zu erwarten ist.

4.2.3 Json

Bei so vielen Kommunikationswegen stellt sich natürlich die Frage, in welchem Format man die Daten austauscht. Neben Python-Scripten und Java-Clients gibt es beim Web-Interface noch JavaScript ([JavaSc]), welches mit dem HTTP-Server kommunizieren kann (z.B. um Daten anzufordern).

Im Zusammenhang mit JavaScript und Ajax ([AJAX]) fällt oft der Begriff JSON ([Json]), welcher für "JavaScript Object Notation" steht und ein Datenaustauschformat beschreibt. Es dient dem selben Zweck wie XML, ist aber viel einfacher aufgebaut und kann so schneller geschrieben bzw. gelesen werden. Wie man dem Namen entnehmen kann, basiert es auf der JavaScript-Syntax und kann somit direkt von JavaScript verarbeitet werden. Es existieren auch viele Implementierungen für viele Sprachen, für Python ist vor allem CJson ([CJson]) empfehlenswert, für Java Json-Lib ([JsonLib]), welches auch das alte JDK 1.3.1 unterstützt.

Für die Kommunikation mit dem Glue-Server wird grundsätzlich Json als Datenformat benutzt, wobei dies von den Clients versteckt wird (siehe Kapitel 4.9.2).

4.2.4 SSH

Die Testläufe werden auf anderen Maschinen im Netzwerk ausgeführt, deswegen braucht das System eine Möglichkeit, auf diese Maschinen zuzugreifen. Eine denkbare Lösung ist ein Dienst, der auf diesen Rechnern läuft und auf Aufträge wartet. So ein Dienst müsste aber auf jedem einzelnen Rechner eingerichtet werden und würde darüber hinaus ein zusätzliches Sicherheitsrisiko darstellen.

Besser wäre eine Lösung, die ohne zusätzliche Dienste funktioniert. Und so einen Dienst gibt es tatsächlich auf allen Rechnern im Netzwerk, die für die Testläufe zur Verfügung stehen: SSH - Secure Shell ([OpenSSH]).

Mit SSH kann man sich mit seinem Benutzernamen und Passwort auf einem Rechner einloggen. Da aber das System schlecht ein Passwort in einer Shell eingeben kann, wird ein Weg benötigt, sich ohne Passworteingabe anmelden zu

können. Auch dafür gibt es eine Lösung: der Public Key. Mit dem Kommando "ssh-keygen" können zwei Schlüssel generiert werden, ein privater Schlüssel und ein öffentlicher Schlüssel. Dies muss mit dem Benutzer gemacht werden, unter dem das System später laufen soll. Anschließend muss der öffentliche Schlüssel in die Datei ".ssh/authorized_keys" im Home-Verzeichnis des Benutzers, der die Testläufe ausführen wird, eingefügt werden. Dank NFS-gemounteter Home- Verzeichnisse muss dies nur einmal gemacht werden. Jetzt kann das System mit

```
scp FILE USER@MACHINE/DIR
```

die Datei "FILE" auf die Maschine mit dem Namen "MACHINE" in das Verzeichnis "DIR" kopieren. Mit

```
ssh USER@MACHINE /home/USER/EXE
```

kann die ausführbare Datei "EXE" auf der Maschine "MACHINE" ausgeführt werden.

Damit gibt es einen Weg, die Testläufe auf andere Maschinen zu kopieren und zu starten.

4.3 Web-Server

Im Kapitel 2.4.1 wurde bereits erwähnt, dass das Benutzerinterface Web-basiert sein soll. Somit ist es notwendig, einen HTTP-Server für die Generierung und Auslieferung der Web-Seiten aufzusetzen.

4.3.1 Apache HTTP-Server

Da sämtliche Komponenten des Systems unter Linux laufen müssen, kommen alle HTTP-Server in Betracht, die unter Linux laufen. Neben dem wohl am meisten benutzten HTTP-Server Apache, gibt es noch andere, kleinere HTTP-Server:

- lighttpd ([LightHTTP])
- LiteSpeed ([LiteSp])

- CGIHTTPServer von Python ([PyHTTP])

Wegen der großen Verbreitung und der Möglichkeit, den Python-Interpreter in den Server zu integrieren, wird der Apache HTTP-Server benutzt.

4.3.2 mod_python

Von zentraler Bedeutung für ein Web-Interface ist die Fähigkeit, dynamisch Web-Seiten zu generieren, da ein Benutzerinterface im Allgemeinen nicht statisch ist. Dazu gibt es heutzutage im wesentlichen drei Möglichkeiten:

- CGI ([CGI])
- Fast-CGI ([FastCGI])
- in den Server integrierter Interpreter einer Scriptsprache

CGI ist recht langsam und verschwendet viel Arbeitsspeicher, da bei jedem Seiten-Request der Interpreter geladen wird. Fast-CGI bietet eine bessere Performance, ist aber schwieriger aufzusetzen und die vorhandenen Lösungen scheinen noch nicht sehr ausgereift zu sein. Eine andere interessante Methode besteht darin, den Interpreter der Scriptsprache in den HTTP-Server einzubauen. Damit entfällt das Laden des Interpreters bei einem Request und der Durchsatz steigt im Vergleich zu CGI. Ein weiterer Vorteil dieser Lösung: Es ist einfach aufzusetzen (beim Apache HTTP-Server muss ein Modul kompiliert und in der Konfigurationsdatei aktiviert werden).

Für den Apache HTTP-Server existieren einige Module für verbreitete Scriptsprachen, darunter Python: “mod_python” ([modpy]). Es ist ausgereift, gut dokumentiert und bietet einige weitere Funktionen wie Session-Menagement ([PySession]) oder PSP - Python Server Pages ([PSP]). Python-Scripte, die von mod_python ausgeführt werden, können sowohl auf die gesamte Standardbibliothek von Python als auch auf die Internas des Apache HTTP-Servers zugreifen.

4.3.3 Integration

Beim Einsatz von mod_python werden statt HTML-Seiten Python-Scripte exportiert, die bei entsprechender Konfiguration von Apache an mod_python übergeben und von diesem verarbeitet werden. Die Ausgabe dieser Scripte wird dann an den Client übermittelt.

Für das Web-Interface wurde auf dem HTTP-Server das Verzeichnis "htdocs/wfms"[3] eingerichtet, welches die Python-Scripte enthält. Im Browser kann die Angabe der Endung ".py" entfallen:

```
http://localhost:8000/wfms/login.py
oder einfach
http://localhost:8000/wfms/login
```

Die Python-Scripte kommunizieren mit dem Glue-Server über XML-RPC, um zum Beispiel die benötigten Daten anzufordern.

4.3.4 Session Management

Um die Clients zu identifizieren, wurde auf das Session Management von mod_python zurückgegriffen. Dieses benutzt Cookies ([Cookie]) um einen Browser zu identifizieren. In diesen Cookies wiederum können weitere Daten, wie zum Beispiel der Benutzername, hinterlegt werden. Dieser Benutzername wird von den Python-Scripten benutzt um festzustellen, ob ein Benutzer Zugriff auf eine Seite hat (siehe Kapitel 4.8.2). Eine Sitzung ist in der Standardeinstellung 30 Minuten lang gültig.

4.3.5 Python Server Pages

Python Server Pages bieten eine Möglichkeit, Python-Code in HTML-Daten unterzubringen, welcher vor der Auslieferung ausgeführt wird, um die Seite mit Inhalt zu füllen. Dabei steht dem Entwickler der gesamte Sprachumfang von

[3] htdocs ist das Verzeichnis, das der HTTP-Server Apache standardmässig exportiert.

Python zur Verfügung (Schleifen, Bedingungen usw.). Darüber hinaus können Python-Bibliotheken benutzt werden.

Die Dateien, die solch eine Mischung aus Python und HTML enthalten, haben meist die Endung ".psp" und können, bei geeigneter Server-Konfiguration, direkt im Browser aufgerufen werden. Dies wurde allerdings nicht benutzt, da es besser ist, die PSP-Dateien als Vorlagen (Templates) zu benutzen. Solche Vorlagen können in Python-Scripten "ausgeführt" werden, wobei Daten als Parameter übergeben werden können. Das Ergebnis (HTML-Seite) wird dann an den Client geschickt. Die PSP-Dateien liegen im Verzeichnis "htdocs/wfms".

4.4 DBMS-Server

Im Kapitel 3.1.3 wurde beschlossen, ein relationales Datenbankmanagementsystem (DBMS) für die Verwaltung der Daten zu benutzen. Es bleibt noch zu klären, welches von den vielen existierenden tatsächlich benutzt werden soll.

Der zu verwaltende Datenbestand hat zwar bereits eine beachtliche Größe erreicht (siehe 2.1.1 und 2.1.2), ist aber immer noch nicht so groß (und wird es wahrscheinlich nicht werden), dass ein großes DBMS wie Oracle oder DB2 nötig wäre. Die Wahl kann also auf kleinere DBMS-Systeme beschränkt werden, die unter Linux laufen und keine hohen Kosten verursachen. Hier bieten sich vor allem die freien DBMS-Systeme wie Firebird ([Firebird]), MySQL ([MySQL]) oder PostgreSQL ([Postgre]) an. Von diesen drei wurde PostgreSQL gewählt, da es einen beachtlichen Leistungsumfang ([PostAb]) hat (deckt zum Beispiel die SQL-Standards besser ab als MySQL), schnell ist, geometrische Datentypen und dazu passende Operatoren und Funktionen bietet (siehe [PostGeo] und [PostGeoFunk]) und einfach in der Handhabung ist.

Für den Zugriff auf SQL-Datenbanken von Python aus existiert ein Standard "Python Database API Specification v2.0" ([PyDB]). Diesen zu benutzen bringt den Vorteil, dass man später mit wenig Aufwand auf eine andere Bibliothek, die diesen Standard unterstützt, umsteigen kann. Aus diesem Grund wurden nur die PostgreSQL-Schnittstellen in Betracht gezogen, die diesen Standard implementieren. Und davon gibt es einige: PyGreSQL ([PyGre]), PyPgSQL ([PyPg]), psycopg ([Psyco]). Von diesen drei erschien psycopg (in Version 2) am ausge-

reiftesten: vollständige Unterstützung der "Python Database API Specification v2.0", ist Thread sicher (Level 2) und auf multithreaded Anwendungen ausgelegt.

Für den Zugriff auf die Datenbank wird ein Benutzer verwendet, der vollen Zugriff auf die Daten hat. Die Zugriffskontrolle muss also weiter oben ansetzen (zum Beispiel bei Funktionen wie "lösche die Bild-Sequenz xyz").

4.4.1 Performance

Bei dem bisherigen Datenbestand war die Performance bei den meisten Anfragen sehr gut, allerdings benötigen einige (komplexere Anfragen) doch lange Laufzeit. Deswegen sollen hier die Möglichkeiten für die Optimierung diskutiert werden.

Eine einfache Möglichkeit, die Geschwindigkeit zu steigern, sind Indexe ([ElNa02, Seite 186]) auf oft abgefragte Attribute in großen Relationen. Um zu sehen, wie viel ein Index bringen kann wurde ein Test durchgeführt. Listing 4.7 zeigt die SQL-Query, die für diesen Test benutzt wurde.

Listing 4.7: SQL-Query für die Performancemessung

```
SELECT  *
FROM    label AS l
WHERE   l.track_id = 'Car_1'
```

Zum Zeitpunkt des Tests befanden sich insgesamt 930679 Labels in der Datenbank, von denen 58846 die WHERE-Clausel der Anfrage erfüllten. Getestet wurde drei Mal ohne Index auf track_id und drei Mal mit Index. Ausgeführt wurde der Test in einer virtuellen Maschine (VMWare Server 1.0.2 [vmware]) mit Gast-System SuSE Linux 9.3 mit 768MB RAM, PostgreSQL 8.1.5 (Host-Rechner: 2.1 GHz AthlonXP, 2GB RAM, OpenSuSE 10.2 [openSUSE]). Mit folgender Kommandozeile wurde der Test ausgeführt:

```
$ time psql wfms -f seq3.sql > /dev/null
```

psql ist der PostgreSQL-Client, wfms der Name der Datenbank und mit "-f seq3.sql" wurde die Datei mit der Anfrage angegeben. Die Tabelle 4.1 fasst die Laufzeiten der drei Läufe ohne Index zusammen (in Sekunden).

Lauf	real	user	sys	gesamt
1	2.840	0.566	0.334	3.740
2	2.818	0.566	0.329	3.713
3	2.821	0.570	0.308	3.699
Gesamt	11.152			

Tabelle 4.1: Laufzeiten ohne Index.

Lauf	real	user	sys	gesamt
1	1.486	0.598	0.336	2.420
2	1.460	0.546	0.357	2.363
3	1.448	0.575	0.423	2.446
Gesamt	7.229			

Tabelle 4.2: Laufzeiten mit Index.

Anschließend wurde Index auf das Attribut "track_id" angelegt und der gleiche Test drei Mal durchgeführt (Tabelle 4.2).

Die Differenz der beiden Gesamtlaufzeiten beträgt 3.923 Sekunden, also bringt der Index in diesem Beispiel 35% kürzere Laufzeit. Das ist zwar keine sehr große Steigerung, aber bei komplizierteren Abfragen, bei denen mehrere Attribute mit Indexen abgefragt werden, können sich solche Steigerungen aufsummieren.

Auch bei einigen anderen Relationen sind Indexe auf bestimmte Attribute denkbar:

- frame(name)
- label_document(name)
- sequence(name)
- sequence_channel(name)

Eine andere Möglichkeit die Geschwindigkeit zu steigern ist die Optimierung der Server-Konfiguration. Zu diesem Zweck kann die Datei "postgresql.conf" (meistens im Datenbank-Verzeichnis) editiert werden. Vor allem die beiden Parameter

“shared_buffers” und “max_connections” sind in diesem Zusammenhang interessant. Mit dem ersten kann PostgreSQL angewiesen werden, mehr Speicher zu benutzen, und mit dem zweiten kann verhindert werden, dass zuviele gleichzeitige Verbindungen den Server zu sehr ausbremsen. Im Rahmen dieser Arbeit konnte aber keine signifikante Wirkung auf die Performance festgestellt werden.

4.5 WfMS-Server

Die Entwicklung eines eigenen Workflow Managemen Systems würde den Rahmen dieser Arbeit deutlich sprengen. Außerdem ist es nich sinnvoll, da es schon viele solche Systeme gibt, die zudem ausgereift und im Einsatz erprobt sind. Es sollte also ein solches System gefunden und integriert werden.

Das “richtige” Workflow Management System für eine bestimmte Problemstellung zu finden ist nicht einfach, da es inzwischen viele Systeme mit teilweise sehr unterschiedlichen Ansätzen gibt. Die einfachsten Systeme arbeiten formularbasiert und eignen sich eher für kleinere und einfachere Aufgaben. Ein Beispiel für so ein System ist Lotus Notes ([Lotus]).

Ähnlich funktionieren die dokumentenzentrierten Workflow-Systeme, bei denen eine Art “Umlaufmappe” von einem Bearbeiter zum nächsten weitergereicht wird. Solche Systeme eignen sich for allem für die Automatisierung der Arbeitsabläufe in einem Büro. Vertreter dieser Gattung sind zum Beispiel ProMInanD ([promin]) und Floware ([Floware]).

Leistungsfähiger und flexibler sind die so genannten “Production Workflows”, die den Anwendungscode von der Prozesslogik trennen. Typische Vertreter dieser Art von Workflow-Systemen sind MQ Workflow von IBM ([MqWork]) und Staffware ([Staffware]). Auch das an der Universität Ulm entwickelte ADEPT fällt in diese Kategorie.

ADEPT liegt eine eigene Notation zur Beschreibung von Prozess-Abläufen und Datenflüssen zugrunde, die bedingte und unbedingte Verzweigungen und Schleifen unterstützt. Auch zeitliche Aspekte werden unterstützt: Für jede Aktivität können minimale und maximale Dauer sowie zeitliche Zusammenhänge zwischen einzelnen Aktivitäten angegeben werden (zum Beispiel minimale und maximale Abstände). Zur Zeit wohl einmalig (zumindest was den Leistungsumfang angeht)

ist die Fähigkeit von ADEPT, während der Ausführung einer Prozess-Instanz dynamisch vom vorgegebenen Prozess-Ablauf abzuweichen.

Von ADEPT existiert zur Zeit ein Prototyp (ADEPT 1), welcher allerdings bereits produktiv eingesetzt wird (siehe [ADEPTHos]). ADEPT 2 befindet sich in Entwicklung und soll in Zukunft die Basis von AristaFlow ([Arista]) bilden. Damit ist sichergestellt, dass ADEPT weiter entwickelt wird. Aus diesen Gründen wurde ADEPT 1 für die Workflow-Komponente des Systems benutzt.

4.5.1 ADEPT

Neben dem Server, der die Prozess-Instanzen ausführt, bietet ADEPT noch einige weitere Werkzeuge an:

- ADEPTClient
- ADEPTEditor
- ADEPTOrgManager

Mit dem ADEPTClient können Prozess-Instanzen erstellt und ausgeführt werden. ADEPTEditor dient der Erstellung der Prozess-Vorlagen und mit dem ADEPTOrgManager können Organisationsstrukturen, Rollen und Benutzer verwaltet werden (siehe dazu Kapitel 5.1). Leider laufen diese Anwendungen trotz Java nur unter Windows fehlerfrei, so dass eine Windows-Maschine benötigt wird, um Prozess-Vorlagen zu erstellen oder Benutzer zu verwalten. Der ADEPT-Server läuft dagegen auch unter Linux, in diesem Fall klappt allerdings die Verbindung von Windows-Client zum Server nicht. Als “work-around” kann auf dem Windows-Rechner ein weiterer ADEPT-Server gestartet werden, der sich zum gleichen Oracle-Server verbindet. Es wäre aber wünschenswert, wenn die ADEPT-Werkzeuge in Zukunft auch unter Linux/UNIX laufen würden.

Ein Nachteil von ADEPT ist die Tatsache, dass es nur mit dem Oracle Datenbankserver ([Oracle]) zusammenarbeiten kann (diesen nutzt ADEPT um Benutzerdaten, Prozess-Vorlagen usw. zu speichern). Da aber die kostenlose (auch für kommerzielle Nutzung) Express Edition ([OracXE]) von Oracle ausreicht, entstehen keine zusätzlichen Kosten.

Da auf den Oracle Datenbankserver nicht verzichtet werden kann, stellt sich natürlich die Frage, ob nicht der PostgreSQL-Server weggelassen werden kann. Der Funktionsumfang von Oracle Express Edition würde dem nicht im Wege stehen (sie bietet auch die Möglichkeit, geometrische Daten zu speichern). Problematisch sind eher die Einschränkungen dieser Oracle Version:

- Maximal 4 GB an Daten
- Nur 1 GB Ram
- Nur 1 Prozessor

Vor allem die Beschränkung auf 4GB Daten könnte in Zukunft problematisch werden und den Kauf einer vollständigen Oracle Version erzwingen. Aus diesem Grund wird PostgreSQL für die Verwaltung der Daten und Oracle für ADEPT benutzt.

Ein weiterer wichtiger Punkt ist die ADEPT-API: diese ist beim Prototyp auf Java beschränkt. Vor allem eine C++-API wäre nützlich, da man für diese leicht Wrapper für Script-Sprachen wie Python erstellen kann.

4.5.2 ADEPT als Dienst

Server-Software läuft normallerweise im Hintergrund als Dienst, bzw. Service unter Windows oder Daemon unter UNIX/Linux. Ein Dienst wird meist beim Starten des System aktiviert und beim Herunterfahren beendet.

Da ADEPT eine normale Java-Anwendung ist und trotzdem als Dienst laufen muss, musste eine Möglichkeit gefunden werden, dies zu bewerkstelligen.

Die einfachste Lösung, die ohne eine Anpassung von ADEPT auskommt, ist der Einsatz eines Wrappers, der eine Java-Anwendung kapselt und sie als Dienst laufen lässt. Ein ausgereifter und leicht aufzusetzender Wrapper ist der “Java Service Wrapper” ([JSW]). Mit Hilfe dieses Wrappers kann ADEPT mit

```
./adept.sh start
./adept.sh stop
./adept.sh restart
```

gestartet, gestoppt und neu gestartet werden.

4.6 WfMS-Clients

Für das Abrufen der Benutzerdaten, die Authentifizierung und das Ausführen von Prozess-Instanzen sind die drei WfMS-Clients vorgesehen, die im folgenden beschrieben werden. Die Clients liegen im Verzeichnis "client".

In Kapitel 4.2 wurde bereits diskutiert, ob die WfMS-Clients in Java oder Python geschrieben werden sollen. Um möglichen Problemen aus dem Weg zu gehen, werden die Clients in Java geschrieben.

Als erstes soll hier noch eine Frage beantwortet werden: Soll ein Client mehrere Prozess-Instanzen gleichzeitig ausführen können oder immer nur eine? Falls ein Client mehrere Prozess-Instanzen gleichzeitig ausführen kann, reduziert sich die Zahl der Betriebssystem-Prozesse, die gleichzeitig laufen. Der klare Vorteil dabei ist der geringere Ressourcen-Verbrauch: Für jede laufende Java-Anwendung wird eine Instanz der virtuellen Maschine (JVM) gestartet. Der Nachteil dieser Lösung ist die erhöhte Komplexität des WfMS-Clients, er muss ja mehrere Prozess-Instanzen gleichzeitig verwalten können. Um dieser Komplexität aus dem Weg zu gehen wurde entschieden, die Lösung mit einer Prozess-Instanz pro WfMS-Client zu wählen.

Wenn das System später weiterentwickelt und von vielen Benutzern verwendet wird, könnte der Speicherverbrauch wegen der vielen JVMs deutlich steigen. Deswegen soll hier noch eine dritte Lösung vorgeschlagen werden: Es wird eine Java-Anwendung geschrieben, die als Dienst im Hintergrund läuft und sämtliche WfMS-Clients verwaltet und ausführt. Diese Anwendung kann mit dem Glue-Server über Unix Message Queues kommunizieren. Dieser Ansatz hat den Nachteil, dass bei einem Absturz dieser Anwendung erst einmal nichts mehr geht, bis die Anwendung neu gestartet wurde.

Um die Arbeit zu verteilen und die einzelnen Clients einfacher zu halten, wurden insgesamt drei WfMS-Clients geschrieben:

- Login-Client
- Verwaltungs-Client

- Client zur Ausführung der Prozess-Instanzen

Da diese drei Clients vieles gemeinsam haben, wurde eine abstrakte Basisklasse geschrieben, von der sie abgeleitet werden (siehe Abbildung 4.3).

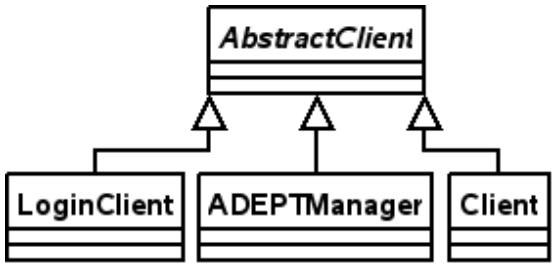

Abbildung 4.3: Hierarchie der WfMS-Client Klassen.

4.6.1 AbstractClient

Die abstrakte Basisklasse ist in der Datei "client/AbstractClient.java" definiert.

In Kapitel 4.7.3 ist beschrieben wie ein WfMS-Client gestartet wird. Wichtig ist dabei, wie die Parameter übergeben werden. Einige der Parameter werden als Aufrufparameter an den Konstruktor von AbstractClient übergeben. Die restlichen Parameter befinden sich in einer Konfigurationsdatei im Arbeitsverzeichnis, die im Konstruktor eingelesen wird. Nach dem Einlesen der Konfigurationsdatei versucht der Konstruktor von AbstractClient eine Verbindung zum ADEPT-Server aufzubauen. Im Erfolgsfall wird eine Nachricht mit dem Inhalt "OK" an den Glue-Server geschickt, sonst lautet die Nachricht "FAILED".

4.6.2 Login-Client

Die Klasse LoginClient (zu finden in "client/LoginClient.java") dient dazu, einen Benutzer zu authentifizieren. Dazu reicht es aus zu versuchen, mit den übergebenen Benutzernamen und Passwort eine Verbindung zum ADEPT-Server aufzubauen. Lehnt der Server diesen Versuch ab, liegt entweder ein Fehler vor oder die Zugangsdaten stimmen nicht. Und genau so verfährt der LoginManager: Die eigentliche Arbeit erledigt der Konstruktor der Basisklasse. LoginManager trennt nur noch die Verbindung und beendet sich.

Um einen Benutzer zu authentifizieren, startet der Glue-Server (genauer der WfMSManager) den Login-Client und wartet auf die Antwort. Lautet diese "OK", wird der Benutzer akzeptiert, sonst nicht.

4.6.3 Verwaltungs-Client

Die Klasse "ADEPTManager" ist in der Datei "client/ADEPTManager.java" definiert.

Der Verwaltungs-Client dient dem Abruf folgender Daten (die ja von ADEPT verwaltet werden, siehe Kapitel 5.1):

- Benutzer
- Rollen
- Fähigkeiten
- Abbildung Fähigkeiten → Benutzer
- Abbildung Rollen → Benutzer

Damit der UserManager (siehe Kapitel 4.7.2) diese Daten vom Verwaltungs-Client anfordern kann, wurden folgende Kommandos für die Nachrichten definiert (eine Mischung aus Englisch und Deutsch, da ADEPT die Bezeichnungen "Rolle" und "Faehigkeit" benutzt):

- GET_ALL_FAEHIGKEITEN
- GET_ALL_USERS
- GET_ALL_ROLLEN
- GET_FAEHIGKEIT_TO_USER
- GET_ROLE_TO_USER

Der Verwaltungs-Client wird vom WfMSManager (Kapitel 4.7.3) beim Hochfahren des Systems gestartet und vom UserManager (Kapitel 4.7.2) mit dem Kommando "TERMINATE" zum Beenden gezwungen (nachdem dieser die Daten ausgelesen hat).

4.6.4 Der allgemeine Client

Die Klasse “Client” ist in der Datei “client/Client.java” definiert.

Dieser WfMS-Client ist der komplexeste von den drei und dient dazu, neue Prozess-Instanzen zu starten und auszuführen. Der Name der Prozess-Vorlage, die benutzt werden soll, wird über die Konfigurationsdatei des Clients übergeben.

Im ADEPTEditor (siehe Kapitel 4.5.1) kann jedem Knoten (Aktivität) eine ganze Zahl (Anwendungs-ID) zugeordnet werden, über die die auszuführende Anwendung/Script identifiziert werden kann. Damit der Client weiß, welches Script welcher Nummer zugeordnet ist, wird eine Datei verwaltet, in der diese Abbildung festgehalten ist. Diese Datei heißt “command_mapping.txt” und liegt im Verzeichnis “clients”. Im gleichen Verzeichnis liegen auch die Python-Scripte (dies kann in der Konfigurationsdatei “GlueServer/glue_server.cfg” geändert werden). Das Format der Mapping-Datei ist ganz einfach: Jede Zeile enthält eine Zahl, der ein Script “zugewiesen” wird. Beispiel: 101 = getMachine.py. Die Nummern <100 wurden für Aktivitäten reserviert, die der Java-Client intern ausführen kann (zum Beispiel auf eine Nachricht vom Glue-Server warten). Falls einer Aktivität die Nummer 1 zugeordnet wurde, wird nichts gestartet und die Aktivität sofort als “ausgeführt” dem ADEPT-Server gemeldet. Damit können die “leeren” Knoten, die bei Verzweigungen und Schleifen entstehen, behandelt werden. Die Mapping-Datei wird im Konstruktor der Client-Classe eingelesen. Das Starten der Scripte übernimmt der Client.

Diese Methode Anwendungen zu starten ist zwar sehr einfach, aber auch unbequem und fehleranfällig. Beim Entwerfen der Prozess-Abläufe im ADEPTEditor muss der Benutzer wissen, welcher Anwendung welche Nummer zugeordnet ist, oder er muss die Mapping-Datei anpassen. Besser wäre es, wenn ADEPT 1 beliebige Anwendungen starten könnte und zwar über deren Namen (Pfad) und nicht über eine nichtssagende Nummer. Auch die automatische Weitergabe der Parameter (Datenflüsse) an die entsprechenden Anwendungs-Prozesse würde den Client erheblich vereinfachen. Anscheinend kann ADEPT aber nur Java-Anwendungen starten. Es wäre zwar möglich gewesen einen Java-Wrapper für Python-Scripte zu schreiben, welcher dann das eigentliche Script starten würde, das wäre aber doch recht aufwendig.

Nach dem Erzeugen einer Instanz der Klasse Client wird die Methode *run* aufgerufen. Diese Methode sucht dann nach dem Template und erzeugt eine neue Prozess-Instanz. Anschließend wird die Prozess-Instanz ausgeführt, bis die Arbeitsliste leer ist.

Falls eine Aktivität das Starten eines Python-Skriptes erfordert, legt der Client im Arbeitsverzeichnis (siehe Kapitel 4.7.3) ein Unterverzeichnis mit der Anwendungs-ID als Namen an. In diesem Verzeichnis werden wiederum zwei weitere Unterverzeichnisse angelegt: "in" für die Eingabeparameter und "out" für die Ausgabeparameter. Anschließend werden die Eingabeparameter in das "in"-Verzeichnis geschrieben, wobei jeder Eingabeparameter in einer eigenen Datei mit seinem Namen landet. Zusätzlich wird die Datei "param" (siehe Kapitel 4.7.3) in das Arbeitsverzeichnis des Scriptes kopiert.

Jetzt wird das Python-Script in einem neuen Prozess gestartet, wobei der Client auf die Beendigung dieses Prozesses wartet. Damit ist klar, dass der Client in der derzeitigen Implementierung nicht in der Lage ist, mehrere Aktivitäten gleichzeitig auszuführen. Das kann aber in einer zukünftigen Version nachgeholt werden.

Wenn der neue Prozess sich beendet, wird sein Exit-Status ausgewertet um festzustellen, ob das Script fehlerfrei ausgeführt wurde. Danach werden die Ausgabeparameter mit den Inhalten der gleichnamigen Dateien im "out"-Verzeichnis gefüllt (die Python-Scripte müssen also ihre Ausgabe in Dateien im "out"-Verzeichnis schreiben).

Jetzt meldet der Client die Beendigung der Aktivität dem ADEPT-Server und liest die Arbeitsliste neu ein.

Wenn die Prozess-Instanz komplett abgearbeitet wurde, schickt der Client eine Nachricht an den Glue-Server mit dem Inhalt:

```
TERMINATE
Benutzername
Nummer des Clients
```

Diese Informationen nutzt der WfMSManager um das Arbeitsverzeichnis des Clients wieder zu löschen.

Die Implementierung ist noch nicht vollständig, es fehlt vor allem eine gute Fehlerbehandlung (wenn zum Beispiel ein Client abstürzt).

4.7 Glue-Server

Der Glue-Server befindet sich im Verzeichnis "GlueServer", die Abbildung 4.4 zeigt seinen Aufbau. Die Klasse "ServerPerspective" ist der eigentliche Glue-

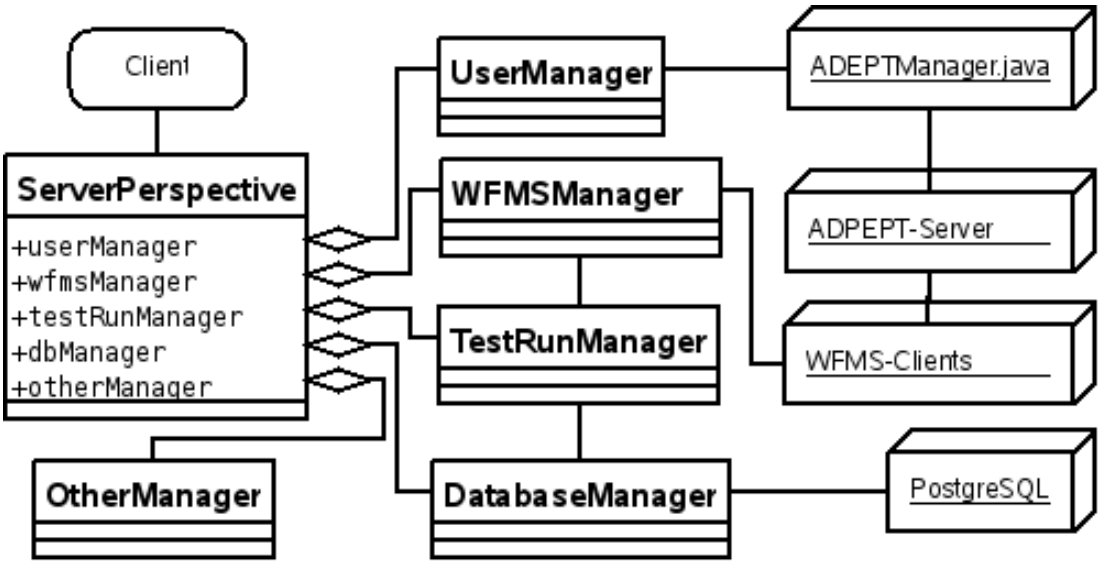

Abbildung 4.4: Interner Aufbau des Glue-Servers.

Server, alle öffentlichen Methoden dieser Klasse werden über XML-RPC exportiert. Die meiste Arbeit verteilt der Glue-Server auf fünf Hilfsklassen:

- DatabaseManager - Schnittstelle zum DBMS-Server
- UserManager - Abfrage der Benutzerdaten
- WFMSManager - Verwaltung der WfMS-Clients
- OtherManager - sonstiges
- TestRunManager - Verwaltung der Testläufe

Der Glue-Server verwaltet je eine Instanz dieser Klassen und leitet die Anfragen entsprechend weiter. Um die Geschwindigkeit und die Antwortzeiten zu optimieren, wäre eine Aufteilung in sechs Prozesse denkbar: ein Prozess für den Glue-Server und je ein Prozess für die fünf Manager-Klassen.

4.7.1 DatabaseManager

Diese Klasse bildet die Schnittstelle zum PostgreSQL-Server und befindet sich in der Datei "sql/DatabaseManager.py". Der Zugriff auf die Datenbank erfolgt über psycopg2, die Einstellungen wie Benutzername und Passwort befinden sich in der Konfigurationsdatei "sql/sql.cfg".

DatabaseManager unterhält keine permanente Verbindung zum SQL-Server, statt dessen wird bei jedem Aufruf eine Verbindung aufgebaut und am Ende wieder abgebaut. Das ist zwar einfacher zu implementieren, hat aber auch Nachteile: Aufbau und Abbau einer Verbindung kosten Zeit und bei vielen Anfragen in kurzer Zeit könnte die maximale Anzahl gleichzeitiger Verbindungen überschritten werden. Es wäre also besser, wenn DatabaseManager in Zukunft eine oder einige wenige, permanente Verbindungen nutzen würde.

Bei der derzeitigen Implementierung wird jeder Aufruf von DatabaseManager in einer eigenen Transaktion abgewickelt, auch wenn dabei nur eine einzige SELECT-Anweisung ausgeführt wird. Bei erfolgreicher Ausführung wird die Transaktion mit einem Commit beendet, sonst wird sie mit einem Rollback abgebrochen.

4.7.2 UserManager

Der UserManager (zu finden in "GlueServer/UserManager.py") wird benutzt, um die Benutzerdaten von ADEPT abzufragen und für eine spätere Nutzung zwischenzuspeichern. Um auf ADEPT zuzugreifen, benutzt UserManager den WfMS-Client "ADEPTManager" (siehe Kapitel 4.6.3). Nach dem Abruf der Benutzerdaten werden diese für einen schnelleren Abruf aufbereitet.

Zur Zeit werden folgende Daten zur Verfügung gestellt:

- Rollen

- Fähigkeiten
- Benutzer
- Zuordnung der Fähigkeiten zu Rollen
- Zuordnung der Rollen zu Benutzern

Die ersten drei können sowohl über die ID als auch über den Namen abgerufen werden. Die beiden letzten können in beide Richtungen abgefragt werden.

Bei der derzeitigen Implementierung werden die Daten beim Erzeugen der User-Manager-Instanz abgerufen und können somit veralten, wenn danach der Datenbestand geändert wird (zum Beispiel können Benutzer gelöscht oder neue angelegt werden). Es sollte also noch eine Möglichkeit vorgesehen werden, den UserManager zum erneuten Einlesen der Daten zu veranlassen. Alternativ könnte der UserManager die Daten erst bei Bedarf vom ADEPTManager anfordern, statt sie einmal komplett einzulesen, was aber weniger performant ist.

4.7.3 WFMSManager

Die Klasse WFMSManager (zu finden in "GlueServer/WFMSManager.py") ist von großer Bedeutung, da sie die WfMS-Clients verwaltet.

Jeder WfMS-Client wird einem Benutzer zugeordnet und jeder Benutzer kann mehrere Clients laufen lassen. Ein Client wird also über den Benutzernamen und eine fortlaufende Nummer adressiert.

Vor dem Starten eines neuen WfMS-Clients legt der WFMSManager ein neues Verzeichnis an, das als Arbeitsverzeichnis für den Client dient. Diese Arbeitsverzeichnisse liegen standardmäßig im Verzeichnis "GlueServer/clients", was aber in der Konfigurationsdatei "GlueServer/glue_server.cfg" geändert werden kann. Der Name des Arbeitsverzeichnises setzt sich aus dem Benutzernamen und der Nummer (wird mit jedem weiteren Client einfach hochgezählt) des Clients zusammen. Anschließend wird in dieses Verzeichnis die Datei "client.cfg" geschrieben, die einige Einstellungen enthält, die jeder Client benötigt (die Schlüssel für die Message Queues, die Adresse für diesen Client, den Namen der Prozess-Vorlage usw.). Zusätzlich wird noch eine Datei mit dem Namen "param" geschrieben,

die beliebige Parameter für die Prozess-Instanz enthält (zum Beispiel welcher Testlauf ausgeführt werden soll). Die Adresse für den Client wird ebenfalls vom WFMSManager vergeben (es ist eine Integer-Zahl, die einfach hochgezählt wird). Jetzt erfolg ein *fork*-Aufruf und der Kind-Prozess startet den Client mit einem *execlp*-Aufruf. Dabei werden noch einige Parameter an den Client übergeben:

- Arbeitsverzeichnis
- Name der Konfigurationsdatei
- Vorname des Benutzers
- Name des Benutzers
- Passwort

Das Passwort sollte später besser über die Konfigurationsdatei übergeben werden, da man dieses sonst mit einem

```
ps ax | grep java
```

auf der Kommandozeile einsehen kann.

Wenn sich ein WfMS-Client beendet, schickt er eine Nachricht an den Glue-Server mit dem Inhalt "TERMINATE", Benutzername und Adresse. Damit kann der WFMSManager wieder aufräumen.

4.7.4 OtherManager

Die Klasse "OtherManager ist in der Datei "GlueServer/OtherManager.py" definiert. Eine Instanz dieser Klasse benutzt der Glue-Server für Funktionen, die keinem anderen Manager zugeordnet wurden. OtherManager bietet folgende Funktionalität:

- Lava-Konfigurationen verwalten
- Label-Dokumente für Label-Aufträge generieren
- Label-Dokumente exportieren

Die Lava-Konfigurationen werden in Verzeichnissen gesucht, die in der Konfigurationsdatei "GlueServer/glue_server.cfg" angegeben sind.

Da OtherManager für seine Arbeit Daten aus der Datenbank lesen muss, bekommt er eine Referenz auf die DatabaseManager-Instanz. Damit die Kommunikation mit dem DatabaseManager intern schneller ablaufen kann, stellt dieser die Funktion *callNoJson* zur Verfügung, die auf Json verzichtet und so das Enkodieren und Dekodieren der Daten überflüssig macht.

4.7.5 TestRunManager

Die Klasse "TestRunManager" ist in der Datei "GlueServer/TestRunManager.py" definiert. Der TestRunManager sorgt dafür, dass die Testläufe zum angegebenen Zeitpunkt ausgeführt werden. Dazu liest er beim Starten alle noch nicht gestarteten Testläufe und definiert mit Hilfe des "sched"-Modules von Python ([Sched]) für jeden Testlauf ein Event, das zum gegebenen Zeitpunkt eine Funktion aufruft. Mit Hilfe des sched-Modules (event scheduler) kann zu einem bestimmten Zeitpunkt eine Funktion aufgerufen werden.

Sobald ein solches Ereignis eintritt, wird mit Hilfe des WfMSManagers der Testlauf gestartet. Diese Lösung ist allerdings noch nicht vollständig, da beim Anlegen von neuen Testläufen der TestRunManager erst einmal nichts davon mitbekommt. Dafür gibt es zwei mögliche Lösungen: Die Methode "saveTestRun()" der Klasse DatabaseManager, die einen neuen Testlauf in die Datenbank schreibt, kann den TestRunManager benachrichtigen, oder aber es wird ein Trigger für die Relation "test_run" definiert, der eine in Python geschriebene Stored Procedure aufruft, die wiederum über die API eine Nachricht an den Glue-Server schickt. Wegen der Einfachheit fiel die Entscheidung auf die erste Variante.

Der DatabaseManager ruft also den TestRunManager auf und übergibt ihm den Namen des neuen Testlaufs, woraufhin dieser den neuen TestLauf einliest und ein Event definiert.

Da TestRunManager auf die Datenbank und den WfMSManager zugreifen muss, werden ihm Referenzen auf DatabaseManager und WfMSManager übergeben.

4.7.6 Logging

Um Meldungen aller Art (Warnungen, Fehler usw.) zu protokollieren, wurde mit Hilfe des "logging"-Modules von Python ([pylog]) ein Logger aufgesetzt (zu finden in "api/specht/tools/Logger.py"), der auf die Standardausgabe schreibt (dazu siehe Kapitel 4.7.7). Dieser Logger wird auch von den Hilfsklassen benutzt.

Es werden mehrere Typen von Meldungen unterschieden:

- DEBUG
- INFO
- WARNING
- ERROR
- CRITICAL

In der Konfigurationsdatei "GlueServer/glue_server.cfg" kann angegeben werden, welche Meldungen protokolliert werden sollen: Steht dort "WARNING", werden "DEBUG"- und "INFO"-Meldungen nicht protokolliert.

Neben dem Typ der Meldung wird noch das Datum, die Uhrzeit, die Python-Datei und die Zeile in dieser Datei (in der die Meldung ausgegeben wurde) im Log verzeichnet.

4.7.7 Glue-Server als Dienst

Der Glue-Server wurde so implementiert, dass er als Daemon (Dienst) laufen kann, allerdings existiert noch keine einfache Möglichkeit, diesen Daemon zu beenden, außer den Prozess mit

```
kill PID
```

(PID - die Prozess ID des Glue-Servers) zum Beenden zu zwingen (Glue-Server fängt das entsprechende Signal ab und fährt dann sauber runter).

Die Standard-Ausgabe und die Standard-Fehlerausgabe werden in je eine Datei umgeleitet, die in der Konfigurationsdatei "GlueServer/glue_server.cfg" angegeben sind. Somit schreibt auch der Logger in die entsprechende Datei (Standardausgabe).

4.8 Web-Interface

Das Web-Interface als Benutzerschnittstelle ist von großer Bedeutung und soll in diesem Kapitel genauer beschrieben werden. Das Fundament des Web-Interface, bestehend aus Apache HTTP-Server und mod_python, wurde bereits im Kapitel 4.3.1 vorgestellt und wird hier nicht weiter vertieft.

Von großer Bedeutung für das Web-Interface ist die Plattformunabhängigkeit. Darunter soll hier sowohl die Unabhängigkeit vom Client-Betriebssystem als auch vom verwendeten Browser verstanden werden. Um diese Unabhängigkeit zu erreichen, wurde darauf geachtet, dass das Web-Interface mit den gängigen Browsern funktioniert:

- Internet Explorer 6 und 7
- Mozilla Firefox ab 1.0
- Opera ab 9.0

Bisher wurden allerdings nicht alle Browser/Betriebssystem-Kombinationen getestet.

Da die Browser die Web-Standards nicht immer korrekt implementieren, beschränkt sich das Web-Interface derzeit auf folgende Technologien:

- HTML
- HTML-Formulare
- JavaScript

Später soll noch CSS - Cascading Style Sheets ([CSS])- hinzukommen, um die Gestaltung des Web-Interface zentral verwalten zu können. Dabei muss besonders auf Kompatibilität mit den Browsern geachtet werden.

4.8.1 JavaScript

HTML und HTML-Formulare reichen nicht immer aus, um eine Web-basierte Benutzerschnittstelle zu implementieren, da es keine Interaktivität bietet. Da man aber oft auf Benutzereingaben reagieren und bestimmte Aktivitäten ausführen muss, kommt man um JavaScript nicht herum. JavaScript ist eine objektbasierte Scriptsprache, deren Syntax der von Java ähnelt. JavaScript-Code kann entweder direkt in eine HTML-Seite eingebaut oder als Verweis angegeben werden (in diesem Fall wird der Code vom Browser nachgeladen). Ausgeführt wird JavaScript-Code im Webbrowser und hat über DOM - Document Object Model ([W3CDOM]) - vollen Zugriff auf die Webseite.

Beispiel für eine sinnvolle Anwendung von JavaScript: Der Benutzer wählt eine Bild-Sequenz aus und will alle Kanäle angezeigt bekommen, die zu dieser Sequenz gehören. Beim Generieren dieser Seite auf dem Server gibt es keine Möglichkeit, so etwas zu berücksichtigen. Ohne JavaScript müsste nach der Auswahl der Bild-Sequenz die Seite neu geladen werden, was ein hohes Datenaufkommen und Verzögerungen verursachen würde. Dagegen kann mit JavaScript nach der Selektion der Bild-Sequenz die Liste der Kanäle angefordert werden. Das kann sogar asynchron geschehen - die Seite wird im Browser nicht blockiert.

Der größte Nachteil von JavaScript ist die Uneinheitlichkeit: Es gibt zwei unterschiedliche Implementierungen, eine von Microsoft (Internet Explorer) und eine von Netscape. Das kann die Entwicklung vom JavaScript-Code erheblich verkomplizieren. Zum Glück gibt es inzwischen eine große Zahl an JavaScript-Bibliotheken, die diese Unterschiede verbergen und so eine einheitliche Schnittstelle zur Verfügung stellen.

MochiKit ([MochiKit]) ist eine solche Bibliothek (eine ebenfalls weit verbreitete ist “prototype” [Prot]), die gut dokumentiert und im Einsatz erprobt ist. Aus diesem Grund wurde MochiKit für das Web-Interface benutzt. Vor allem zwei Funktionen, die MochiKit anbietet, wurden oft benutzt. Die Funktion *$(ID)* gibt das DOM-HTML-Object mit der Id == “ID” zurück und zwar unabhängig vom verwendeten Browser. Die zweite Funktion *loadJSONDoc(URL)* fordert asynchron von der Adresse “URL” ein Json-Dokument an und dekodiert dieses auch gleich zu einem JavaScript-Objekt.

MochiKit selbst besteht aus einer einzigen Datei “MochiKit.js”, die unter “htdocs/wfms/js” abgelegt wurde. In diesem Verzeichnis liegen noch weitere JavaScript-Dateien, da der Großteil des JavaScript-Codes nicht direkt in die PSP-Dateien (zu PSP siehe Kapitel 4.3.5) geschrieben, sondern in Dateien mit der Endung “.js” untergebracht wurde. Das hat den Vorteil, dass man so den JavaScript-Code leicht in andere PSP-Dateien importieren kann.

4.8.2 Gestaltung

Um mit dem Web-Interface arbeiten zu können, muss sich der Benutzer zuerst authentifizieren. Dazu ruft er die Login-Seite “wfms/login” auf (oder jede andere Seite, die ihn dann auf die Login-Seite umleitet). Falls die Anmeldung fehlschlägt,

Abbildung 4.5: Login-Seite.

landet der Benutzer wieder auf der Login-Seite, die eine entsprechende Meldung anzeigt (Abbildung 4.6).

Abbildung 4.6: Login-Failed-Seite.

Nach erfolgreicher Anmeldung landet der Benutzer auf der Startseite “wfms/index” (Abbildung 4.7). Die Anmeldung ist für 30 Minuten gültig, danach wird

Abbildung 4.7: Startseite.

man wieder auf die Login-Seite umgeleitet.

Der Inhalt der Startseite hängt davon ab, welche Rollen dem Benutzer zugeordnet wurden. Zum Beispiel fehlt der Link "Manage label jobs", wenn dem Benutzer die Rolle "LabelJobManager" fehlt. Da dieser "Zugriffsschutz" nicht ausreicht (man kann die entsprechenden Seiten ja auch direkt in der Adressleiste des Browsers angeben), wird bei jedem Zugriff auf eine Seite zuerst geprüft, ob der Benutzer dazu berechtigt ist. Dazu wird über den Glue-Server der UserManager kontaktiert, der anhand des Benutzernamens und der Rollen des Benutzers entscheidet, ob der Zugriff gewährt wird. Falls der Zugriff verweigert wird, erscheint eine entsprechende Meldung. Zur Zeit ist die Zugriffskontrolle an die Rollen geknüpft, was nur einen recht groben Zugriffsschutz erlaubt. Später könnte man auch die Fähigkeiten hinzunehmen.

Der Click auf den Link "Label Documents" führt auf die Seite "Labeling" (Abbildung 4.8). Um diese Seite aufrufen zu können, braucht der Benutzer die Rolle "Labeler". Von hier aus gelangt man auf die Seite für den Import von Label-

Abbildung 4.8: Labeln.

Dokumenten (Abbildung 4.9). Die Rollen “Labeler” und “LabelDocumentManager” berechtigen einen Benutzer zum Aufruf dieser Seite. Über den zweiten Link

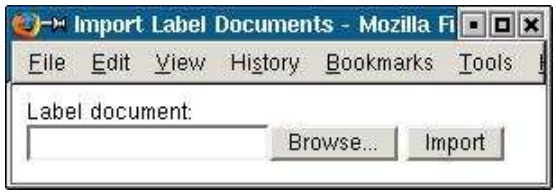

Abbildung 4.9: Label-Dokument Import.

“View label jobs” gelangt man auf die Seite für den Abruf der Label-Aufträge (Abbildung 4.10). Voraussetzung dafür ist die Rolle “Labeler”. Auf dieser Seite

Abbildung 4.10: Label-Aufträge.

kann der Benutzer sowohl die Konfigurationsdatei für Lava als auch das Label-Dokument für den Label-Auftrag herunterladen.

Der Link “Manage label jobs” auf der Startseite führt den Benutzer, vorausgesetzt er verfügt über die Rolle “LabelJobManager”, auf die Seite für die Verwaltung der Label-Aufträge (Abbildung 4.11). Diese Seite besteht aus zwei Teilen: Oben ist ein Filter, in dem man die Auswahl der unten angezeigten Label-Aufträge einschränken kann. Unten werden die Label-Aufträge aufgelistet. Mit dem Button “Delete” können die ausgewählten Label-Aufträge (siehe die Checkboxen links neben den Aufträgen) gelöscht werden. Mit einem Click auf “New” gelangt man auf die Seite zum Anlegen neuer Label-Aufträge (Abbildung 4.12). Mit einem

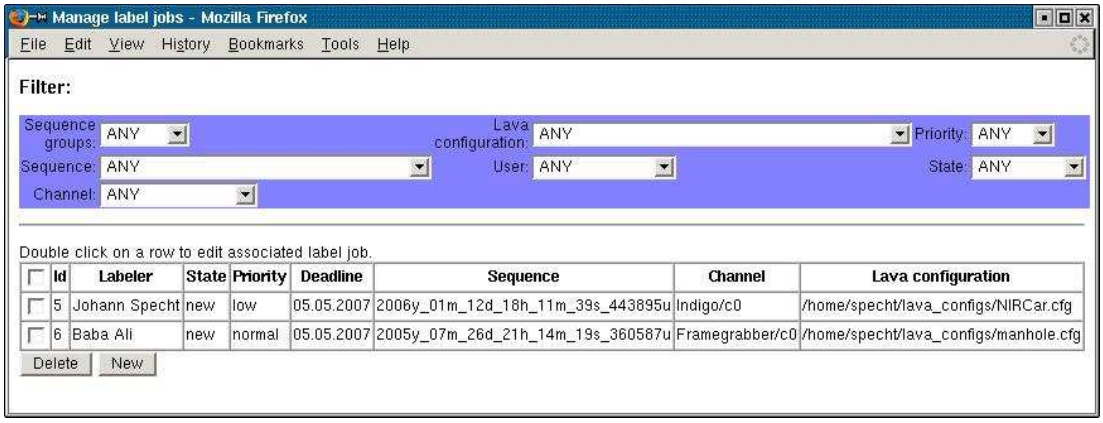

Abbildung 4.11: Verwaltung der Label-Aufträge.

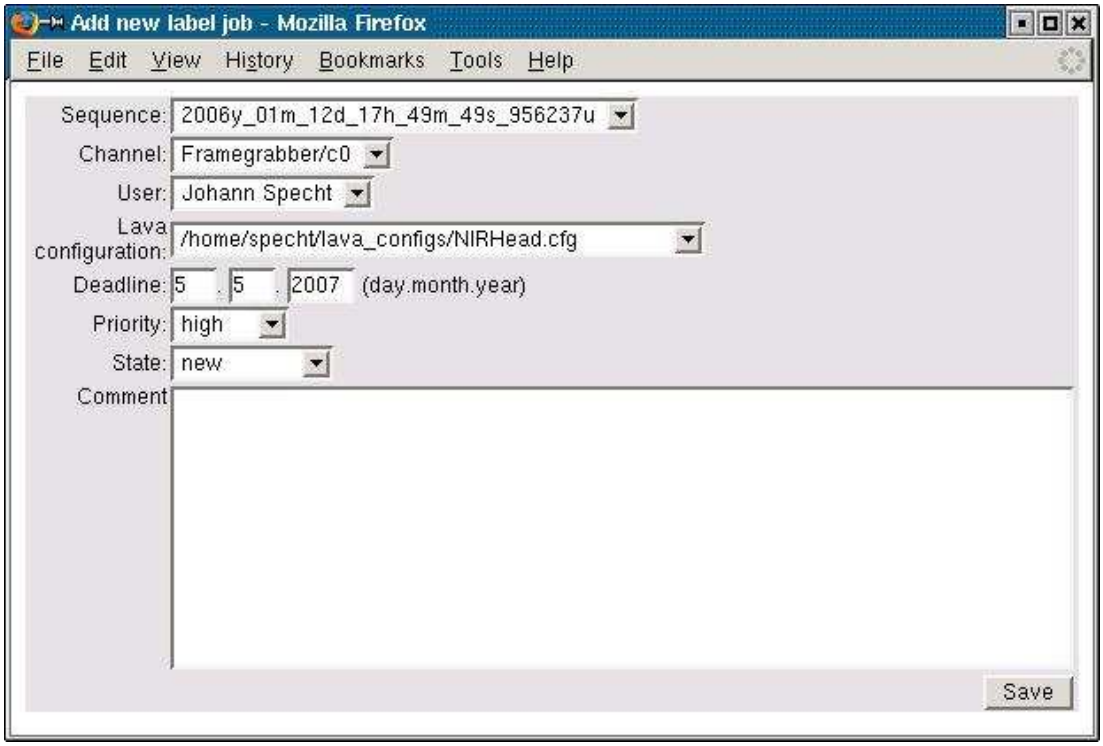

Abbildung 4.12: Einen neuen Label-Auftrag anlegen.

Doppelklick auf einen Label-Auftrag ruft man die Seite zum Editieren eines Auftrags (entspricht der Abbildung 4.12) auf. Für das Anlegen neuer Aufträge sowie das Löschen und Editieren von bestehenden Aufträgen wird ebenfalls die Rolle "LabelJobManager" vorausgesetzt.

Der Link "Manage detection systems" auf der Startseite führt auf die Seite zum Verwalten der Erkennungssysteme (Abbildung 4.13). Um diese Seite aufrufen zu dürfen, benötigt der Benutzer die Rolle "DetectionSystemManager".

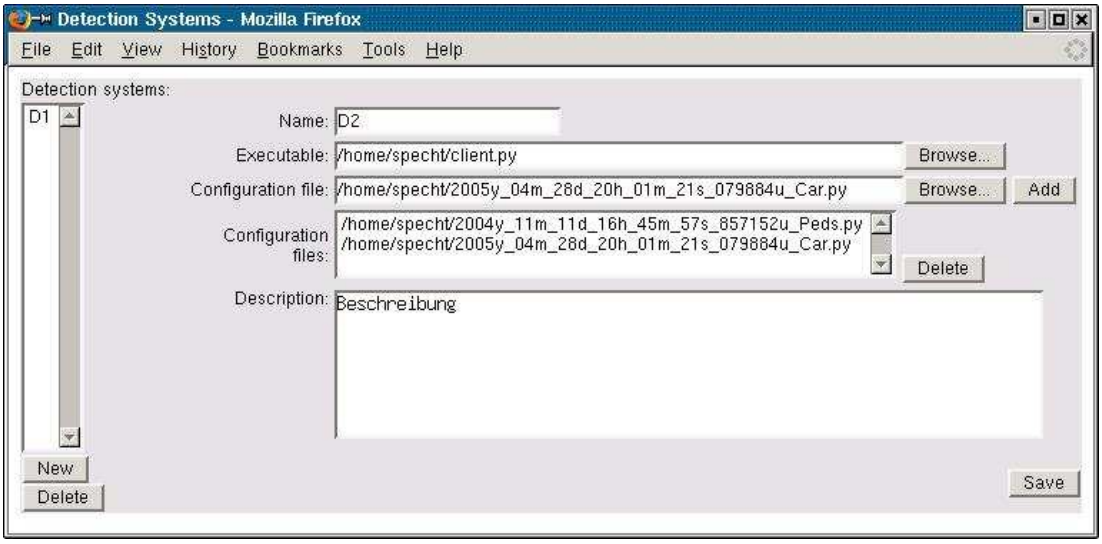

Abbildung 4.13: Verwaltung der Erkennungssysteme.

Mit "Manage test runs" gelangt man von der Startseite auf die Seite für die Verwaltung der Testläufe (Abbildung 4.14). Voraussetzung dafür ist die Rolle "TestRunManager".

Auf die Seite für die Verwaltung der Bild-Sequenzen gelangt man über den Link "Manage Sequences" (Abbildung 4.15). Hierfür braucht man die Rolle "SequenceManager".

Als Letztes kann mit "Access Database" eine Seite aufgerufen werden, auf der beliebige SQL-Queries abgesetzt werden können. Das Ergebnis wird auf der Seite in Tabellenform dargestellt. Aus Sicherheitsgründen dürfen nur Benutzer diese Seite aufrufen, die über die Rolle "DatabaseManager" verfügen

Den Abbildungen kann man entnehmen, dass das Web-Interface noch sehr schlicht gehalten ist und kein einheitliches Aussehen hat.

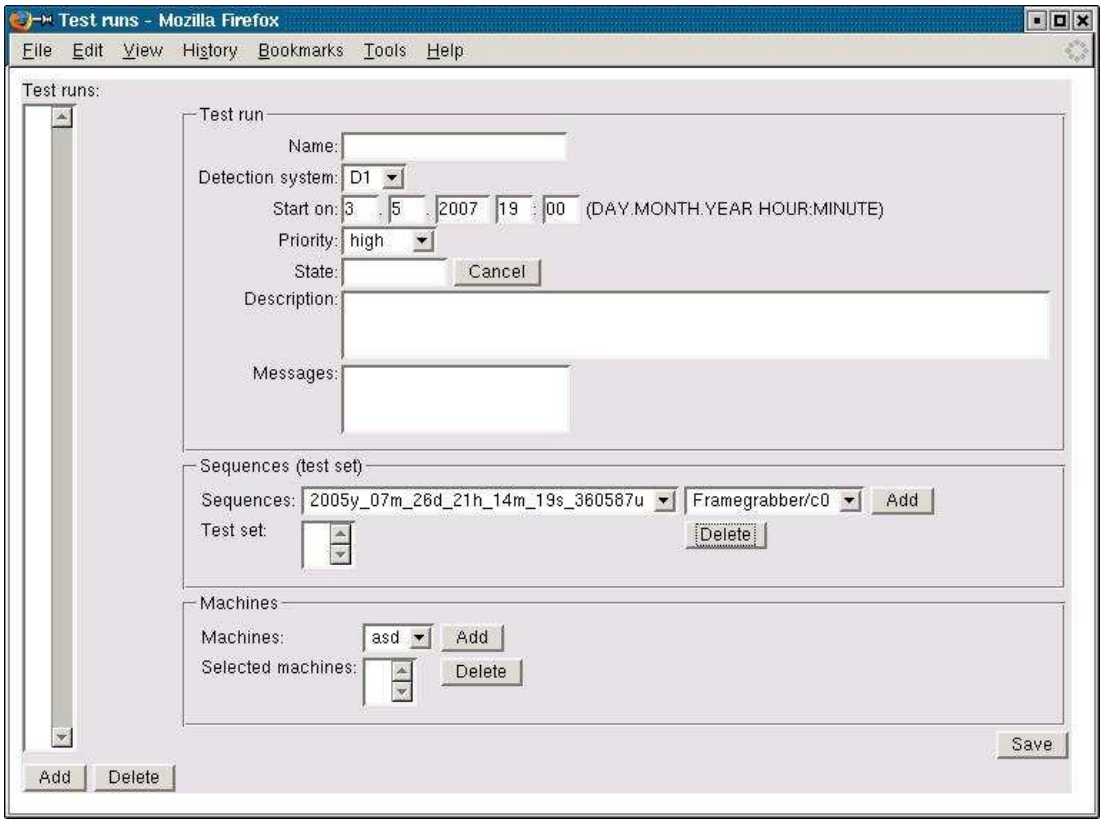

Abbildung 4.14: Verwaltung der Testläufe.

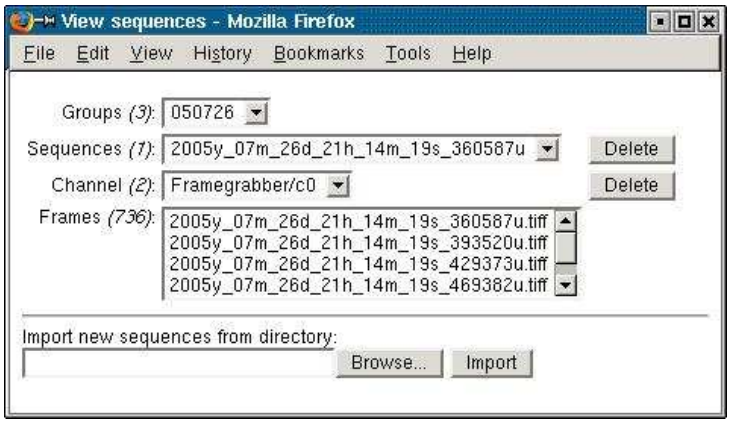

Abbildung 4.15: Verwaltung der Bild-Sequenzen.

Anzumerken ist noch, dass das Web-Interface nicht die gesamte Funktionalität abdeckt. Vor allem Lava und die ADEPT-Werkzeuge stellen normale GUI-Anwendungen dar, deren Funktionalität aber in Zukunft durchaus in das Web-Interface integriert werden könnte. Mit modernen Web 2.0 Technologien wie Ajax ([AJAX]) ist eine Web-basierte Lösung für das Labeln durchaus denkbar.

4.9 Python-API

Die Python-API befindet sich im Verzeichnis "api/specht".

Die Python-API kann in vier Bereiche unterteilt werden, die in eigenen Kapiteln genauer beschrieben werden:

- "DATA" : Zugriff auf die Datenbank
- "USER" : Zugriff auf die Benutzerdaten
- "WFMS" : Zugriff auf die Workflow-Funktionalität
- "OTHER" : Sonstige Funktionalität

Bis auf eine Ausnahme (SQLObject, siehe Kapitel 4.9.3) kommunizieren alle API-Komponenten mit dem Glue-Server über XML-RPC. Damit kann der Rest des Systems von den Client-Anwendungen versteckt werden: Diese wissen weder etwas von PostgreSQL-Server noch "sehen" sie den ADEPT-Server oder die WFMS-Clients. Auch die Zugriffskontrolle kann hier ansetzen und die Zugriffe filtern.

Es ist allerdings möglich, dass der Glue-Server in Zukunft zu einem "Flaschenhals" wird und so das System ausbremst. Um dem entgegen zu wirken, könnte der Glue-Server später auf mehrere Prozesse verteilt werden (seihe Kapitel 4.7).

4.9.1 API-Architektur

Die einzige Möglichkeit mit dem Glue-Server zu kommunizieren ist XML-RPC. Dafür exportiert der Glue-Server die Funktion *call*, die zwei Parameter erhält: eine Adresse und eine Liste (Array) mit Parametern. Die Adresse ist von zentraler Bedeutung, da dies die einzige Möglichkeit ist, dem Glue-Server zu sagen,

an wen er die Anfrage weiterleiten soll. Der Aufbau der Adresse entspricht den Pfadangaben in UNIX-Systemen: Die Wurzel ist "/", danach folgt der "Name" des Managers (eines der Werte "data" für DatabaseManager, "other" für OtherManager, "user" für UserManager, "wfms" für WFMSManager), der Rest hängt vom angesprochenen Manager ab (meist steht hier der Name einer Funktion). Die einzelnen Bestandteile der Adresse werden mit "/" von einander getrennt. Beispiel: "/other/getLavaConfigs" ruft OtherManager.getLavaConfigs() auf. Welche Einträge der zweite Parameter enthält, hängt vom Manager/Funktion ab.

Die vier Manager haben die gleiche Schnittstelle wie der Glue-Server: Jeder von ihnen hat nur eine nach außen sichtbare Funktion: *call*. Wenn die call-Methode des Glue-Servers aufgerufen wird, entfernt diese den Namen des Managers aus der Adresse und übergibt den Rest an die call-Methode des Managers. Der zweite Parameter wird unverändert weitergegeben. Beispiel:
GlueServer.call('/other/getLavaConfigs', param) →
OtherManager.call('/getLavaConfigs', param)
Die Rückgabe des Managers wird vom Glue-Server unverändert an den Client zurückgegeben.

Alle vier Manager geben grundsätzlich einen Tupel zurück (als JSON-String enkodiert), welcher aus zwei Komponenten besteht: True/False und das eigentliche Ergebnis oder Fehlermeldung. Falls an erster Stelle False steht, ist ein Fehler aufgetreten und an der zweiten Stelle steht dann die Fehlermeldung. Ansonsten steht an der zweiten Stelle das Ergebnis (das kann eine beliebige Python-Datenstrucktur sein). Listing 4.8 zeigt ein Beispiel dafür, wie eine Fehlermeldung zurückgegeben wird.

Listing 4.8: Rückgabe einer Fehlermeldung

```
(False, 'Unknown function "getLListResult"!')
```

4.9.2 GlueServerInterface

Die zentrale Komponente der Python-API ist die Klasse "GlueServerInterface" (in "api/specht/GlueServerInterface.py"). Mit dieser Klasse kann auf einfache Art und Weise auf den Glue-Server zugegriffen werden. Einfach eine Instanz erzeugen und entweder die Methode *call(param)* oder *callJson(param)* aufrufen. Das

Besondere an diesen beiden Funktionen ist die Tatsache, dass man mit einem Aufruf mehrere Anfragen schicken kann. Dazu legt man ein Python-Dictionary an und erzeugt für jeden Aufruf einen Eintrag:

Listing 4.9: Nutzung des GlueServerInterface

```
gsi = GlueServerInterface()
param = {'result1' : ['/data/getListResult/getColorByID',
                      {'id' : 1}],
         'result2' : ['/other/getLabelDocumentByName',
                      {'name' : 'NAME'}] }
result = gsi.call(param)
```

In diesem Beispiel werden zwei Aufrufe abgesetzt: eine Farbe über die ID lesen und ein Label-Dokument exportieren. Mit den beiden Schlüsseln "result1" und "result2" (die frei gewählt werden können) kann das jeweilige Ergebnis gelesen werden: result['result1'] oder result['result2']. Die Methode *call* gibt also ein Dictionary zurück mit den Schlüsseln, die im Parameter übergeben wurden. Jeder Schlüssel identifiziert die Rückgabe der jeweiligen Remote-Funktion. Also: *result1* identifiziert die Rückgabe der Funktion *getColorByID* und *result2* die Rückgabe von *getLabelDocumentByName*. Man kann an diesem Beispiel auch erkennen, dass meistens ein Dictionary an die Remote-Funktion übergebenen wird:

```
{'id' : 1}
```

Die Methode *call()* kehrt erst zurück, wenn alle Aufrufe abgearbeitet wurden. Neben *call* gibt es noch die Methode *callJson*, die das Ergebnis als JSON-String zurückgibt (wird für das Web-Interface benötigt). Beide verpacken den Parameter vor dem Verschicken als JSON-String.

4.9.3 DB-API

Die DB-API befindet sich im Verzeichnis "api/specht/db" und kann in zwei Bereiche unterteilt werden: die low-level API und die high-level API.

Die low-level API entspricht dem, was oben beschrieben wurde und kann somit mit Hilfe der GlueServerInterface-Klasse benutzt werden (die Adresse beginnt

mit "/data"). Es gibt insgesamt drei Möglichkeiten, auf den DatabaseManager zuzugreifen:

- SQL-Query wird direkt übergeben
- es wird der Name der Datei mit der Query übergeben
- es wird eine Methode aufgerufen

Um eine beliebige Query auszuführen, steht die Methode *getRawResultForQuery* zur Verfügung. Diese bekommt als Parameter ein Dictionary mit dem Schlüssel "query" und einen String mit der Query als Wert:

Listing 4.10: Aufruf von getRawResultForQuery

```
gsi = GlueServerInterface()
param = {'result' : ['/data/getRawResultForQuery',
                     {'query' : 'SELECT * FROM color'}]}
result = gsi.call(param)
```

Es ist klar, dass der Zugriff auf diese Funktion eingeschränkt werden sollte, da sonst jeder beliebige Anfragen (auch DELETE) an den SQL-Server schicken kann. Zum Beispiel könnte das Recht, diese Funktion auszuführen, an die Rolle "DatabaseManager" gebunden werden.

Die am meisten benutzte Methode mit dem DatabaseManager zu kommunizieren besteht darin, den Namen einer Datei anzugeben, welche die Query enthält. Dies geht allerdings nur indirekt:

Listing 4.11: Aufruf einer Query-Datei

```
gsi = GlueServerInterface()
param = {'result' : ['/data/getListResult/getAllColors']}
result = gsi.call(param)
```

In diesem Beispiel lautet der Name der Datei "getAllColors" (die Endung ".sql" muss nicht angegeben werden). Die Funktion *getListResult* ist eine der drei Funktionen, welche die Ergebnis-Tupel aufbereiten:

- *getListResult*: Ergebnis als Liste von Listen zurückgeben

- *getDictResult*: Ergebnis als Liste von Dictionaries zurückgeben (jedes Ergebnis-Tupel ist ein Dictionary mit Attribut-Namen als Schlüssel)
- *getRawResult*: Ergebnis unverändert zurückgeben (Liste mit zwei Einträgen: der erste Eintrag enthält die Angaben zu den Attribut-Namen und der zweite die Ergebnis-Tupel)

Die Query-Dateien befinden sich standardmäßig im Verzeichnis "sql/queries", dies kann aber über die Konfigurationsdatei "sql/sql.cfg" geändert werden. Dieser Ansatz bietet den Vorteil, neue Query-Dateien hinzufügen oder vorhandene anpassen zu können, ohne das System neu starten zu müssen.

Der Zugriff auf die Query-Dateien kann leicht vom DB-Manager überwacht werden, indem für jede Datei angegeben wird, welche Rollen/Fähigkeiten den Zugriff gestatten.

Die dritte Möglichkeit auf den DB-Manager zuzugreifen, besteht darin, eine vorgefertigte Funktion aufzurufen. Einige Beispiele: *insertNewLabelJob*, *getAllLabelJobsForUser*, *saveDetectionSystem*, *saveTestRun*.

Neben dieser low-level Schnittstelle, die hauptsächlich für das Web-Interface benutzt wird, existiert noch eine (noch nicht vollständige) high-level API. Diese basiert auf SQLObject ([SQLObj]), einem objektrelationalen Wrapper für Python. SQLObject bildet eine Schicht über der SQL-Datenbank und versteckt so SQL vom Entwickler. Dabei werden Relationen als Klassen repräsentiert und Tupel als Instanzen dieser Klassen. Dabei wird der Großteil der Flexibilität der SQL-Queries bewahrt, so dass man weitgehend auf SQL verzichten kann. Einige Abstriche muss man aber doch machen: SQLObject unterstützt keine geometrischen Datentypen und greift direkt auf die Datenbank zu. Wenn man also die Geometrie-Daten der Label verarbeiten will, muss man den Weg über die low-level API gehen.

Wegen SQLObject wurde bei sämtlichen Relationen in der Datenbank ein Primärschlüssel mit dem Namen "id" vom Typ Integer definiert, der von PostgreSQL automatisch hochgezählt wird. Der Grund dafür ist, dass SQLObject bei allen Relationen einen einfachen (also nicht aus mehreren Attributen zusammengefügten) Primärschlüssel vom Typ Integer erwartet. Zusätzlich müssen Fremdschlüssel mit "id" enden, zum Beispiel "frame_id".

Der direkte Zugriff von SQLObject auf die Datenbank (über psycopg) stellt ein ernstes Problem dar, da keine Zugriffskontrolle realisiert werden kann. Als Kompromiss wurde beschlossen, für SQLObject einen Datenbank-Benutzer anzulegen, der nur lesenden Zugriff auf die Relationen hat. Sämtliche Schreibzugriffe werden als Methoden implementiert, die über die low-level API auf die Datenbank zugreifen, wo die Zugriffskontrolle stattfinden kann (siehe Kapitel 5.2.2).

Eine Übersicht über die high-level API gibt die Abbildung 4.16.

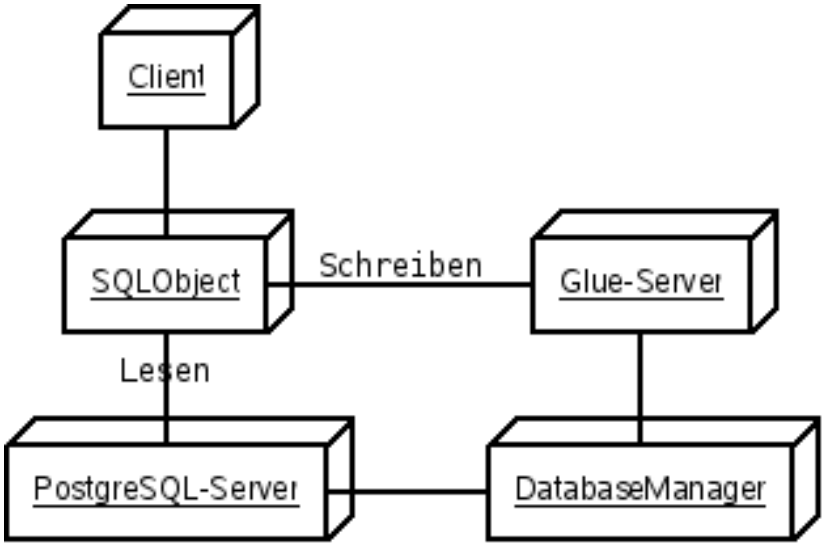

Abbildung 4.16: SQLObject: Integration.

4.9.4 User Management API

Für den Zugriff auf die Benutzerdaten steht der "Adressraum" "/user/*" zur Verfügung. Der Aufruf einer Funktion erfolgt genauso wie bei der low-level DB-API.

Die User-API bietet folgende Funktionen an:

- *getAllLabeler* - gibt alle zum Labeln berechtigte Benutzer zurück
- *getAllLabelManager* - gibt alle Benutzer zurück, die Label-Aufträge verwalten dürfen

- *getAllRolesForUser* - gibt alle Rollen für den angegebenen Benutzer zurück
- *canAccess** - eine Reihe von Funktionen, die prüfen, ob ein Benutzer Zugriff auf bestimmte Funktionalität hat

Die *canAccess**-Funktionen:

- *canAccessLabelJobManagement*
- *canAccessDetectionSystemManagement*
- *canAccessTestRunManagement*
- *canAccessMachineManagement*
- *canAccessLabelDocumentManagement*
- *canAccessDatabaseManagement*
- *canAccessSequenceManagement*

Diese Funktionen werden zum Beispiel vom Web-Interface benutzt um zu prüfen, ob ein Benutzer eine bestimmte Seite aufrufen darf.

4.9.5 WFMS-API

Die WFMS-API besteht derzeit aus einer einzigen Funktion: *sendMessageTo*. Mit dieser Funktion kann eine Nachricht an einen WfMS-Client geschickt werden. Wichtig ist dabei die Adressierung: "/wfms/sendMessageTo/USERNAME/CLIENTNUMMER", also zum Beispiel "/wfms/sendMessageTo/Johann Specht/1". Als Parameter erwartet *sendMessageTo* ein Dictionary mit zwei Einträgen: Typ der Nachricht und die Nachricht selbst. Beispiel:

Listing 4.12: Parameter für sendMessageTo

```
{'msgType'   :  'ERROR',
 'msg'       :  'Configuration file not found!'}
```

Neben “ERROR” gibt es noch “FAILED”, “TERMINATE” (zwingt einen WfMS-Client sich zu beenden), “RESULT” und “CMD” als Nachrichtentyp. Mit “CMD” können Kommandos wie “START_CLIENT” (wird intern benutzt) verschickt werden, “RESULT” wird angegeben wenn das Ergebnis eines Testlaufs eingeschickt wird.

4.9.6 Other-API

Für die Funktionalität, die keinem anderen Manager zugeordnet ist, steht der OtherManager zur Verfügung. Er bietet folgende vier Funktionen an:

- *getLavaConfigs* - gibt eine Liste mit allen bekannten Lava-Konfigurationen zurück
- *getLavaConfigurationByPath* - liefert die angeforderte Lava-Konfiguration
- *getLabelJobDocumentById* - generiert ein Label-Dokument für einen Label-Auftrag
- *getLabelDocumentByName* - liefert das angeforderte Label-Dokument

Kapitel 5

Modellierung der Daten und Prozesse

In diesem Kapitel wird es vor allem um die Modellierung der Daten (Kapitel 5.3) und der Prozesse (Kapitel 5.4) gehen, zuerst aber werden noch die Benutzerverwaltung (Kapitel 5.1) und die Zugriffsrechte (Kapitel 5.2) behandelt.

5.1 Benutzerverwaltung

Das als WfMS-Server eingesetzte ADEPT bringt bereits eine eigene Benutzerverwaltung mit, die in diesem Kapitel kurz vorgestellt werden soll.

Genau genommen bietet ADEPT mehr als nur Benutzerverwaltung. Man kann nämlich auch Organisationsstrukturen, Fähigkeiten, Rollen und Beziehungen zwischen Mitarbeitern verwalten. Die Abbildung 5.1 zeigt Entities der Benutzerverwaltung und deren Beziehungen untereinander.

Der Organisationstyp gibt an, um was für eine Organisationseinheit es sich handelt, einige Beispiele: “Abteilung” oder “Niederlassung”. Die Organisationseinheit ist dann eine bestimmte Ausprägung eines Organisationstyps, zum Beispiel “Abteilung Informationssysteme“ oder “Niederlassung Ulm”. Der Organisationstyp kann neben einem Namen noch über Rollen verfügen. Für eine Organisationseinheit kann eine Stelle angegeben werden, die den Leiter dieser Einheit definiert.

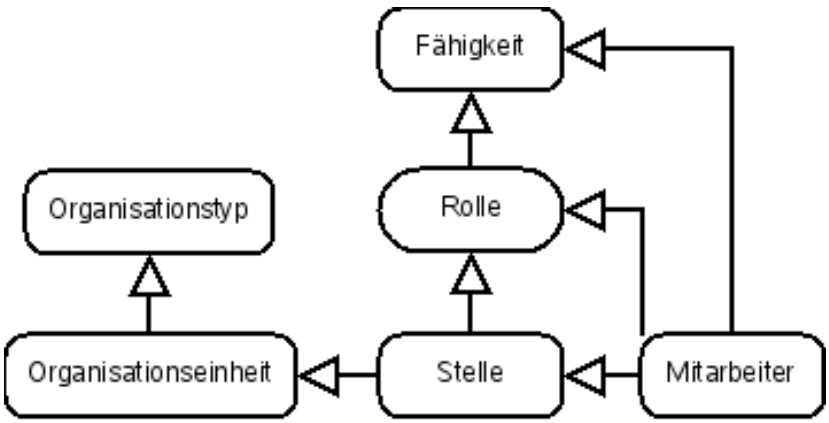

Abbildung 5.1: Organisation der Benutzerverwaltung.

Die nächst kleinere Einheit ist die Stelle. Eine Stelle kann einer Organisationseinheit zugeordnet werden, darüber hinaus kann eine Rolle angegeben werden. Eine Stelle kann einen disziplinarischen und einen fachlichen Vorgesetzten haben. Wichtig ist auch die Möglichkeit, eine Stelle mit einem Mitarbeiter zu besetzen oder von einem Mitarbeiter vertreten zu lassen.

Ein Mitarbeiter kann mehrere Fähigkeiten und Rollen besitzen und eine Stelle besetzen oder vertreten. Einer Rolle können mehrere Fähigkeiten zugeordnet werden, wobei die Zuweisung einer Rolle auch die entsprechenden Fähigkeiten vererbt.

Damit verfügt ADEPT bereits über eine recht leistungsfähige Benutzerverwaltung. Es fehlt allerdings die Möglichkeit, Projekte zu definieren und diesen Mitarbeiter zuzuordnen.

Im Kapitel 3.3.1 wurde bereits die Frage diskutiert, wie man auf der Kommandozeile feststellen kann, um welchen Benutzer es sich handelt. Dazu muss das System den UNIX-Namen des Benutzers auf den ADEPT-Namen abbilden können. Die sicherste Lösung wäre die Angabe des UNIX-Namens beim Anlegen des Benutzers in ADEPTOrgManager. Das einzige Attribut, das dafür in Frage kommt, ist wohl das Eingabefeld “Beschreibung des Mitarbeiters”, das eigentlich für einen Kommentar gedacht ist.

Eine vermutlich bessere Lösung ist die Nutzung des gleichen Namens für beide Namensräume, was auf den UNIX-Namen hinausläuft. Dazu gibt man im Feld

"Name" beim Anlegen eines neuen Benutzers den UNIX-Namen an und das Feld für den Vornamen bleibt leer.

5.2 Zugriffsrechte

Wie die Zugriffskontrolle für das Web-Interface realisiert wurde, ist bereits in Kapitel 4.8.2 besprochen worden. Hier soll die Zugriffskontrolle auf dem Level der Python-API thematisiert werden.

5.2.1 Low-level DB-API

Die Implementierung der low-level DB-API wurde im Kapitel 4.9.3 besprochen. Vor allem wurden dort die drei Möglichkeiten vorgestellt, auf die Daten zuzugreifen. Die gefährlichste Methode ist die Übergabe der SQL-Query an das System. Der Zugriff auf diese Methode sollte auf die Rolle "DatabaseManager" beschränkt werden.

Die zweite Möglichkeit auf die Daten zuzugreifen besteht darin, den Namen einer Datei mit der Query anzugeben. Hier kann der Zugriff sehr fein eingeschränkt werden, indem für jede Datei angegeben wird, welche Rollen oder Fähigkeiten ein Benutzer besitzen muss, um die Datei benutzen zu dürfen. Man könnte sogar einzelnen Benutzern Zugriff gewähren. Es läuft also auf ACL (Access Control List) hinaus. Die Zugriffsrechte können problemlos in der Datenbank gespeichert werden.

Das Gleiche gilt auch für die dritte Methode: Aufruf von Methoden der Klasse DatabaseManager. Nur sind hier die Methoden die Entities, für die Zugriffsrechte definiert werden.

5.2.2 High-level DB-API

Die high-level DB-API baut auf SQLObject ([SQLObj], siehe Kapitel 4.9.3) auf, welches direkt auf die Datenbank zugreift. Dieser direkte Zugriff stellt ein ernstes Problem dar, weil damit jeder Benutzer automatisch vollen Zugriff auf die Da-

ten bekommt. Da aber nicht auf SQLObject verzichtet werden kann, muss eine Lösung gefunden werden.

Eine Möglichkeit wäre eine Art Proxy, welcher zwischen SQLObject und PostgreSQL geschaltet wird und auf SQL-Ebene die Zugriffe filtert. Diese Lösung ist allerdings nicht praktikabel, da es viel zu aufwendig ist, auf SQL-Ebene zu filtern.

Als Kompromiss wurde folgende Lösung gewählt: SQLObject verwendet einen Benutzer, der nur lesenden Zugriff auf sämtliche Daten hat. Alle Operationen, die Schreibzugriff erfordern, werden als Methoden der entsprechenden Klassen implementiert, die den Weg über die low-level API gehen.

5.2.3 Benutzerverwaltungs-API

Der Zugriff auf die Benutzerverwaltungs-API (oder kurz User-API) ist auf das System beschränkt, da die Benutzer keinen Zugriff benötigen. Wenn allerdings in einer späteren Implementierung die Benutzerverwaltung in das Web-Interface und in die API integriert wird, muss der Zugriff auf die Funktionen der User-API kontrolliert werden. Dies kann auf die gleiche Art geschehen, wie bei der DB-API.

5.2.4 WFMS-API

Die einzige Funktion, die diese API zur Zeit bietet, ist *sendMessageTo*, mit der Nachrichten an einen WfMS-Client geschickt werden können. Das Versenden von Nachrichten sollte sinnvollerweise dem Benutzer erlaubt sein, dem dieser Client gehört. Das kann der WfMSManager leicht prüfen.

5.2.5 Other-API

Diese API bietet folgende vier Funktionen an:

- *getLavaConfigs*
- *getLavaConfigurationByPath*
- *getLabelJobDocumentById*

- *getLabelDocumentByName*

Die ersten beiden Funktionen sind nicht kritisch, da sie nur lesenden Zugriff bieten und keine sensitiven Daten liefern.

Die anderen beiden Funktionen bieten zwar ebenfalls nur lesenden Zugriff, aber es sollte trotzdem nicht jeder alle Label-Dokumente bzw. Label-Jobs lesen dürfen. Der Zugriff auf die Label-Jobs kann auf den Betreuer und den Bearbeiter beschränkt werden. Bei Label-Dokumenten ist es etwas komplizierter: Die Bearbeiter können auf jeden Fall lesenden Zugriff bekommen. Zusätzlich bekommt der jeweilige Betreuer auch Schreibzugriff genauso wie Benutzer mit der Rolle "LabelDocumentManager".

5.3 Datenverwaltung

Als erstes wird in diesem Kapitel beschrieben, nach welchen allgemeinen Regeln die Relationen angelegt wurden. Danach wird analysiert, welche Daten genau verwaltet werden müssen und wo sie am sinnvollsten untergebracht werden können. Und schließlich werden die Relationen vorgestellt.

5.3.1 SQL-Relationen

Da SQL nicht zwischen Groß- und Kleinschreibung unterscheidet, werden die Namen der Relationen und der Attribute klein geschrieben. Bei zusammengesetzten Namen wird zwischen den einzelnen Wörtern ein Unterstrich gesetzt: "sequence_channel" oder "sequence_channel_id". Nur englische Bezeichnungen werden benutzt. Es wird Wert darauf gelegt, Attribute oder Attribut-Kombinationen, deren Werte nur einmal pro Relation vorkommen können, mit "UNIQUE" zu kennzeichnen: "UNIQUE (category_id, sequence_id)". Damit werden mehrfache Vorkommen eines Wertes verhindert und der Datenbankserver kann eventuell zusätzliche Optimierungen vornehmen.

Bei schwachen Entities wird immer "ON DELETE CASCADE" für die Fremdschlüssel der übergeordneten Relationen angegeben, damit keine Inkonsistenzen entstehen. Bei sonstigen Fremdschlüsseln wird immer "ON DELETE SET NULL"

angegeben, damit immer entweder ein gültiger Fremdschlüssel oder NULL im entsprechenden Attribut steht.

Fremdschlüssel bekommen den Namen der Relation und des Attributs, auf den sie verweisen: “frame_id”. Jede Relation hat als primären Schlüssel das Attribut mit dem Namen “id” vom Typ “SERIAL” (wird automatisch hochgezählt). Es gibt somit keine zusammengesetzten Primärschlüssel. Dies wurde vor allem im Hinblick auf SQLObject so festgelegt (siehe 4.9.3).

Es wurde Wert auf die Einhaltung der ersten drei Normalformen ([ElNa02, Kapitel 14.3.2 bis 14.3.4]) gelegt (1NF, 2NF, 3NF) um Anomalien wie “Einfüge-Anomalie”, “Lösch-Anomalie” oder “Änderungs-Anomalie” und Inkonsistenzen zu vermeiden. Für die ersten zwei Normalformen ist dies auch gewährleistet, da die Relationen keine mehrwertigen Attribute haben und alle Primärschlüssel nicht zusammengesetzt sind. Die Einhaltung der 3NF wird an entsprechenden Stellen in diesem Kapitel diskutiert. An dieser Stelle noch kurz die Definition der “Transitiven Abhängigkeit” (siehe [ElNa02, Seite 524]), die benötigt wird, um die Einhaltung der 3NF zu beurteilen:
“ Eine funktionale Abhängigkeit $X \rightarrow Y$ in einem Relationsschema R ist eine transitive Abhängigkeit, wenn es eine Attributmenge Z enthält, die weder ein Kandidatenschlüssel noch eine Teilmenge eines Schlüssels von R ist, und wenn sowohl $X \rightarrow Z$ als auch $Z \rightarrow Y$ gilt. ” Beim Vorliegen einer transitiven Abhängigkeit ist eine Relation nicht in der dritten Normalform.

Für einige Attribute (wie Farbe, Priorität, Tag) wurden eigene Relationen angelegt, die die möglichen Werte aufnehmen. Die anderen Relationen verweisen über den Fremdschlüssel auf diese Werte. Damit soll erzwungen werden, dass nur dem System bekannte Werte angegeben werden können. Man kann zum Beispiel keine Priorität für einen Testlauf angeben, die nicht in der Tabelle “priority” existiert. Um das noch zu verschärfen, kann der Schreibzugriff auf diese Relationen eingeschränkt werden. Alle diese Tabellen bestehen nur aus zwei Attributen: Primärschlüssel “id” und “name” als String, damit sind sie in der 3NF. Der Nachteil dieser Lösung ist allerdings die erhöhte Komplexität der SQL-Queries, da zusätzliche Joins benötigt werden (wenn zum Beispiel die Farbe eines Labels im Ergebnistupel erscheinen soll), und somit auch eine längere Laufzeit. Da aber alle diese Tabellen nicht viele Tupels enthalten, fällt das nicht sehr ins Gewicht.

5.3.2 Bild-Sequenzen

Den mit Abstand größten Teil der Daten stellen die Bild-Sequenzen dar (siehe 2.1.1). Der eigentliche Inhalt, die Bilddaten der Tiff-Bilder, ist aus Sicht eines Datenbankmanagementsystems ein Strom von Binärdaten. Sinnvolle Abfragen dieser Daten mit SQL-Queries wären nur möglich, wenn man diese Daten interpretieren würde, was aber kein DBMS kann. Man müsste also vor dem Import der Bilddaten diese auswerten und dann in die Datenbank importieren. Dazu können die Tags, die in den Tiff-Bildern enthalten sein können, herausgelesen und in die DB geschrieben werden, was SQL-Abfragen die Möglichkeit geben würde, diese Tags abzufragen. Die eigentlichen Pixel-Daten könnten als BLOB in die DB geschrieben werden, was aber nicht sinnvoll ist, da Binärdaten nicht sinnvoll abgefragt werden können.

Als Kompromiss wurde eine gemischte Lösung gewählt: Die eigentlichen Bilder verbleiben im Dateisystem und Verwaltungsinformationen wandern in die DB. Die Tags wurden nicht ausgewertet, was aber in Zukunft durchaus nachgeholt werden kann.

Bild-Sequenzen können zu Gruppen zusammengefasst werden, wobei eine Sequenz mehreren Gruppen angehören und eine Gruppe mehrere Sequenzen enthalten kann (die Kardinalität ist also n:m). Beim Import von Bild-Sequenzen wird automatisch das sechsstellige Aufnahmedatum (siehe Abbildung 2.2) als Gruppe genommen. Die Gruppen (oder auch Kategorien, “group” ist ein reserviertes Wort in SQL) werden in der Tabelle “category” gespeichert (3NF):

Listing 5.1: Relation “category”

```
CREATE TABLE category (
id          SERIAL NOT NULL PRIMARY KEY,
name        VARCHAR(256) UNIQUE NOT NULL )
```

Die Zuordnung einer Bild-Sequenz zu Kategorien erfolgt in der Relation “sequence_category” (3NF):

Listing 5.2: Relation "sequence_category"

```
CREATE TABLE sequence_category (
id              SERIAL NOT NULL PRIMARY KEY,
category_id     INTEGER REFERENCES category(id)
                ON DELETE CASCADE,
sequence_id     INTEGER REFERENCES sequence(id)
                ON DELETE CASCADE,
UNIQUE (category_id, sequence_id) )
```

Das Paar "category_id, sequence_id" ist ein Kandidatenschlüssel und deswegen als unique markiert. Diese Relation ist in der 3NF, da es keine transitive Abhängigkeiten gibt.

Folgende Informationen sind bei Bild-Sequenzen wichtig: Name, Pfad und Zeitpunkt des Imports. Daraus ergibt sich eine einfache Tabelle:

Listing 5.3: Relation "sequence"

```
CREATE TABLE sequence (
id              SERIAL NOT NULL PRIMARY KEY,
name            VARCHAR(256) UNIQUE NOT NULL,
path            VARCHAR(1024),
imported        TIMESTAMP DEFAULT CURRENT_TIMESTAMP )
```

Wie man sehen kann kann der Zeitpunkt des Imports (Attribut "imported") automatisch von PostgreSQL gesetzt werden. Attribut "name" ist hier ein Schlüsselkandidat, da eindeutig. Diese Relation ist ebenfalls in der 3NF.

Die Kanäle der Bild-Sequenzen werden in der Relation "sequence_channel" gespeichert (in 3NF wie auch die restlichen Relationen in diesem Kapitel):

Listing 5.4: Relation "sequence_channel"

```
CREATE TABLE sequence_channel (
id              SERIAL NOT NULL PRIMARY KEY,
sequence_id     INTEGER REFERENCES sequence(id)
                ON DELETE CASCADE,
name            VARCHAR(256) NOT NULL,
UNIQUE(sequence_id, name) )
```

Die Kardinalität hier ist 1:n, da ein Kanal immer zu einer Sequenz gehört und eine Sequenz beliebig viele Kanäle enthalten kann. Das Paar "sequence_id, name" ist ein Kandidatenschlüssel und deswegen unique.

Die Informationen zu einzelnen Bildern/Frames werden in der Relation "frame" abgelegt (3NF):

Listing 5.5: Relation "frame"

```
CREATE TABLE frame (
id              SERIAL NOT NULL PRIMARY KEY,
name            VARCHAR(256) NOT NULL,
time_stamp      TIMESTAMP DEFAULT NULL )
```

Bisher wurde hier nur der Name des Frames und Zeitpunkt der Aufnahme (mikrosekunden genau) gespeichert. Letzterer wird beim Import einer Bild-Sequenz aus dem Frame-Namen extrahiert und kann später in Queries benutzt werden.

Die Zuordnung der Frames zu Kanälen erfolgt über die Relation "frame_to_channel" (3NF):

Listing 5.6: Relation "frame_to_channel"

```
CREATE TABLE frame_to_channel (
id                      SERIAL NOT NULL PRIMARY KEY,
frame_id                INTEGER REFERENCES frame(id)
                        ON DELETE CASCADE,
sequence_channel_id INTEGER
                        REFERENCES sequence_channel(id)
                        ON DELETE CASCADE,
UNIQUE (frame_id, sequence_channel_id) )
```

Der Grund für diese zusätzliche Relation, ist der Wunsch ein Frame auch mehreren Kanälen bzw. Sequenzen zuordnen zu können. Damit kann man zum Beispiel "virtuelle" Bild-Sequenzen zusammenstellen, die Frames mit bestimmten Eigenschaften enthalten. Daraus ergibt sich eine Kardinalität von n:m.

Darüber hinaus existieren noch einige Relationen, die die Tags und Kommentare für Bild-Sequenzen, Kanäle und Frames aufnehmen (Kardinalität 1:n). Es ist klar, dass Tags und Kommentare schwache Entities sind.

Listing 5.7: Relation "sequence_tag"

```
CREATE TABLE sequence_tag (
id              SERIAL NOT NULL PRIMARY KEY,
sequence_id     INTEGER REFERENCES sequence(id)
                ON DELETE CASCADE,
tag_id          INTEGER REFERENCES tag(id)
                ON DELETE CASCADE,
UNIQUE (sequence_id, tag_id) )
```

Listing 5.8: Relation "sequence_comment"

```
CREATE TABLE sequence_comment (
id              SERIAL NOT NULL PRIMARY KEY,
sequence_id     INTEGER REFERENCES sequence(id)
                ON DELETE CASCADE,
comment         VARCHAR(1024) )
```

Die nachfolgenden Relationen enthalten einen Fremdschlüssel mit dem Namen "label_document_id". Der Grund dafür ist, dass die Tags und Kommentare für Kanäle und Frames aus Label-Dokumenten stammen können und somit die Notwendigkeit besteht, das entsprechende Label-Dokument zu referenzieren. Die Kardinalität hier ist ebenfalls 1:n (Label-Dokument kann höchstens einmal angegeben werden).

Listing 5.9: Relation "sequence_channel_tag"

```
CREATE TABLE sequence_channel_tag (
id                      SERIAL NOT NULL PRIMARY KEY,
sequence_channel_id     INTEGER
                        REFERENCES sequence_channel(id)
                        ON DELETE CASCADE,
tag_id                  INTEGER REFERENCES tag(id)
                        ON DELETE CASCADE,
label_document_id       INTEGER REFERENCES label_document(id)
                        ON DELETE SET NULL,
UNIQUE (sequence_channel_id, label_document_id, tag_id) )
```

Listing 5.10: Relation "sequence_channel_comment"

```
CREATE TABLE sequence_channel_comment (
id                      SERIAL NOT NULL PRIMARY KEY,
sequence_channel_id     INTEGER
                        REFERENCES sequence_channel(id)
                        ON DELETE CASCADE,
label_document_id       INTEGER REFERENCES label_document(id)
                        ON DELETE SET NULL,
comment VARCHAR(1024) )
```

Listing 5.11: Relation "frame_tag"

```
CREATE TABLE frame_tag (
id                      SERIAL NOT NULL PRIMARY KEY,
frame_id                INTEGER REFERENCES frame(id)
                        ON DELETE CASCADE,
label_document_id       INTEGER REFERENCES label_document(id)
                        ON SET NULL,
tag_id                  INTEGER REFERENCES tag(id)
                        ON DELETE CASCADE,
UNIQUE (frame_id, label_document_id, tag_id) )
```

Listing 5.12: Relation "frame_comment"

```
CREATE TABLE frame_comment (
id                      SERIAL NOT NULL PRIMARY KEY,
frame_id                INTEGER REFERENCES frame(id)
                        ON DELETE CASCADE,
label_document_id       INTEGER REFERENCES label_document(id)
                        ON DELETE SET NULL,
comment                 VARCHAR(1024) )
```

5.3.3 Label-Dokumente, Label

Für die Speicherung der Label-Dokumente ist die Relation "label_document" vorgesehen:

Listing 5.13: Relation "label_document"

```
CREATE TABLE label_document (
id                          SERIAL NOT NULL PRIMARY KEY,
name                        VARCHAR(256),
sequence_channel_id         INTEGER REFERENCES
                            sequence_channel(id)
                            ON DELETE SET NULL,
cfg_file_md5                CHAR(32) NULL,
cfg_file_name               VARCHAR(256) NULL,
cfg_file_path               VARCHAR(1024) NULL,
prev_label_file_md5         CHAR(32) NULL,
prev_label_file_name        VARCHAR(256) NULL,
prev_label_file_path        VARCHAR(1024) NULL,
user_name                   VARCHAR(64) NULL,
app_version                 INTEGER NULL,
app_name                    VARCHAR(32) NULL,
format_version              INTEGER NULL,
format_name                 VARCHAR(32) NULL,
document_date               TIMESTAMP
                            DEFAULT CURRENT_TIMESTAMP )
```

Wie man sieht wird ein Label-Dokument nicht als schwache Entity (abhängig vom Kanal) gespeichert. Damit wird verhindert, dass beim Löschen der zugehörigen Bild-Sequenz/Kanals das Label-Dokument auch gelöscht wird. Das macht insofern Sinn, als dass die Bild-Sequenz noch im Dateisystem vorhanden sein kann und Label-Dokumente nicht ungefragt gelöscht werden dürfen. Das Attribut "document_date" nimmt das Datum, welches im Label-Dokument gespeichert ist, auf. Es enthält also nicht den Zeitpunkt des Imports.

Die Label-Daten werden in der Relation "label" gespeichert:

Listing 5.14: Relation "label"

```
CREATE TABLE label (
id                  SERIAL NOT NULL PRIMARY KEY,
parent_id           INTEGER REFERENCES label(id)
                    ON DELETE CASCADE,
track_id            VARCHAR(64) NOT NULL,
label_document_id   INTEGER REFERENCES label_document(id)
                    ON DELETE CASCADE,
frame_id            INTEGER REFERENCES frame(id)
                    ON DELETE SET NULL,
label_type_id       INTEGER REFERENCES label_type(id)
                    ON DELETE SET NULL,
color_id            INTEGER REFERENCES color(id)
                    ON DELETE SET NULL,
point_data          POINT DEFAULT NULL,
box_data            BOX DEFAULT NULL,
path_data           PATH DEFAULT NULL,
polygon_data        POLYGON DEFAULT NULL )
```

Da ein Label-Dokument beliebig viele Label enthalten kann und ein Label immer zu einem Label-Dokument gehört, ergibt sich eine Kardinalität von 1:n.

Wichtig bei dieser Relation sind vor allem die vier Attribute "point_data", "box_data", "path_data" und "polygon_data", die alle der Aufnahme der geometrischen Daten dienen. Eigentlich würde dafür ein Attribut reichen (wenn man "POLYGON" als Datentyp nimmt), allerdings würde man dabei die Label alle gleich behandeln, unabhängig davon ob es Rechtecke, Punkte oder Polygone sind. PostgreSQL bietet aber für alle diese Datentypen spezifische Operationen und Funktionen an. Zum Beispiel kann mit dem Operator "&&" geprüft werden, ob sich zwei Rechtecke schneiden. Mit *area(object)* kann die Fläche eines Rechtecks berechnet werden. Aus diesem Grund gibt es vier Attribute, die folgendermaßen benutzt werden:

- "point_data" für Punkte
- "box_data" für Rechtecke

- "path_data" für offene Polygone und Freihandwerkzeug
- "polygon_data" für geschlossene Polygone

Diese Unterscheidung erschwert leider das Schreiben von Queries, da man bei jedem Tupel wissen muss, welches dieser vier Attribute das richtige ist. Es ermöglicht aber die Nutzung des gesamten Vorrats an Operatoren und Funktionen. Um festzustellen von welchem Typ ein Label ist, kann das Attribut "label_type_id" abgefragt werden.

Auch Label-Dokumente und Label können mit Tags und Kommentaren versehen sein (beides steht im Label-Dokument), welche in diesen Relationen gespeichert werden (schwache Entities, 1:n, 3NF):

Listing 5.15: Relation "label_document_tag"

```
CREATE TABLE label_document_tag (
id                      SERIAL NOT NULL PRIMARY KEY,
label_document_id       INTEGER REFERENCES label_document(id)
                        ON DELETE CASCADE,
tag_id                  INTEGER REFERENCES tag(id)
                        ON DELETE CASCADE,
UNIQUE (label_document_id, tag_id) )
```

Listing 5.16: Relation "label_document_comment"

```
CREATE TABLE label_document_comment (
id                      SERIAL NOT NULL PRIMARY KEY,
label_document_id       INTEGER REFERENCES label_document(id)
                        ON DELETE CASCADE,
comment                 VARCHAR(1024) )
```

Listing 5.17: Relation "label_tag"

```
CREATE TABLE label_tag (
id              SERIAL NOT NULL PRIMARY KEY,
label_id        INTEGER REFERENCES label(id)
                ON DELETE CASCADE,
tag_id          INTEGER REFERENCES tag(id)
                ON DELETE CASCADE,
UNIQUE (label_id, tag_id) )
```

Listing 5.18: Relation "label_comment"

```
CREATE TABLE label_comment (
id              SERIAL NOT NULL PRIMARY KEY,
label_id        INTEGER REFERENCES label(id)
                ON DELETE CASCADE,
comment         VARCHAR(1024) )
```

Alle vier Relationen enthalten schwache Entities, die automatisch gelöscht werden, wenn das zugehörige Label-Dokument oder Label gelöscht wird.

5.3.4 Label-Aufträge

Für die Speicherung der Label-Aufträge (oder auch Label-Jobs) ist die Relation "label_job" vorgesehen:

Listing 5.19: Relation "label_job"

```
CREATE TABLE label_job (
id SERIAL NOT NULL PRIMARY KEY,
sequence_channel_id INTEGER
                    REFERENCES sequence_channel(id)
                    ON DELETE SET NULL,
labeler             VARCHAR(256) NOT NULL,
manager             VARCHAR(256) NOT NULL,
lava_configuration  VARCHAR(2048),
deadline            DATE,
priority_id         INTEGER REFERENCES priority(id)
                    ON DELETE SET NULL,
job_state_id        INTEGER REFERENCES job_state(id)
                    ON DELETE SET NULL,
comment             VARCHAR(4096),
label_document_id   INTEGER REFERENCES label_document(id)
                    ON DELETE SET NULL )
```

Ein Label-Auftrag bezieht sich immer auf genau ein Kanal und für ein Kanal können mehrere Label-Aufträge existieren, es ist also eine 1:n Beziehung.

Das Attribut "labeler" nimmt den Namen des Bearbeiters auf und das Attribut "manager" den Namen des Betreuers. Das Attribut "deadline" gibt an, bis wann der Bearbeiter fertig sein muss. Zusätzlich kann ein Auftrag priorisiert werden ("priority_id"). Der Status eines Auftrags wird im Attribut "'job_state_id" gespeichert. Wenn der Bearbeiter das Ergebnis einschickt, wird die ID des entsprechenden Label-Dokuments im Attribut "label_document_id" gespeichert.

5.3.5 Erkennungssysteme

Im Kapitel 2 wurde bereits erwähnt, dass ein Erkennungssystem durch eine ausführbare Datei und eine/mehrere Konfigurationsdateien definiert wird. Erstere enthält Binärcode und ist somit aus Sicht eines DBMS ein BLOB. Die Konfigurationsdateien haben leider keine einheitliche Struktur und können deswegen schlecht auf Relationen abgebildet werden. Aus diesem Grund werden sowohl die ausführbaren Dateien als auch die Konfigurationsdateien nicht direkt in der

Datenbank abgelegt. Stattdessen wird für jedes Erkennungssystem ein Unterverzeichnis in einem System-Verzeichnis (ist über die Konfigurationsdatei "sql/sql.cfg"einstellbar) erstellt. Anschließend werden die Dateien dorthin kopiert und MD5-Summen für diese berechnet. Am Schluss werden die Namen der Dateien zusammen mit den MD5-Summen in der Datenbank abgelegt.

Die MD5-Summen sind dazu da, Änderungen an den Dateien festzustellen. Die Notwendigkeit dafür leitet sich aus der Feststellung ab (Kapitel 2), dass jede Änderung eines Bestandteils eines Erkennungssystems ein neues Erkennungssystem hervorbringt. Falls also eine solche Änderung festgestellt wird , kann sie vom System gemeldet werden.

Für die Speicherung der Erkennungssysteme sind folgende Relationen vorgesehen (dsystem steht für "Detection System"):

Listing 5.20: Relation "dsystem"

```
CREATE TABLE dsystem (
id                  SERIAL NOT NULL PRIMARY KEY,
name                VARCHAR(256) UNIQUE,
path                VARCHAR(1024),
executable_name     VARCHAR(1024),
executable_md5      VARCHAR(32),
description         VARCHAR(1024),
imported            TIMESTAMP DEFAULT CURRENT_TIMESTAMP )
```

Listing 5.21: Relation "dsystem_config_file"

```
CREATE TABLE dsystem_config_file (
id          SERIAL NOT NULL PRIMARY KEY,
name        VARCHAR(64) NOT NULL,
path        VARCHAR(1024),
md5         VARCHAR(32),
dsystem_id  INTEGER REFERENCES dsystem(id)
            ON DELETE CASCADE,
UNIQUE(name, dsystem_id) )
```

Da ein Erkennungssystem mehrere Konfigurationsdateien haben kann und eine Konfigurationsdatei zu genau einem Erkennungssystem gehört, ergibt sich eine Kardinalität von 1:n.

Jedes Erkennungssystem bekommt einen eindeutigen Namen. Im Attribut “path” wird der vollständige Pfad zu dem Verzeichnis gespeichert, in dem sich die Dateien befinden. “executable_name” gibt den Namen der ausführbaren Datei an und “executable_md5” die MD5-Summe.

Die Informationen über die Konfigurationsdateien eines Erkennungssystems werden in der Tabelle “dsystem_config_file” als schwache Entities gespeichert.

5.3.6 Testläufe

Die Daten, welche die Testläufe definieren, werden ebenfalls in der Datenbank gespeichert. Ein Testlauf wird durch folgende Angaben definiert:

- Name (eindeutig)
- Erkennungssystem (ID)
- Startzeit
- Priorität
- Status
- Beschreibung
- Bild-Sequenzen
- Maschinen auf denen es ausgeführt werden kann

Darüber hinaus können Nachrichten (Messages), die ein Testlauf verschickt, gespeichert werden.

Die Maschinen werden in einer eigenen Relation verwaltet:

Listing 5.22: Relation "machine"

```
CREATE TABLE machine (
id                  SERIAL NOT NULL PRIMARY KEY,
name                VARCHAR(256) UNIQUE NOT NULL,
machine_state_id    INTEGER REFERENCES machine_state(id)
                    ON DELETE SET NULL )
```

Listing 5.23: Relation "machine_state"

```
CREATE TABLE machine_state (
id      SERIAL NOT NULL PRIMARY KEY,
name    VARCHAR(32) UNIQUE NOT NULL )
```

Mit "machine_state_id" wird der Status der Maschine angegeben: "frei", "reserviert", "belegt".

Die zentrale Relation bei den Testläufen ist "test_run":

Listing 5.24: Relation "test_run"

```
CREATE TABLE test_run(
id                  SERIAL NOT NULL PRIMARY KEY,
name                VARCHAR(64) UNIQUE,
dsystem_id          INTEGER REFERENCES dsystem(id)
                    ON DELETE SET NULL,
start_time          TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
priority_id         INTEGER REFERENCES priority(id)
                    ON DELETE SET NULL,
test_run_state_id   INTEGER REFERENCES test_run_state(id)
                    ON DELETE SET NULL,
description         VARCHAR(1024) )
```

Da für einen Testlauf mehrere Maschinen angegeben werden können (von denen eine ausgewählt wird), wird eine weitere Relation benötigt:

Listing 5.25: Relation "test_run_to_machine"

```
CREATE TABLE test_run_to_machine (
id              SERIAL NOT NULL PRIMARY KEY,
test_run_id     INTEGER REFERENCES test_run(id)
                ON DELETE CASCADE,
machine_id      INTEGER REFERENCES machine(id)
                ON DELETE CASCADE,
UNIQUE (test_run_id, machine_id) )
```

Eine Maschine kann durchaus bei mehreren Testläufen gleichzeitig angegeben werden (also eine n:m Beziehung), die Testläufe "konkurieren" dann um diese Maschine.

Ähnlich ist es bei den Bild-Sequenzen: Ein Testlauf kann mehrere Sequenzen verarbeiten und eine Sequenz kann von mehreren Testläufen verarbeitet werden. Also wird eine weitere (n:m) Relation benötigt:

Listing 5.26: Relation "test_run_channel"

```
CREATE TABLE test_run_channel (
id                      SERIAL NOT NULL PRIMARY KEY,
test_run_id             INTEGER REFERENCES test_run(id)
                        ON DELETE CASCADE,
sequence_channel_id     INTEGER
                        REFERENCES sequence_channel(id)
                        ON DELETE CASCADE,
UNIQUE (test_run_id, sequence_channel_id,
        label_document_id) )
```

Bleiben noch die Nachrichten. Eine Nachricht gehört immer zu einem Testlauf und dieser kann beliebig viele Nachrichten haben, damit ergibt sich eine 1:n Beziehung:

Listing 5.27: Relation "test_run_message"

```
CREATE TABLE test_run_message (
id              SERIAL NOT NULL PRIMARY KEY,
test_run_id     INTEGER REFERENCES test_run(id)
                ON DELETE CASCADE,
message         VARCHAR(1024) )
```

5.4 Workflow

Die Abbildung 5.2 zeigt die Prozess-Vorlage für das Ausführen von Testläufen (ohne die leeren Knoten für die Verzweigung). Diese Vorlage ist bisher einfach gehalten, kann aber durchaus noch sinnvoll erweitert werden, indem zum Beispiel der Empfang der Nachrichten in einer Schleife erfolgt um mehr als eine Nachricht empfangen zu können. Die durchgezogenen Pfeile zeigen den Prozess-Fluss und die gestrichelten Pfeile den Datenfluss. Die Aktivität "GetMachine" holt die Liste

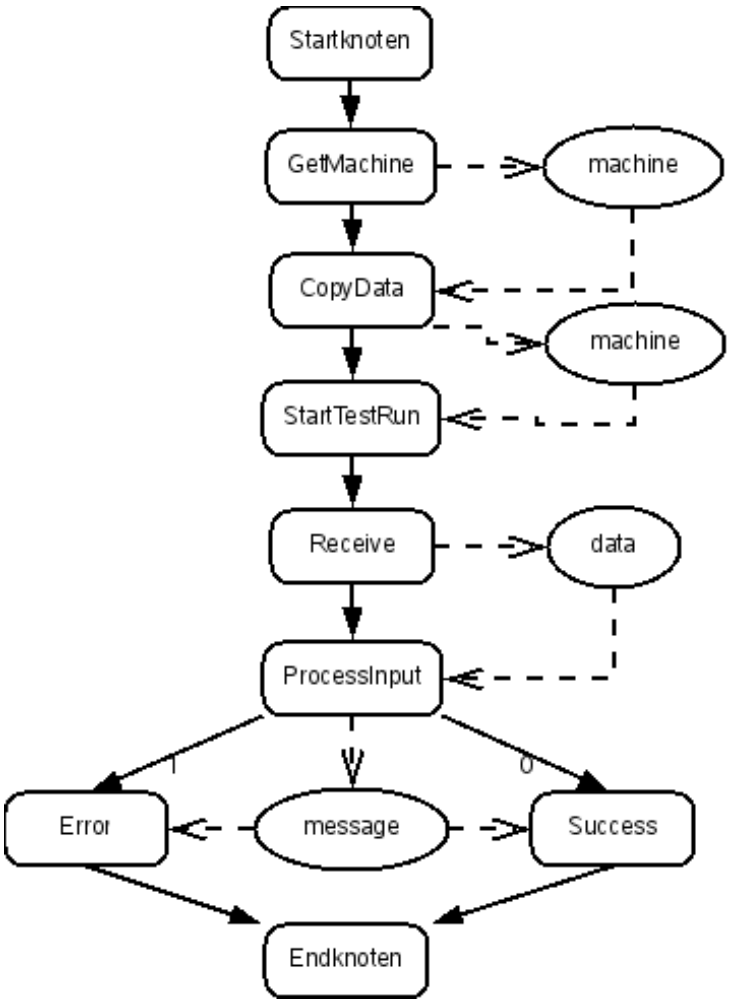

Abbildung 5.2: Implementierung der Testläufe.

der Maschinen für diesen Testlauf, sucht aus dieser eine aus, die nicht belegt ist, und gibt diese als Ausgabeparameter "machine" aus. Zusätzlich wird die freie Maschine nun als belegt markiert. Das sollte nicht in einer eigenen Aktivität

passieren, da sonst Mehrfachbelegung wahrscheinlicher wird (sollte also möglichst atomar passieren). Diese Aktivität kehrt erst zurück, wenn eine freie Maschine gefunden wurde. Wenn noch keine Maschine verfügbar ist, wird der Status des Testlaufs in "not ready" geändert. Sobald eine Maschine zur Verfügung steht, ändert sich der Status in "ready".

Nachdem nun eine freie Maschine gefunden wurde, können die benötigten Daten auf diese Maschine kopiert werden: "CopyData". Dieser Kopiervorgang erfolgt mit Hilfe des Kommandos "scp" (siehe Kapitel 4.2.4). Den Namen der Maschine gibt CopyData wieder als Ausgabeparameter weiter. Vor dem Kopieren wird der Status des Testlaufs in "starting" geändert.

Die nächste Aktivität, "StartTestRun", startet nun den Testlauf (mit Hilfe von SSH siehe Kapitel 4.2.4) und ändert den Status des Testlaufs in "running".

Die Aktivität "Receive" wartet (blockierend) auf eine Nachricht. Sobald eine Nachricht eintrifft, wird der Inhalt in den Ausgabeparameter "data" geschrieben.

Für die Auswertung der Nachricht ist die Aktivität "ProcessInput" zuständig. Falls der Typ der Nachricht "ERROR" lautet, wird zur Aktivität "Error" verzweigt, die die Nachricht in die Datenbank schreibt und den Status des Testlaufs in "error" ändert. Ansonsten wird die Aktivität "Success" ausgeführt, die das Ergebnis (in der Nachricht enthalten) in die Datenbank schreibt und den Status in "done" ändert.

Neben Testläufen wurden leider keine weiteren Prozesse mit ADEPT abgewickelt. Der Import der Bild-Sequenzen erfolgt über die Methode *importFromDirectory* der Klasse "Sequence" der DB-API. Für den Import von Label-Dokumenten existiert mit "sql/importLabels.py" ein Kommandozeilenwerkzeug.

Kapitel 6

Diskussion

In den Kapiteln 3, 5 und 4 wurden mögliche Lösungen besprochen und das implementierte System vorgestellt. In diesem Kapitel soll das Ergebnis dieser Arbeit im Hinblick auf Vollständigkeit und Richtigkeit der getroffenen Entscheidungen diskutiert werden.

Eines der ausgereiftesten Teile des Gesamtsystems ist die Verwaltung der Daten. Sämtliche Daten (abgesehen von den Teilen, die im Dateisystem verbleiben) werden in der Datenbank verwaltet und über die DB-API (Kapitel 4.9.3) zur Verfügung gestellt. Was hier vor allem noch fehlt, ist die Zugriffskontrolle. Diese ist nur bei der SQLObject-Integration teilweise gegeben und sollte noch bei Schreibzugriffen implementiert werden. Dies ist auch gleich der nächste Punkt: Die Integration von SQLObject ist noch nicht vollständig, es fehlen vor allem noch die Methoden für den Schreibzugriff. Ansonsten deckt SQLObject sämtliche Relationen ab. Die Nutzung von SQLObject bringt außerdem noch ein anderes Problem mit sich: fehlende Unterstützung der geometrischen Datentypen. Um geometrische Daten zu verarbeiten, muss man sich zur Zeit noch auf die SQL-Ebene begeben. Es ist allerdings möglich, dass eine künftige Version von SQLObject auch geometrische Datentypen unterstützt.

PostgreSQL hat sich als ein ausgereiftes und schnelles DBMS mit guter Anbindung an Python herausgestellt. SQLObject ist ein vielversprechender, objektrelationaler Wrapper mit beachtlichem Leistungsumfang und guter Dokumentation.

Die Workflow-Funktionalität ist noch nicht vollständig, von den vorhandenen Prozessen wird einzig die Ausführung eines Testlaufs vom Workflow-Server bewerkstelligt. Auch andere Prozesse, wie Import von Label-Dokumenten oder Bild-Sequenzen, müssen noch integriert werden. Abgesehen von der Ausführung der Prozesse wird auch die Benutzerverwaltung des Workflow-Servers (ADEPT 1) benutzt, die der Aufgabenstellung sehr nahe kommt. Nur die Unterstützung von Projekten fehlt noch.

Der Eindruck von ADEPT 1 als Workflow-Server fällt unterschiedlich aus, was aber vor allem seinem Prototypen-Status zuzuschreiben ist. Problematisch erwies sich die teilweise sehr knapp gehaltene Dokumentation der ADEPT-API. Auch der Betrieb von ADEPT unter Linux führte zu Komplikationen. Zwar läuft der ADEPT-Server unter Linux, aber die GUI-Werkzeuge von ADEPT können sich nicht zum Server verbinden wenn sie unter Windows laufen. Und unter Linux funktionieren diese Werkzeuge nicht richtig - sie starten zwar, lassen sich aber nicht richtig bedienen (zum Beispiel sind die Dialoge auf wenige Bildpunkte reduziert und lassen sich nicht vergrößern). Da sie aber in Java geschrieben sind sollte sich der Aufwand für eine Portierung auf Linux in Grenzen halten. Auch eine C oder C++ API wäre wünschenswert, da man damit leicht Script-Sprachen, wie Python, unterstützen kann. Und die Fähigkeit, externe Anwendungen zu starten, ist bei ADEPT ebenfalls noch ausbaufähig (siehe dazu Kapitel 4.6.4).

Positiv ist dagegen die Benutzerverwaltung aufgefallen (siehe oben). Auch die Fähigkeit von ADEPT, während der Ausführung die Prozess-Abläufe ändern zu können, kann später sehr nützlich werden (wurde aber aus Zeitgründen noch nicht im System integriert). Insgesamt kann man sagen, dass die Entscheidung für ADEPT durchaus richtig war, zumal mit ADEPT 2 in Zukunft eine weiterentwickelte Version zur Verfügung stehen wird.

Das Web-Interface bietet bereits für die meisten Aufgaben entsprechende Seiten an und schützt diese auch vor unerlaubtem Zugriff. Diese Seiten sehen allerdings noch sehr schlicht aus und es fehlt ein einheitliches Look-and-Feel. Beides kann später mit Hilfe von CSS - Cascading Style Sheets - nachgeholt werden. Neben der Unvollständigkeit und dem schlichten Aussehen ist noch die mangelnde Fehelertoleranz bei Benutzereingaben zu erwähnen (zum Beispiel werden Datumsangaben nicht auf ihre Korrektheit hin geprüft). Dagegen werden bereits Fehler, die vom System gemeldet werden, in dem Web-Interface angezeigt.

Techniken wie PSP und Session Management von mod_python haben gute Dienste geleistet und stützen somit die Entscheidung mod_python und somit auch Apache HTTP-Server einzusetzen. Ein Verzicht auf Apache (Python bringt bereits einen HTTP-Server mit) würde zwar die Architektur des Systems vereinfachen, dafür müsste man aber andere Lösungen für das Session Management und die Webseiten-Vorlagen finden und integrieren.

Ein Wegfall von Apache HTTP-Server würde auch den Glue-Server nicht überflüssig machen, da dieser nicht nur mit mod_python Scripten kommuniziert, sondern auch eine Schnittstelle für die API bietet und die Workflow-Clients verwaltet. Somit ist weiterhin die Berechtigung für den Einsatz von XML-RPC gegeben: Dieses ist das Kommunikationsmedium für die API (die über Rechnergrenzen hinweg funktionieren muss). XML-RPC hat sich auch im Einsatz bewährt: Es gab keine Auffälligkeiten oder Probleme.

Die Entscheidung über UNIX Message Queues mit den Workflow-Clients zu kommunizieren hat sich ebenfalls als richtig erwiesen (was natürlich nicht heißt, dass es keine besseren Lösungen gibt): Es hat bis jetzt zuverlässig und schnell funktioniert. Was allerdings noch verbesserungswürdig ist, ist die Fehlertoleranz: Bei einem Absturz verbleiben nicht abgeholte Nachrichten in der Warteschlange und können später fälschlicherweise von anderen Clients abgeholt werden (die Adressen beginnen nach einem Neustart wieder bei 1). Außerdem unterstützt die aktuelle Implementierung den Versand von Nachrichten an “alle” (Adresse 0) noch nicht.

Die Benutzerverwaltung ist dank ADEPT bereits recht ausgereift, es fehlt nur die Unterstützung für Projekte. Zur Zeit wertet die API den Benutzernamen nicht aus und bietet somit noch keinen Zugriffsschutz. Mögliche Lösungen dafür wurden allerdings im Kapitel 5.1 diskutiert.

Die API bietet bereits Zugriff auf die meisten Daten/Funktionen. Sie ist allerdings, abgesehen von der DB-API, noch nicht objectorientiert. Dies sollte eine künftige Version nachholen, indem alle System-Komponenten, die über die API zugänglich sein sollen, als Klassen in der API abgebildet werden.

Kapitel 7

Zusammenfassung

Das Ziel dieser Arbeit war die Konzeption und die Implementierung eines Workflow Management Systems zur Entwicklung und Qualitätssicherung von Erkennungssystemen. Neben dem Workflow muss das System auch große Datenbestände verwalten, ein Web-basiertes Benutzerinterface bieten und über eine Programmierschnittstelle (API) zugänglich sein. Zusätzlich sollte auch Zugriffsschutz gewährleistet sein.

Dafür wurde der Prototyp des Workflow Management Systems ADEPT 1 eingesetzt, welcher somit den Rahmen vorgab. Die Notation für die Beschreibung der Prozesse, die Werkzeuge für das Erstellen von Prozess-Vorlagen und die Verwaltung der Benutzer, sowie die Benutzerverwaltung wurden von ADEPT übernommen.

Um Prozess-Instanzen ausführen zu können, wurde ein eigener Client in Java geschrieben, der über die ADEPT-API mit dem ADEPT-Server interagiert, wobei die eigentliche Arbeit externe Python-Scripte erledigen, die von dem Client aufgerufen werden.

Die Datenverwaltung übernimmt das Datenbankmanagementsystem PostgreSQL, welches frei verfügbar und leistungsfähig ist. Die Daten werden über eine objektorientierte API zugänglich gemacht, wobei auch die Möglichkeit besteht über SQL-Anfragen die Daten zu benutzen.

Dem Web-basierten Benutzerinterface liegen der HTTP-Server Apache und mod_python zugrunde. Letzteres ist für die Generierung der Web-Seiten zuständig und kann über die API mit dem Rest des Systems kommunizieren.

Teilweise wurde auch der Zugriffsschutz realisiert. Am vollständigsten ist dieser beim Benutzerinterface, zum Teil wurde er auch beim Zugrif auf die Daten über die API realisiert.

Insgesamt ist das umgesetzte System noch in einem Prototypen-Stadium.

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Literaturverzeichnis

[ElNa02] Ramez Elmasri, Shamkant B. Navathe: *Grundlagen von Datenbanksystemen*. Dritte Auflage, Pearson Studium, 2002

[Wenz07] Christian Wenz: JavaScript und AJAX. Siebte Auflage, Galileo Computing, 2007

[Lubk06] Mark Lubkowitz: *Webseiten programmieren und gestalten*. Zweite Auflage, Galileo Computing, 2006

[Lutz06] Mark Lutz: *Programming Python*. Third Edition, O'Reilly, 2006

[Krug02] Guido Krüger: *Handbuch der Java-Programmierung*. Dritte Auflage, Addison-Wesley, 2002

[Hero99] Helmut Herold: *Linux-Unix Systemprogrammierung*. Zweite Auflage, Addison-Wesley, 1999

[Statis] Statistisches Landesamt Baden-Württemberg, Stuttgart: *Straßenverkehrsunfälle und dabei verunglückte Personen in Baden-Württemberg seit 1970*. http://www.statistik-bw.de/UmweltVerkehr/Landesdaten/MUnfaelle.asp. [Stand: 09.05.2007].

[Daimler] DaimlerChrysler: *Homepage*. http://www.daimlerchrysler.com/dccom_de

[SuSELi] Novell: *SUSE Linux 9.3*. http://www.novell.com/de-de/products/linuxprofessional. [Stand: 09.05.2007].

[openSUSE] openSUSE: *openSUSE 10.2*. http://de.opensuse.org/Stabile_Version. [Stand: 09.05.2007])

[Python] Python: *Release 2.4.4*. http://www.python.org/download/releases/2.4.4

[GNUGCC] GCC - GNU Compiler Collection: *Release 3.3.5*. http://gcc.gnu.org/gcc-3.3

[Java13] Java: *Release 1.3.1*. http://java.sun.com/j2se/1.3/download.html

[ADEPT] Universität Ulm Institut für Datenbanken und Informationssysteme: *ADEPT*. http://www.informatik.uni-ulm.de/dbis. [Stand: 09.05.2007].

[Apache] Apache: *Release 2.2.4*. http://httpd.apache.org

[modpy] mod_python: *Homepage*. http://www.modpython.org. [Stand: 09.05.2007].

[W3CDOM] W3C: *DOM - Document Object Model*. http://www.w3.org/DOM. [Stand: 09.05.2007].

[MochiKit] MochiKit: *Homepage*. http://mochikit.com. [Stand: 09.05.2007].

[Scons] SCons - Open Source software construction tool: *Homepage*. http://www.scons.org. [Stand: 09.05.2007].

[SWIG] SWIG - Simplified Wrapper and Interface Generator: *Homepage*. http://www.swig.org. [Stand: 09.05.2007].

[Dia] live.gnome.org: *Dia - Illustrationsprogramm und UML-Werkzeug*. http://live.gnome.org/Dia

[MqWork] IBM: *IBM WebSphere MQ Workflow*. http://www-306.ibm.com/software/integration/wmqwf

[Staffware] Staffware: *Homepage*. http://www.staffware.com

[TIFF] Adobe Systems: *TIFF 6.0 Specification*. http://partners.adobe.com/public/developer/tiff/index.html

[OpenSSH] OpenSSH: *Homepage*. http://www.openssh.org

[RMI] Sun Developer Network: *RMI - Remote Method Invocation*. http://java.sun.com/javase/technologies/core/basic/rmi/index.jsp

[Jithon] The Jython Project: *Homepage*. http://www.jython.org/Project/index.html

[JPype] JPype: *Homepage*. http://jpype.sourceforge.net

[JPypeEx] JPype: *Anwendungsbeispiel*. http://jpype.sourceforge.net/doc/user-guide/userguide.html#using

[JNI] Sun Microsystems: *JNI - Java native Interface*. http://java.sun.com/j2se/1.4.2/docs/guide/jni/index.html

[RPC] IETF: *Remote Procedure Call Protocol Specification Version 2*. http://tools.ietf.org/html/rfc1831

[SOAP] W3C: *SOAP Specifications*. http://www.w3.org/TR/soap

[twissp] Twisted: *Twisted Spread*. http://twistedmatrix.com/projects/core/documentation/howto/pb.html. [Stand: 09.05.2007].

[XMLRPC] XML-RPC: *Homepage*. http://www.xmlrpc.com

[PyXMLRPC] Python Dokumentation: *XML-RPC*. http://www.python.org/doc/2.4/lib/module-xmlrpclib.html

[ThXMLRPC] ActiveState Programmer Network: *Simple Threaded XML-RPC Server*. http://aspn.activestate.com/ASPN/Cookbook/Python/Recipe/425043. [Stand: 09.05.2007].

[JavaSc] Ecma International: *ECMAScript Language Specification*. http://www.ecma-international.org/publications/files/ecma-st/ECMA-262.pdf. [Stand: 09.05.2007].

[AJAX] Mozilla Developer Center: *AJAX - Asynchronous JavaScript and XML*. http://developer.mozilla.org/de/docs/AJAX. [Stand: 09.05.2007].

[Json] JSON: *Homepage*. http://www.json.org. [Stand: 09.05.2007].

[CJson] CJson: *Homepage*. http://cheeseshop.python.org/pypi/python-cjson. [Stand: 09.05.2007].

[JsonLib] JSON-LIB: *Homepage*. http://json-lib.sourceforge.net. [Stand: 09.05.2007].

[LightHTTP] LIGHTTPD: *Homepage*. http://lighttpd.net

[LiteSp] LITESPEED WEB SERVER: *Homepage*. http://litespeedtech.com

[PyHTTP] PYTHON DOKUMENTATION: *CGIHTTPServer - CGI-capable HTTP request handler*. http://www.python.org/doc/2.4/lib/module-CGIHTTPServer.html

[CGI] CGI - COMMON GATEWAY INTERFACE: *RFC Project Page*. http://cgi-spec.golux.com

[FastCGI] FAST-CGI: *Homepage*. http://www.fastcgi.com

[PySession] MOD_PYTHON DOKUMENTATION: *Session Management*. http://www.modpython.org/live/current/doc-html/pyapi-sess.html. [Stand: 09.05.2007].

[PSP] MOD_PYTHON DOKUMENTATION: *PSP - Python Server Pages*. http://www.modpython.org/live/current/doc-html/pyapi-psp.html. [Stand: 09.05.2007].

[Cookie] IETF: *Cookie - RFC2965*. http://tools.ietf.org/html/rfc2965

[Firebird] FIREBIRD: *Homepage*. http://www.firebirdsql.org

[MySQL] MYSQL: *Homepage*. http://www.mysql.com

[Postgre] POSTGRESQL: *Homepage*. http://www.postgresql.org

[PostAb] POSTGRESQL: *About*. http://www.postgresql.org/about. [Stand: 09.05.2007].

[PostGeo] POSTGRESQL DOKUMENTATION: *Geometrische Datentypen*. http://www.postgresql.org/docs/8.2/interactive/datatype-geometric.html. [Stand: 09.05.2007].

[PostGeoFunk] POSTGRESQL DOKUMENTATION: *Funktionen und Operatoren für geometrische Datentypen*. http://www.postgresql.org/docs/8.2/interactive/functions-geometry.html. [Stand: 09.05.2007].

[PyDB] Python: *Python Database API Specification v2.0*. http://www.python.org/dev/peps/pep-0249. [Stand: 09.05.2007].

[PyGre] PyGreSQL: *Homepage*. http://www.pygresql.org

[PyPg] PyPgSQL: *Homepage*. http://pypgsql.sourceforge.net

[Psyco] Psycopg2: *Homepage*. http://initd.org/tracker/psycopg/wiki/PsycopgTwo. [Stand: 09.05.2007].

[Lotus] IBM: *Lotus Notes*. http://www-306.ibm.com/software/de/lotus

[promin] iABG: *ProMInanD*. http://www.iabg.de/index_en.php

[Floware] SER: *Floware*. http://www.ser.de

[ADEPTHos] ADEPT: *Process-Management in Hospital Environments*. http://www.informatik.uni-ulm.de/dbis/01/forschung/projects/adept/partners/ClinFlow.htm. [Stand: 09.05.2007].

[Arista] AristaFlow Project: *Homepage*. http://www.aristaflow.de

[Oracle] Oracle: *Oracle Database*. http://www.oracle.com/database/index.html

[OracXE] Oracle: *Oracle Database 10g Express Edition*. http://www.oracle.com/technology/products/database/xe/index.html. [Stand: 09.05.2007].

[JSW] Java Service Wrapper: *Homepage*. http://wrapper.tanukisoftware.org/doc/english/introduction.html. [Stand: 09.05.2007].

[CSS] W3C: *Cascading Style Sheets*. http://www.w3.org/Style/CSS. [Stand: 09.05.2007].

[Sched] Python Dokumentation: *sched - Event scheduler*. http://www.python.org/doc/2.4/lib/module-sched.html

[Prot] Prototype: *Homepage*. http://www.prototypejs.org

[SQLObj] SQLObject: *Homepage*. http://www.sqlobject.org/index.html. [Stand: 09.05.2007].

[SCP] Secure Shell: *scp - Secure Copy*. http://www.uni-koeln.de/rrzk/netze/ssh/sshscp.html

[vmware] VMWare: *VMWare Server 1.0.2*. http://register.vmware.com/content/eula102.html

[pylog] Python Dokumentation: *logging - Logging facility for Python* http://www.python.org/doc/2.4/lib/module-logging.html

Erklärung

Ich erkläre hiermit, dass ich diese Diplomarbeit selbständig verfasst, noch nicht anderweitig für andere Prüfungszwecke vorgelegt, keine anderen als die angegebenen Quellen und Hilfsmittel benützt sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

Ulm, den 11. Mai 2007