UBS

HOCHSCHULE DER MEDIEN

Diplomarbeit

im Studiengang *Medieninformatik*

Entwicklung eines Frameworks zum Loggen und Auswerten applikationsspezifischer Parameter

eingereicht im Wintersemester 2008/2009
durch Michael Zender, Matrikelnummer 15226
an der *Hochschule der Medien Stuttgart*

1. Prüfer: Professor Walter Kriha
Hochschule der Medien, Stuttgart

2. Prüfer: MSc Computer Science & Systems Design Raju Varghese
UBS AG, Zürich

Eingereicht am: 31. Oktober 2008

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Diplomarbeit selbständig und ohne fremde Hilfe verfasst habe. Sämtliche in dieser Diplomarbeit verwendeten Hilfsmittel und Quellen sind im Literaturverzeichnis im Anhang aufgeführt.

Ort, Datum

Unterschrift

Zusammenfassung

Im Rahmen der vorliegenden Diplomarbeit wird ein Framework entwickelt, welches einem Anwendungsentwickler die Möglichkeit gibt auf einfache Art und Weise und in kurzer Zeit einer bereits entwickelten oder in Entwicklung befindlichen Java Anwendung Reportingfunktionalität hinzuzufügen. Das entwickelte Framework bietet dabei sowohl eine einfache Integration in eine Applikation als auch die Möglichkeit die generierten Reports über eine Web-Applikation aufzurufen.

Die Entwicklung des Frameworks fand in einer Abteilung für die Entwicklung von Web-Applikationen der UBS AG statt. Somit ergab sich für das zu entwickelnde Framework diese Umgebung als primäres Einsatzgebiet.

Nach der Analyse der Ausganssituation in der oben genannten Abteilung und der möglichen Einsatzszenarien für dieses Framework beschäftigt sich diese Diplomarbeit vor allem mit dessen Konzeption und Implementation. Die Anwendung des Frameworks wird schließlich anhand eines kurzen Beispiels am Schluss der Diplomarbeit demonstriert.

Inhaltsverzeichnis

Abbildungsverzeichnis

Codebeispielverzeichnis

Kapitel 1

Einstieg

In diesem Kapitel wird zunächst die dieser Diplomarbeit zugrunde liegende Problemstellung erläutert und das verfolgte Ziel dargestellt. Anschließend wird die Vorgehensweise zum Erreichen des gesetzten Ziels kurz umrissen und die Gliederung dieser Arbeit erläutert.

1.1 Problemstellung

Für viele Applikationen ist es interessant und teilweise zur Überwachung auch notwendig Informationen aufzuzeichnen (zu loggen), die Aufschluss über ihren internen Zustand oder ihre Benutzung geben. Da die geloggten Daten jedoch im Rohzustand nur sehr schwer zu interpretieren sind benötigt man zusätzlich ein Werkzeug, welches die Darstellung der aufgezeichneten Daten in tabellarischer, noch besser jedoch in grafischer Form ermöglicht.

Im Bereich Web-Applikationen der *UBS* in Zürich werden zur Überwachung der verschiedenen Anwendungen die Server-Log-Dateien mit Hilfe der Statistiksoftware *WebTrends* ausgewertet. Diese enthalten zum Beispiel Informationen darüber, wann und von wo aus auf welche Ressource oder Applikation (*URL*) auf einem Server zugegriffen wurde. Auch Informationen über aufgetretene Fehler sind in diesen Log-Dateien zu finden (z.B. wenn das in der URL angegebene Objekt auf dem Server nicht gefunden wurde oder der Benutzer nicht über ausreichende Rechte verfügt um auf den Server zuzugreifen). Applikationsspezifische Parameter, die nicht in den Log-Dateien der Server auftauchen können jedoch auf diese Weise nicht ausgewertet werden. Um dieses Problem zu umgehen werden in wenigen Applikationen diese spezifischen Parameter in Log-Dateien aufgezeichnet, die anschließend den Anforderungen von WebTrends entsprechend umgeformt werden um ausgewertet werden zu können.

Dieses Vorgehen hat jedoch zwei Nachteile: Zum Einen muss für jede Applikation der Code zum aufzeichnen der interessanten Parameter neu

geschrieben werden. Zum Anderen müssen die geloggten Daten wie bereits erwähnt in ein von WebTrends lesbares Format transformiert werden, was häufig mit viel manuellem Aufwand verbunden ist.

1.2 Ziel der Arbeit

Ziel dieser Diplomarbeit ist es ein Framework zu entwickeln, welches Entwickler beim Hinzufügen von Reportingfunktionalität zu einer Applikation unterstützt. Zum Einen soll dieses Framework ein API zur Verfügung stellen welches das Schreiben interessanter Daten in Log-Dateien vereinheitlicht und vereinfacht. Um alle aufgezeichneten Daten zentral verwalten zu können soll das Framework des Weiteren eine Komponente beinhalten, die den Inhalt der zuvor geschriebenen Log-Dateien in eine Datenbank einfügt. Auf diese Datenbank soll schließlich eine dritte Komponente zugreifen die dem Benutzer eine grafisch aufbereitete Sicht auf die Daten ermöglicht um ihn beim Auswerten derselben zu unterstützen.

1.3 Gliederung der Arbeit

In diesem Abschnitt soll nun kurz der Aufbau dieser Arbeit und damit auch die Vorgehensweise bei der Entwicklung des Frameworks erläutert werden.

Kapitel 2 beschäftigt sich zunächst mit verschiedenen Usecases für das zu entwickelnde Framework. Anschließend werden darauf basierend die Anforderungen aufgestellt und die grobe Struktur des Frameworks entwickelt.

In Kapitel 3 wird näher auf die in dieser Diplomarbeit verwendete Reporttechnologie *BIRT* (Business Intelligence and Reporting Tools) eingegangen.

Um die Implementierung des Frameworks geht es Kapitel 4. Es ist weiter unterteilt in die Implementierung der einzelnen Teilkomponenten *Konfiguration*, *Report Design Generator*, *Writer*, *Collector* und *Viewer*.

Anhand eines kleinen Beispiels wird in Kapitel 5 die Verwendung des Frameworks demonstriert.

Kapitel 6 beschließt diese Diplomarbeit mit einer Zusammenfassung und der Evaluation des entwickelten Frameworks anhand der aufgestellten Anforderungen.

Im Anhang dieser Arbeit befindet sich neben den erstellten Build Skripten für die verschiedenen entwickelten Komponenten, entwickeltem Javascript Code und Templates auch eine CD mit dem kompletten entwickelten Quellcode und dieser Arbeit als PDF-Datei.

Kapitel 2

Design

In diesem Kapitel werden zunächst einige Usecases für das zu entwickelnde Framework vorgestellt. Darauf aufbauend werden anschließend die Anforderungen die an das Framework gestellt werden aufgestellt und die grobe Struktur des Frameworks skizziert.

2.1 Usecases

Bevor anschließend die Anforderungen an das zu entwickelnde Framework aufgestellt werden soll nun ein kurzer Überblick über einige mögliche Usecases gegeben werden um ein besseres Verständnis für dessen Einsatzbereiche herzustellen.

2.1.1 Unterstützung bei der Fehlersuche

Tritt während des Betriebs einer Applikation eine Anomalie auf (z.B. unerwartet hohe Antwortzeiten), so ist es oftmals nicht möglich rein durch die Analyse der Logdateien des Servers die Ursache des Problems zu lokalisieren. Häufig ist es erst durch die Analyse applikationsinterner Parameter bzw. deren Gegenüberstellung möglich Rückschlüsse auf die eigentliche Ursache des Problems zu ziehen.

2.1.2 Erkennung zukünftiger Probleme

Zeichnet man applikationsspezifische Parameter über einen längeren Zeitraum auf und wertet diese anschließend aus, so lassen sich daraus unter Umständen Rückschlüsse auf die zukünftige Entwicklung dieser Parameter ziehen. Beobachtet man zum Beispiel bei einer Applikation steigende Nutzerzahlen (und damit auch möglicherweise steigende Antwortzeiten), so könnte man mit diesen gewonnen Informationen rechtzeitig Maßnahmen ergreifen, die einen weiterhin stabilen Betrieb der betroffenen Applikation ermöglichen.

2.1.3 Überwachen von Applikationen

Ein weiterer Einsatzbereich für das Framework wäre die simple Überwachung bestimmter Parameter um zum Beispiel applikationsinterne Fehler, die sich nicht nach außen hin manifestieren zu erkennen.

2.1.4 Erfassen und Auswerten von Nutzungsdaten

Abgesehen von der Benutzung des Frameworks um im Fehlerfall ein geeignetes Werkzeug zur Analyse zu besitzen könnte man es auch dazu einsetzen das Verhalten des Benutzers einer Applikation (zum Beispiel seine Navigation innerhalb der Applikation) aufzuzeichnen und auszuwerten. Dadurch wäre es möglich die zukünftige Entwicklung einer Applikation direkter auf die Bedürfnisse der Benutzer abzustimmen.

2.2 Anforderungen

Aufgrund der dargelegten Ausgangssituation und der behandelten Usecases ergeben sich folgende Anforderungen:

2.2.1 Struktur der Logdaten konfigurierbar

Da sich die Struktur der zu loggenden Daten von Applikation zu Applikation stark unterscheidet muss es möglich sein für jede Anwendung ihre jeweiligen Parameter zu konfigurieren. Jeder Parameter einer Applikation soll einen Namen erhalten und außerdem den Typ des Wertes, der aufgezeichnet werden soll. Zusätzlich müssen pro Applikation noch gewisse Attribute wie zum Beispiel deren Name konfiguriert werden können.

2.2.2 API zum Loggen von Daten

Zum Loggen von Daten an geeigneter Stelle soll dem Applikationsentwickler ein API zur Verfügung stehen. Dieses API muss Methoden zur Verfügung stellen, die einen Satz aus Schlüssel-Wert-Paaren als Parameter entgegennehmen. Der Schlüssel eines solchen Paares entspricht dabei dem Namen eines Parameters und dessen Wert dem Wert, der für diesen Parameter aufgezeichnet werden soll. Die aufgezeichneten Werte sollen in einer Datei abgespeichert werden.

2.2.3 Persistente Speicherung der Logdaten

Um die spätere Auswertung der Daten zu vereinfachen sollen die über das erwähnte API in Dateien abgelegten Informationen in einer Datenbank persistent gespeichert werden. Dies soll zum Einen die Verwaltung der Daten

vereinfachen. Zum Anderen soll dadurch aber auch der Zugriff auf die Daten bzw. das Auswählen bestimmter Daten vereinfacht werden. Die Aggregierung dieser Log-Dateien und das Einfügen in die Datenbank soll periodisch einmal am Tag erfolgen.

2.2.4 Webbasierter Zugriff auf Reports

Der Zugriff auf die Daten zum Zweck der Anzeige und Analyse soll über ein webbasiertes Interface möglich sein. Zusätzlich soll dieses Interface dem Analysten die Möglichkeit geben ein Start- und ein Enddatum anzugeben und einen Report aus den für diese Zeitspanne vorliegenden Daten zu generieren. Dabei sollen die Daten in tabellarischer und/oder grafischer Form dargestellt werden.

2.2.5 Generieren der Reportvorlagen

Da die Struktur der Daten wie in Abschnitt 2.2.1 konfigurierbar sein soll muss natürlich auch die Gestalt der generierten Reports dynamisch sein und sich an der Konfiguration orientieren. Nach Möglichkeit sollte die Generierung der Reports in zwei Schritten erfolgen:

1. Im ersten Schritt wird aus der Konfiguration eine Art Vorlage für den Report erstellt. Diese Vorlage enthält neben der Definition der Datenquellen das komplette Layout und die Formatierung für einen Report.

2. Im zweiten Schritt wird diese Vorlage dazu benutzt um daraus den fertigen Report zu generieren. Dafür wird die Vorlage gelesen, die Daten anhand der definierten Datenquellen aggregiert und der Report anhand der Layout- und Formatierungsangaben erstellt.

2.3 Reporttechnologie

Eine der Fragen, die ganz zu Beginn geklärt werden musste war die nach der Reporttechnologie, die in dem Framework Verwendung finden sollte. Bei der Suche nach geeigneten OpenSource Lösungen ergaben sich *JasperReports* und *Eclipse BIRT* als zwei mögliche Kandidaten, die auch die im Abschnitt 2.2.5 erläuterte Anforderung erfüllen. Diese sollen in diesem Abschnitt anhand der an die Reporttechnologie gestellten Anforderungen evaluiert und anschließend miteinander verglichen werden um eine Entscheidung für eines der beiden Produkte zu treffen.

2.3.1 Anforderungen

Um die Reportlösungen entsprechend evaluieren zu können wurden zunächst folgende Anforderungen aufgestellt:

Bearbeiten der Reportvorlage durch ein API Um die Ersetllung der Report Vorlage (siehe Abschnitt 2.2.5) aus der Konfiguration zu vereinfachen sollte die gewählte Reporttechnologie möglichst über ein API verfügen, mit Hilfe dessen dieser Schritt in einem Generator einfach möglich ist.

Bezug der Reportdaten aus einer Datenbank Da die geloggten Daten zur persistenten Speicherung in eine Datenbank eingefügt werden, muss die gewählte Reporttechnologie über die Möglichkeit verfügen die Daten für den zu erstellenden Report aus einer Datebank zu beziehen. Diese Möglichkeit sollte unabhängig vom Typ der Datenbank gegeben sein.

Diagrammfunktionalität Zur besseren Visualisierung der Daten sollte die gewählte Reporttechnologie die Möglichkeit bieten aus den Reportdaten Diagramme zu erzeugen und diese auf dem generierten Report anzuzeigen.

Integrierbar in Web-Applikation Zur Erfüllung der in Abschnitt 2.2.4 erläuterten Anforderung sollte es einfach möglich sein die Reports zum einfachen Zugriff in eine Web-Applikation zu integrieren.

2.3.2 BIRT

BIRT (Business Intelligence and Reporting Tools) ist ein Top-Level Projekt innerhalb der Eclipse Foundation für Reporting und Business Intelligence. In *BIRT* wird zunächst eine Vorlage für den Report (*Report Design*) erstellt aus der dann zur Laufzeit der Report generiert wird. Für die Bearbeitung des Report Designs steht sowohl ein auf der Entwicklungsumgebung *Eclipse* basierender grafischer Editor als auch ein Java API zur Verfügung. Für den Zugriff auf die Daten für einen Report benutzt *BIRT* das *Open Data Access* Framework (ODA) welches ebenfalls ein Eclipse Projekt ist. Diagramme können ebenfalls in Reports integriert werden. Um die Reports einfacher in Web-Applikationen integrieren zu können enthält *BIRT* eine fertige Web-Applikation zum Anzeigen von Reports, die abgesehen von ihrer Konfiguration ohne Anpassungen auf einem Application-Server installiert werden kann.

Ein weniger technischer Vorteil von *BIRT*, der während der Entwicklung allerdings von großem Nutzen ist, ist die aktive Entwicklercommunity und die gute Dokumentation sowie die Fülle an Beispielen für die Verwendung der *BIRT* APIs.

2.3.3 Jasper

Jasper[1] ist eine OpenSource Bibliothek die es ermöglicht Reportingfunktionalität in Java Applikationen zu integrieren. Die Reports können dabei in den verschiedensten Formaten generiert werden.

Auch bei Jasper verläuft die Generierung eines Reports in zwei Schritten: Zunächst wird ein Report Design erstellt welches Informationen enthält aus denen dann später der fertige Report generiert wird. Für die Bearbeitung der Report Designs stehen sowohl ein Java API als auch mehrere grafische Werkzeuge (u.a. für Eclipse) zur Verfügung. Da diese Unterstützung durch grafische Werkzeuge allerdings erst im Nachhinein entwickelt wurde muss für die Realisierung eines Reports trotzdem noch eine nicht unerhebliche Menge Code geschrieben werden um die Infrastruktur für die Generierung eines Reports bereitzustellen (z.B. Zugriff auf Datenquellen). Insbesondere fehlt der Jasper Bibliothek im Gegensatz zu *BIRT* eine Web-Applikation zum Ausführen der Report Designs bzw. zum Betrachten der generierten Reports.

Report Designs können bei Jasper ebenfalls Diagramme enthalten. Die Generierung dieser Diagramme erfolgt durch die Einbindung der OpenSource Bibliothek *JFreeChart*[2].

2.3.4 Entscheidung für BIRT

Nach dem Vergleich der beiden Reporttechnologien wurde die Entscheidung zu Gunsten von *BIRT* gefällt. Ausschlaggebend war zunächst die bessere Unstützung bei der Erstellung der Report Designs einerseits durch den *BIRT Report Designer*, andererseits durch ein Java API. Ein weiteres Argument für *BIRT* war die gute Integrationsmöglichkeit von BIRT Reports in Web-Applikationen vor allem mit Hilfe der *BIRT Report Viewer* Web-Applikation und in die bei UBS bereits im Einsatz befindliche Entwicklungsumgebung Eclipse.

2.4 Struktur

In diesem Kapitel soll nun kurz auf den Aufbau des zu entwickelnden Frameworks und die einzelnen Komponenten und ihre Aufgabe innerhalb des Frameworks eingegangen werden. Hierzu wird zunächst in Abbildung 2.1 ein Überblick über das Framework gegeben und im Anschluss auf die einzelnen Komponenten näher eingegangen.

Die genaue Beschreibung der Implementierung und Funktionsweise der einzelnen Komponenten ist in Kapitel 4 ab Seite 23 zu finden.

[1] `http://sourceforge.net/projects/jasperreports/`
[2] `http://www.jfree.org/jfreechart/`

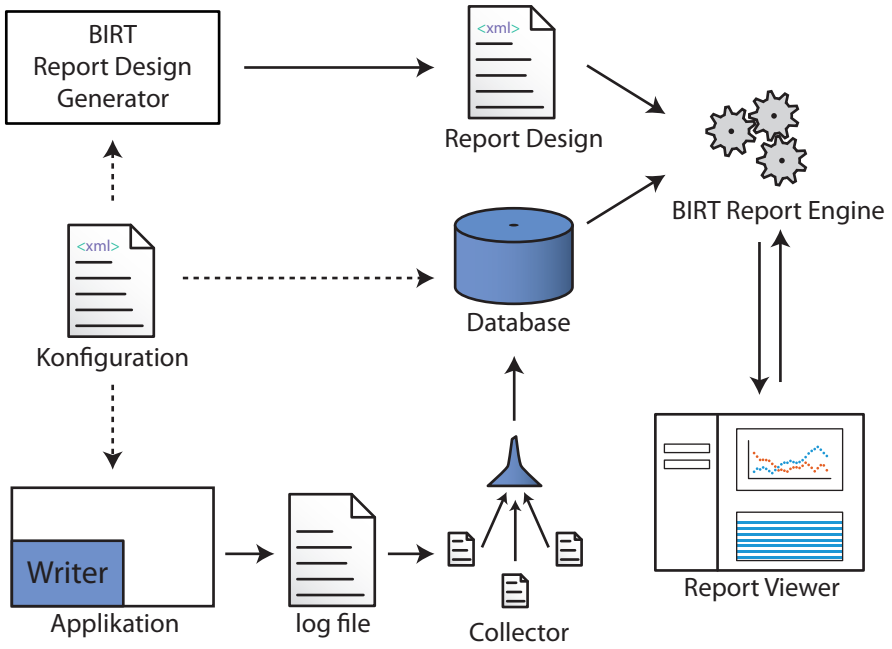

Abbildung 2.1: Das Design des Frameworks im Überblick

2.4.1 Konfiguration

Um die in Abschnitt 2.2.1 festgelegte Anforderung zu erfüllen muss für jede Applikation, die das Framework benutzt eine Konfiguration erstellt werden. Da mehrere Komponenten des Frameworks auf diese Konfiguration zugreifen müssen ist es sinnvoll die Zugriffsklassen in einer eigenen Komponente zusammenzufassen und diese dann in die anderen Komponenten, die die Konfiguration benötigen zu integrieren.

2.4.2 Report Design Generator

Der Report Design Generator übernimmt die Aufgabe aus der vom Benutzer erstellten Konfiguration ein BIRT Report Design zu generieren. Dieses dient als Grundlage für den späteren Report.

2.4.3 Writer

Der Writer ist ein Java API, welches dem Entwickler Methoden zur Verfügung stellt mit Hilfe derer er an den geeigneten Stellen im Programmcode die gewünschten Parameter loggen kann. Das API muss dabei unabhängig sein von der Struktur der Parameter.

2.4.4 Collector

Der Collector ist ein Serverprogramm, welches periodisch die Log-Dateien der einzelnen Applikationen „einsammelt“ und in die entsprechenden Tabellen der Datenbank einfügt.

2.4.5 Viewer

Beim Viewer handelt es sich um eine Web-Applikation, welche dem Analysten die Möglichkeit bietet aus einer Liste der verfügbaren Reports einen auszuwählen. Zusätzlich kann er die Zeitspanne die ihn interessiert auswählen und schließlich den Report ausführen und anzeigen lassen.

Kapitel 3

BIRT

Im August 2004 trat die Firma Actuate[1], Marktführer im Bereich Enterprise Reporting Applikationen, der Eclipse Foundation als strategischer Entwickler bei. Damit stellte Actuate rund 10 Jahre Erfahrung im Bereich Business Intelligence und Enterprise Reporting der Eclipse Community zur Verfügung. Auf Basis dieser Erfahrung schlug Actuate gleichzeitig das *Business Intelligence and Reporting Tool* (BIRT) Projekt als neues Top Level Eclipse Projekt vor. Nach einer 30-tägigen Überprüfung durch die Mitglieder der Eclipse Community wurde BIRT schließlich als erstes Eclipse Business Intelligence Projekt bestätigt.

Nach einem weiteren halben Jahr im Juni 2005 wurde BIRT schließlich in der Version 1.0 veröffentlicht. Seither wurde BIRT laufend weiterentwickelt. Version 2.0 erschien bereits ein halbes Jahr später im Januar 2006. Die aktuelle Version 2.3 wurde als Teil des *Ganymede* Releases im Juni 2008 veröffentlicht. Im Rahmen dieser Diplomarbeit wurde BIRT in der im April 2008 aktuellen Version 2.2 benutzt.

Im folgenden Abschnitt soll ein kurzer Überblick über die BIRT Architektur und die zur Verfügung stehenden APIs gegeben werden.

3.1 BIRT Architektur

Wie in Abbildung 3.1 zu sehen besteht BIRT aus mehreren Komponenten, die bei der Erstellung eines Reports verschiedene Aufgaben übernehmen. Diese werden in den folgenden Abschnitten genauer beschrieben.

Darüberhinaus werden durch die BIRT Architektur zwei Phasen bei der Erstellung eines Reports gekennzeichnet:

Designphase Zunächst muss von Hand, unter Zuhilfenahme des BIRT Report Designers ein Report Design erstellt werden (siehe Abschnitt 3.2.1).

[1] http://www.actuate.com/

Generierungsphase Liegt das Report Design vor, so kann es immer wieder durch die Report Engine ausgeführt werden um einen Report zu generieren.

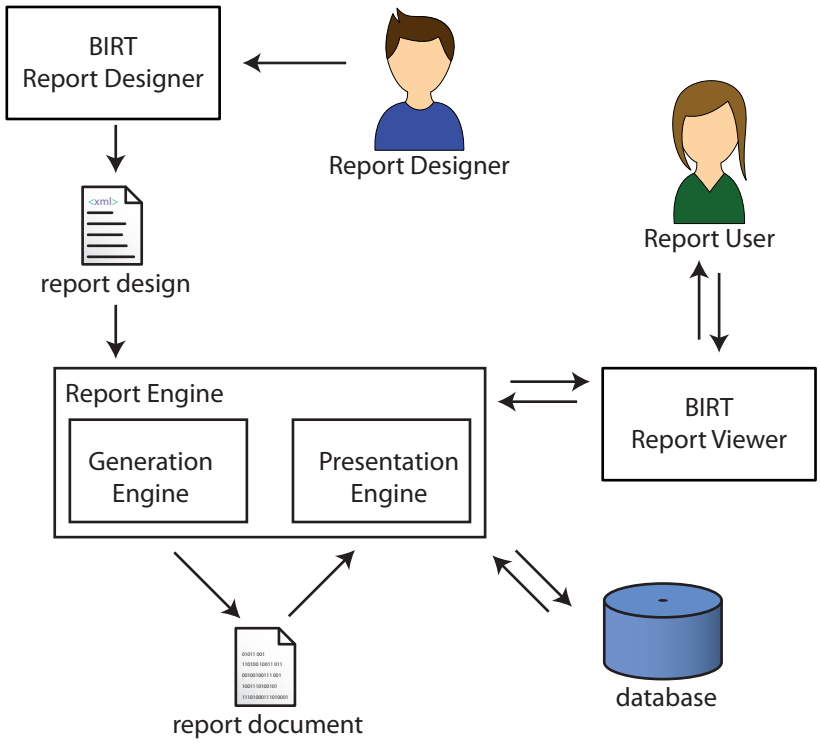

Abbildung 3.1: BIRT Architektur

3.1.1 Report Design Engine

Die Report Design Engine stellt Dienste zum Erstellen und Validieren einer Report Design Datei (siehe Abschnitt 3.2.1) zur Verfügung. Sie wird vom BIRT Report Designer und jeder anderen Java Applikation die Report Designs erstellt benutzt. Während der Reporterstellung wird die Report Design Engine dazu benutzt das Report Design zu lesen und ein Report Dokument zu erzeugen.

Eine kurze Erläuterung des APIs für die Report Design Engine ist im Abschnitt 3.5.1 zu finden.

3.1.2 Report Engine

Die Report Engine erstellt aus den in einem Report Design vorhandenen Definitionen den fertigen Report. Die Erstellung des Reports geschieht in

zwei Schritten, für die jeweils eine eigene Komponente zuständig ist. Diese beiden Komponenten übernehmen bei der Erstellung eines Reports folgende Aufgaben:

Generation Engine Die Generation Engine kann ein Report Design sowohl lesen als auch interpretieren. Sie benutzt die Data Engine um die Daten für den zu erstellenden Report aus den im Report Design definierten Data Sources zu beziehen und entsprechend zu transformieren. Die Generation Engine erzeugt aus dem geladenen Report Design und den aggregierten Daten ein Report Document (siehe Abschnitt 3.2.4).

Presentation Engine Das Report Document, welches von der Generation Engine erzeugt wurde wird von der Presentation Engine dazu verwendet den fertigen Report zu erstellen. Dabei macht sie sich wiederum die Data Engine zu Nutze, diesmal allerdings um Daten aus dem Report Document zu extrahieren.

Anschließend bedient sich die Presentation Engine eines sogenannten *Emitters*, der den generierten Report im gewünschten Format ausgibt.

Behält man das durch die Generation Engine erzeugte Report Document, so ist es möglich basierend darauf verschiedene Versionen (z.B. HTML, PDF, Word) eines Reports zu erzeugen ohne jedesmal die Daten neu aggregieren zu müssen. Ein weiterer Vorteil dieser Vorgehensweise zeigt sich bei der Betrachtung eines mehrseitigen Reports, wenn jede Seite (beispielsweise über einen Webbrowser) einzeln angefordert wird. Da jede neue Seite aus den Daten im Report Document erstellt wird erspart man sich eventuell teure Abfragen auf die zugrundeliegenden Datenquellen.

Auch für die Report Engine steht dem Entwickler ein API zur Verfügung mit dem er die Erstellung von Reports in die eigene Applikation integrieren kann. In Abschnitt 3.5.2 wird das API zur Report Engine kurz erläutert.

3.1.3 Data Engine

Die Data Engine ist, wie der Name erkennen lässt, für die in einem Report dargestellten Daten zuständig. Auch diese Komponente ist intern in zwei Teile gegliedert:

Data Access Benutzt das ODA Framework[2] um Daten von den verschiedenen unterstützten Datenquellen zu beziehen.

[2]Das Open Data Access Framework ist Teil des *Eclipse Data Tools Platform* Projekts und bietet einheitlichen Zugriff auf verschiedene Datenquellen (XML, Web Service, JDBC, etc.). Weitere Informationen unter `http://www.eclipse.org/datatools/`

Data Transform Führt verschiedene Operationen auf den abgefragten Daten aus (Sortieren, Gruppieren, Filtern, Aggregieren)

Wie bereits weiter oben beschrieben bezieht die Data Engine ihre Daten entweder aus den Data Sources des Report Designs (aufgerufen durch Generation Engine) oder aus einem Report Document (aufgerufen durch Presentation Engine).

3.1.4 Report Viewer

Der Report Viewer stellt die Schnittstelle für den Benutzer zu den verschiedenen zur Verfügung stehenden Reports dar. Der Funktionsumfang des Report Viewers hängt dabei sehr stark von der Umgebung ab in der die Reports ausgeführt werden sollen. Wie der Name dieser Komponente vermuten lässt zeigt sie den von der Report Engine erzeugten Report an.

Je nach Einsatzgebiet der BIRT Reports bietet der Viewer dem Benutzer zusätzlich eine oder mehrere der folgenden Funktionen:

- Auswahl aus einer Liste aller zur Verfügung stehender Reports.
- Eingabe eventuell vorhandener Reportparameter (siehe Abschnitt 3.3.4 für weitere Informationen).
- Anzeige des Inhaltsverzeichnisses eines Reports.
- Blättern in einem Report mit mehreren Seiten.
- Exportieren des Reports in ein anderes Format zur dauerhaften Speicherung.
- Exportieren der Daten eines Reports in eine CSV-Datei.
- Drucken des Reports.

Ebenfalls abhängig vom Einsatzgebiet der BIRT Reports kann der Report Viewer entweder als eigenständige Applikation, als Web-Applikation oder als Teil einer anderen Applikation umgesetzt sein.

Im Rahmen des BIRT Projekts wird auch eine umfangreiche Viewer Web-Applikation entwickelt, die bis auf die Auflistung der vorhandenen Reports alle der oben genannten Funktionen bietet. Allerdings ist es ohne weiteres möglich eine eigene Viewer Applikation zu entwickeln, um spezielle Anforderungen wie zum Beispiel Security in den Griff zu bekommen.

3.2 BIRT Dateitypen

3.2.1 Report Design

Die Basis jedes BIRT Reports ist das sogenannte Report Design. In diesem Dokument sind alle Eigenschaften die zum Erstellen eines Reports benötigt werden definiert (Datenquellen, Formatierung, Layout). Zum Erstellen und Bearbeiten des Report Designs stellt BIRT einen Report Designer zum Einen als Eclipse Plugin und zum Anderen als eigenständige Rich Client Anwendung zur Verfügung.

Das Report Design ist ein XML Dokument dessen Struktur auf einem XML Schema basiert.

3.2.2 Report Template

Neben dem Report Design gibt es sogenannte Report Templates. Diese dienen als Vorlage, bei der Erstellung eines neuen Report Designs. Dabei kann ein Report Template zum Beispiel alle für einen Report benötigten Elemente enthalten während ein anderes Report Template nur ganz bestimmte Elemente enthält. Wird ein neues Report Design auf Basis eines Report Templates erstellt werden alle Elemente aus dem Report Template in das Design kopiert. Von dieser Basis aus kann nun das Report Design weiter entwickelt werden. Das neu erstellte Report Design ist somit unabhängig von dem Report Template auf dem es basiert.

3.2.3 Report Library

Report Libraries hingegen folgen einem etwas anderen Konzept. Zwar enthalten auch sie Report Elemente, die bei der Erstellung eines neuen Report Designs wiederverwendet werden können, allerdings werden diese nicht in das neue Report Design kopiert, sondern lediglich referenziert. Daraus folgt, dass dieses Report Design immer von der verwendeten Report Library abhängt. Diese wird also immer benötigt, wenn aus dem Report Design ein Report erstellt werden soll. Um bei der Verwendung mehrerer Report Libraries in einem Report Design zu vermeiden dass es zu Namenskonflikten zwischen Elementen in den verschiedenen Report Libraries kommt werden die Elemente einer Library in einem eigenen Namespace zusammengefasst, der in der Regel dem Dateinamen der Report Library entspricht.

3.2.4 Report Document

Ein Report Document ist eine Datei im Binärformat, die neben dem Report Design und den Daten, welche aus den im Report Design definierten Datenquellen aggregiert wurden, noch zusätzlich Folgendes enthält:

- Informationen über die Seiteneinteilung des Reports
- Informationen über das Inhaltsverzeichnis des Reports

Ein Report Document wird von der Report Engine als Zwischenschritt bei der Reporterstellung generiert. Im Folgenden können daraus entweder der fertige Report im gewünschten Format (HTML, PDF, DOC, XLS...) erstellt oder die aggregierten Daten als CSV Datei extrahiert werden.

Der BIRT Web Viewer benutzt das Report Document um dem Benutzer das Blättern im Report zu ermöglichen ohne für jede Seite auf die geblättert wird die Daten neu aggregieren zu müssen.

3.2.5 Wiederverwendung verschiedener Report Elemente

Zusammenfassend bieten sowohl Report Templates als auch Report Libraries die Möglichkeit die Erstellung eines Report Designs zu vereinfachen, da häufig benutzte Elemente wiederverwendet werden können. Zusätzlich ist es mit diesen beiden Mitteln einfacher möglich Report Designs mit einer einheitlichen Optik zu erstellen.

3.3 Report Elemente

Im Folgenden sollen nun einige wichtige Report Elemente und ihre Funktion erläutert werden.

3.3.1 Masterpage

Masterpages definieren Vorlagen für Reportseiten, wobei zum Beispiel die Seitengröße, die Ausrichtung der Seite und die Größe eventuell vorhandener Seitenränder festgelegt werden können. Zusätzlich können in der Kopf- und Fußzeile einer Seite verschiedene Elemente platziert werden (z.B. Logo, Seitenzahl, etc.). In einem Report Design können nun für verschiedene Report Elemente verschiedene Masterpages festgelegt werden. Angenommen es sind zwei verschiedene Masterpages vorhanden, eine im Hochformat und eine im Querformat, so kann man beispielsweise festlegen, dass eine im Report Design vorhandene Tabelle auf einer Seite im Hochformat, ein Diagramm dagegen auf einer Seite im Querformat angezeigt werden soll.

3.3.2 Data Source

In BIRT werden Data Sources dazu benutzt um die Verbindungsparameter zu einer Datenquelle zusammenzufassen. Sind die Daten für einen Report zum Beispiel in einer relationalen Datenbank abgelegt, für die ein JDBC

Treiber verfügbar ist, so würde die Data Source Definition folgende Angaben enthalten:

1. Treiberklasse des JDBC Treibers
2. URL unter der die Datenbank zu erreichen ist
3. Der Benutzername der für die Verbindung zur Datenbank benötigt wird
4. Das zum Benutzernamen gehörige Passwort

Eine relationale Datenbank ist allerdings nur eine Möglichkeit Daten zu speichern, die in einem Report verwendet werden sollen. Darüber hinaus sind noch andere Datenquellen verfügbar:

XML Data Source Die Daten werden aus einer XML-Datei gelesen.

Flat File Data Source Die Daten werden aus einer CSV-Datei gelesen.

Scripted Data Source Der Zugriff auf die Daten erfolgt per Javascript. Da in BIRT der Zugriff auf Java Objekte per Javascript möglich ist kann es sich hierbei um jede mit Hilfe von Java erreichbare Datenquelle handeln.

3.3.3 Data Set

In einem Data Set wird festgelegt, *welche* Daten aus einer Data Source extrahiert werden sollen. Dabei hängen die Möglichkeiten von der verwendeten Data Source ab.

Im Zusammenhang mit einer JDBC Data Source zum Beispiel hat man die Möglichkeit entweder eine SQL *SELECT* Abfrage oder eine in der Datenbank angelegte *stored procedure* festzulegen, die jeweils an die Datenbank gesendet wird um die gewünschten Daten zu erhalten. Eine Besonderheit des SQL Data Sets ist die Möglichkeit in der an die Datenbank gesendeten Abfrage Platzhalter in Form von Fragezeichen zu notieren. Für diese Platzhalter kann jeweils ein korrespondierender Reportparameter definiert werden. Zur Laufzeit des Reports wird dann das Fragezeichen in der Abfrage durch den Wert des Reportparameters ersetzt bevor die Anfrage an die Datenbank gesendet wird.

Basiert das Data Set allerdings auf einer XML-Datei, so muss man die Daten in eine relationale Struktur überführen. Dies geschieht in einem XML Data Set mit Hilfe von XPath-Ausdrücken. Bei einem Flat File Data Set welches seine Daten aus einer CSV-Datei bezieht ist die transformation in eine relationale Struktur einfacher und praktisch durch das Dateiformat bereits vorgegeben.

Gemeinsam haben alle Data Sets, dass am Ende die Daten in tabellarischer Form vorliegen und durch folgende Möglichkeiten weiter bearbeitet werden können:

computed fields Mit Hilfe von „computed fields“ ist es möglich der aus einem Data Set resultierenden Tabelle weitere Spalten hinzuzufügen, deren Werte sich aus denen der anderen Spalten berechnen.

filter Mit Filtern lässt sich die Ergebnismenge eines Data Sets weiter einschränken. Ein möglicher Filter wäre zum Beispiel, dass alle Zeilen der Ergebnistabelle weggelassen werden, bei denen in einer Spalte der Wert außerhalb eines bestimmten Intervalls liegt.

3.3.4 Report Parameter

Wenn man ein Report Design erstellt legt man genau fest, welche Daten im Report angezeigt werden sollen. Der Benutzer des Reports hat später keine Möglichkeiten dies zu beeinflussen es sei denn er ändert das Report Design ab, was im Normalfall allerdings wenig praktikabel ist. Genau dieses Problem lösen Report Parameter. Durch sie kann der Report Designer dem Benutzer des Reports die Möglichkeit geben bestimmte Elemente des Reports zu beeinflussen.

In einem Report, der beispielsweise statistische Daten für verschiedene Regionen enthält, definiert der Report Designer für das im Report Design definierte Data Set einen Filter, der die Ergebnisse nach der Region filtert. Zusätzlich definiert er einen Report Parameter, der es dem Benutzer des Reports ermöglicht den Wert für den Filter, sprich die gewünschte Region, an den Report zu übergeben. Auf diese Weise kann der Benutzer des Reports zur Laufzeit bestimmen für welche Region die Daten angezeigt werden sollen.

3.3.5 Tabelle

Die einfachste Möglichkeit das Ergebnis eines Data Sets anzuzeigen ist in einer Tabelle. Eine Tabelle in BIRT hat mehrere Bereiche:

Header Der Header-Bereich wird für jede Tabelle nur einmal an deren Anfang dargestellt. Dieser Bereich kann zum Beispiel einen Titel für die Tabelle oder eine Überschrift für jede Spalte enthalten.

Detail Der Detail-Bereich einer Tabelle wird für jede Zeile im Resultat eines Data Sets wiederholt. Das heißt in diesem Bereich werden später im Report die aggregierten Daten angezeigt.

Footer Auch der Footer-Bereich wird wie der Header-Bereich ein einziges Mal allerdings am Ende der Tabelle dargestellt. Hier können zum Beispiel zusammenfassende Informationen über den Inhalt der Tabelle angezeigt werden (z.B. Summe oder Durchschnitt einer Spalte).

Damit eine Tabelle überhaupt auf die Daten eines Data Sets zugreifen kann muss die Tabelle (wie jedes andere Report Element auch) mit dem Data Set assoziiert werden.

3.3.6 Diagramm

Mit Hilfe des Diagramm Elements kann man die Daten in einem Report grafisch darstellen. Dieses Element ist in seinen Konfigurationsmöglichkeiten sehr komplex.

Zunächst kann man zwischen verschiedenen Typen von Diagrammen wählen, darunter eher einfache wie zum Beispiel Balkendiagramm, Flächendiagramm oder Kreisdiagramm aber auch komplexere wie zum Beispiel das Gantt-Diagramm. Anschließend kann man genau festlegen welche Daten wie auf dem Diagramm zu sehen sein sollen. Zu guter Letzt können BIRT Diagramme noch formatiert und so optisch genau den Wünschen des Report Designers angepasst werden.

Eine weitere Besonderheit ist, dass man das Ausgabeformat des Diagramms festlegen kann. Dabei kann man zwischen Rasterformaten (BMP, JPEG, PNG) und SVG wählen. Letzteres ist insbesondere bei webbasierten Reports im HTML Format interessant, da hierfür das Diagramm um gewisse interaktive Features erweitert werden kann (z.B. Umschalten der Sichtbarkeit von Datenreihen oder Anzeigen von Tooltips mit zusätzlichen Informationen). Wird aus dem Report Design ein Report im PDF-Format generiert stehen diese interaktiven Features selbstverständlich nicht mehr zur Verfügung.

3.4 Scripting in Reports

Reichen die zur Verfügung stehenden Optionen der einzelnen Report Elemente nicht aus, so kann man das Report Design um eigene *event handler* erweitern, mit denen man verschiedene Aspekte der Reporterstellung zusätzlich beeinflussen kann. Ein event handler besteht aus Code, der während der Erstellung eines Reports bei bestimmten Ereignissen aufgerufen wird und kann entweder in Javascript oder Java implementiert werden. Die Möglichkeiten die Reporterstellung zu beeinflussen sind bei beiden Programmiersprachen gleich. Jedoch bieten beide Optionen individuelle Vorteile:

Vorteile von in Javascript programmierten event handlern:

- Einen Javascript event handler zu einem Report Element hinzuzufügen ist wesentlich unkomplizierter als dies bei einem in Java programmierten event handler der Fall ist. Der BIRT Report Designer stellt dafür ein Eingabefeld zur Verfügung.
- Durch die gegenüber Java einfachere Syntax und die dynamische Typisierung von Javascript ist es für den Report Designer einfacher kleine event handler zu entwickeln.

Vorteile von Java bei der Programmierung von event handlern:

- Verwendet man den BIRT Report Designer als Plugin für Eclipse, so kann man zur Entwicklung von event handlern in Java auf die Eclipse Java IDE zurückgreifen (sehr komfortabler Java Editor, integrierter Java Debugger).
- Im Gegensatz zu Javascript event handlern, die in das Report Design eingebettet sind liegen Java event handler in eigenen Dateien vor und können so einfacher gefunden und bearbeitet werden.
- Ein weiterer Vorteil dieses Umstands besteht darin, dass der gleiche Java event handler in mehreren Report Elementen innerhalb des gleichen Report Designs oder sogar in mehreren verschiedenen Report Designs verwendet werden kann.

3.5 BIRT APIs

In diesem Abschnitt werden die verschiedenen APIs, die *BIRT* zur Verfügung stellt kurz erklärt. Diese APIs werden auf der einen Seite in den zur Verfügung stehenden Werkzeugen zum Bearbeiten von Report Designs und zum Erzeugen von Reports verwendet. Auf der anderen Seite stehen sie allerdings auch zur Entwicklung eigener Werkzeuge zum Bearbeiten von Report Designs oder zum Generieren von Reports zur Verfügung.

3.5.1 Design Engine API

Das BIRT Design Engine API (auch Report Model API) wird von Entwicklern dazu benutzt um Werkzeuge zum Erstellen und Bearbeiten von Report Designs, Report Templates und Report Libraries zu entwickeln. Der BIRT Report Designer basiert beispielsweise komplett auf diesem API. Selbstverständlich kann es auch ohne die Anbindung an eine grafische Benutzeroberfläche dazu verwendet werden die eben genannten Artefakte zu erzeugen.

Das API bietet, unabhängig vom Kontext in dem es verwendet wird, folgende Möglichkeiten:

- Lesen und schreiben von Report Designs, Report Templates und Report Libraries
- Zugriff auf die Kommandoliste (für Rückgängig/Wiederholen Funktionalität)
- Semantische Repräsentation des Report Designs, des Report Templates oder der Report Library
- Zur Verfügung stellen von Metainformationen über das Report Object Model
- Überprüfen von Eigenschaftswerten von Report Elementen auf ihre Gültigkeit

3.5.2 Report Engine API

Die BIRT Report Engine ermöglicht es aus einem Report Design einen Report zu erzeugen. Zu diesem Zweck kann sie in den verschiedensten Umgebungen eingesetzt werden:

Eigenständige Java Applikation In einer Java Kommandozeilenanwendung oder einer Java Anwendung mit grafischer Benutzeroberfläche kann die Report Engine dazu benutzt werden um einen Report zu erzeugen und abzuspeichern.

BIRT Report Viewer Der BIRT Report Designer benutzt die Report Engine um eine Vorschau des in Bearbeitung befindlichen Report Designs zu erzeugen. Analog kann die Report Engine dazu benutzt werden in einer speziell gebauten Report Designer Anwendung eine Vorschau des Report Designs zur Verfügung zu stellen.

Web Applikation In einer Web Applikation kann die Report Engine dazu benutzt werden einen webbasierten Report zur Verfügung zu stellen.

In jeder dieser Umgebungen übernimmt die Report Engine lediglich die Erstellung des Reports. Weitere Aspekte (Zusammensetzen von URLs, Caching, Security, usw.) bleiben dabei der Applikation selbst überlassen.

Im Rahmen des BIRT Projekts wird auf Basis des Report Engine APIs die *BIRT Viewer Web-Applikation* entwickelt, eine fertige Web-Applikation, die auf einem Anwendungsserver installiert werden kann um Reports über ein Netzwerk zur Verfügung zu stellen.

3.5.3 Chart Engine API

Das BIRT Chart Engine API ist ein eigenständiges, von BIRT Reports unabhängig definiertes API. Aus diesem Grund ist es möglich, BIRT Charts

nicht nur in BIRT Reports, sondern auch in andere Java Anwendungen zu integrieren.

Das Chart Report Element für BIRT Reports ist als Erweiterung von BIRT über den Erweiterungsmechanismus von Eclipse realisiert. Ähnlich wie das Report Design ist auch die Definition eines Charts in XML realisiert. Allerdings basiert die Struktur der Chartdefinition nicht auf einem XML Schema, sondern auf einem ECore Modell[3]. Auch die Zugriffsklassen des Chart Engine API wurde aus dem Chart ECore Modell generiert. Aus diesem Grund ist das API zum Teil unnötig komplex. Auch die Dokumentation ist zum Teil nur spärlich bis gar nicht vorhaden, so dass in seltenen Fällen nur noch der Blick in den XML-Code der Chartdefinition Rückschlüsse über die zu verwendenden Klassen und Methoden zulässt.

3.6 Weitere Informationen

Für einen genaueren Einblick in das Reporting Framework *BIRT* können die beiden Bücher [PHH06] und [WFB+06] empfohlen werden. Das erste der genannten Bücher verschafft dem Leser einen guten Einblick in die Erstellung von Report Designs mit Hilfe des *BIRT* Report Designers. Das zweite Buch behandelt die Möglichkeiten *BIRT* in eigene Applikationen zu integrieren oder durch das *Extension Framework* zu erweitern. Auch auf die Möglichkeiten *BIRT* Reports durch Scripts in Java oder Javascript zu erweitern wird in diesem Buch detailliert eingegangen.

[3] ECore Modelle werden mit Hilfe des *Eclipse Modeling Frameworks* erstellt. Diese bieten neben der Modellierung auch umfangreiche Generierungsmöglichkeiten von Zugriffsklassen für Modellinstanzen.

Kapitel 4

Implementierung

Dieses Kapitel befasst sich mit der Implementierung der einzelnen Frameworkskomponenten und einiger zusätzlicher Komponenten die zur weiteren Unterstützung des Entwicklers bei der Verwendung des Frameworks entwickelt wurden.

In Abschnitt 4.1 wird auf die Entwicklung der Bibliothek zum Lesen der Frameworkskonfiguration eingegangen.

Um den Generator für die BIRT Report Designs geht es in Abschnitt 4.2 ab Seite 37.

Anschließend folgt die Beschreibung der Implementation der drei Frameworkskomponenten Writer (ab Seite 46), Collector (ab Seite 62) und Viewer (ab Seite 69).

Der Abschnitt 4.6 ab Seite 82 beschäftigt sich mit der Entwicklung einiger Werkzeuge, die die Anwendung des Frameworks und einige dazu nötige Arbeitsschritte unterstützen.

4.1 Konfiguration

Wie in Abschnitt 2.2.1 beschrieben muss das Framework für jede Anwendung im Hinblick auf die Struktur der geloggten Daten konfiguriert werden können. Da die meisten Komponenten des Frameworks auf dieser Konfiguration basieren (näheres dazu jeweils in den Kapiteln über die einzelnen Komponenten) lag es nahe, den Zugriff auf die Konfiguration über ein Java API zu ermöglichen welches separat entwickelt und im Anschluss in die einzelnen Komponenten, welche Zugriff auf die Konfiguration benötigen, eingebunden wurde.

In den folgenden Abschnitten wird zunächst auf den Aufbau der Konfiguration eingegangen, sprich welche Aspekte des Frameworks konfigurierbar sind. Im Anschluss werden die Implementierung des erwähnten APIs erläutert und Besonderheiten bei dessen Entwicklung angesprochen.

4.1.1 Aufbau der Konfiguration

In diesem Abschnitt soll der Aufbau der Konfiguration veranschaulicht werden.

Wie bereits in Abschnitt 2.2.1 festgelegt muss es möglich sein Attribute festzulegen, die das gesamte Reporting der Applikation betreffen. Dazu gehören folgende Konfigurationsmerkmale:

Name der Applikation Der Name der Applikation ist ein wichtiges Attribut um die Zuordnung und Gruppierung von Reports zu ermöglichen.

Verzeichnis für Log-Dateien Für die Writer-Komponente ist es wichtig zu wissen, in welches Verzeichnis die Log-Dateien geschrieben werden sollen. Auch diese Angabe soll für die gesamte Applikation gelten.

Verbindungsparameter für Datenbank Gemäß der in Abschnitt 2.2.3 formulierten Anforderung sollen die geloggten Daten in einer Datenbank persistent gespeichert werden. Aus diesem Grund müssen für die verschiedenen Komponenten, die auf die Daten zugreifen müssen (Collector zum Einfügen der geloggten Daten, *BIRT* zum Generieren des Reports) die entsprechenden Verbindungsparameter konfiguriert werden.

Neben den allgemeinen Informationen für das Reporting einer Applikation muss natürlich vor allem die Struktur der zu loggenden Daten in der Konfiguration festgelegt werden. Zunächst ist in diesem Zusammenhang wichtig jedem zu loggenden Parameter einen Namen zu geben um eine eindeutige Zuordnung zu ermöglichen. Vor allem für das Einfügen der Daten in die Datenbank aber auch für die spätere Formatierung der

Daten im Report ist es zusätzlich von Bedeutung den Typ eines Parameters mit anzugeben.

Um dem Entwickler die Möglichkeit zu geben mehrere Parameter gleichzeitig zu loggen, da diese unter Umständen semantisch zusammengehören, werden diese in einem sogenannten *LogRecord* zusammengefasst. Dabei kann ein *LogRecord* entweder einen oder mehrere Parameter enthalten.

Damit bereits in der Konfiguration Einfluss auf die Gestaltung des Reports genommen werden kann können für jeden *LogRecord* zu diesem Zweck weitere Angaben gemacht werden. Zum Beispiel kann konfiguriert werden, ob auf dem Report ein Diagramm auftauchen soll und wenn ja welche Parameter in dem Diagramm dargestellt werden sollen.

Aus den eben genannten Konfigurationsmöglichkeiten ergibt sich eine baumartige Hierarchie wie auch in Abbildung 4.1 dargestellt.

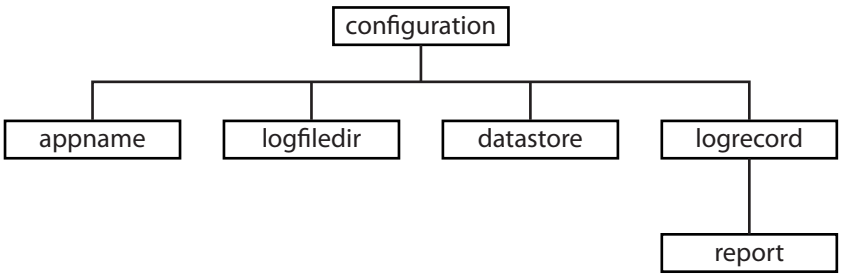

Abbildung 4.1: Baumartige Hierarchie der Konfiguration

Zur Repräsentation einer solchen baumartigen Struktur in einer Datei ist XML gut geeignet, da bei diesem Format die baumartige Struktur sowohl semantisch (durch die Verschachtelung der einzelnen Elemente) als auch optisch (bei entsprechender Einrückung von Unterelementen) klar ersichtlich ist. Deshalb wurde diese Technologie auch für diesen Zweck gewählt. Die genaue Syntax der XML Konfigurationsdatei und die Bedeutung der einzelnen Elemente wird im folgenden Abschnitt genauer erklärt.

4.1.2 Syntax der XML Konfigurationsdatei

In diesem Abschnitt wird die Syntax der XML Konfigurationsdatei genauer erläutert. Dabei wird auch näher auf die Funktion der einzelnen Definitionen eingegangen.

<config> Das Element `<config>` ist das Wurzelelement der XML Konfigurationsdatei. Es enthält ein `<logfiledir>` Element, ein

<datastore> Element und ein oder mehrere <logrecord> Elemente. Die genaue Bedeutung dieser Elemente wird in den folgenden Absätzen näher erklärt. Außerdem besitzt das <config> Element ein Attribut mit dem Namen appname welches den Namen der Applikation enthält für die diese Konfiguration bestimmt ist.

<logfiledir> Das <logfiledir> Element enthält das Verzeichnis in welchem die Writer-Komponente die Dateien mit den geloggten Daten ablegt.

<datastore> Dieses Element enthält fünf weitere Unterlemente, die die einzelnen Verdindungsparameter für die Datenbank enthalten. Diese sind im Einzelnen:

1. <driver>: Enthält den Namen der JDBC Treiberklasse die für die Verbindung zu der Datenbank benötigt wird.
2. <url>: Enthält die URL über die die Verbindung zur Datenbank aufgebaut werden muss.
3. <username>: Dieses Element enthält den Benutzernamen für die Datenbankverbindung.
4. <password>: Enthält das Passwort das zu dem definierten Benutzernamen gehört.
5. <prefix>: Um zu vermeiden, dass innerhalb einer Datenbank die Tabellen verschiedener Applikationen den gleichen Namen haben und es somit zu Problemen kommt kann hier ein Präfix angegeben werden, der vor den Namen jeder Tabelle dieser Applikation gestellt wird.

<logrecord> Dieses Element kann in der Konfiguration einer Applikation ein- oder mehrmals vorkommen und enthält alle Informationen, die für die Generierung eines Report Designs notwendig sind. Dazu gehört zunächst das Attribut name das jeden LogRecord mit einem Namen versieht mit Hilfe dessen die einzelnen LogRecords unterschieden werden. Des Weiteren enthält das Element <logrecord> mindestens einmal das Element <logentry> und höchstens einmal das Element <report>.

<logentry> Dieses Element repräsentiert einen zu loggenden Parameter. Es hat keinen Inhalt, dafür zwei Attribute, nämlich zum Einen den Namen des Parameters im Attribut name und zum Anderen, im Attribut type, dessen Typ. Für weitere Erläuterungen zu den verfügbaren Datentypen siehe Abschnitt 4.1.4. Zusätzlich zu den hier definierten Werten wird immer die aktuelle Zeit eines Ereignisses erfasst.

Zum einen ist die Zeit zu der ein Ereignis auftritt häufig von Interesse, zum anderen wird sie für die Darstellung der Daten auf einem Diagramm benötigt.

<report> Das Element `<report>` ermöglicht wie bereits weiter oben erwähnt die Beeinflussung des Report Design Generators. Es enthält die Elemente `<library>`, `<include>`, `<group>` und `<chart>` die verschiedene Bereiche des resultierenden Report Designs beeinflussen.

<library> Mit Hilfe dieses Elements kann man den Report Design Generator dazu veranlassen einen Verweis auf eine Report Library in das generierte Report Design einzufügen. Das Element ist leer, enthält allerdings zwei Attribute die Angaben zu der zu importierenden Report Library enthalten. Das Attribut `path` enthält ihren Pfad; das Attribut `namespace` den Namespace der Report Library in dem generierten Report Design.

<include> Das `<include>` Element weist den Report Design Generator an Report Elemente aus einer eingebundenen Report Library in das generierte Report Design zu übernehmen. Auch dieses Element ist leer und hat mehrere Attribute. Das Attribut `type` legt fest von welchem Typ das zu importierende Element ist, `namespace` bestimmt aus welcher Report Library das Element importiert werden soll und `ref` gibt den Namen des Elements in der Report Library an.

<group> *BIRT* ermöglicht es zum Beispiel in einer Tabelle die Daten zu gruppieren. Dieses Element ermöglicht es eine solche Gruppierung festzulegen. Es hat wiederum selbst keinen Inhalt, dafür mehrere Attribute:

1. `name`: Gibt der Gruppe einen eindeutigen Namen.
2. `groupon`: Verweist auf das `<logentry>` Element welches den Parameter repräsentiert anhand dessen gruppiert werden soll.
3. `interval`: Bezieht sich die Angabe des Attributs `groupon` auf einen `<logentry>` vom Typ *date*, *time* oder *datetime*, so kann über dieses Attribut angegeben werden, dass die angezeigten Datensätze nicht nach exakten Werten sondern über ein angegebenes Intervall (z.B. *hour*, *day*, *week*) gruppiert werden sollen.
4. `range`: Mit dem optionalen Parameter *range* können bei Verwendung des Parameters *interval* mehrere Gruppen zusammengefasst werden. Ist der Wert des *interval*-Parameters beispielsweise *day* und der des *range*-Parameters *5*, so werden immer 5 *aufeinanderfolgende* Tage zu einer Gruppe zusammengefasst.

<chart> Ist dieses Element in der Konfiguration vorhanden wird zu dem generierten Report Design ein Diagramm hinzugefügt. Der Titel des Diagramms kann dabei durch das Attribut `name` des `<chart>` Elements festgelegt werden. Weiter enthält dieses Element ein oder mehrere `<series>` Elemente.

<series> Über das leere Element `<series>` kann konfiguriert werden welche Parameter eines LogRecords in dem Diagramm dargestellt werden sollen. Das `column` Attribut dieses Elements verweist auf einen mit einem `<logentry>` Element definierten Parameter. Mit dem optionalen `caption` Attribut kann zusätzlich die Beschriftung auf dem Diagramm für den dargestellten Parameter festgelegt werden.

Abbildung 4.2 gibt einen genauen Überblick über die Syntax der Konfiguration.

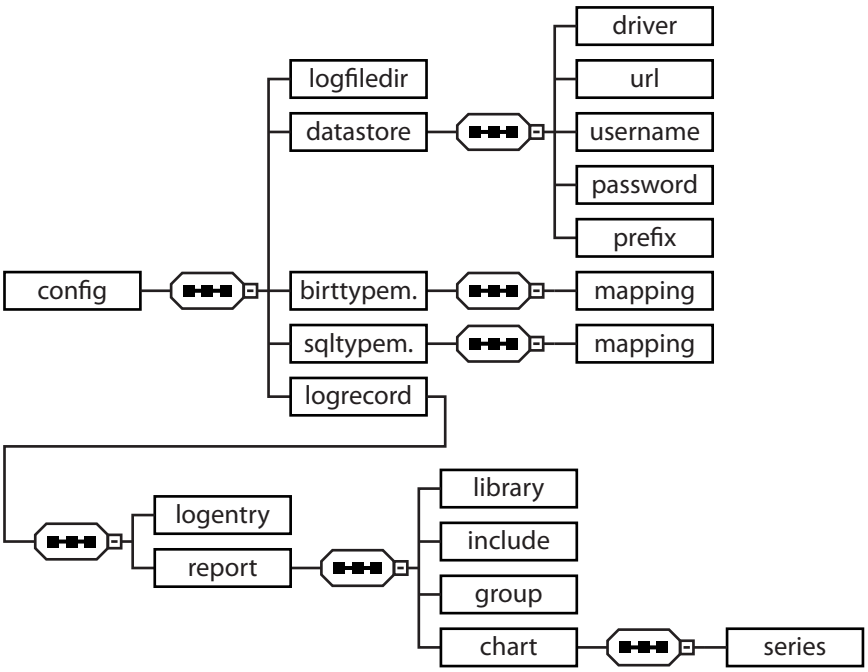

Abbildung 4.2: Überblick über die Syntax der Konfiguration

4.1.3 Implementierung

Wie bereits in der Einleitung zu diesem Kapitel erwähnt sollte der Zugriff auf die Konfiguration über ein Java API möglich gemacht werden. Als Ba-

sis für die Entwicklung des APIs diente die *configuration* Bibliothek, die im Rahmen des *Apache Commons* Projektes[1] entwickelt wird. Diese Bibliothek bietet eine generische Schnittstelle um Konfigurationsinformationen aus verschiedenen Quellen (darunter auch XML) zu lesen. Ist wie in diesem Fall die Konfigurationsquelle eine XML-Datei, so bietet *commons configuration* die Möglichkeit diese anhand einer *DTD*[2] zu validieren. Für die Auswahl einzelner Konfigurationswerte aus der XML-Datei können *XPath*-Ausdrücke[3] verwendet werden. Ein weiteres Feature von *commons configuration* welches bei der Implementierung des APIs Verwendung fand ist die Möglichkeit mehrere Konfigurationsquellen zu einer einzigen zu kombinieren.

Die Klassenstruktur des APIs wurde eng an die XML-Struktur der Konfigurationsdatei angelehnt. Das heißt, für jedes Element in der Konfiguration, welches mehrere Attribute oder Unterelemente besitzt wurde eine Klasse angelegt, die über Methoden Zugriff auf ihre Attribute sprich auf die Attribute des korrespondierenden Elements bietet (z.B. Element `LogRecord` → Klasse `LogRecordCfg`). Um Objekte einer dieser Klassen zu erzeugen muss ein Objekt vom Typ `HierarchicalConfiguration`[4], welches die Optionen eines Elements der Konfigurationsdatei enthält, an deren Konstruktor übergeben werden. Der Konstruktor extrahiert nun alle nötigen Informationen aus dem übergebenen Objekt und setzt die entsprechenden internen Attribute seiner Klasse. Das heißt jede Klasse ist selbst dafür verantwortlich, dass ihre Attribute korrekt gesetzt werden. Lediglich die Klasse `Config` die Zugriff auf alle Optionen der Konfiguration bietet weicht von diesem Schema ab und erhält als Konstruktorargument den Pfad zu der Konfigurationsdatei, die geladen werden soll.

Dieses Vorgehen hat mehrere Vorteile:

1. In Abschnitt 4.1.5 wird noch näher auf die testgetriebene Entwicklung dieser kleinen Bibliothek eingegangen. An dieser Stelle sei schon einmal vorweg genommen, dass der Umstand, dass jede Klasse für ihre korrekte Initialisierung selbst verantwortlich ist sehr zur Test- und Wartbarkeit des Codes beigetragen hat.

2. Bei Erweiterungen der Struktur der Konfigurationsdatei muss nur eine, maximal zwei Klassen geändert werden. Wenn zum Beispiel ein

[1] *Commons* ist ein Apache Projekt welches die Erstellung wiederverwendbarer Javakomponenten zum Ziel hat (`http://commons.apache.org/`).

[2] Eine *Document Type Definition* ist eine Textdatei in der über eine spezielle Syntax die Struktur eines XML-Dokuments festgelegt wird, also welche Elemente vorhanden sein dürfen und welche Attribute sie haben müssen bzw. können.

[3] XPath ist eine Sprache mit deren Hilfe Ausdrücke zum Auswählen von Elementen in einem XML Dokument formuliert werden können.

[4] Die Klasse HierarchicalConfiguration ist Teil der Bibliothek *commons configufation* und repräsentiert Konfigurationsdaten die aus hierarchischen Quellen wie zum Beispiel XML-Dateien gelesen werden.

weiteres Attribut zu einem Element hinzukommt, reicht es die Klasse die dieses Element repräsentiert entsprechend anzupassen. Wird ein ganz neues Element eingeführt muss dafür eine neue Klasse erstellt und die Klasse, die das Element repräsentiert dessen Kind das neue Element ist, entsprechend angepasst werden.

3. Für die Verwendung der Bibliothek in den einzelnen Komponenten ist diese Strukturierung ebenso von Vorteil, da alle für einen bestimmte Funktion benötigten Informationen über ein einzelnes Objekt zugänglich sind.

Um überprüfen zu können ob eine Konfigurationsdatei alle nötigen Elemente enthält wurde eine entsprechende *DTD* entwickelt und der entwickelten Bibliothek für das Lesen der Konfigurationsdateien hinzugefügt. Beim Laden einer Konfigurationsdatei wird anhand dieser *DTD* immer überprüft, ob diese den darin beschriebenen Regeln entspricht oder nicht. Wird beim Laden einer Konfigurationsdatei ein Verstoß gegen die *DTD* festgestellt, so wird ein entsprechender Fehler zurückgegeben, der den Verstoß beschreibt, um ihn schneller beheben zu können.

4.1.4 Datentypen

Für die Konfiguration der einzelnen Werte eines LogRecords stehen folgende Datentypen zur Auswahl: *string* (Zeichenkette), *date* (Datum), *time* (Uhrzeit), *datetime* (Datum mit Uhrzeit), *integer* (Ganzzahl) und *double* (Fließkommazahl). Die Zuordnung dieser internen Typen zu den Typen von *BIRT* bzw. der verwendeten Datenbank ist ebenfalls über die Klasse `Config` mit Hilfe der Methoden `getBirtTypeMappings()` bzw. `getSQLTypeMappings()` möglich. Die beiden Methoden liefern jeweils eine Java Map zurück, in denen der interne Datentyp dem jeweiligen Datentyp von *BIRT* bzw. *SQL* zugeordnet ist. In Abbildung 4.3 ist die standardmäßig vorgenommene Zuordnung zu sehen. Um diese Zuordnung den eigenen Wünschen anzupassen ist es möglich nach dem Element `<datastore>` die optionalen Elemente `<birttypemapping>` bzw. `<sqltypemapping>` in die Frameworkskonfiguration zu integrieren. Diese können wiederum mindestens ein leeres Unterlement `<mapping>` besitzen, welches zwei Attribute besitzt, über die eine bestimmte Zuordnung neu definiert werden kann. Das Attribut *type* bekommt dabei den Namen des neu zuzuordnenden internen Typs, das Attribut *mappedtype* enthält den Namen des diesem internen Typ neu zugeordneten Typs von *BIRT* bzw. *SQL*. Diese Möglichkeit ist vor allem bei Datebanken hilfreich, da diese unter Umständen eigene Datentypen besitzen, die unter Umständen optimiert und deshalb besser geeignet sind als die standardmäßig verwendeten Datentypen.

Intern	Birt	SQL
string	string	varchar
integer	integer	integer
date	date	date
double	float	double
time	time	time
datetime	date-time	datetime

Abbildung 4.3: Typzuordnung

4.1.5 Testgetriebene Entwicklung

Da die meisten Frameworkkomponenten das Konfigurations-API benutzen und zusätzlich die Struktur der Konfigurationsdatei während der Entwicklung der einzelnen Komponenten ständigen Änderungen unterlag war es wichtig während der Entwicklung des APIs ständig die korrekte Funktion aller Klassen und ihrer Methoden sicherstellen zu können. Aus diesem Grund wurde dieses kleine API *testgetrieben* entwickelt.

Ganz allgemein wird bei der *testgetriebenen Entwicklung* das gewünschte Verhalten einer sogenannten Unit (kleinste testbare Einheit, z.B. Java-Methode) in sogenannten *Unit-Tests* spezifiziert (getestet) *bevor* die entsprechende Funktionalität vorhanden ist. Anschließend wird die getestete Einheit implementiert bis der Test nicht mehr fehlschlägt. Wird ein neues Feature benötigt oder muss eine Änderung gemacht werden wird der Test entsprechend erweitert oder geändert damit er den neuen Bedürfnissen entspricht. Anschließend wird wieder die getestete Einheit so lange angepasst bis die Tests ohne Fehler ausgeführt werden können.

Gerade im Hinblick auf die Entwicklung von APIs also von Code, der später von Anderen benutzt werden soll hat dieses Vorgehen den Vorteil, dass der Entwickler, in dem Moment in dem er die Tests für das API, das er entwickeln möchte, schreibt, die Rolle des API-Benutzers einnimmt und so mehr Augenmerk auf die Benutzbarkeit der API-Methoden legt. Ein weiterer Vorteil ist die aus diesem Vorgehen resultierende hohe Testabdeckung[5], da neue bzw. andere Funktionalität nur dann hinzugefügt wird, wenn ein entsprechender Testfall existiert. Dadurch ist es sicherer Anpassungen am Code vorzunehmen, da die korrekte Funktionalität der getesteten Einheit über die Unit-Tests immer verifiziert werden kann. Des Weiteren wird durch den Umstand den Test zuerst zu entwickeln eine bessere Entkopplung der einzelnen Einheiten motiviert, da Einheiten mit we-

[5]Mit Testabdeckung ist hier das Verhältnis der tatsächlich getesteten Codezeilen zur Gesamtzahl aller Codezeilen gemeint.

nigen Abhängigkeiten einfacher zu testen sind. Zu guter Letzt dienen die programmierten Testfälle als *ausführbare Dokumentation* sprich als Anwendungsbeispiel für das entwickelte API.

Für die Implementation der Unit-Tests für das Konfigurations-API wurde das Framework *JUnit v4.0* benutzt. Typischerweise schreibt man für jede neue Klasse eine Testklasse und zumindest für jede `public`-Methode eine Testmethode. Die Testmethoden werden in *JUnit v4.0* mit Hilfe der Annotation[6] `@Test` gekennzeichnet. Des Weiteren stellt JUnit verschiedene Methoden zum Überprüfen erwarteter Ergebnisse zur Verfügung. So überprüft zum Beispiel die Methode `assertEquals(String, String)` ob die beiden als Argumente übergebenen String Objekte gleich sind. Trifft dies nicht zu löst JUnit eine Exception aus, was zum Fehlschlagen des Tests führt.

Um eine Einheit möglichst unabhängig von Anderen testen zu können bzw. um Einheiten zu testen die Abhängigkeiten zu (noch) nicht existierenden Systemen besitzen ist es notwendig diese bestehenden Abhängigkeiten durch sogenannte *Mocks* (engl. für Attrappe) zu ersetzen, die das Verhalten des *gemockten* Objekts, sprich der Abhängigkeit, simulieren.

```
1  package genericreports.config;
2
3  import static org.easymock.classextension.EasyMock.*;
4  import static junit.framework.Assert.*;
5
6  import org.apache.commons.configuration.
       HierarchicalConfiguration;
7  import org.junit.Before;
8  import org.junit.Test;
9
10 public class SeriesCfgTest
11 {
12   private SeriesCfg seriesCfg;
13   private HierarchicalConfiguration configMock;
14
15   @Before
16   public void setUp() throws Exception
17   {
18     configMock = createMock(HierarchicalConfiguration.class
         );
19
20     expect(configMock.getString("@column")).andReturn("
         column").anyTimes();
21     expect(configMock.getString("@caption")).andReturn("
         caption").anyTimes();
```

[6] Annotationen in Java ermöglichen die Einbindung von Metainformationen in den Quellcode. Diese Metainformationen stehen sowohl dem Compiler als auch zur Laufzeit über das Java Reflection API zur Verfügung. Siehe dazu auch [Wik08c]

```
22
23      checkOrder(configMock, false);
24
25      replay(configMock);
26
27      seriesCfg = new SeriesCfg(configMock);
28    }
29
30    @Test
31    public void testConstructor()
32    {
33      SeriesCfg seriesCfg = new SeriesCfg(configMock);
34
35      assertEquals("column", seriesCfg.getColumn());
36      assertEquals("caption", seriesCfg.getCaption());
37    }
38
39  }
```

Codebeispiel 4.1: Unit-Test am Beispiel der Klasse `SeriesCfgTest`

Im Codebeispiel 4.1 ist auszugsweise der Testcode für die Klasse `SeriesCfg` aus dem Konfigurations-API zu sehen. Die erste Methode in dieser Testklasse `setUp()` in Zeile 15 ist mit der Annotation `@Before` versehen. Dadurch wird diese Methode vor jeder anderen Testmethode durch das *JUnit* Framework aufgerufen.

Wie bereits weiter oben erwähnt wird jedes Objekt aus dem Konfigurations-API über einen eigenen Konstruktor erzeugt, welcher ein Objekt vom Typ `HierarchicalConfiguration` aus der *commons configuration* Bibliothek als Parameter übernimmt. Dieses Objekt enthält alle zum Erzeugen dieses eines Objektes nötigen Informationen aus der Konfigurationsdatei. Damit der Test unabhängig von der Konfigurationsdatei ausgeführt werden kann muss für das Konstruktorargument ein sogenanntes Mockobjekt erstellt werden welches sich genauso verhält wie das vom getesteten Code erwartete Objekt.

In der `setUp()` Methode wird zunächst in Zeile 18 mit Hilfe der statischen Methode `createMock(Class<T>)` aus der Bibliothek *EasyMock*[7] ein solches Mockobjekt erzeugt. In den Zeilen 20f wird das erzeugte Mockobjekt konfiguriert. Es werden zwei Aufrufe auf dem Mockobjekt konfiguriert. Zunächst erwartet es einen Aufruf der Methode `getString(String)` mit dem Parameter `"@column"`. Der Rückgabewert für diesen Aufruf soll der Wert `"column"` sein und der Aufruf wird in einer beliebigen Anzahl erwartet. Der zweite Aufruf der konfiguriert wird

[7] EasyMock ist ein Java Framework welches in Unit-Tests zur Laufzeit Mockobjekte für Java Interfaces erzeugt. Über eine Erweiterung ist es möglich auch Mockobjekte für Klassen erzeugen zu lassen. Weitere Informationen zu EasyMock auf `http://www.easymock.org/`.

verhält sich analog. In Zeile 23 wird das Mockobjekt so konfiguriert, dass es die konfigurierten Aufrufe in beliebiger Reihenfolge erwartet.

In Zeile 30ff wird eine Testmethode definiert (zu erkennen an der Annotation `@Test`). Wie am Namen der Methode zu erkennen ist soll hier der Konstruktor der getesteten Klasse getestet werden. Zunächst wird in der Testmethode ein Objekt der Klasse `SeriesCfg` erzeugt. Als Parameter wird das in der `setUp()` Methode erzeugte Mockobjekt übergeben. In den Zeilen 35f wird nun das erzeugte Objekt auf die erwarteten Werte getestet. Die Methode `assertEquals(String, String)` vergleicht die beiden übergebenen Argumente mit Hilfe der Methode `equals()`. Gibt diese den Wert `true` zurück läuft der Test weiter. Liefert die `equals()` Methode false, so wird der Test mit einem Fehler abgebrochen.

4.1.6 Build mit Ant

Damit das API in den anderen Projekten einfach verwendet werden kann sollten die resultierenden Klassen in einem Java Archiv (*JAR*, vgl. [Wik08b]) zusammengefasst werden.

Zwar enthält die verwendete Entwicklungsumgebung Eclipse einen Assistenten mit Hilfe dessen der Export der entsprechenden Klassen in eine JAR-Datei möglich ist jedoch erfordert ein solches Vorgehen die manuelle Durchführung mehrerer Schritte. Abgesehen davon dass ein solches Vorgehen nicht sehr bequem ist, ist es außerdem fehleranfällig. Viel besser ist es solche Aufgaben ganz oder zumindest teilweise zu automatisieren.

Ein Werkzeug zum automatisieren solcher Aufgaben das sich im Java-Umfeld großer Verbreitung erfreut ist *Ant*[8]. Das Besondere an *Ant* ist, dass es komplett in Java implementiert und somit zwischen verschiedenen Plattformen portabel ist. Die einzelnen *Task* genannten Funktionen von *Ant* können in einer XML-Datei, dem sogenannten *Build-Skript*, zur Bewältigung komplexer Aufgaben kombiniert werden.

Im Build-Skript für die Bibliothek mit den Klassen für das Konfigurations-API werden zunächst alle im Projekt vorhandenen Java-Klassen (API-Klassen und Testklassen) kompiliert. Im zweiten Schritt werden die in den Testklassen implementierten Testmethoden ausgeführt. Schlägt einer der Tests fehl wird der Vorgang nach dem Ausführen aller Tests abgebrochen. Erst wenn alle Tests ordnungsgemäß ausgeführt wurden werden die kompilierten API-Klassen und alle weiteren nötigen Resourcen in ein Java Archiv gepackt welches anschließend in den Classpath einer Java-Applikation aufgenommen werden kann um die darin enthaltenen Klassen verfügbar zu machen.

[8] Apache Ant ist ein OpenSource Projekt welches die Entwicklung eines auf Java basierenden Werkzeugs zur automatisierung von Software Builds zum Ziel hat. Weitere Informationen unter `http://ant.apache.org/`.

4.1.7 Beispiel einer Konfiguration

Das Codebeispiel 4.2 zeigt eine beispielhafte Konfiguration. Die Bedeutung der einzelnen Konfigurationsteile kann dem Abschnitt 4.1.2 entnommen werden. An dieser Stelle folgt nur eine kurze Zusammenfassung der Konfiguration.

Nach den allgemeinen Informationen zur Applikation wie dem Verzeichnise für die Log-Dateien in Zeile 5 und den Verbindungsparametern zur Datenbank in Zeile 7ff werden in dieser Datei zwei LogRecords konfiguriert (*QueryExecutionTime* in Zeile 15 und *RequestCount* in Zeile 27). Neben der Definition der einzelnen Parameter (`<logentry>` Elemente) enthalten die beiden Definitionen jeweils die Konfiguration für ein Diagramm (Element `<chart>`) auf dem die genannten Parameter visualisiert werden sollen.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE config PUBLIC "-//UBS//DTD XML GenericReports
       Config v1.0//EN" "config.dtd">
3  <config appname="ExampleApplication">
4
5    <logfiledir>logs</logfiledir>
6
7    <datastore>
8      <driver>org.hsqldb.jdbcDriver</driver>
9      <url>jdbc:hsqldb:hsql://localhost/</url>
10     <username>SA</username>
11     <password></password>
12     <prefix>exapp_</prefix>
13   </datastore>
14
15   <logrecord name="QueryExecutionTime">
16     <logentry name="exectime" type="double" />
17     <logentry name="logincount" type="integer" />
18
19     <report>
20       <chart name="Query Execution Time">
21         <series column="exectime" caption="query execution
               time" />
22         <series column="logincount" />
23       </chart>
24     </report>
25   </logrecord>
26
27   <logrecord name="RequestCount">
28     <logentry name="requestcount" type="integer" />
29
30     <report>
31       <chart name="Request Count">
32         <series column="requestcount" />
```

```
33        </chart>
34      </report>
35    </logrecord>
36
37  </config>
```

Codebeispiel 4.2: Beispielkonfiguration

4.2 Report Design Generator

Gemäß der in Abschnitt 2.2.5 formulierten Anforderung soll das Report Design für jeden Report basierend auf der Konfiguration generiert werden. Da wie in Abschnitt 2.3.4 beschrieben *BIRT* als Technologie für das Reporting gewählt wurde stand für diese Aufgabe das *Report Design API* von *BIRT* zur Verfügung.

4.2.1 Aufbau der generierten Reports

Bevor in den folgenden Abschnitten genauer auf die Implementierung des Generators eingegangen wird soll hier kurz beschrieben werden wie die aus der Konfiguration generierten Report Designs aufgebaut sein sollen.

Zunächst enthält jedes generierte Report Design Elemente, die nicht von der Konfiguration abhängen also in jedem generierten Report Design gleich sind. In erster Linie gehören dazu die im Report Design verwendeten Masterpages (siehe 3.3.1). Damit diese Elemente nicht für jedes Report Design neu generiert werden müssen wird eine *Report Library* (siehe 3.2.3) in das generierte Report Design eingebunden, die entsprechend gemeinsam genutzte Elemente enthält.

Als weiteres Element muss das generierte Report Design natürlich die Definition der Datenquelle enthalten. Wie bereits in 3.3.2 und 3.3.3 beschrieben muss in einem Report Design zunächst in einer *Data Source* die Quelle der Daten für einen Report festgelegt werden. Sie enthält im Falle einer Datenbank als Datenquelle die entsprechenden Verbindungsparameter. In unserem Fall werden diese in der Konfiguration im Element `<datasource>` und seinen Unterelementen angegeben. Anschließend wird über ein *Data Set* spezifiziert welche Daten aus der angegebenen *Data Source* gelesen werden sollen. Repräsentiert die *Data Source* wie in unserem Fall eine relationale Datenbank, so enthält das *Data Set* im wesentlichen eine SQL-Select-Abfrage die festlegt, welche Spalten aus einer bestimmten Tabelle in der angegebenen Datenbank gelesen werden und dem Report zur Verfügung stehen sollen. Der Aufbau dieser SQL-Abfrage lässt sich aus den `<logrecord>` Elementen und ihren `<logentry>` Kindelementen in der Konfiguration entnehmen. Ein Beispiel für eine aus einer `<logrecord>` Konfiguration erstellten SQL-Abfrage ist in Abbildung 4.4 zu sehen.

Wie bereits in Abschnitt 2.2.4 erwähnt muss es dem Benutzer über die Viewer Komponente möglich sein Start- und Enddatum für den Report auszuwählen. Um die vom Benutzer gewählen Werte für diese beiden Parameter an das Report Design zu übergeben werden im Reportdesign zwei Reportparameter (siehe 3.3.4) definiert. Wie bereits in Abschnitt 3.3.3 beschrieben können in der SQL-Abfrage eines *Data Sets* gewisse Platzhalter („?“) durch die Werte von Reportparametern ersetzt werden. Wie in Ab-

```
<logrecord name="QueryExecutionTime">
  <logentry name="exectime" type="double" />
  <logentry name="logincount" type="integer" />
</logrecord>
```

```
SELECT time, exectime, logincount
FROM queryexecutiontime
WHERE time >= ? AND time <= ?
ORDER BY time ASC
```

Abbildung 4.4: Konfiguration eines LogRecords und zugehörige SQL-Abfrage

bildung 4.4 zu erkennen enthält die aus der Konfiguration generierte SQL-Abfrage zwei Platzhalter, wobei der erste durch das Startdatum und der zweite durch das Enddatum ersetzt wird. Da die beiden Reportparameter für jedes Report Design unabhängig von der zugrundeligenden Konfiguration gleich sind werden sie aus der weiter oben erwähnten *Report Library* in das generierte Report Design importiert.

Schließlich muss das Report Design natürlich noch Elemente zur Darstellung der Daten enthalten. Da die Daten in einer relationalen Datenbank in tabellarischer Form vorliegen liegt es nahe sie in einer Tabelle auf dem Report anzuzeigen. Allerdings kann man durch diese Art der Darstellung die Daten nur sehr schwer visuell analysieren. Vor allem bei einer großen Menge von Daten lässt sich so kaum etwas aus dem Report ablesen. Aus diesem Grund wird zusätzlich auf dem Report ein Diagramm angezeigt, welches die ausgewählten Daten auch grafisch darstellt (zur Konfiguration des Diagramms siehe 4.1.2).

Zusätzlich zu den Report Designs für jeden konfigurierten LogRecord wir ein Report Design generiert, welches in einem Diagramm für jeden LogRecord die Häufigkeit darstellt mit der die entsprechenden Werte aufgezeichnet wurden. Um diese Häufigkeit zu berechnen wird direkt auf die Möglichkeiten von *BIRT* Diagrammen zurückgegriffen, nicht nur absolute Werte darzustellen sondern beispielsweise, wie in diesem Fall, für mit einem Zeitstempel versehenen Einträge in der Datenbank die Häufigkeit der Einträge pro Zeiteinheit.

Der hier erläuterte Aufbau der generierten Report Designs reizt natürlich nicht alle Möglichkeiten von *BIRT* aus. Und obwohl die von den generierten Report Designs genutzten Funktionen für viele Anwendungen ausreichen wird es Anwendungen geben die Anforderungen an die Report Designs stellen die über die Möglichkeiten der generierten Report Designs hinausgehen. Aus diesem Grund sind die generierten Report Designs als eine Art Gerüst zu sehen, die bereits viele Elemente enthalten die für je-

den Report benötigt werden (vor allem die Definition der Datenquelle und der Reportparameter). Werden an einen Report allerdings weitere Anforderungen gestellt so kann das generierte Report Design über den *BIRT Report Designer* jederzeit um weitere Aspekte erweitert werden.

4.2.2 Art der Implementation

Vor Beginn der Implementierung des Report Design Generators stellte sich die Frage nach der Art und Weise wie der Generator vom Entwickler benutzt werden sollte.

Die erste Möglichkeit, die hierfür in Betracht gezogen wurde war, den Report Design Generator als eigenständige Java Applikation zu entwickeln. Der Entwickler müsste dann, nachdem er die Konfiguration erstellt hat, den Report Design Generator entweder über die Kommandozeile oder über eine grafische Benutzeroberfläche aufrufen um aus der erstellten Konfiguration die entsprechenden Report Designs zu generieren. Allerdings ist diese Möglichkeit wenig komfortabel, da der Generator so als zusätzliches Werkzeug zu den bereits vorhandenen Entwicklungswerkzeugen zur Verfügung stehen würde. Das eigentliche Ziel war jedoch das Framework und alle dazu nötigen Arbeitsschritte möglichst nahtlos in die Entwicklung einer Applikation zu integrieren.

Da die Entwicklungsumgebung für Java Web-Applikationen bei der UBS auf Eclipse basiert lag es nahe die Generierung der Report Designs über ein Plugin in die Entwicklungsumgebung zu integrieren. Da wie in Abschnitt 3 beschrieben *BIRT* ebenfalls auf Eclipse basiert wäre ein weiterer Vorteil dieser Art von Implementation, dass die generierten Report Designs bei Bedarf gleich nach der Generierung in der Entwicklungsumgebung weiterbearbeitet werden könnten. Da die Erstellung und Bearbeitung der Konfiguration ebenfalls in Eclipse erledigt werden kann wäre so der gesamte nötige Workflow von der Erstellung der Konfiguration, über die Generierung der Report Designs bis hin zu deren eventuellen Nachbearbeitung in der gleichen Entwicklungsumgebung möglich, die auch für die restliche Applikationsentwicklung genutzt wird.

Da außer einer vermutlich geringeren Entwicklungs- bzw. Einarbeitungszeit die Umsetzung des Report Design Generators als eigenständige Java Applikation gegenüber dessen Umsetzung als Eclipse Plugin keine weiteren Vorteile hatte wurde letztgenannte Variante für die Implementation des Generators gewählt.

In den nun folgenden Abschnitten wird zunächst grundlegend auf die Entwicklung von Eclipse Plugins eingegangen und anschließend die Implementierung des Generators näher erläutert.

4.2.3 Entwicklung von Eclipse Plugins

Zur Unterstützung bei der Plugin-Entwicklung stellt Eclipse das *Plugin Development Environment (PDE)* bereit welches wiederum selbst ein Eclipse-Plugin ist. Es enthält einige nützliche Werkzeuge, so zum Beispiel Views, die Informationen über installierte Plugins bzw. deren Abhängigkeiten zu anderen Plugins enthalten.

Ein Plugin besteht aus mehreren Erweiterungen (extensions), die ein anderes Plugin, sei es nun aus dem Eclipse-Kern oder nicht, um weitere Funktionalitäten erweitern. Damit ein Plugin erweitert werden kann muss es sogenannte Erweiterungspunkte (extension points) definieren in denen genau festgelegt ist wie es erweitert werden kann. Sowohl Erweiterungen als auch Erweiterungspunkte werden in der Datei *plugin.xml* festgelegt. Zur Bearbeitung dieser Datei stellt das *Eclipse PDE* einen umfangreichen grafischen Editor zur Verfügung der den Entwickler bei der Definition von Erweiterungen und Erweiterungspunkten unterstützt.

In der Datei *MANIFEST.MF* welche in jedem Plugin im Unterverzeichnis *META-INF* vorhanden sein muss werden allgemeine Informationen wie der Name des Plugins und seine Version angegeben. Außerdem werden in dieser Datei andere Plugins definiert zu denen das neue Plugin Abhängigkeiten hat. Auch zur Bearbeitung dieser Datei steht ein grafischer Editor zur Verfügung, der zum Beispiel die Auswahl der Abhängigkeiten aus einer Liste ermöglicht. Die Implementation der Pluginfunktionalität erfolg in Java-Klassen. Für die Entwicklung des Java-Codes kann auf die sehr leistungsfähige Java Entwicklungsumgebung von Eclipse (unter Anderem auch auf dem Debugger) zurückgegriffen werden.

Muss ein Plugin beim Starten und beim Beenden gewisse Arbeiten erledigen (z.B. Initialisierung von internen Komponenten, Freigeben belegter Ressourcen) benötigt es einen sogenannten *Activator*. Ein *Activator* ist eine Java Klasse, die das Interface `BundleActivator` implementiert. Dieses Interface schreibt die beiden Methoden `start(BundleContext)` und `stop(BundleContext)` vor, welche jeweils vom Eclipse Framework zu Beginn und Ende der Lebenszeit eines Plugins aufgerufen werden.

Um ein in Entwicklung befindliches Plugin einfach testen zu können bietet das *PDE* die Möglichkeit direkt aus der Entwicklungsumgebung heraus eine zweite Instanz von Eclipse zu starten, die sogenannte *Runtime Workbench* (im Gegensatz zur *Development Workbench*, in der die Entwicklung der Plugins stattfindet). Zu diesem Zweck muss in der Eclipse Entwicklungsumgebung eine sogenannte *launch configuration* angelegt werden, in der alle Parameter für die zu startende *Runtime Workbench* festgelegt werden können. Einer der wichtigsten Parameter ist die Liste der Plugins, die in der *Runtime Workbench* verfügbar sein sollen. Hierbei hat man entweder die Möglichkeit alle in der *Development Workbench* verfügbaren Plugins oder nur ganz bestimmte davon in der *Runtime Workbench*

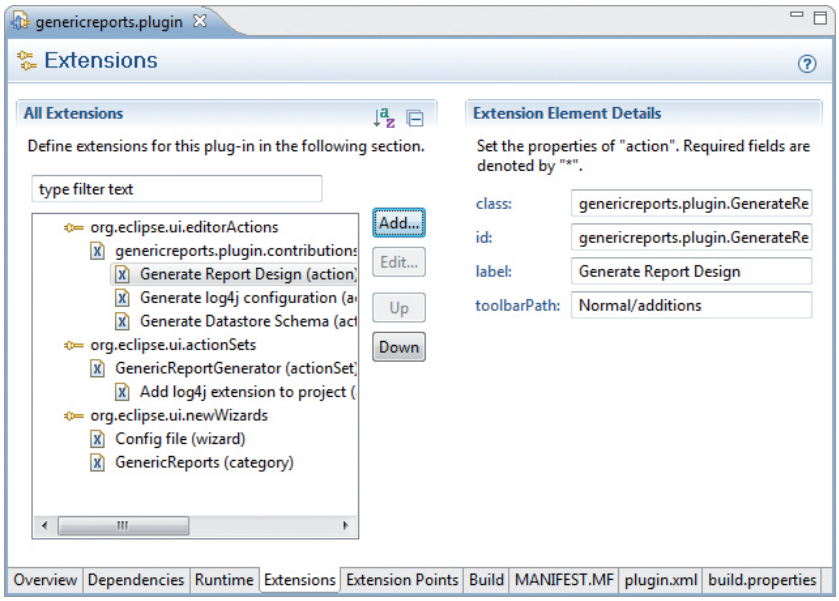

Abbildung 4.5: Eclipse Editor für plugin.xml

zu laden. In letzterem Fall ist darauf zu achten, dass alle von dem zu testenden Plugin benötigten Abhängigkeiten erfüllt werden. Sind alle nötigen Einstellungen gemacht kann die *Runtime Workbench* mit dem in Entwicklung befindlichen Plugin gestartet werden. Die eben getätigte Konfiguration wird gespeichert und kann so jederzeit wieder aufgerufen werden ohne die Einstellungen erneut vornehmen zu müssen.

Ist die *Runtime Workbench* gestartet hat man die gleichen Möglichkeiten, die einem auch beim Entwickeln von normalen Java Applikationen in Eclipse zur Verfügung stehen. So ist es zum Beispiel möglich in einer zu einem Plugin gehörenden Java Klasse einen Haltepunkt zu setzen, um die Ausführung des Plugins an einer bestimmten Stelle zu unterbrechen und dem Entwickler so die Möglichkeit zu geben die Werte einzelner Variablen bzw. Attribute zu überprüfen und die Ausführung des Plugincodes Schritt für Schritt nachzuvollziehen.

Ein weiteres Hilfsmittel bei der Entwicklung von Plugins ist das *Error Log*. Dabei handelt es sich um einen View, in dem Fehler, die während der Ausführung von Eclipse auftreten aufgelistet und gespeichert werden. Dieser enthält neben einer Beschreibung des Fehlers auch das verursachende Plugin und die genaue Zeit zu der der Fehler aufgetreten ist. Wenn verfügbar wird zu jedem Fehler ein Stacktrace[9] mit gespeichert, der möglicherweise genauere Hinweise auf die Ursache des Fehlers geben kann.

[9] Als Stacktrace bezeichnet man die Auflistung aller in einem Aufruf-Stack vorhadenen

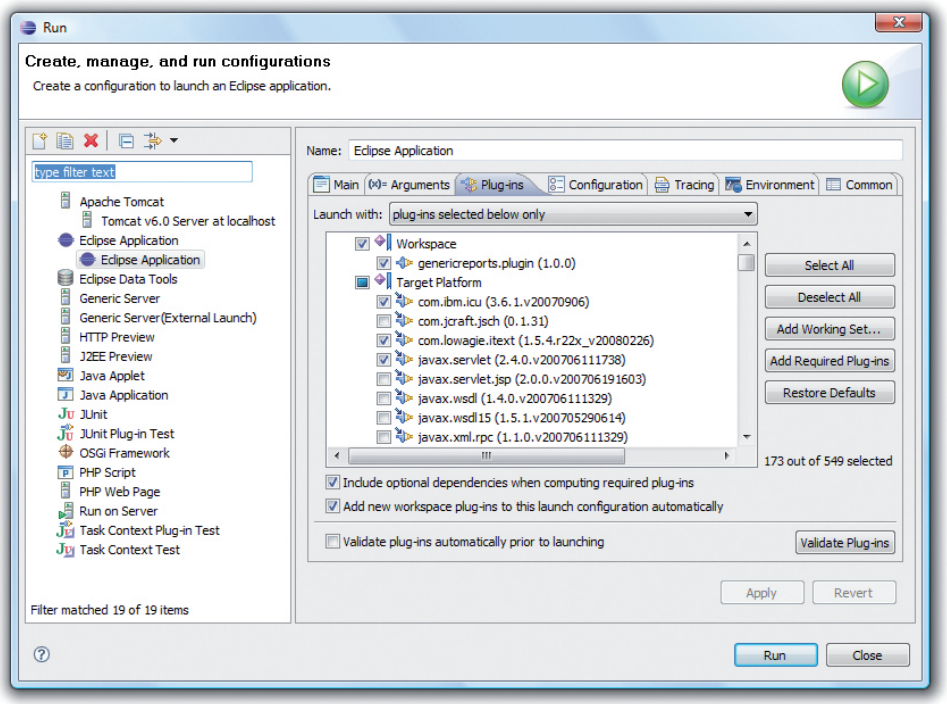

Abbildung 4.6: Plugin *launch configuration*

Weiterführende Informationen zur Entwicklung von Eclipse Plugins sind in [CR06] zu finden.

4.2.4 Implementierung

Wie bereits zu Beginn dieses Abschnitts erwähnt ist der Zweck des Report Design Generators aus der Konfiguration die entsprechenden Report Designs zu generieren. Wie in Abschnitt 4.2.2 beschreiben soll der Generator als Erweiterung der Entwicklungsumgebung *Eclipse* in einem Plugin umgesetzt werden.

Über eine sogenannte *Action* können Schaltflächen oder Menüeinträge in Eclipse zur Verfügung gestellt werden über die beliebige Funktionen aufgerufen werden können. Je nachdem in welchem Teil der Eclipse Benutzeroberfläche eine Action angezeigt werden soll gibt es von dieser verschiedene Ausprägungen. Da die Generierung der Report Designs auf einer Konfigurationsdatei im XML-Format basiert wurde für die Implementierung der Action für den Report Design Generator der extension point `org.eclipse.ui.editorActions` verwendet. Dieser ermöglicht es zu

Stackframes.

einem beliebigen Eclipse Editor (in diesem Fall dem XML-Editor) Menüs oder Actions hinzuzufügen die dann entweder über das Kontextmenü oder die Werkzeugleiste des entsprechenden Editors zur Verfügung stehen.

Die Action alleine ist zunächst nur die Repräsentation einer Schaltfläche in der Eclipse Benutzeroberfläche. Der Code, welcher die Funktionalität der Action implementiert befindet sich in einem sogenannten *ActionDelegate* (Java-Klasse, die ein entsprechendes Interface, z.B. `IEditorActionDelegate`, implementiert). Durch diese Trennung von grafischer Repräsentation und Funktionalität kann Eclipse die Schaltfläche der Action bereits anzeigen ohne das Plugin zu laden welches die Implementation der Funktionlität der Action enthält. Erst wenn die Action zum ersten mal benutzt wird lädt Eclipse das entsprechende Plugin nach (*lazy plugin initialization*). Dies hat zwar den Nachteil, dass der erste Aufruf der Action etwas länger dauert, dafür startet Eclipse zu Beginn schneller und benötigt auch weniger Speicher, da immer nur die Plugins geladen werden, die auch tatsächlich benutzt werden.

Die Implementation des *ActionDelegate* für die Report Design Generator Action befindet sich in der Klasse `GenerateReportDesignActionDelegate` (implementiert das Interface `IEditorActionDelegate`). In dessen `run(IAction)` Methode (diese Methode wird aufgerufen, wenn die Action ausgelöst wird) wird zunächst aus der Datei im aktuellen XML-Editor über das in Abschnitt 4.1 beschriebene API die Konfiguration geladen. Anschließend wird ein neues Objekt der Klasse `ReportDesignGeneratorRunnable` erzeugt welche die eigentliche Implementation des Generators enthält. Da diese Klasse das Interface `IRunnableWithProgress` implementiert kann beim Ausführen des Generators auf den *progress service* von Eclipse zurückgegriffen werden. Dieser stellt wie schon der Name erkennen lässt eine Fortschrittsanzeige für die laufende Aufgabe in einem Dialog zur Verfügung. In diesem Dialog befindet sich auch eine Schaltfläche um die Generierung vor deren Ende abzubrechen.

Die Generierung der Report Designs selbst erfolgt wie bereits erwähnt mit dem *Report Design API* von *BIRT*. Um gegenüber einer Implementierung des Generators in einer einzigen Klasse die Übersicht zu erhöhen und die Wartbarkeit des Generatorcodes zu verbessern wurden die einzelnen Schritte zur Generierung der Report Designs in eigenen Klassen implementiert (siehe Abbildung 4.7). Wie auf der Abbildung zu sehen gibt es das Interface `Command` für diese Klassen, welches zwei Methoden vorschreibt: Die erste Methode, `getCommandMessage()`, gibt eine kurze Beschreibung des in der jeweiligen Klasse implementierten Arbeitsschritts als `String` Objekt zurück. Diese Beschreibung wird dem Benutzer während des Generierungsvorgangs in der Fortschrittsanzeige angezeigt. Die zweite Methode, `execute()`, enthält in der das Interface implementierenden Klasse die Implementation des jeweiligen Arbeitsschritts. Die Klasse

`AbstractCommand` implementiert das eben beschriebene Interface, überlässt jedoch die Implementation der Methoden `getCommandMessage()` und `execute()` ihren Subklassen mit den konkreten Implementationen der einzelnen Arbeitsschritte. Hauptaufgabe dieser Klasse ist es den Subklassen gemeinsame Hilfsmethoden zur Verfügung zu stellen. Die Klassen welche die einzelnen Arbeitsschritte des Generators implementieren erben schließlich alle von der Klasse `AbstractCommand`.

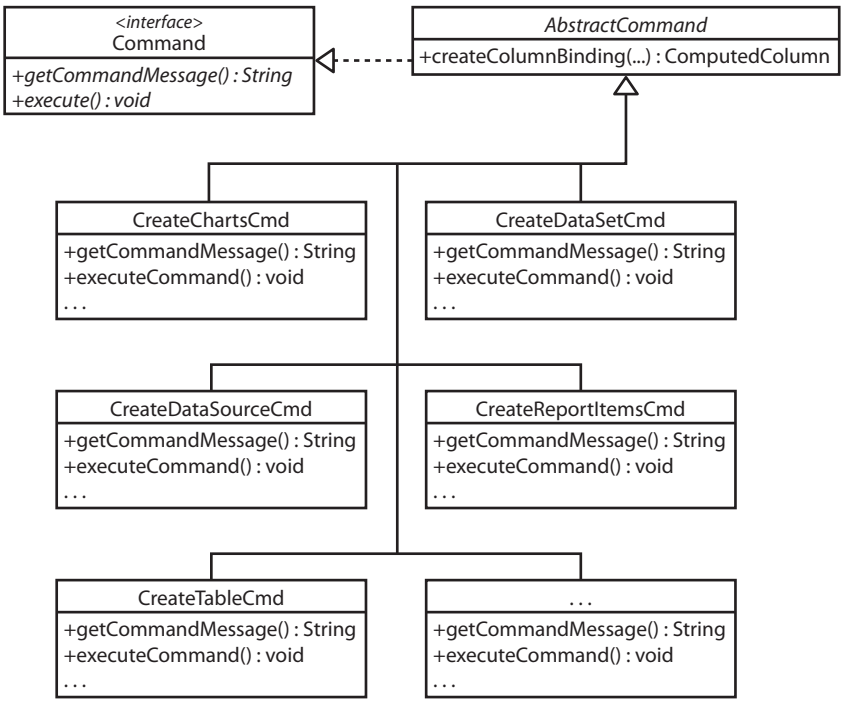

Abbildung 4.7: Klassenhierarchie zur Implementation der einzelnen Generatorschritte

In der Methode `run(IProgressMonitor)` der Klasse `ReportDesignGeneratorRunnable` findet die Generierung der Report Designs anhand der geladenen Konfiguration statt. Zu diesem Zweck werden für jedes zu generierende Report Design Objekte der die einzelnen Arbeitsschritte implementierenden Klassen instanziiert und in eine Liste eingefügt. Anschließend wird über diese Liste iteriert und auf jedem einzelnen in ihr gespeicherten Objekt die Methode `execute()` aufgerufen um den entsprechenden Generatorschritt auszuführen. Zusätzlich wird die mit Hilfe der Methode `getCommandMessage()` erhaltene textuelle Beschreibung des Arbeitsschritts dem Benutzer über die Benut-

zeroberfläche angezeigt. Außerdem wird in jeder Iteration überprüft, ob der Benutzer die Schaltfläche *Abbrechen* betätigt hat und gegebenenfalls die Generierung der Report Designs abgebrochen.

Mit der Entscheidung den Report Design Generator als Eclipse Plugin zu realisieren wurde gleichzeitig der Grundstein gelegt für die Implementation weiterer Werkzeuge zur Unterstützung bei der Anwendung des im Rahmen dieser Diplomarbeit entwickelten Frameworks. In Abschnitt 4.6 ab Seite 82 wird diese Idee auch bereits aufgegriffen um einige weitere, von der Konfiguration abhängige Artefakte, zu erzeugen.

4.2.5 Build mit Ant

Die in Abschnitt 4.2.3 erwähnte Möglichkeit das entwickelte Plugin über die Eclipse Entwicklungsumgebung zu starten ist natürlich für den produktiven Einsatz eines Plugins nicht praktikabel. Das Plugin muss sich vielmehr in die bereits in Verwendung befindliche Entwicklungsumgebung integrieren lassen. Um diese Integration möglichst einfach zu gestalten müssen alle zum Plugin gehörenden Dateien in einem Java Archiv (*JAR*) zusammengefasst werden. Auch für diese Aufgabe stellt Eclipse dem Entwickler einen entsprechenden Assistenten zur Seite. Allerdings ist wie schon in Abschnitt 4.1.6 beschrieben die Verwendung eines solchen Assistenten auf die Dauer lästig und, da wiederholt manuelle Schritte durchgeführt werden müssen, auch fehleranfällig.

Auch in diesem Fall wurde wie bereits in dem im vorigen Absatz erwähnten Abschnitt Ant für die Erstellung des gepackten Plugins verwendet. Das Script welches diesen Vorgang steuert ist im Anhang auf Seite 108 zu finden.

4.2.6 Benutzung des Report Design Generators

Die tatsächliche Anwendung des Report Design Generators soll hier nicht näher erläutert werden. Statt dessen wird an dieser Stelle auf das Kapitel 5 verwiesen, in dem die Benutzung aller Frameworkskomponenten an einem praktischen Beispiel gezeigt wird.

4.3 Writer

Wie im Abschnitt 2.2.2 festgelegt muss das Framework eine Komponente zur Verfügung stellen, die in Form eines APIs dem Applikationsentwickler die Möglichkeit gibt an den geeigneten Stellen im Applikationscode die gewünschten und vorher konfigurierten Parameter aufzuzeichnen. In den folgenden Abschnitten werden zunächst die Anforderungen, die von dieser Komponente erfüllt werden müssen festgelegt. Anschließend erfolgt eine kurze Evaluierung von Alternativen für die Umsetzung der Writer-Komponente und schließlich wird genauer auf die gewählte Umsetzung eingegangen.

4.3.1 Anforderungen an die Writer-Komponente

In diesem Abschnitt werden die Anforderungen, die an die Writer-Komponente gestellt wurden näher erläutert.

Speichern der gewünschten Daten Die Hauptaufgabe der Writer-Komponente innerhalb des Frameworks ist es dem Applikationsentwickler in Form eines APIs die Möglichkeit zu gebem die gewünschten und vorher konfigurierten Werte in eine Datei zu schreiben.

Automatischer Zeitstempel Zusätzlich zu den vom Entwickler an die API-Methoden übergebenen Werten soll bei jedem Aufruf ein Zeitstempel mitgespeichert werden um die geloggten Werte zeitlich einordnen zu können.

Semantische Annotation Um eine Zuordnung der Daten beim Einlesen der Datei durch die Collector-Komponente zu ermöglichen ist es notwendig, dass beim Schreiben der Daten diese in irgendeiner Form mit ihrer in der Konfiguration festgelegten Bezeichnung annotiert werden.

Einfach zu benutzen Eine weitere etwas weniger technische Anforderung an die Writer Komponente war, dass diese vom Entwickler einfach zu benutzen sein sollte und neben dem normalen Applikationscode möglichst wenig zusätzlichen Code erfordern sollte.

4.3.2 Design des Writer-APIs

Vor dem Entwurf eines Designs für das Writer-API wurden zunächst bereits vorhandene Bibliotheken für Logging begutachtet. Einer genaueren Analyse wurden dabei die beiden Frameworks *log4j* und *Java Logging APIs*

unterzogen. Beide sind dazu gedacht Textmeldungen an verschiedene Ausgabemedien (zum Beispiel Konsole, Datei, etc.) zu senden um über Fehler oder andere Vorgänge innerhalb der Applikation zu informieren. Im Verlauf dieser Analyse wurde deutlich, dass die beiden genannten Frameworks *log4j* und *Java Logging APIs* nicht nur als Vorlagen für das Writer-API sondern auch mehr oder weniger gut direkt zum Aufzeichnen der gewünschten Daten geeignet sind.

Im folgenden Abschnitt sollen nun die beiden APIs im direkten Vergleich miteinander beschrieben werden. Anschließend wird anhand dieses Vergleichs festgestellt, ob eines der APIs die in Abschnitt 4.3.1 gestellten Anforderungen erfüllt.

4.3.2.1 Vergleich von *log4j* und den *Java Logging APIs*

In diesem Abschnitt werden die beiden Logging Frameworks *log4j* und *Java Logging APIs* in einem direkten Vergleich unter verschiedenen Gesichtspunkten vorgestellt.

Log4j ist Teil des Apache Logging Services Projekts[10] in dessen Rahmen Logging Frameworks für verschiedene Programmiersprachen entwickelt werden (Java, C#, .NET, PHP). Die *Java Logging APIs* hingegen sind seit Version 1.4 fester Bestandteil des Java Sprachstandards und wurden bei ihrer Spezifikation in weiten Teilen dem bereits existierenden *log4j* nachempfunden weshalb beide Frameworks ähnliche Konzepte haben.

Logger genannte Objekte bieten in beiden Fällen Methoden an mit denen der Entwickler Log-Nachrichten an das Framework übergeben kann. Die Logger übergeben diese Nachricht an nachgelagerte *Appender* (log4j) bzw. *Handler* (Java Logging APIs) genannte Objekte, die das schreiben der einzelnen Log-Nachrichten auf verschiedene Ausgabemedien ermöglichen (Konsole, Datei, Socket, etc.). Ebenfalls in beiden Frameworks verfügbar sind Objekte, die sich um die Formatierung der auszugebenden Nachricht kümmern, im log4j-Umfeld werden diese *Layout*, im Zusammenhang mit den Java Logging APIs *Formatter* genannt. Um das Verhalten der Frameworks festzulegen bieten beide die Möglichkeit die entsprechende Konfiguration entweder im Programmcode oder über eine Konfigurationsdatei vorzunehmen welche in jedem Fall im *.properties*-Format, im Falle von log4j zusätzlich im XML-Format verfasst werden kann.

Obwohl beide Frameworks typischerweise für die Ausgabe von *Strings* verwendet werden erlaubt log4j an die entsprechenden Methoden der *Logger*-Klasse Objekte beliebigen Typs zu übergeben. Im Normalfall wird die Ausgabe von Objekten, die nicht vom Typ `String` sind über die `toString()` des entsprechenden Objekts erzeugt. Wird mehr Kontrolle über die Ausgabe dieser Objekte benötigt, besteht bei log4j die Möglich-

[10] http://logging.apache.org/

keit sogenannte *Renderer* zu registrieren, die für die Ausgabe von Objekten eines bestimmten Typs zuständig sind.

4.3.2.2 Entscheidung für log4j

Nach einem gründlichen Vergleich der beiden Frameworks fiel die Wahl auf log4j. Ausschlaggebend war, dass es gegenüber den Java Logging APIs bedingt durch seine lange Geschichte sehr weit ausgereift ist. Vor allem bei der Vielfalt an Appendern und Layouts können die Java Logging APIs nicht mithalten. Speziell für die Verwendung in diesem Framework ist das Konzept der Renderer ein sehr nützliches Feature. Ebenfalls dadurch bedingt, dass es schon viel länger verfügbar ist, ist seine Verbreitung wesentlich höher und auch die Menge an verfügbarer Dokumentation ist um einiges größer.

Eine genauere Beschreibung von log4j und seiner Funktionsweise ist im nun folgenden Abschnitt zu finden.

4.3.3 Log4j

Log4j ist ein weit verbreitetes Java Framework zum einfachen loggen von Meldungen in Applikationen. Es wurde im April 1999 zum ersten Mal veröffentlicht und wird seitdem unter der Apache Software Lizenz (Open-Source) von einer großen Entwicklergemeinde weiterentwickelt. Als Besonderheit sei hier erwähnt, dass *log4j* als Mindestanforderung die Java Umgebung in Version 1.1 vorraussetzt. Aktuell liegt *log4j* in der Version 1.2 vor die als sehr weit verbreitet und extrem stabil gilt. Die Entwicklung dieser Version beschränkt sich auf das Beheben von Bugs und kleinere Erweiterungen.

Log4j wurde aufgrund seiner großen Akzeptanz auch für einige andere Sprachen adaptiert (darunter C++, C#, Python, Ruby).

Abbildung 4.8 gibt einen Überblick über die Architektur von *log4j* wobei die grau dargestellten Komponenten optional sind. Die einzelnen Komponenten werden in den folgenden Abschnitten näher erläutert.

4.3.3.1 Logger

Logger sind die zentralen Objekte in *log4j*. Über sie werden die einzelnen Loganweisungen an *log4j* gesendet. Zu diesem Zweck ist in der Klasse `Logger` die Methode log(), die für verschiedene Argumente überladen ist, jedoch immer den Level (siehe Abschnitt 4.3.3.2) und das zu loggende Objekt als Argument entgegennimmt. Zusätlich gibt es in der Klasse `Logger` Methoden, die beim Aufruf als einziges Argument das zu loggende Objekt erwarten und intern die Methode `log()` mit einem bestimmten Level aufrufen (zum Beispiel die Methode `debug()` für den Level *DEBUG*).

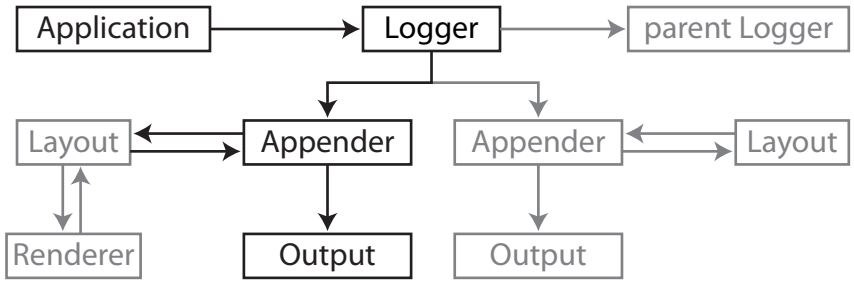

Abbildung 4.8: Überblick über die Architektur von log4j

Zusätzlich verfügen Logger Objekte über einen Namen anhand dessen sie nicht nur identifiziert werden können sondern durch welchen sie gleichzeitig in eine Baumstruktur eingeordnet werden. Die Beziehungen zwischen den einzelnen Logger Objekten folgen dabei folgender Regel:

Ein Logger ist genau dann Vorfahr eines anderen Loggers, wenn sein Name gefolgt von einem Punkt das Präfix des Namens des anderen Loggers ist.

Diese Regel soll anhand eines Beispiels kurz erläutert werden: Der Logger mit dem Namen *java* ist der direkte Vorfahre des Loggers mit dem Namen *java.lang*. Ein anderer Logger mit dem Namen *java.lang.String* wiederum ist dessen Kind.

Eine gängige Praxis ist es, die Logger nach dem voll qualifizierten Klassennnamen der Klasse in der sie verwendet werden zu benennen. Auf diese Weise entsteht eine Loggerhierarchie, die der Packagestruktur der Javaklassen der jeweiligen Applikation entspricht.

Die Wurzel dieser Hierarchie ist der sogenannte *Rootlogger*, welcher immer vorhanden ist und nicht explizit erstellt werden muss.

Diese Hierarchie hat in *log4j* zweierlei Auswirkungen:

1. Logger erben ihren Level von ihrem übergeordneten Logger, wenn für sie kein eigener Level definiert ist.

2. Loganweisungen, die an einen Logger geschickt werden schickt dieser nach der Verarbeitung an seinen übergeordneten Logger weiter. Dieses Weiterleiten von Loganweisungen kann unterbunden werden, wenn bei einem Logger Objekt das Attribut *additivity* auf `false` gesetzt wird.

4.3.3.2 Level

Jede Loganweisung erhält einen sogenannten Level, durch den gewissermaßen ihre Dringlichkeit zum Ausdruck gebracht wird. In *log4j* gibt es deren 5 nämlich *FATAL*, *ERROR*, *WARN*, *INFO* und *DEBUG* (in absteigender Reihenfolge der Dringlichkeit nach sortiert).

Für jeden Logger ist ebenfalls ein Level definiert. Der Level eines Loggers ist entweder direkt definiert oder von seinem Vorfahr geerbt. Durch den Vergleich der beiden Level (dem der Loganweisung und dem des Loggers) wird bestimmt, ob eine Loganweisung weiter bearbeitet oder unterdrückt wird. Eine Loganweisung wird nur dann weiter bearbeitet, wenn ihr Level größer oder gleich (im Bezug auf die Dringlichkeit) dem Level des Loggers ist.

4.3.3.3 Appender

In *log4j* werden die Ausgabekanäle für die Loganweisungen *Appender* genannt. Sie bestimmen, wohin die in den Loganweisungen enthaltenen Meldungen und Informationen geschrieben werden sollen. Im *log4j* Framework sind bereits einige Appender enthalten darunter unter anderem:

ConsoleAppender Die in einer Loganweisung enthaltene Meldung und Information wird in der Konsole ausgegeben von der aus die Applikation gestartet wurde.

FileAppender Die geloggte Information wird in eine Datei geschrieben. Von diesem Appender gibt es zusätzlich Varianten, die entweder Dateien nur bis zu einer bestimmen Größe beschreiben und danach eine neue Datei erstellen in die ab dann geschrieben wird oder in einem bestimmten Interval (zum Beispiel stündlich, täglich oder monatlich) eine neue Datei öffnen.

SocketAppender Der SocketAppender verbindet sich über ein Netzwerk zu einem durch IP-Adresse und Port festgelegten Socket und sendet die geloggten Informationen dort hin. Zur weiteren Verarbeitung muss an diesem Socket ein Programm lauschen und die empfangenen Nachrichten entgegennehmen.

4.3.3.4 Layout

Die sogannten Layouts sind für die Formatierung der in der Loganweisung enthaltenen Information zuständig. Ein sehr gebräuchliches Layout, das bereits in *log4j* integriert ist ist das PatternLayout. Dieses Layout kann man mit einer Zeichenkette konfigurieren, die bestimmte Platzhalter enthält, die dann durch die jeweiligen Informationen aus der Loganweisung

ersetzt werden (z.B. die Zeit der Loganweisung, der Thread in dem die Loganweisung getätigt wurde, etc.).

Durch selbst entwickelte Layouts ist es möglich die Formatierung der in den Loganweisungen enthaltenen Informationen den eigenen Anforderungen anzupassen. Dazu muss man eine Klasse implementieren, die von der Klasse `Layout` erbt. In der Methode `format(LoggingEvent)` dieser Klasse wird die gewünschte Ausgabe erzeugt. Das an diese Methode übergebene `LoggingEvent` Objekt enthält neben dem geloggten Objekt unter anderem den Zeitpunkt und den Thread in dem der Aufruf des Loggers erfolgte. Weitere Informationen wie zum Beispiel über den Ort des Aufrufs (Datei, Zeile, Klasse, Methode) werden bei Bedarf ermittelt.

4.3.3.5 Renderer

Der `log()` Methode eines Loggers werden im Normalfalls Objekte vom Typ `String` als Argument und damit als zu loggende Information übergeben. Der Inhalt dieser `String` Objekte wird dann unverändert an das *Layout* übergeben welches gegebenenfalls weitere Informationen wie zum Beispiel die Zeit hinzufügt und die so formatierte Lognachricht an den *Appender* übergibt.

Für Objekte, die nicht vom Typ `String` sind hält *log4j* einen *Renderer* bereit, der die für jede Javaklasse definierte Methode `toString()` auf dem Objekt aufruft um es zu serialisieren. Für weitergehende Anforderungen ermöglicht *log4j* die Implementation einer eigenen Rendererklasse. Diese muss das Interface `ObjectRenderer` und damit auch die dort vorgeschriebene Methode `doRender(Object)` implementieren. Zusätzlich muss man in der Konfiguration angeben, für welchen Objekttyp dieser Renderer verwendet werden soll.

Wird nun ein Objekt an die `log()` Methode eines Loggers übergeben für den ein entsprechender Renderer konfiguriert ist, so wird dieser Renderer dazu verwendet das Objekt zu serialisieren.

Eine weitere Besonderheit dieses Konzeptes ist, dass ein selbst definierter ObjectRenderer nicht nur für genau den Typ verwendet wird für den er konfiguriert wurde, sondern auch für dessen Subtypen. Hierzu ein Beispiel: Angenommen es liegt die in Abbildung 4.9 dargestellte Vererbungsbeziehung vor und für Objekte der Klasse `Article` ist die Klasse `ArticleRenderer` als Renderer konfiguriert. Wird nun an die `log()` Methode eines Loggers ein Objekt vom Typ `Book` übergeben, so wird aufgrund der Vererbungsbeziehung automatisch die Klasse `ArticleRenderer` zum serialisieren des Objekts benutzt.

Abbildung 4.9: Vererbungsbeziehung zwischen den Klassen Article und Book

4.3.3.6 Konfiguration

Log4j kann auf verschiedene Arten und Weisen konfiguriert werden auf die hier kurz eingegangen werden soll.

Konfiguration im Code *Log4j* kann durch Aufrufe des APIs konfiguriert werden. Allerdings ist diese Möglichkeit wenig praktikabel, da so bei einer Änderung der Konfiguration der Programmcode neu übersetzt und das Programm neu installiert werden müsste.

Konfiguration mit Hilfe einer Konfigurationsdatei Diese Möglichkeit der Konfiguration ist um Längen flexibler und einfacher zu handhaben. Um sie zu nutzen muss eine Konfigurationsdatei, wahlweise im Java Properties Format (key=value) oder im XML Format, mit den entsprechenden Definitionen im Classpath der Anwendung abgelegt werden. Dadurch ist es möglich die Konfiguration von *log4j* zu ändern ohne die Applikation neu kompilieren zu müssen. Zusätzlich wird die Konfigurationsdatei bei Änderungen neu gelesen und angewendet wodurch die Konfiguration von *log4j* sogar zur Laufzeit angepasst werden kann.

Zur Vervollständigung des Abschnitts zur Konfiguration folgt nun ein einfaches Beispiel mit anschließender Erklärung:

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">
3
4  <log4j:configuration debug="false" xmlns:log4j="http://
       jakarta.apache.org/log4j/">
5
6    <renderer renderedClass="java.util.Map" renderingClass="
         org.example.log4j.renderer.MapRenderer" />
7
8    <appender name="FILE" class="org.apache.log4j.
         DailyRollingFileAppender">
9      <param name="file" value="log/org.example.log" />
10     <param name="datePattern" value=".yyyy-MM-dd" />
11
12     <layout class="org.apache.log4j.PatternLayout">
```

```
13        <param name="ConversionPattern" value="%-4r [%t] %-5p
              %c %x - %m%n" />
14      </layout>
15    </appender>
16
17    <appender name="CONSOLE" class="org.apache.log4j.
          ConsoleAppender">
18      <param name="target" value="System.out" />
19
20      <layout class="org.apache.log4j.PatternLayout">
21        <param name="ConversionPattern" value="%-4r [%t] %-5p
              %c %x - %m%n" />
22      </layout>
23    </appender>
24
25    <logger name="org.example" additivity="false">
26      <level value="DEBUG" />
27
28      <appender-ref ref="FILE" />
29    </logger>
30
31    <root>
32      <level value="WARN" />
33
34      <appender-ref ref="CONSOLE" />
35    </root>
36
37  </log4j:configuration>
```

Codebeispiel 4.3: *log4j* Beispielkonfiguration in XML

Die vorliegende beispielhafte *log4j* Konfigurationsdatei definiert in Zeile 6 einen *Renderer* für alle Objekte des Typs `java.util.Map`. Anschließend werden zwei verschiedene *Appender* konfiguriert.

Der erste in Zeile 8ff mit dem Namen *FILE*, dem Typ `DailyRollingFileAppender` und folgenden Optionen:

- Die Datei in die die Loganweisungen geschrieben werden liegt im Unterverzeichnis „log“ der Applikation und heißt „org.example.log“.
- Der Wert „.yyyy-MM-dd“ des Parameters *datePattern* sorgt dafür, dass der Appender immer um Mitternacht die aktuelle Logdatei umbenennt und nachfolgende Loganweisungen in eine neue Datei schreibt.

Zusätzlich enthält die Konfiguration dieses Appenders in Zeile 12ff die Definition eines Layouts vom Typ `PatternLayout`.

Der zweite Appender wird in Zeile 17ff definiert, hat den Namen *CONSOLE* und ist vom Typ `ConsoleAppender`, gibt also die Loganweisungen an der Konsole aus.

Im Anschluss an die Konfiguration der Appender werden zwei Logger definiert:

Der erste Logger, der in Zeile 25ff konfiguriert wird hat den Namen „org.example“. Sein *additivity* Attribut ist auf `false` gesetzt. Außerdem ist sein Level auf den Wert „DEBUG“ gesetzt, das heißt alle Loganweisungen die an ihn geschickt werden werden auch verarbeitet (zur Erinnerung: *DEBUG* ist der von der Dringlichkeitsstufe am niedrigsten eingestufte Level). Die Loganweisungen leitet der Logger an den Appender mit dem Namen „FILE“ weiter.

Der zweite Logger der in dieser Konfigurationsdatei konfiguriert wird ist der Rootlogger (Zeile 31ff). Obwohl man den Rootlogger eigentlich nicht explizit konfigurieren muss kann man in der Konfiguration gewisse Standardwerte wie hier seinen Level oder seinen Appender neu definieren.

Davon ausgehend, dass in der Applikation die diese Konfigurationsdatei verwendet alle Packagenamen mit „org.example“ beginnen und die Logger entsprechend der Klassen in der Applikation (z.B. „org.example.Main“) benannt werden ergeben sich bei Verwendung oben stehender Konfiguration folgende Umstände:

- Loganweisungen, die aus Klassen kommen die zur Applikation gehören (sprich deren voll qualifizierter Klassenname mit „org.example“ beginnt) werden ohne Einschränkungen an den mit „FILE“ bezeichneten Appender geschickt und somit in die Datei „log/org.example.log“ geschrieben.
- Loganweisungen, die aus anderen Klassen kommen (beispielsweise eine Bibliothek, die ebenfalls *log4j* für das Logging benutzt) werden erst ab dem Level *WARN* (sprich nicht *INFO* und *DEBUG*) an der Konsole ausgegeben.

4.3.3.7 Weiterführende Informationen

Für eine detailliertere Beschreibung der einzelnen Features von *log4j* sei hier das Buch [GS03] empfohlen. Es enthält eine lückenlose Beschreibung aller Features sowie viele Konfigurationsbeispiele mit entsprechenden Erklärungen.

4.3.4 Format der Log-Dateien

Ein weiterer Aspekt, der bei der Implementation der Writer-Komponente eine Rolle spielte war die Wahl des Dateiformats für die zu schreibende Log-Datei. Da die aufzuzeichnenden Daten eine festgelegte tabellarische

Struktur haben, die für die weitere Verarbeitung erhalten bleiben muss, ist es notwendig, dass das gewählte Dateiformat diese Struktur in gewisser Weise erhält, damit die Collector-Komponente die Daten später wieder strukturiert einlesen kann.

Zwei Alternativen für ein tabellarisch strukturiertes Dateiformat sind *CSV* und *XML*. In den folgenden Abschnitten soll kurz auf die beiden Dateiformate eingegangen werden und anhand des Vergleichs der beiden Technologien eine Entscheidung für eines der beiden Formate getroffen werden.

4.3.4.1 CSV

In einer CSV-Datei (***C**omma **S**eparated **V**alues*) können Daten in tabellarischer Form sehr einfach gespeichert werden. Die Zeilen der repräsentierten Tabelle entsprechen dabei den Zeilen in der Datei. Die Werte der einzelnen Zellen innerhalb einer Zeile werden durch ein Trennzeichen separiert (typischerweise ein Strichpunkt). Um Bezeichner für die einzelnen Spalten einzuführen ist es üblich in der ersten Zeile der Datei die Spaltennamen ebenfalls durch Strichpunkte getrennt zu notieren. Durch die simple Formatierung der Datei entsteht wenig Overhead bei der Speicherung der gewünschten Daten. Um auf CSV-Dateien zuzugreifen stehen diverse APIs für Java zur Verfügung.

4.3.4.2 XML

Auch in einer XML-Datei können leicht tabellarische Daten abgelegt werden. Die Einteilung in Zeilen und Spalten erfolgt dabei durch die Verschachtelung der einzelnen Elemente. So steht das Wurzelelement der XML-Datei für die gesamte Tabelle. Dessen Unterelemente repräsentieren die Zeilen und deren Unterelemente wiederum die einzelnen Zellen der Tabelle. Die Benennung der Spalten ergibt sich somit aus den Namen der die einzelnen Zellen repräsentierenden Elemente. Dadurch, dass jede Zelle in einem eigenen Element eingeschlossen ist ergibt sich gegenüber dem CSV-Format ein größerer Overhead beim Speichern der Daten in einer Datei. Für den Zugriff auf XML-Dateien stehen in Java bereits APIs in Form von Interfaces zur Verfügung für die verschiedene Implementierungen existieren.

4.3.4.3 Entscheidung für XML

An dieser Stelle soll zunächst kurz auf die Art und Weise wie das API vom Entwickler benutzt werden soll und auf die Collector-Komponente eingegangen werde, da diese beiden Aspekte bei der Wahl des Dateiformats mit berücksichtigt werden mussten.

Wie in der Anforderung in Abschnitt 2.2.2 beschrieben soll der Entwickler die zu loggenden Daten als Schlüssel-Wert-Paare an die Methoden des Writer-APIs übergeben. Der Schlüssel ist dabei der Name eines Wertes wie in der Konfiguration festgelegt (vgl. auch in Abschnitt 4.1.2 die Hinweise zum `<logentry>` Element), der Wert wird jeweils vom Applikationsentwickler entsprechend übergeben. Des weiteren werden ein oder mehrere solcher Schlüssel-Wert-Paare in der Konfiguration in einem Logrecord zusammengefasst und bilden so ein Datenpaket, welches als Ganzes an das Writer-API übergeben werden muss. In Java kann eine solche Struktur am einfachsten in einem Objekt vom Typ `java.util.Map` abgelegt werden.

Die Collector-Komponente übernimmt in dem Framework die Aufgabe die aufgezeichneten Daten aus den Dateien in eine Datenbank einzufügen. Ein wichtiger Punkt bei diesem Vorgang ist natürlich, dass die Daten in der Datei wieder eindeutig ihrer Bedeutung zugeordnet werden können, damit ein korrektes Einfügen in die Datenbank möglich ist. Der Zusammenhang zwischen der Writer- und der Collector-Komponente ist in Abbildung 4.10 dargestellt.

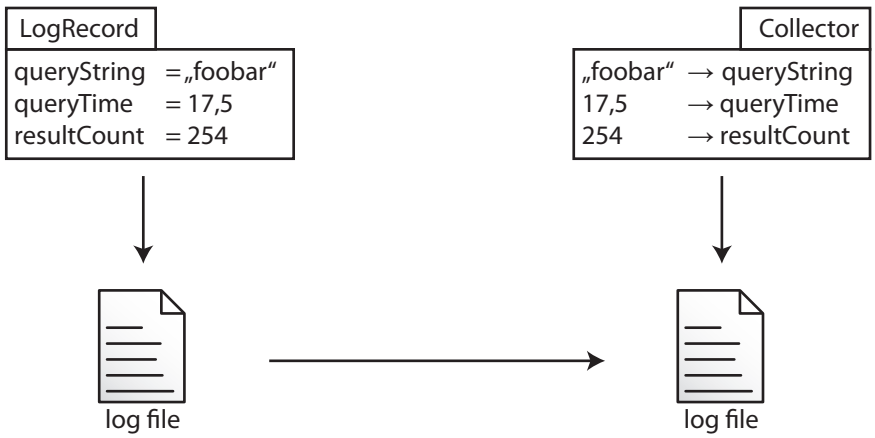

Abbildung 4.10: Zuordnung der Daten zu ihrer Bedeutung

Um bei einer CSV-Datei diese geforderte Zuordnung machen zu können ist es wichtig, dass die Daten in jeder Zeile immer in der gleichen Reihenfolge abgelegt sind und auch der in Abschnitt 4.3.4.1 erwähnten Kopfzeile entsprechen. Um dies zu gewährleisten müsste bei jedem Schreibvorgang der Daten in die entsprechende Datei von der Konfiguration die richtige Reihenfolge abgefragt werden um die Daten korrekt schreiben zu können.

Bei einer Speicherung der Daten im XML-Format hingegen kann die Zuordnung der einzelnen Daten zu ihrer Bedeutung über den Namen des sie umschließenden Elements gemacht werden. Dieser wird über die oben

erwähnte Map-Struktur beim Schreiben der Daten vom Entwickler mit übergeben.

Um das Schreiben der gewünschten Daten in die entsprechende Datei so einfach wie möglich zu halten wurde als Format für die Datei mit den geloggten Daten XML gewählt, da hier kein Zugriff auf die Konfiguration seitens der Writer-Komponente nötig ist. Zwar ist dadurch der Aufwand beim Einlesen der Datei durch die Collector-Komponente höher, da deren Ausführung die Applikation selbst jedoch nicht beeinflusst fällt dieser Umstand hier eher weniger ins Gewicht.

4.3.5 Implementation und Konfiguration

In diesem Abschnitt wird nun ausführlicher auf die Implementation des Writer-APIs mit Hilfe des Logging Frameworks log4j eingegangen.

Zunächst soll genauer auf die Konfiguration von log4j für die Verwendung in der Writer-Komponente eingeganen werden. Da die Daten jedes LogRecords in eine eigene Datei geschrieben werden sollen um das Lesen selbiger später durch die Collector-Komponente zu vereinfachen muss für jeden LogRecord ein eigener Logger mit einem eigenen Appender konfiguriert werden. Wie bereits einige Male erwähnt sollen die Daten für einen bestimmten LogRecord in einem `Map`-Objekt an den Logger übergeben werden. Damit die in diesem Objekt gespeicherten Daten auch korrekt ausgegeben werden (zur Erinnerung: Anhand der Argumente in Abschnitt 4.3.4.3 wurde entschieden, dass die Dateien mit den geloggten Daten im XML-Format abgelegt werden sollen) muss zusätzlich ein Renderer registriert werden, der die Ausgabe des Map-Objekts im korrekten Format übernimmt. Ebenfalls für die korrekte Formatierung der Ausgabe verantwortlich ist ein entsprechendes Layout. Auf die Implementation dieser beiden Klassen wird weiter unten näher eingegangen.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">
3
4  <log4j:configuration debug="false" xmlns:log4j="http://
       jakarta.apache.org/log4j/">
5    <renderer renderedClass="java.util.Map" renderingClass="
       genericreports.log4j.renderer.MapXMLRenderer" />
6
7    <appender name="QueryExecutionTime" class="org.apache.
       log4j.DailyRollingFileAppender">
8      <param name="file" value="logs/QueryExecutionTime.log"
         />
9      <param name="append" value="true" />
10     <param name="datePattern" value=".yyyy-MM-dd" />
11
12     <layout class="genericreports.log4j.layout.
         XMLReportLayout">
```

```
13        <param name="includeLocationInfo" value="false" />
14        <param name="includeProperties" value="false" />
15      </layout>
16    </appender>
17
18    <appender name="RequestCount" class="org.apache.log4j.
          DailyRollingFileAppender">
19      <param name="file" value="logs/RequestCount.log" />
20      <param name="append" value="true" />
21      <param name="datePattern" value=".yyyy-MM-dd" />
22
23      <layout class="genericreports.log4j.layout.
            XMLReportLayout">
24        <param name="includeLocationInfo" value="false" />
25        <param name="includeProperties" value="false" />
26      </layout>
27    </appender>
28
29
30    <logger name="QueryExecutionTime" additivity="false">
31      <level value="ALL" />
32
33      <appender-ref ref="QueryExecutionTime" />
34    </logger>
35
36    <logger name="RequestCount" additivity="false">
37      <level value="ALL" />
38
39      <appender-ref ref="RequestCount" />
40    </logger>
41
42  </log4j:configuration>
```

Codebeispiel 4.4: log4j Konfiguration für Writer-Komponente

In Codebeispiel 4.4 ist eine entsprechende log4j-Konfiguration für die Writer-Komponente zu sehen. Die Konfiguration der einzelnen Appender und Logger orientiert sich an der in Abschnitt 4.1.7 vorgestellten Frameworkskonfiguration. In den Zeilen 7 und 18 werden jeweils die zu den beiden LogRecords korrespondierenden Appender vom Typ *DailyRollingFileAppender* definiert, dessen einzelne Optionen bereits in Abschnitt 4.3.3.6 erläutert wurden. An dieser Stelle zu erwähnen ist, dass der Name des Appenders und der des entsprechenden LogRecords gleich sind. Dies ist zwar nicht zwingend erforderlich, erhöht allerdings die Übersicht in der log4j-Konfiguration. Der Wert des *file*-Parameters der Appender setzt sich jeweils aus dem in der Konfiguration definierten Verzeichnis für Log-Dateien (`<logfiledir>`) und wiederum dem Namen des LogRecords zusammen. Wie in dem Codebeispiel zu sehen wird für jeden Appender ein Layout

vom Typ `XMLReportLayout` konfiguriert. Ab Zeile 30 werden die benötigten Logger definiert. Auch diese bekommen den gleichen Namen wie die zugehörigen LogRecords. Ein wichtiges Detail an dieser Stelle ist das Attribut *additivity*, welches bei jedem Logger den Wert *false* bekommt. Dies verhindert, dass dieser Logger Lognachrichten an seinen übergeordneten Logger weitergibt und so die aufzuzeichnenden statistischen Daten in der normalen Log-Ausgabe der Applikation auftauchen. Da alle an diesen Logger geschickten Daten aufgezeichnet werden sollen, das Level der Log-Nachrichten also keine Rolle spielt, wird der Level jedes Loggers auf *ALL* gesetzt. Schließlich wird jedem Logger noch der entsprechende Appender zugewiesen.

Wie bereits erwähnt müssen die zu speichernden Daten ins XML-Format gebracht werden, damit sie von der Collector-Komponente gelesen werden können. Um dies zu erreichen macht sich die Writer-Komponente zwei Konzepte des Frameworks log4j zu Nutze, nämlich *Renderer* und *Layout*.

Wie bereits in Abschnitt 4.3.3.5 erwähnt sind Renderer Klassen, die das Interface `ObjectRenderer` implementieren. Dieses Interface schreibt die Methode `doRender(Object)` vor, deren Aufgabe es ist das übergebene Objekt für die Ausgabe zu serialisieren. Für die Writer-Komponente wurde die Klasse `MapXMLRenderer` implementiert, die übergebene Objekte vom Typ `Map` als XML serialisiert. Die Namen der XML-Elemente entsprechen dabei den Schlüsseln der in der Map abgelegten Objekte, deren `toString()`-Methode dazu verwendet wird um die Daten selbst zu serialisieren.

Das für das Framework eingesetzte Layout ist in der Klasse `XMLReportLayout` implementiert. Diese erbt von der Klasse `Layout` und implementiert folglich die Methode `format(LoggingEvent)`. Wie bereits in Abschnitt 4.3.3.4 beschrieben ist die Aufgabe dieser Methode die Informationen in dem übergebenen `LoggingEvent` Objekt für die Ausgabe entsprechend zu formatieren. Im Fall der Writer-Komponente stellt diese Methode der Ausgabe des Renderers den Zeitstempel in einem Element mit dem Namen `<time>` voran und schließt beides zusammen in einem `<logevent>` Element ein. Auf diese Weise wird der Zeitstempel jedem LogRecord automatisch hinzugefügt, der Entwickler muss diesen also nicht bei jedem Aufruf des Loggers selbst mitgeben. Eine weitere Aufgabe des XMLReportLayouts ist es die gesamte Ausgabe in einem `<log>`-Element einzuschließen. Für diesen Zweck existieren in der Klasse `Layout` die Methoden `getHeader()` und `getFooter()`, die jeweils einen String zurückgeben, welcher der Ausgabe eines Appenders vorangestellt respektive an diese angehängt wird.

```
1 <log>
2   <logevent>
3     <time>2008-10-01 11:11:47.925</time>
```

```
4     <exectime><![CDATA[249]]></exectime>
5     <logincount><![CDATA[4]]></logincount>
6   </logevent>
7   <logevent>
8     <time>2008-10-01 11:13:56.925</time>
9     <exectime><![CDATA[281]]></exectime>
10    <logincount><![CDATA[5]]></logincount>
11  </logevent>
12  ...
13 </log>
```

Codebeispiel 4.5: Beispiel einer Log-Datei

In Codebeispiel 4.5 ist ein Auszug aus einer durch die Writer-Komponente erzeugte Log-Datei zu sehen. Gut zu erkennen ist, dass jeder einzelne LogRecord in einem entsprechenden Element eingeschlossen ist. Jeder dieser LogRecords enthält dabei in jeweils einem eigenen Element seine Zeit und die entsprechenden geloggten Daten. Diese sind in Elementen eingeschlossen, deren Name dem des jeweiligen LogEntry-Elements aus der Konfiguration entspricht. Des Weiteren ist zu sehen, dass die Werte der einzelnen Elemente in sogenannten *CDATA-Sections* eingeschlossen sind. Diese Maßnahme ist nötig, da in den übergebenen Werten unter Umständen XML Sonderzeichen vorhanden sein können (zur Erinnerung: Die hier geloggten Werte können auch vom Typ *String* sein.). Durch die Notation mit den *CDATA-Sections* ignoriert der Parser eventuelle vorhandene XML Sonderzeichen, die ansonsten die korrekte Funktion des Parsers verhindern würden.

4.3.6 Erweiterbarkeit

Für den Fall, dass die gewünschten Daten allesamt in einem Container-Objekt vorliegen lässt sich die Writer-Komponente entsprechend erweitern um den Aufruf des Loggers zu vereinfachen. Diese Erweiterung ist über die Implementation einer zusätzlichen Renderer-Klasse möglich, die von der im vorigen Abschnitt beschriebenen Klasse `MapXMLRenderer` erbt. Die Methode `doRender(Object)` der neuen Klasse erwartet Objekte vom Typ des o.g. Container-Objekts und fügt die Werte von dessen Attributen in eine Map ein. Als Schlüssel sind hier wieder die Namen der LogEntry-Elemente des entsprechenden LogRecords aus der Konfiguration zu nehmen. Zum korrekten Serialisieren dieser Map wird einfach die `doRender(Object)`-Methode der Klasse `MapXMLRenderer` (der Superklasse) benutzt.

Anschließend muss natürlich noch die neu implementierte Klasse als Renderer für den entsprechenden Container-Objekt-Typ der log4j-Konfiguration hinzugefügt werden.

4.3.7 Build mit Ant

Auch für die Writer-Komponente wurde ein Skript für Ant verfasst um die beiden entwickelten Klassen zur einfacheren Verwaltung in einem Java Archiv zusammenzufassen. Dieses Skript befindet sich im Anhang dieser Diplomarbeit ab Seite 106.

4.3.8 Einbindung der Writer-Komponente

Auf die Anwendung der Writer-Komponente soll an dieser Stelle nicht näher eingegangen werden. Es sei jedoch auf das Kapitel 5 verwiesen, in dem die Funktionsweise des gesamten Frameworks anhand eines Anwendungsbeispiels genauer gezeigt wird.

4.4 Collector

Gemäß der in Abschnitt 2.2.3 formulierten Anforderung sollen die geloggten Informationen in einer Datenbank persistent gespeichert werden. Wie bereits in Abschnitt 3.3.2 beschrieben wäre es zwar möglich bei der Auswertung direkt auf die, wie in Abschnitt 4.3.4.3 erwähnt, im XML-Format vorliegenden Log-Dateien zurückzugreifen. Jedoch ergeben sich beim Datenzugriff über eine Datenbank wesentlich mehr Möglichkeiten die gesammelten Daten auszuwerten. Dies ist besonders im Hinblick auf die Anpassung der vom Report Design Generator generierten Report Designs an eigene Bedürfnisse interessant.

4.4.1 Verbindung zur Datenbank mit JDBC

Wie zu Beginn dieses Abschnitts über die Implementation der Collector-Komponente nochmals erwähnt sollen die Daten die von der Writer-Komponente in die Log-Dateien geschrieben wurden in eine Datenbank eingefügt werden um diese zum Einen persistent zu speichern und zum Anderen um die Auswertung der Daten zu vereinfachen.

Für den Zugriff auf relationale Datenbanken steht in Java das API *JDBC* (***J**ava **D**ata**b**ase **C**onnectivity*) zur Verfügung. Es stellt Methoden zur Verfügung welche den Zugriff auf bzw. das Verändern von Daten in einer relationalen Datenbank aus einem Java-Programm heraus ermöglichen. Das API selbst ist dabei unabhängig von der verwendeten Datenbank. Für jede Datenbank, die den Zugriff über *JDBC* unterstützt ist eine Implementation des APIs in einem sogenannten JDBC-Treiber vorhanden. Der Applikationsentwickler benutzt zur Programmierung einer Datenbankapplikation mit *JDBC* nur die im Java Sprachstandard verfügbaren Interfaces und Klassen. Die Applikation bleibt dadurch unabhängig von der verwendeten Datenbank. Lediglich zur Laufzeit einer Applikation müssen die entsprechenden Treiberklassen für die verwendete Datenbank im *CLASSPATH* der Applikation gefunden werden können.

Eine wichtige Eigenschaft von JDBC, die auch wie später noch beschrieben in der Implementation der Collector-Komponente Verwendung fand ist die Möglichkeit sogenannte *PreparedStatements* zu erstellen. Wie der Name schon andeutet stellen diese *PreparedStatements* die Möglichkeit zur Verfügung bestimmte Abfragen zur Vorbereitung an die Datenbank zu senden. In diesen Abfragen ist es zudem möglich durch Fragezeichen markierte Platzhalter zu definieren. Zu einem späteren Zeitpunkt können die definierten Platzhalter mit konkreten Werten belegt werden und die so entstandene Abfrage zur Ausführung an die Datenbank gesendet werden. Da diese PreparedStatements in optimierter Form in der Datenbank gespeichert werden bieten sie einen potentiellen Geschwindigkeitsvorteil gegenüber einem wiederholten Ausführen der gleichen Abfrage ohne vorige Vor-

bereitung. Des weiteren bieten *PreparedStatements* die Möglichkeit mehrere Ausführungen mit verschiedenen Werten im sogenannten *Batch* des *PreparedStatements* zu speichern und gebündelt auszuführen.

4.4.2 Datenbank

Die Datenbank dient als peristenter Speicher für die von der Collector-Komponente aus den Log-Dateien extrahierten Daten. Bei der späteren Generierung der Reports greift die BIRT Report Engine auf die Datenbank zu um die für einen Report benötigten Daten zu beziehen. Am besten lässt sich die tabellarische Struktur der geloggten Informationen in einer relationalen Datenbank abbilden. Wie in Abschnitt 4.4.1 erläutert, wird für die Verbindung zur Datenbank das Java-API JDBC benutzt. Aus diesem Grund kann hier nahezu jede relationale Datenbank verwendet werden, für die ein JDBC-Treiber verfügbar ist, der alle nötigen Features unterstützt. Das einzige von der Collector-Komponente benutzte erweiterte JDBC-Feature ist das in Abschnitt 4.4.1 erwähnte PreparedStatement. Auch die vom Report Design Generator generierten Report Designs verwenden in den Data Sets nur einfache SELECT-Abfragen um die Verwendung vieler verschiedener Datenbanksysteme zu ermöglichen.

Um während der Entwicklung des Frameworks den nötigen freien Zugriff auf die Datenbank zu haben, das heißt insbesondere schnell Tabellen anlegen und löschen zu können wurde *HSQLDB* als Entwicklungsdatenbank eingesetzt. *HSQLDB* ist eine vollständig in Java implementierte Datenbank die sowohl in Java-Applikationen eingebettet als auch im eigenständigen Modus als Server betrieben werden kann. Neben der geringen Größe war einer der größten Vorteile von *HSQLDB*, dass sie ohne Installation auskommt, was auf dem Rechner des Autors aufgrund von Zugriffsbeschränkungen nicht möglich gewesen wäre.

4.4.3 Lesen der Log-Dateien

Bevor die Daten in der Log-Datei in eine Datenbank eingefügt werden können müssen sie natürlich zunächst wieder korrekt aus der Log-Datei gelesen werden. Wie in Abschnitt 4.3.4.3 beschrieben liegen die Log-Dateien im XML-Format vor (siehe Codebeispiel 4.5 auf Seite 59). Um Daten im XML-Format zu lesen stehen in Java grundsätzlich zwei Ansätze zur Verfügung:

SAX Beim lesen einer XML-Datei generiert ein *SAX-Parser* (***S**imple **A**PI for **X**ML*) verschiedene Ereignisse (z.B. Start bzw. Ende eines Elements, Text wurde gelesen, Kommentar wurde gelesen). Diese Ereignisse werden an einen sogenannten *Handler* gesendet, der entsprechend darauf reagiert. Während der Parser universell eingesetzt wird benötigt man für jede Anwendung einen anderen Handler, der die

vom Parser generierten Ereignisse entsprechend den Anwendungsbedürfnissen behandelt. Der größte Vorteil dieses Ansatzes ist die Geschwindigkeit mit der XML-Dokumente gelesen und verarbeitet werden können sowie der geringe Speicherverbrauch.

DOM Im Gegensatz zu SAX steht beim DOM (***D**ocument **O**bject **M**odel*) ein API zur Verfügung welches direkten Zugriff auf den XML-Dokumentbaum erlaubt. Um dies zu ermöglichen wird eine Repräsentation des gesamten zu lesenden XML-Dokuments im Speicher aufgebaut. Anschließend ist es mit o.g. API möglich beliebig auf diesen Baum zuzugreifen und ihn zu verändern (Ändern von Knoten, löschen und hinzufügen von Knoten). Dieser große Unterschied zu SAX ist auch der größte Vorteil von DOM. Auf der anderen Seite ist beim DOM der Speicherverbrauch von der Größe des Dokuments abhängig.

Da die einzige Aufgabe der Collector-Komponente darin besteht die vom Writer erzeugten Log-Dateien zu lesen und entsprechend in die Datenbank einzufügen (also keine Manipulationen des Dokumentbaums gemacht werden müssen) ging die Entscheidung über das zu verwendende API zum Lesen der Log-Dateien klar zu Gusten von SAX aus.

Im nun folgnden Absatz wird kurz auf die Besonderheiten dieses APIs eingegangen und seine Verwendung näher erläutert.

4.4.4 SAX

Wie bereits im vorigen Abschnitt erwähnt erzeugt der Parser beim Lesen eines XML-Dokuments eine Serie von Ereignissen (z.B. Start bzw. Ende eines Elements, Text gelesen, etc.). Um nun in einer Applikation diese Ereignisse entsprechend zu verarbeiten, kann man beim Parser zum einen den sogenannten *content handler* registrieren. Dabei handelt es sich um eine Klasse, die das Interface *ContentHandler* implementiert und entsprechend den oben genannten Ereignissen Methoden enthält, die vom Parser entsprechend aufgerufen werden. Um auf Fehler während des Parsens entsprechend reagieren zu können kann man zusätzlich ebenfalls beim Parser einen *error handler* registrieren, der Methoden zur Behandlung von Fehlern enthält.

Die Java Interfaces des SAX gehören zum Java-Sprachstandard. Bei der Implementierung kann man zwischen mehreren zum Teil unter einer OpenSource-Lizenz verfügbaren Bibliotheken wählen. Eine dieser Bibliotheken wird zum Beispiel bei *Xerces*[11] entwickelt, einem Projekt welches bei der Apache Software Foundation beheimatet ist und die Entwicklung von Software zum parsen und manipulieren von XML-Dokumenten zum Ziel

[11] Nähere Informationen zu Xerces unter `http://xerces.apache.org/`

hat. Es stehen allerdings noch jede Menge anderer Parserimplementationen zur Verfügung, die zum Teil auch für unterschiedliche Zwecke verschieden gut geeignet sind.

Anders als DOM wurde SAX nicht formal spezifiziert. Vielmehr gilt die Java-Implementation als Referenz für die Implementation von SAX in anderen Sprachen. Im *Xerces*-Projekt werden zum Beispiel neben dem SAX-Parser für Java auch Implementationen für die Sprachen C++ und Perl entwickelt.

Weitere Informationen zu SAX wie z.B. eine Liste mit weiteren Parser-Implementationen und eine ausführliche Dokumentation ist im Internet auf der offiziellen SAX-Seite unter der URL `http://www.saxproject.org/` zu finden.

4.4.5 Implementation

In diesem Abschnitt wird nun näher auf die Implementation der Collector-Komponente eingegangen. Zunächst einmal musste dafür ein geeigneter SAX-Parser ausgewählt werden. Als maßgebliches Argument bei der Auswahl einer Parserimplementation entpuppte sich bereits zu Beginn ein Problem, welches von log4j verursacht wird:

Im Normalfall haben die von der Writer-Komponente erzeugten Log-Dateien das in Codebespiel 4.5 auf Seite 59 dargestellte Format. Wie bereits in Abschnitt 4.3.5 erwähnt werden die beiden Tags des `<log>`-Elements, welches alle LogRecords in einer Log-Datei enthält durch die Methoden `getHeader()` und `getFooter()` der Klasse `XMLReportLayout` erzeugt. Das Problem dabei ist, dass zum Beispiel der `DailyRollingFileAppender` beim Rotieren der Log-Datei die Methode `getFooter()` nicht aufruft und somit das schließende `</log>`-Tag auch nicht in der Ausgabe erscheint. Die resultierende Log-Datei ist somit kein wohlgeformtes[12] XML-Dokument.

Da zum Beispiel der SAX-Parser aus dem Xerces-Projekt ein Dokument während dem Parsen auf wohlgeformtheit überprüft würde beim Parsen einer Log-Datei bei der das schließende `</log>`-Tag fehlt jedesmal ein Fehler verursacht. An dieser Stelle musste also ein SAX-Parser benutzt werden, der sich gegenüber nicht wohlgeformten XML-Dokumenten tolerant verhält. Ein Parser mit dieser Eigenschaft ist zum Beispiel der *NekoHTML*-Parser[13]. Dieser ist wie der Name bereits vermuten lässt für das Parsen von HTML-Dokumenten gedacht. Da diese nicht den strengen syntakti-

[12] Ein wohlgeformtes XML-Dokument entspricht den XML-Syntaxregeln und darf zum Beispiel nur ein Wurzelelement besitzen. Ferner müssen alle Elemente korrekt geschlossen werden und korrekt verschachtelt sein.

[13] Der NekoHTML-Parser wird unter der Apache Lizenz v2.0 entwickelt. Weitere Informationen zu diesem Projekt sind auf der Projekthomepage unter `http://nekohtml.sourceforge.net/` zu finden.

schen Regeln von XML genügen müssen können sie in der Regel nicht so ohne Weiteres von einem XML-Parser verarbeitet werden. Der *NekoHTML*-Parser schafft hier Abhilfe, indem er zum Beispiel nicht korrekt geschlossene Tags automatisch schließt oder fehlende Elternelemente eines Elements hinzufügt. Für das parsen der Log-Dateien besonders wichtig ist die Möglichkeit das zu parsende Element als Fragment also als nicht vollständiges HTML-Dokument zu behandeln, da der Parser ansonsten alle fehlenden obligatorischen HTML-Tags automatisch hinzufügt (z.B. `<html>`, `<body>`).

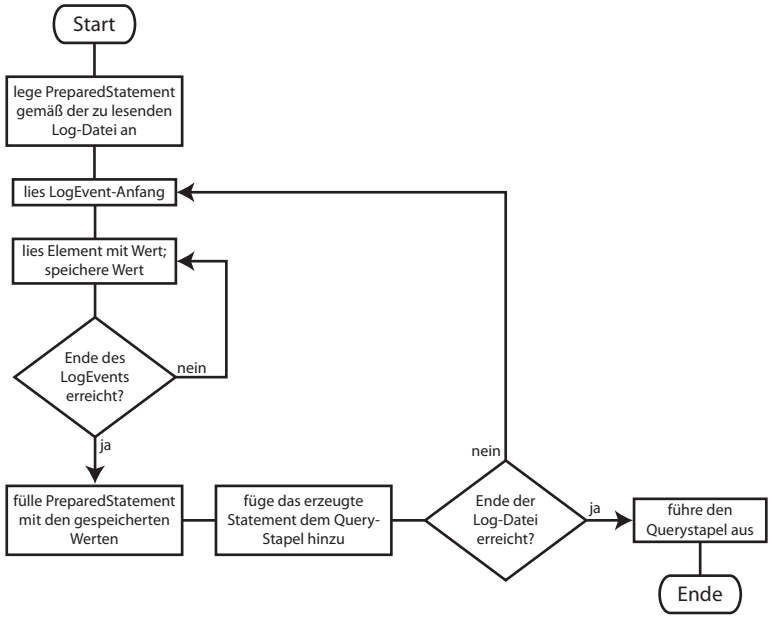

Abbildung 4.11: Schematischer Ablauf beim parsen einer Log-Datei

Um die vom Parser beim parsen der Log-Dateien generierten Ereignisse auch entsprechend verarbeiten zu können wurde ein *content handler* (siehe Abschnitt 4.4.4) entwickelt, der nach dem in Abbildung 4.11 dargestellten Schema vorgeht.

Zunächst wird aus der Konfiguration des LogRecords zu dem die zu parsende Log-Datei gehört ein *PreparedStatement* wie in Abbildung 4.12 erzeugt. Anschließend startet das Parsen der Log-Datei und die Verarbeitung der dabei generierten Ereignisse. Die Ereignisse, die von dem entwickelten *content handler* verarbeitet werden sind dabei folgende:

startElement Das öffnende Tag eines Elements wurde gelesen. Unter anderem wird der Name des Elements an die entsprechende Methode übergeben.

```
<logrecord name="QueryExecutionTime">
  <logentry name="exectime" type="double" />
  <logentry name="logincount" type="integer" />
</logrecord>
```

```
INSERT INTO queryexecutiontime
(time, exectime, logincount)
VALUES
(?, ?, ?)
```

Abbildung 4.12: Konfiguration eines LogRecords und zugehöriges SQL-PreparedStatement

endElement Das schließende Tag eines Elements wurde gelesen. Auch hier wird jeweils der Name des Elements mit übergeben.

characters Normaler Text wurde gelesen. Der gelesene Text wird an die Methode übergeben.

endDocument Das Ende des geparsten Dokuments wurde erreicht.

Da das Parsen der Log-Datei nicht in einer Schleife abläuft, sondern der *content handler* durch die oben genannten Methodenaufrufe über gewisse Ereignisse informiert wird muss dieser bestimmte Werte speichern damit bei jedem neuen Ereignis nachvollzogen werden kann welcher Teil der Log-Datei gerade geparst wird.

Wie in Abbildung 4.11 gut zu erkennen wiederholen sich die vom Parser gesendeten Ereignisse für jeden in der Log-Datei gespeicherten LogRecord. Wie in Abschnitt 4.3.5 beschrieben und auch im Codebeispiel 4.5 auf Seite 59 zu sehen sind die geloggten Werte eines LogRecords jeweils in einem Element welches entsprechend der Konfiguration benannt ist in einem `<logevent>`-Element zusammengefasst. Empfängt der *content handler* das Ereignis für das öffnende `<logevent>`-Tag, ist klar, dass jedes folgende *startElement*-Ereignis zu einem Element gehört, welches einen Wert enthält. Wird ein solches *startElement*-Ereignis vom *content handler* in diesem Zustand emfpangen, bedeutet das weiter, dass nachfolgend gelesener Text den Wert repräsentiert, der unter dem Namen des zuvor gelesenen Starttags von der Writer-Komponente gespeichert wurde. Das *endElement*-Ereignis für das Endetag dieses Elements bewirkt, dass nachfolgender Text nicht mehr gespeichert wird. Der gelesene Wert wird unter dem Namen des ihn umschließenden Elements in einer Map abgelegt. Anschließend werden weitere Werte auf die gleiche Weise gelesen, bis das schließende Tag des <*logevent*>-Elements geparst wird. Wird das *endElement*-Ereignis für das <*logevent*>-Element vom *content handler* empfangen, so werden die

Platzhalter in dem zu Beginn erzeugten *PreparedStatement* durch die Werte ersetzt, die im gerade geparsten LogRecord gespeichert waren und das so entstandene *INSERT-Statement* dem Batch des *PreparedStatements* hinzugefügt. Anschließend wird entweder wieder ein startElement-Ereignis für ein `<logevent>`-Element emfpangen oder das Ende der Log-Datei erreicht. Im ersten Fall wird die Map mit den gespeicherten Werten geleert und die oben beschriebene Prozedur beginnt von neuem. Wird das Ende der Log-Datei durch den Empfang eines endDocument-Ereignisses signalisiert, wird der Batch des *PreparedStatements* ausgeführt und somit die gelesenen Daten in die Datenbank eingefügt. Diese gebündelte Ausführung der *INSERT-Statements* hat den Vorteil, dass nicht für jeden LogRecord der in die Datenbank eingefügt werden soll eine Anfrage zur Ausführung an die Datenbank gesendet werden muss.

4.4.6 Verwendung der Collector-Komponente

Die Collector-Komponente bzw. das Parsen der Log-Dateien und das Eintragen der extrahierten Daten in die Datenbank ist ein wiederkehrender Prozess (so wie das rotieren der Log-Dateien durch die Writer-Komponente ebenfalls ein zum Beispiel täglich vollzogener Vorgang ist). Aus diesem Grund ist es natürlich wenig praktikabel diesen von Hand auszulösen. Sinnvollerweise läuft die Collektor-Komponente als Serverprogramm, welches automatisch ausgelöst seinen Dienst zu einer vorher festgelegten Zeit zum Beispiel einmal am Tag ausführt.

Da die in Abschnitt 4.5 beschriebene Viewer-Komponente ebenfalls als Serverprogramm ausgeführt wird und auf eine ähnliche Art und Weise implementiert ist wie die Collector-Komponente soll hier nicht näher auf deren Implementierung eingeganen werden. Statt dessen wird dieser Punkt in dem erwähnten Abschnitt über die Viewer-Komponente in Abschnitt 4.5.5 noch einmal aufgegriffen und genauer erläutert.

4.5 Viewer

Neben der Aufzeichnung und Speicherung potentiell interessanter Daten ist natürlich eine einfache und schnelle Möglichkeit zur (grafischen) Darstellung und Auswertung dieser aufgezeichneten Daten enorm wichtig. Diese ist auch in den Anforderungen an das Framework in Abschnitt 2.2.4 auf Seite 5 formuliert. Die Viewer-Komponente, deren Implementation in diesem Abschnitt erläutert wird, soll zunächst dem Benutzer alle verfügbaren Reports zur Auswahl anbieten. Des Weiteren soll es möglich sein die vom Report Design Generator erzeugten Report Designs bei Bedarf auszuführen und die dabei generierten Reports anzeigen.

Um den Zugriff auf die Reports weiter zu vereinfachen soll die Viewer-Komponente als Web-Applikation implementiert werden. Dadurch kann der Benutzer ohne die Installation einer Software auf seinem Rechner über den Browser zur Auswertung auf die Reports zugreifen.

4.5.1 Design der Viewer-Komponente

Wie bereits erwähnt soll die Viewer-Komponente den Zugriff auf alle mit Hilfe des Frameworks erstellten Report Designs ermöglichen. Zusätlich muss es für jeden Report der generiert werden soll möglich sein über einen Kalender Start- und Enddatum auszuwählen. Eine Skizze des gewünschten grafischen Aufbaus der Applikation ist in Abbildung 4.13 zu sehen:

- Auf der Linken Seite des Applikationsfensters soll sich die Navigation befinden, die die Eingabe des Datums und die die Auswahl des gewünschten Reports ermöglicht. Die verfügbaren Reports sollen dabei nach der jeweiligen Applikation zu der sie gehören gruppiert werden.

- Auf der rechten Seite des Fensters soll nach Eingabe sämtlicher benötigter Informationen der generierte Report zu sehen sein. Außerdem soll es hier möglich sein in einem mehrseitigen Report zu blättern und verschiedene Exportoptionen zu wählen.

Für die Erzeugung und Darstellung der Reports aus den vom Report Design Generator generierten Report Designs soll die in Abschnitt 3.1.4 erwähnte BIRT Web-Viewer Applikation in die Viewer-Komponente integriert werden, damit die Generierung er Reports nicht erneut implementiert werden muss, zumal die BIRT Web-Viewer Applikation bereits mit einer ausreichenden Fülle von Features ausgestattet ist.

Um die Implementation der Viewer-Komponente geht es im nun folgenden Abschnitt 4.5.2, mit der Integration des BIRT Web-Viewers beschäftigt sich der Abschnitt 4.5.4.

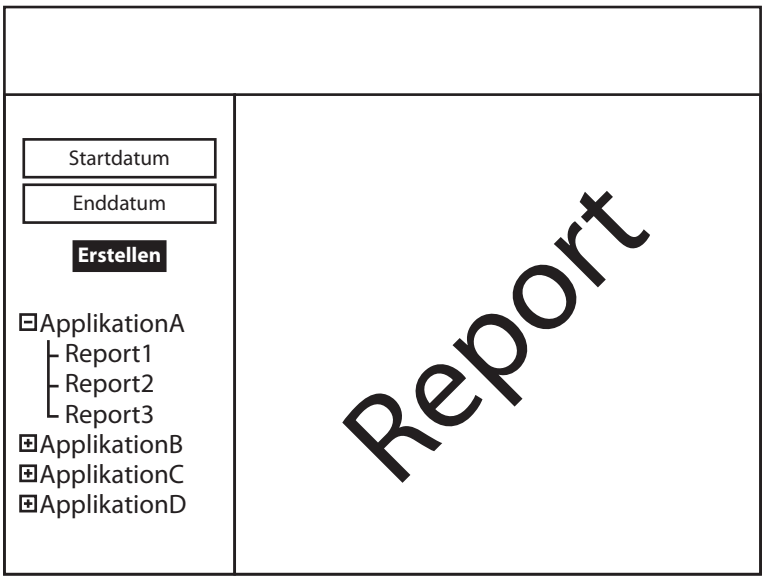

Abbildung 4.13: Layoutplan der Viewer Web-Applikation

4.5.2 Implementation mit Hilfe des Frameworks *Spring*

Um die Implementation der Viewer-Komponente zu vereinfachen und um deren Modularität und somit auch deren Erweiterbarkeit zu erhöhen wurde das Java-Framework *Spring* eingesetzt. Im nun folgenden Abschnitt wird kurz auf das Framework eingegangen um vor allem die für die Implementation der Viewer-Komponente wichtigen Konzepte zu erläutern während der Abschnitt danach sich mit der Implementation der Viewer-Komponenten selbst beschäftigt.

Spring ist ein OpenSource Framework, welches von der Firma *SpringSource* (`http://www.springsource.com/`) kreiert wurde und auch laufend weiterentwickelt wird. Neben den Kernfunktionen von Spring, die von jeder Java-Applikation genutzt werden können gibt es verschiedene Erweiterungen, die die Entwicklung von Web-basierten Applikationen auf Basis der *Java Enterprise Platform* unterstützen. Für die Entwicklung der Viewer-Komponente waren vor allem folgende Teile von Spring von Bedeutung:

Inversion of Control Container (IoC) Diese Komponente ist in Spring von zentraler Bedeutung und stellt einen einheitlichen Weg zur Verfügung Java Objekte zu konfigurieren und Abhängigkeiten dieser Objekte untereinander zu verwalten. Objekte, die durch den IoC con-

tainer verwaltet werden bezeichnet man in Spring auch als *managed objects* oder *Beans*. Die Konfiguration des IoC Containers erfolgt in aller Regel in einer XMl-Datei (dem *Application Context*, die die Definition der einzelnen *Beans* sprich alle zum Erzeugen der Objekte notwendigen Informationen enthält. Der Zugriff auf Objekte im Container erfolgt entweder durch *dependency lookup*[14] oder durch *dependency injection*[15].

Movel-View-Controller Framework In Spring enthalten ist auch ein eigenes MVC-Framework[16] zur strukturierten Entwicklung von Web-Applikationen. Als Model kann im Spring WebMVC jedes Objekt, angefangen bei einem einfachen String bis hin zu einem Datenbankabstraktionsobjek, dienen. Der Controller-Teil einer Spring WebMVC-Applikation besteht zunächst aus der Klasse `DispatcherServlet`, die alle HTTP-Anfragen, die an diese Applikation gesendet werden entgegennimmt (*front controller*). Anschließend wird mit Hilfe des sogenannten *handler mappings* entschieden, welche Controller-Klasse innerhalb der Applikation eine bestimmte Anfrage entgegennimmt und verarbeitet. Zur Erzeugung der HTML-Seiten (View), können verschiedene Technologien wie zum Beispiel *Java Server Pages*[17] oder *Velocity*[18] eingesetzt werden.

Scheduler Framework Ferner unterstützt Spring die Ausführung von Aktionen, die nicht vom Benutzer sondern automatisch basierend auf einem zuvor festgelegten Zeitplan ausgelöst werden. Zum festlegen dieses Zeitplans können verschiedene Implementation zum Einsatz kommen, zum Beispiel die im Java-Sprachstandard vorhandene Klasse `Timer` oder das OpenSource Framework *Quartz*[19].

Neben den hier vorgestellten verfügt Spring noch über einige weitere Komponenten, die die Entwicklung von Java-Applikationen im Enterprise-

[14] Der Aufrufer fragt beim Container-Objekt nach einer Bean mit einem bestimmten Namen oder von einem bestimmten Typ.

[15] Der Container verwaltet die Abhängigkeiten selbst und übergibt Objekte an andere Objekte entweder durch Aufruf eines geeigneten Konstruktors, durch Setzen eines Attributs (*setter injection*) oder durch Aufrufen einer geeigneten Factorymethode.

[16] MVC ist ein gängiges Designpattern zur Umsetzung von Applikationen mit grafischer Benutzeroberfläche (z.B. Web-Applikationen). Dabei werden die Business-Logik und die Daten einer Applikation (Model) von der Darstellung (View) getrennt; im Controller findet die Behandlung von Benutzereingaben sowie die Aktualisierung von Model und View statt.

[17] JSP ist eine Technologie, die hauptsächlich zum Erzeugen von HTML- oder XML-Dokumenten auf Webservern gedacht ist. Dabei kann Java-Code in Verbindung mit statischen Inhalten in einer JSP verwendet werden. Ein spezieller Compiler wandelt die JSP in Java-Quellcode um, der wiederum vom Java-Compiler zu Java-Bytecode kompiliert wird, der schließlich auf einem Applikationsserver ausgeführt werden kann.

[18] Eine kurze Erklärung zu Velocity ist in Abschnitt 4.6.1.1 zu finden.

[19] `http://opensymphony.com/quartz/`

Umfeld unterstützen (z.B. Datenzugriffskomponente für relationale Datenbanken, Transaktionsmanagement, aspektorientierte Programmierung, Authentisierung und Autorisierung). Die Erläuterung dieser Komponenten ginge jedoch weit über den Rahmen dieser Arbeit hinaus, zumal sie für die Implementation der Viewer-Komponente nicht notwendig sind. Jedoch sei an dieser Stelle noch angemerkt, dass durch die Verwendung von Spring die Basis gelegt wurde für eine spätere Erweiterung beispielsweise um eine Authentisierung und anschließende Autorisierung des aktuellen Benutzers um einer etwaigen Zugriffsbeschränkung für bestimmte Reports Rechnung zu tragen.

Weitere Informationen zum Spring Framework sind in der offiziellen Dokumentation unter [spr08] zu finden. Eine weitere sehr gute Hilfe bei der Implementation von Spring-basierten Applikationen ist [WB07]. Für einen kurzen Überblick siehe [Wik08d].

4.5.3 Implementation der Viewer-Komponente

Wie bereits mehrfach erwähnt soll die Viewer-Komponente mit Hilfe des Frameworks *Spring*, insbesondere dessen MVC-Framework, als Java Web-Applikation umgesetzt werden. Grundsätzlich muss dafür wie für jede Java Web-Applikation zunächst ein sogenannter *Deployment Descriptor* erstellt werden, eine *web.xml* genannte XML-Datei, die die Beschreibung der Applikation für den jeweiligen Applikationsserver enthält. Der wichtigste Bestandteil des *Deployment Descriptors* ist die Beschreibung der einzelnen Servlets[20] einer Applikation.

Bei der Verwendung des Spring MVC-Frameworks zur Implementation einer Java Web-Applikation ist wie bereits weiter oben erwähnt nur ein einziges Servlet nach außen hin sichtbar, der sogenannte *front controller* in Form der Klasse `DispatcherServlet`, die Teil des Spring MVC-Frameworks ist. Im Fall der Viewer-Komponente heißt dieses Servlet *genericreports.webapp*. Die eigentliche Konfiguration der Web-Applikation erfolgt wie bereits erwähnt im sogenannten *Application Context*. Dies ist ebenfalls eine XML-Datei, deren Name sich aus dem Namen des entsprechenden Servlets gefolgt von der Zeichenkette „-servlet.xml“ zusammensetzt. Durch diese Konvention leitet der *front controller* eingehende Anfragen an die richtige Spring Applikation weiter.

4.5.3.1 Controller-Architektur der Viewer-Komponente

Wie bereits durch die Abbildung 4.13 angedeutet werden für die Viewer Web-Applikation Frames benutzt um die verschiedenen Bereiche (Ti-

[20] Ganz allgemein gesehen ist ein Servlet ein Objekt in einem Java Applikationsserver, welches Anfragen entgegennimmt (üblicherweise über HTTP) und basierend auf dieser Anfrage eine Antwort generiert, die an den Aufrufer zurückgeschickt wird.

tel, Navigation, Report) voneinander abzugrenzen. Frames bieten in HTML die Möglichkeit den Anzeigebereich im Browser in verschiedene Bereiche zu unterteilen, in denen dann wiederum verschiedene HTML-Dokumente angezeigt werden können. Obwohl der Einsatz von Frames heutzutage eher kritisch gesehen wird[21] wurde hier diese Option gewählt um vor allem die Integration des BIRT Web-Viewers zu vereinfachen (siehe Abschnitt 4.5.4). Da wie erwähnt in jedem Frame ein anderes HTML-Dokument angezeigt wird werden zunächst zwei verschiedene HTML-Dokumente zur Anzeige benötigt:

title.html Wird im oberen Bereich angezeigt und enthält nur den Titel für die Viewer Web-Applikation.

navigation.html Zeigt im Bereich auf der linken Seite alle verfügbaren Reports an und ermöglicht die Auswahl der gewünschten Datumsspanne für einen Report.

Darüberhinaus werden noch zwei weitere Seiten zur Verfügung gestellt:

help.html Diese Seite gibt Hinweise zur Navigation und wird beim Start der Applikation im Frame rechts, der für die Anzeige der Reports bestimmt ist, angezeigt. Später ist diese Seite über einen Link auf der Navigationsseite zu erreichen.

admin.html Diese Seite enthält verschiedene Möglichkeiten Aktionen innerhalb der Viewer Web-Applikation zu Testzwecken aufzurufen, die im produktiven Betrieb zeitgesteuert ablaufen. Da diese Möglichkeiten nur während der Entwicklung von Bedeutung waren soll darauf nicht näher eingegangen werden.

Für jede dieser Seiten wird im *Application Context* (genericreports.webapp-servlet.xml) jeweils ein Controller angelegt, der Anfragen nach den jeweiligen Seite entgegennimmt, die benötigten Informationen bezieht und an den entsprechenden View übergibt, der dann daraus das HTML-Dokument erzeugt, welches an den Aufrufer zur Anzeige zurückgeschickt wird. Welcher Controller zur Bearbeitung einer Anfrage jeweils aufgerufen werden soll wird über das *handler mapping* festgelegt. Dazu wird im *applicatoin context* eine *Bean* vom Typ `SimpleUrlHandlerMapping` angelegt die entsprechend der URL in einem Request diesen an ein bestimmtes Controller-Objekt zur Behandlung weiterleitet. Dieser Ablauf ist auch in Abbildung 4.14 noch einmal bildlich dargestellt.

[21] Siehe dazu den Abschnitt *Criticism* in [Wik08a].

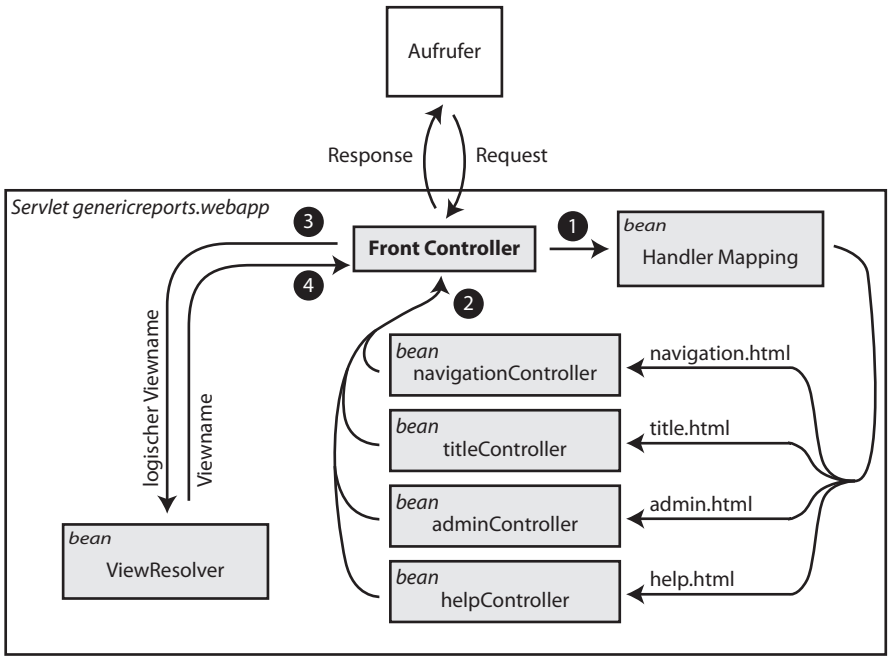

Abbildung 4.14: Anwendungsstruktur der Viewer-Komponente

4.5.3.2 Model, View und Controller für die Navigation

Wie bereits erwähnt soll in der Navigation die Auswahl aller zur Verfügung stehender Reports gruppiert nach den jeweiligen Applikationen möglich sein. Da diese Informationen bereits in den für den Report Design Generator erstellten XML-Konfigurationsdateien vorhanden sind sollten diese dazu benutzt werden um die entsprechenden Auswahlmöglichkeiten in der Navigation anzuzeigen. Zu diesem Zweck müssen die Konfigurationsdateien in einem bestimmten Verzeichnis auf dem Applikationsserver abgelegt werden (siehe dazu auch Abschnitt 4.5.3.3. Um bequem auf die verschiedenen Konfigurationsdateien zugreifen zu können wurde die Klasse `ConfigRepository` implementiert, die mit Hilfe der zu diesem Zweck entwickelten Bibliothek (siehe Abschnitt 4.1) die Konfigurationsdateien lädt und für den Zugriff durch andere Objekte bereithält. Damit ein Objekt dieser Klasse innerhalb der Applikation zur Verfügung steht wird im *Application Context* eine entsprechende Bean-Konfiguration erstellt, die in Codebeispiel 4.6 abgedruckt ist. Dort ist auch zu sehen, dass für die *Bean* mit der ID *configRepository* zusätzlich eine Eigenschaft (Unterelement `<property>`) mit dem Namen *configFileDir* definiert wird, die den Pfad zu dem Verzeichnis mit den Konfigurationsdateien enthält. Um als *Bean* im *Application Context* instanziiert werden zu können muss die Klasse

```
1 <bean id="configRepository" class="genericreports.webapp.
     ConfigRepository">
2   <property name="configFileDir" value="${configrep.basedir
       }" />
3 </bean>
4
5 <bean id="navigationController" class="genericreports.
     webapp.controller.NavigationController">
6   <property name="configRepository" ref="configRepository"
       />
7 </bean>
```

Codebeispiel 4.6: Konfiguration der Beans ConfigRepository und NavigationController

`ConfigRepository` folgende Voraussetzungen erfüllen:

- Sie benötigt einen parameterlosen Konstruktor
- Es muss eine Methode vorhanden sein, die es erlaubt die Eigenschaft *configFileDir* nach dem instanziieren der Klasse zu setzen (`setConfigFileDir(String)`)

Beim Start der Viewer Web-Applikation wird die Klasse `ConfigRepository` automatisch instanziiert und steht unter der ID *configRepository* innerhalb des *Application Contexts* zur Verfügung.

Im unteren Teil des Codebeispiels 4.6 ist die Konfiguration der NavigationController-Bean zu sehen. Auch diese Bean hat eine Eigenschaft mit dem Namen *configRepository*. Der Wert dieser Eigenschaft ist die Referenz auf die zuvor konfigurierte ConfigRepository-Bean. Nach dem instanziieren der Klasse `NavigationController` mit Hilfe des parameterlosen Konstruktors ruft der *Application Context* also eine entsprechende Methode (`setConfigRepository(ConfigRepository)`) auf und übergibt eine Referenz auf das zuvor kreierte `ConfigRepository`-Objekt (*setter injection*). Auf diese Weise hat die Klasse `NavigationController` Zugriff auf die vorhandenen Konfigurationsdateien ohne selbst die Abhängigkeit zum `ConfigRepository`-Objekt auflösen zu müssen.

Die Klasse `NavigationController` selbst ist relativ einfach. Neben der bereits erwähnten Methode zum setzen der Referenz auf das `ConfigRepository`-Objekt hat sie noch eine zweite Methode, `handleRequest()`, die vom implementierten Interface `Controller` vorgeschrieben wird. Diese Methode gibt ein Objekt vom Typ `ModelAndView` zurück, welches, wie der Name vermuten lässt, Modeldaten (in diesem Fall die Liste der verfügbaren Konfigurationen) auf der einen und den logischen Namen des Views auf der anderen Seite enthält.

Für die Implementierung der Views wurden sogenannte JSP (***J**ava **S**erver **P**ages*) benutzt. Wie bereits erwähnt erlauben diese die Kombination von statischen mit dynamisch generierten Inhalten. Zu diesem Zweck ist es möglich Java Code (sogenannte *Scriptlets*) in gesondert markierten Bereichen zu notieren, der die dynamischen Inhalte erzeugt. Um die Übersicht in der JSP zu erhöhen können diese Scriptlets in sogenannte *Tags* ausgelagert werden. Der Java Code eines Scriptlets wird dabei in eine Java Klasse verlagert, die ein bestimmtes Interface implementieren muss (`Tag` oder `BodyTag`). Zusätlich können diese Tags in einer *Tag Library* zusammengefasst werden. Diese wird in einer XML-Datei beschrieben, dem sogenannten *Taglibrary Descriptor*, und im Deployment Descriptor der Web-Applikation eingebunden. Die Syntax der Notation der Tags in der JSP folgt der von HTML. In der JSTL (***J**SP **S**tandard **T**ag **L**ibrary*) sind bereits einige nützliche Tags zum Beispiel für die bedingte Ausführung von Code in einer JSP oder für verschiedene Schleifenkonstrukte vorhanden.

Wie bereits erwähnt gibt die Methode `handleRequest()` des Controllers ein Objekt zurück, welches den logischen Namen des Views enthält, der an den Aufrufer geschickt werden soll. Um zum logischen Namen eines Views nun die richtige JSP zu finden muss im *Application Context* eine sogenannte *View Resolver*-Bean definiert werden. Der View Resolver der Viewer Web-Applikation ist vom Typ `InternalResourceViewResolver`, das heißt die Auflösung der logischen View-Namen erfolgt auf interne Resourcen der Web-Applikation. Wie im Codebeispiel 4.7 zu sehen werden für den View Resolver drei Eigenschaften definiert:

1. In der Eigenschaft *viewClass* ist definiert, dass die Views vom Typ `JstlView` sind, also JSPs mit Unterstützung für die JSTL.

2. Die beiden Eigenschaften *prefix* und *suffix* werden dem logischen View-Namen voran- bzw. nachgestellt um den Pfad zur entsprechenden JSP zu konstruieren. Der Pfad zum View *navigation* wäre also entsprechend der Konfiguration „/WEB-INF/jsp/navigation.jsp“.

Auf die Implementation des Views für die Navigation selbst soll hier nur kurz eingegangen werden. Abbildung 4.15 zeigt den fertig gerenderten View im Browser. Wie bereits erwähnt wird die im `ConfigRepository` geladene Liste an Konfigurationsobjekten an den View übergeben. In der JSP wird über diese Liste iteriert um alle verfügbaren Reports nach der jeweiligen Applikation gruppiert aufzulisten. Über das Symbol vor dem Namen der jeweiligen Applikation kann die Liste der für diese Applikation verfügbaren Reports expandiert und wieder verborgen werden. Die Auswahl eines Reports erfolgt über einen Klick auf den jeweiligen Link in der Liste, um den Summary Report einer Applikation auszuwählen muss auf den Namen der jeweiligen Applikation geklickt werden. Das Auswählen

```
1 <bean id="internalResourceViewResolver" class="org.
     springframework.web.servlet.view.
     InternalResourceViewResolver">
2   <property name="viewClass" value="org.springframework.web
       .servlet.view.JstlView" />
3   <property name="prefix" value="/WEB-INF/jsp/" />
4   <property name="suffix" value=".jsp" />
5 </bean>
```

Codebeispiel 4.7: Konfiguration des ViewResolvers

der Reports sowie das expandieren und verbergen der Reports einer Applikation erfolgt komplett auf der Clientseite mit Hilfe von Javascript. Zusätzlich befindet sich im oberen Teil der Navigation das Formular für die Eingabe des Start- und Enddatums für den zu generierenden Report. Auf die genaue Verwendung dieser beiden Formularfelder wird in Abschnitt 4.5.4 näher eingegangen.

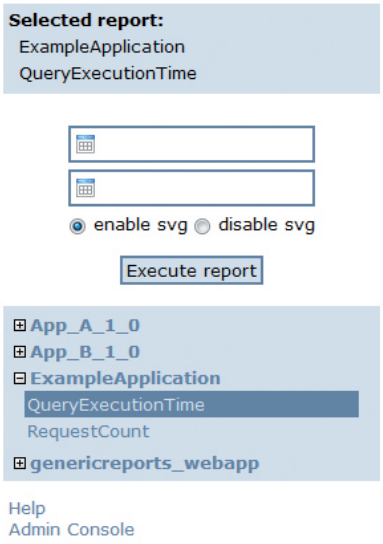

Abbildung 4.15: Die Navigation der Viewer Web-Applikation

4.5.3.3 Konfiguration der Viewer Web-Applikation

Einige Werte, die im Application Context konfiguriert werden variieren zwischen verschiedenen Installationen der Viewer Web-Applikation. Dazu gehört zum Beispiel das Verzeichnis in dem die Konfigurationsdateien abgelegt sind die die Informationen für die Navigation enthalten, im

Abschnitt 4.5.5 werden noch weitere Beispiele erläutert. Um diese Einstellungen zentral in einer Datei verwalten zu können stellt Spring die Klasse `PropertyPlaceholderConfigurer` zur Verfügung, die als Bean im Application Context angelegt werden kann. Über die Eigenschaft *locations* dieser Bean kann man den Pfad zu einer Datei im Java Properties Format konfigurieren, in der diese ausgelagerten Einstellungen gespeichert sind. Anschließend kann man die Werte aus dieser Datei über eine spezielle Notation im Application Context einfügen. Ein Beispiel hierfür findet sich im Codebeispiel 4.6 bei der Konfiguration der Eigenschaft *configFileDir*. Bei der Angabe des Wertes dieser Eigenschaft wird der Wert, der unter dem Schlüssel *configrep.basedir* angegeben ist dynamisch eingefügt. Diese Technik des Auslagerns von Konfigurationswerten kann an jeder beliebigen Stelle im Application Context angewendet werden.

4.5.4 Integration des BIRT WebViewers

Wie bereits erwähnt soll für die Generierung und die Darstellung der im Rahmen des BIRT Projektes entwickelte Web-Viewer zum Einsatz kommen. Der Web-Viewer steht als Java Web-Applikation in einer WAR-Datei[22] zur Verfügung, die auf vielen gängigen Java Application Servern installiert werden kann.

Vor der Installation müssen allerdings einige Anpassungen an der Konfiguration in der Datei *web.xml* des BIRT Web-Viewers gemacht werden. Dort befinden sich einige Parameter, in denen neben der standardmäßig eingestellten Sprache auch einige Pfade in denen die BIRT Web-Viewer Applikation zum Beispiel die angeforderten Report Designs sucht oder temporäre Dateien bei der Reportgenerierung ablegt konfiguriert werden. Um diese Einstellungen zu ändern muss die Datei *web.xml* aus der WAR-Datei extrahiert und nach den Änderungen wieder in das Archiv zurückkopiert werden. Zusätlich müssen noch die JDBC-Treiber für die Datenbanken auf die bei der Reportgenerierung zugegriffen werden muss in das entsprechende Verzeichnis im Web Archiv[23] kopiert werden. Anschließend kann der BIRT Web-Viewer wie jede andere Web-Applikation auch auf dem Application Server installiert werden.

Einmal installiert stellt die BIRT Web-Viewer Applikation über mehrere URLs verschiedene Servlets zur Verfügung, über die auf verschiedene Möglichkeiten zur Reportgenerierung zugegriffen werden kann. Über verschiedene Parameter der URL können bestimmte Informationen die zur Generierung des Reports wichtig sind übergeben werden. Allen voran na-

[22] Eine WAR-Datei (Web Archive) ist ein Java Archiv, in dem alle nötigen Dateien für eine Web-Applikation zusammengefasst werden können um die Installation auf dem Application Server zu vereinfachen.

[23] [BIRT WAR]/WEB-INF/platform/plugins/
org.eclipse.birt.report.data.oda.jdbc_[version]/drivers

türlich der Dateiname des Report Designs aber auch die Werte eventuell vorhandener Report Parameter. Für die Integration des BIRT Web-Viewers in die Viewer-Komponente wurde das unter der URL */frameset* erreichbare Servlet verwendet. Dieses generiert nicht nur den entsprechenden Report im HTML-Format sondern stellt auch umfassende Möglichkeiten zur Verfügung im Report zu navigieren oder aus dem generierten Report beispielsweise ein PDF zu erstellen. Um die entsprechenden Paramter an die URL mit anzuhängen wird in der Navigation ein HTML Formular eingesetzt, welches die eingegebenen Daten (Start- bzw. Enddatum für die Reportgenerierung, Name des Report Designs) per *HTTP GET*[24] an die BIRT Web-Viewer Applikation übergibt. Der Name des gewählten Report Designs wird dabei per Javascript gesetzt, wenn über das Auswahlmenü im unteren Teil der Navigation ein Report ausgewählt wird. Die Antwort der BIRT Web-Viewer Applikation, sprich der generierte Report, wird im rechten Teil des Browserfensters angezeigt, die Navigation steht weiterhin zur Auswahl eines anderen Datums oder Reports zur Verfügung.

Beim Absenden des Formulars werden, noch bevor der HTTP Request gesendet wird, die vom Benutzer eingegebenen Daten per Javascript auf Korrektheit überprüft (beides gültige Datumsangaben, Startdatum liegt vor Enddatum). Außerdem wird überprüft, ob ein Report gewählt wurde. Schlägt eine der Überprüfungen fehl wird das Senden der Formulardaten abgebrochen und eine entsprechende Fehlermeldung angezeigt.

Weitere Informationen über die Verwendung der BIRT Web-Viewer Applikation ist in [WFB+06] und auch im Internet in [ecl08] zu finden.

4.5.5 Integration der Collector-Komponente

Wie bereits in Abschnitt 4.4.6 angesprochen sollte die Collector-Komponente ebenfalls als Serverprogramm entwickelt werden welches periodisch ausgeführt werden kann. Da das Framework Spring wie erwähnt die Möglichkeit bietet mit Hilfe des *Scheduler Frameworks* vorher definierte Aufgaben periodisch auszuführen wurde die Collector-Komponente mit in die Web-Viewer Applikation integriert.

Zunächst wurde die Klasse `LogFileCollector` im Application Context als Bean definiert und alle nötigen Abhängigkeiten konfiguriert. Wie in Codebeispiel 4.8 zu sehen gehört dazu der Parser für die Log-Dateien, der ebenfalls in einer eigenen Bean definiert ist. Neben dem Verzeichnis in dem die zu lesenden Log-Dateien liegen bzw. in welchem sie nach dem Lesen archiviert werden wird auch eine Referenz auf die bereits erläuterte Bean *ConfigRepository* konfiguriert. Diese wird benötigt, da aus den Namen der in den Konfigurationsdateien angegebenen Applikationsnamen

[24] Bei HTTP GET werden die Formulardaten als URL-Parameter übertragen im Gegensatz zu HTTP POST bei dem diese im Body des HTTP Requests übertragen werden.

```
1 <bean id="logFileCollector" class="genericreports.webapp.
     collector.LogFileCollector">
2   <property name="logFileParser" ref="logFileParser" />
3   <property name="configRepository" ref="configRepository"
       />
4   <property name="logFileDir" value="${collector.logfiledir
       }" />
5   <property name="archiveDir" value="${collector.archivedir
       }" />
6 </bean>
7
8 <bean id="logFileParser" class="genericreports.webapp.
     collector.LogFileParser" />
```

Codebeispiel 4.8: Konfiguration der Collector-Bean

und LogRecordnamen die Dateinamen der zu lesenden Log-Dateien konstruiert werden.

Für die zeitgesteuerte Ausführung der Collector-Komponenten wird die bereits erwähnte Bibliothek *Quartz* benutzt, da sie gegenüber der Java-Klasse `Timer`, die nur die Angabe eines Intervalls zur zeitgesteuerten Ausführung erlaubt, die Möglichkeit bietet die genaue Uhrzeit festzulegen zu der bestimmte Aufgaben ausgeführt werden sollen. Für jede zeitgesteuert auszuführende Aufgabe müssen zwei Beans definiert werden:

JobBean Diese Bean enthält Informationen über die auszuführende Aufgabe insbesondere die Klasse, die diese implementiert und alle Angaben die zur Instanziierung dieser Klasse nötig sind.

JobTriggerBean Legt fest wie oft bzw. wann eine Aufgabe ausgeführt werden soll.

In Codebeispiel 4.9 ist die Definition dieser beiden Beans für die Collector-Komponente zu sehen. Bei der Definition der Bean *collectorJobTrigger* fällt wieder auf, dass der Wert für die Eigenschaft *cronExpression* über die in Abschnitt 4.5.3.3 erwähnte Properties-Datei konfiguriert wird. Der Wert dieser Eigenschaft gibt an, zu welchen Zeiten eine konfigurierte Aufgabe ausgeführt wird und gemäß Abbildung 4.16 aus 7 Teilen besteht. Jeder dieser Teile kann dabei durch einen festen Wert (z.B. 15), ein Intervall (z.B. 15-17), eine Liste von Werten (z.B. 15,16,17) oder einen Platzhalter (*) definiert werden. Dabei kann entweder der Wochentag oder der Tag im Monat angegeben werden, die jeweils andere Angabe ist durch ein Fragezeichen zu markieren. Um die Ausführung des Collectors also zum Beispiel für jeden Tag um 4 Uhr zu terminieren ist der Wert *0 0 4 * * ?*[25] für die Eigenschaft *cronExpression* in der *collectorJobTrigger* Bean nötig.

[25] Ausführung zur nullten Sekunde und nullten Minute der vierten Stunde jedes Tages in

```
1  <bean id="collectorJob" class="org.springframework.
       scheduling.quartz.JobDetailBean">
2    <property name="jobClass" value="genericreports.webapp.
       jobs.LogFileCollectorJob" />
3    <property name="jobDataAsMap">
4      <map>
5        <entry key="logFileCollector" value-ref="
           logFileCollector" />
6      </map>
7    </property>
8  </bean>
9
10 <bean id="collectorJobTrigger" class="org.springframework.
       scheduling.quartz.CronTriggerBean">
11   <property name="jobDetail" ref="collectorJob" />
12   <property name="cronExpression" value="${collector.
       cronexpression}" />
13 </bean>
```

Codebeispiel 4.9: Konfiguration des Jobs für die COllector-Komponenten

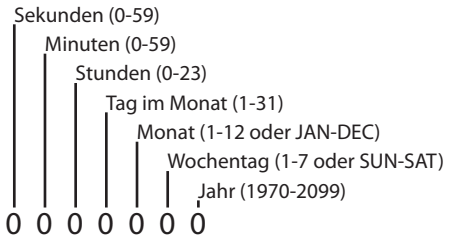

Abbildung 4.16: Aufbau des Wertes für die zeitgesteuerte Ausführung

jedem Monat egal an welchem Wochentag; die Angabe des Jahres kann wenn nicht benötigt entfallen.

4.6 Zusätzliche Erweiterungen

Um den Einsatz des entwickelten Frameworks weiter zu unterstützen wurden zusätzlich zu den beschriebenen Kernkomponenten weitere Werkzeuge entwickelt. Da wie in Abschnitt 4.2.2 bereits erläutert für die Implementation des Report Design Generators ein Eclipse Plugin entwickelt wurde lag es nahe auch alle weiteren, die Arbeit mit dem Framework unterstützenden Werkzeuge als Eclipse Erweiterungen zu implementieren.

In den folgenden Abschnitten wird nun kurz auf die Entwicklung der einzelnen Werkzeuge eingegangen. Die Entwicklung von Eclipse Plugins im Allgemeinen wurde bereits in Abschnitt 4.2.3 näher erläutert weshalb in den folgenden Abschnitten nur noch besondere Konzepte näher erklärt werden auf die bisher nicht eingegangen wurde.

4.6.1 Assistent zum Erstellen der Konfiguration

Um die Erstellung der zentralen Konfiguration für das Framework zu vereinfachen wurde ein Assistent entwickelt, der es ermöglicht über eine grafische Benutzeroberfläche eine neue Konfigurationsdatei zu erstellen, in der bereits einige Parameter festgelegt sind. Alle noch fehlenden Parameter können dann vom Entwickler noch nachgetragen werden.

Für die Entwicklung von Assistenten für neue Resourcen stellt Eclipse den Erweiterungspunkt `org.eclipse.ui.newWizards` zur Verfügung. Dieser Erweiterungspunkt erlaubt die Entwicklung von Assistenten, die die Erstellung von neuen Resourcen (Dateien, Ordner oder Projekte) unterstützen. Die so definierten Assistenten können entweder standardmäßig über den Dialog *New* (zu Erreichen über die Menüleiste von Eclipse: *File* → *New* → *Other...*) oder direkt über eine *Action* aufgerufen werden. Ein neuer Assistent kann entweder in eine bestehende oder eine eigene Kategorie eingeordnet werden. Neue Kategorien können ebenfalls über den gleichen Erweiterungspunkt definiert werden.

Die Implementation der Funktionalität eines Assistenten erfolgt in einer Klasse, die das Interface `INewWizard` implementiert. Um nicht für jeden Assistenten die rund 20 Methoden die dieses Interface vorschreibt implementieren zu müssen steht die abstrakte Klasse `Wizard` zur Verfügung, die bis auf drei alle Methoden des besagten Interfaces implementiert. Die Methode `init()` dient der Initialisierung des Assistenten, `addPages()` kann überschrieben werden um dem Assistenten neue Seiten hinzuzufügen und `performFinish()` schließlich wird ausgeführt, wenn der Benutzer alle benötigten Informationen in den Assistenten eingegeben hat und die Schaltfläche *Finish* betätigt hat.

Ein Assistent kann eine oder mehrere Abschnitte (Seiten) haben zwischen denen der Benutzer während der Eingabe der benötigten Informationen hin- und herwechseln kann. Die Implementation einer solchen Seite

erfolg in einer Klasse, die von der abstrakten Klasse `WizardPage` erbt. In dieser Klasse muss die Methode `createControl()` implementiert werden die die Benutzeroberflächenelemente der neuen Seite zurückgibt. Die Klasse `WizardPage` stellt zusätzlich einige Methoden zur Verfügung, die zum Beispiel das einheitliche Anzeigen von Fehlermeldungen zu fehlenden oder falschen Angaben ermöglichen.

Der Assistent zum Erzeugen einer neuen Konfigurationsdatei sollte folgende Konfigurationsparameter vom Benutzer abfragen und anschließend eine neue Konfigurationsdatei erstellen, die eine neue Konfiguration mit den im Assistenten eingegebenen Werten enthält:

- Name der Applikation
- Verzeichnis der Log-Dateien
- Verbindungsparameter für die Datenbank

Zusätzlich musste es möglich sein auszuwählen, in welchem Projekt die neue Konfigurationsdatei erstellt werden soll.

In der Klasse `NewConfigFileWizardPage`, die wie oben beschrieben von der abstrakten Klasse `WizardPage` erbt ist die grafische Benutzeroberfläche für den Assistenten implementiert. Da die Menge an Information, die der Benutzer eingeben muss recht überschaubar ist wurde nur eine einzige Seite für den Assistenten implementiert über die alle benötigten Informationen eingegeben werden können. Die Konstruktion der Benutzeroberfläche erfolgt wie bereits erähnt in der Methode `createControl()`. Zusätzlich besitzt die Klasse eine Methode um die eingegebenen Informationen bei der Generierung der neuen Konfigurationsdatei abzufragen. Abbildung 4.17 zeigt wie sich der Assistent dem Benutzer in Eclipse präsentiert.

Die Implementation der Assistentenfunktionalität ist in der Klasse `NewConfigFileWizard` zu finden, die das Interface `INewWizard` implementiert. Wie im vorigen Abschnitt beschrieben muss diese Klasse drei Methoden implementieren, von denen nur die `performFinish()` Methode die eigentliche Funktionalität enthält. Diese überprüft zunächst ob eine Datei mit dem Namen der neuen Konfigurationsdatei (dieser setzt sich aus dem Namen der Applikation wie im Assistenten eingegeben und der Endung *.xml* zusammen) bereits vorhanden ist und fragt gegebenenfalls nach, ob die Datei überschrieben werden soll. Anschließend wird die Generierung der neuen Konfigurationsdatei gestartet.

4.6.1.1 Generierung mit Velocity

Zur Generierung der neuen Konfigurationsdatei kommt *Velocity*[26] von Apache zum Einsatz, eine OpenSource Bibliothek die es ermöglicht basie-

[26] Nähere Informationen zu Velocity unter `http://velocity.apache.org/`

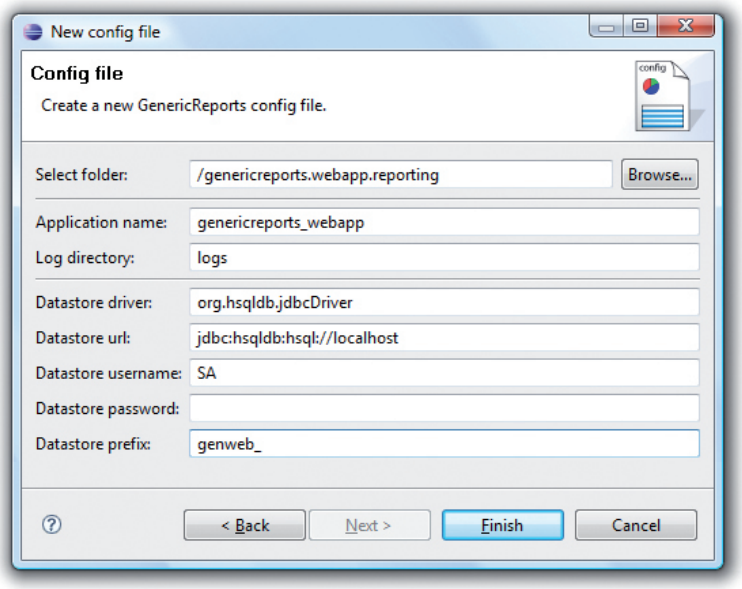

Abbildung 4.17: Assistent zum Erstellen einer neuen Konfiguration

rend auf einem Set von Werten (*context*) und einer Vorlage (*template*), die gewisse Platzhalter enthält die durch die Werte aus dem *context* ersetzt werden eine Ausgabe zu erzeugen. In Abbildung 4.18 ist dieser Vorgang dargestellt. Zur Formulierung der *templates* steht in Velocity die *Velocity Template Language* (*VTL*) zur Verfügung. Neben dem einfachen Einfügen von im *context* vorhandenen Werten sind mit der *VTL* auch die bedingte Einbindung von im *template* vorhandenem Text in die Ausgabe oder die Ausführung von Schleifen während der Templateverarbeitung möglich.

Für die Generierung der neuen Konfigurationsdatei wurde ein *template* erstellt, welches bis auf die Definition der *Logrecords* alle nötigen wie in Abschnitt 4.1.2 beschriebenen Elemente enthält. An den entsprechenden Stellen sind Platzhalter definiert, die dann bei der Anwendung des *templates* mit den entsprechenden Werten aus dem context gefüllt werden. Da das Element `<prefix>`, ein Unterelement des Elements `<datastore>` optional ist wird es im template auch nur eingefügt, wenn der Wert dafür auch im Assistenten angegeben wurde. Um dieses bedingte Einfügen von Text während der Anwendung des templates zu erreichen wurde hier auf die `#if` Anweisung der VTL zurückgegriffen. Das Template für die Generierung einer neuen Konfigurationsdatei ist im Anhang auf Seite 115 zu finden.

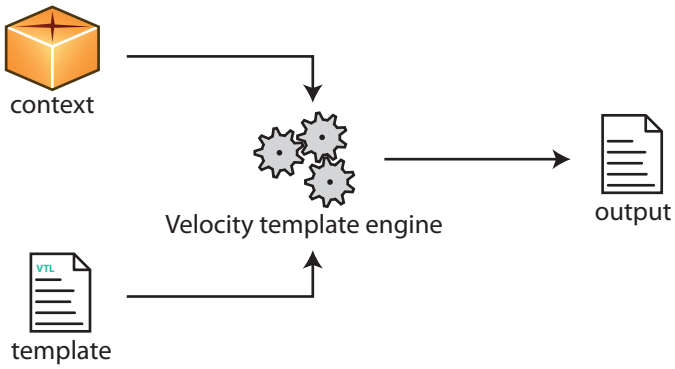

Abbildung 4.18: Templateverarbeitung mit Apache Velocity

4.6.2 log4j Konfigurationsgenerator

Wie in Abschnitt 4.3.5 beschrieben muss für die korrekte Funktion der Writer-Komponente eine log4j-Konfigurationsdatei erstellt werden. Da der Aufbau dieser Konfiguration jedoch komplett auf der Konfiguration des Frameworks (siehe Abschnitt 4.1) basiert kann diese dazu benutzt werden um die log4j-Konfigurationsdatei zu generieren.

Um auch diesen Generator einfach in die Entwicklungsumgebung integrieren zu können wurde er auf die gleiche Art und Weise in Eclipse eingebunden wie der Generator für die Report Designs: Der Werkzeugleiste des Eclipse XML-Editors wurde eine weitere Action hinzugefügt, über die der Generator aufgerufen werden kann.

Der Generator selbst benutzt wie schon der Assistent zum Erstellen einer neuen Konfiguration die Bibliothek Velocity von Apache (siehe Abschnitt 4.6.1.1). Da wie in Abschnitt 4.3.3.6 beschrieben die log4j-Konfigurationsdatei wahlweise im .properties- oder im XML-Format erstellt werden kann bietet auch der entwickelte Generator die Möglichkeit die Konfiguration für die Writer-Komponente wahlweise in dem einen oder in dem anderen Format zu generieren. Zusätzlich kann der Benutzer das Intervall auswählen in dem die Log-Dateien von den verwendeten *DailyRollingFileAppendern* rotiert werden (stündlich, täglich, wöchentlich, etc.). Um diese Optionen festzulegen wird dem Benutzer vor dem Generieren ein entsprechender Dialog präsentiert (siehe Abbildung 4.19). Der Generator erzeugt die vollständige Konfiguration, eine manuelle Anpassung ist nicht nötig, jedoch durchaus möglich, für den Fall, dass zum Beispiel zum Zweck des normalen Loggings einer Applikation entsprechende Logger und Appender definiert werden müssen.

Die beiden Templates (für die log4j-Konfigurationsdatei im .properties- und im XML-Format) sind im Anhang auf den Seiten 116 und 117 zu fin-

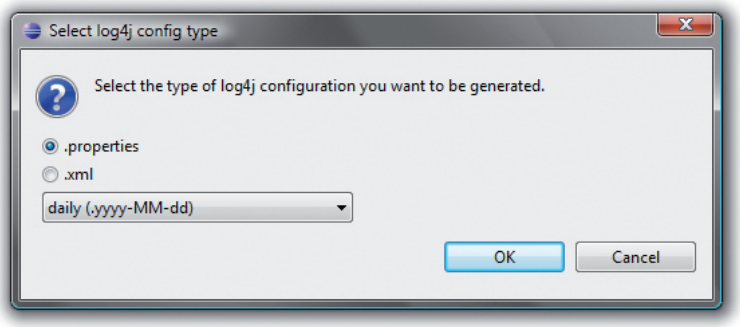

Abbildung 4.19: Auswahl der Optionen für die Generierung der log4j Konfiguration

den. Im Velocity-Kontext befinden sich jeweils das über den im vorigen Abschnitt erwähnten Dialog (Abbildung 4.19) ausgewählte Intervall für die Appender sowie das Verzeichnis für die Log-Dateien und eine Liste aller LogRecords, die in der jeweiligen Konfigurationsdatei festgelegt wurden. Im Template werden diese Werte dann an den entsprechenden Stellen eingefügt. Um die entsprechenden Konfigurationsblöcke für alle im Kontext enthaltenen LogRecords zu erzeugen wird das `#foreach` Schleifenkonstrukt verwendet (siehe z.B. in Zeile 3 von Codebeispiel B.2 auf Seite 116).

4.6.3 Import der log4j-Erweiterung

Wie in Abschnitt 4.3.8 beschrieben werden die beiden Klassen, die zusätzlich zur Bibliothek log4j benötigt werden in einem Java Archiv zusammengefasst um die Einbindung in die jeweilige Applikation zu vereinfachen. Da die Bibliothek bzw. die darin enthaltenen Klassen zur optionalen Entwicklung eigener Renderer (siehe Abschnitt 4.3.6, in jedem Fall aber zur Laufzeit der Applikation benötigt werden wurde zusätzlich eine einfache Möglichkeit entwickelt diese zu einem Projekt in Eclipse hinzuzufügen. Aus diesem Grund ist die Bibliothek in dem Plugin enthalten, welches die hier beschriebenen Erweiterungen enthält. Über eine Action, die in der Eclipse Werkzeugleiste auftaucht kann sie in das aktuelle Projekt importiert werden. Des weiteren kann die Bibliothek optional auch gleich zum *CLASSPATH* des jeweiligen Projekts hinzugefügt werden (wichtig zur Entwicklung eigener Renderer).

In Abbildung 4.20 ist der Dialog abgebildet, in dem der Benutzer einen Zielpfad in einem der aktuellen Projekte für die Bibliothek auswählen kann. In einem weiteren Dialog kann der Benutzer auswählen, ob die Bibliothek zusätzlich zum *CLASSPATH* des Projekts hinzugefügt werden soll.

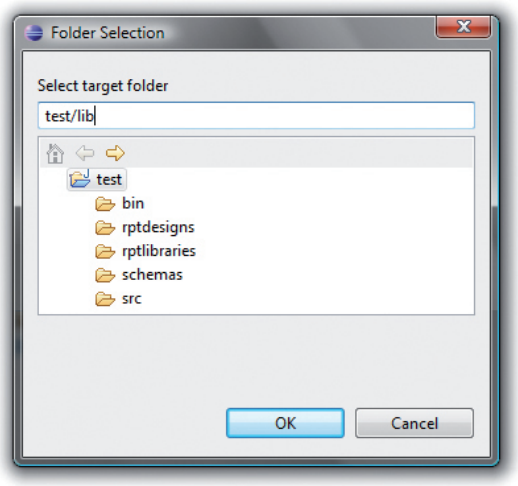

Abbildung 4.20: Auswahl des Zielverzeichnisses für die log4j Bibliothek

4.6.4 Schemagenerator für die Datenbank

Damit die durch die Writer-Komponente geloggten Daten von der Collector-Komponente in die Datenbank eingefügt werden können müssen natürlich zunächst die entsprechenden Tabellen in der Datenbank angelegt werden. Der Aufbau einer Tabelle (Schema) ergibt sich komplett aus der Konfiguration des entsprechenden LogRecords:

1. Der Name der Tabelle setzt sich aus dem optional in der Datenbankkonfiguration angegebenen Präfix und dem Namen des LogRecords zusammen.

2. Die Anzahl, Namen und Typen der Spalten ergeben sich aus den für diesen LogRecord definierten LogEntries.

In Abbildung 4.21 ist im oberen Teil die Konfiguration eines LogRecords, im unteren Teil der Aufbau der entsprechenden Datenbanktabelle zu sehen. Die zusätzliche Spalte *time*, die im LogRecord nicht konfiguriert ist wird dabei automatisch hinzugefügt wie bereits in Abschnitt 4.1.2 über die Syntax der Konfiguration erwähnt.

Aufgrund des extrem einfachen Templates wäre ein weiterer Weg bei der Generierung der Schema-Datei gewesen ihren Inhalt komplett im Programmcode zu erzeugen und dann entsprechend in eine Datei zu schreiben. Trotzdem wurde zur Generierung der Datei mit dem Schema wieder Velocity gewählt, um Änderungen an der Ausgabe einfacher bewerkstelligen zu können. Außerdem erhöht der Einsatz von Velocity in jedem Fall die Übersicht über den Programmcode. Das Template für die Schema-Dateien ist auf Seite 118 zu finden.

```
<logrecord name="QueryExecutionTime">
  <logentry name="exectime" type="double" />
  <logentry name="logincount" type="integer" />
</logrecord>
```

```
CREATE TABLE exapp_QueryExecutionTime
(
  time datetime,
  exectime double,
  logincount integer
)
```

Abbildung 4.21: Konfiguration eines LogRecords und Aufbau der zugehörigen Tabelle

4.6.5 Aufruf der Werkzeuge

In Abbildung 4.22 sind die verschiedenen Schaltflächen zum Aufrufen der im Rahmen dieser Diplomarbeit entwickelten Generatoren und Werkzeuge zu sehen. Von links nach rechts finden sich die Werkzeuge *Import der log4j Erweiterung* für die Integration der Writer-Komponente, *Schemagenerator für die Datenbank*, *log4j Konfigurationsgenerator* und *Report Design Generator*. Eclipsetypisch kann dabei die Position der einzelnen Schaltflächen variieren.

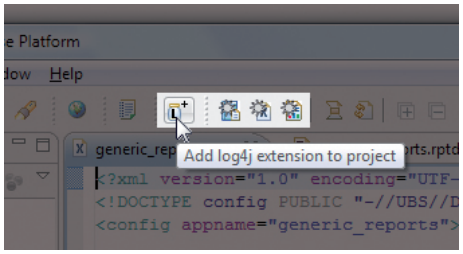

Abbildung 4.22: Schaltflächen zum Aufrufen der verschiedenen Werkzeuge in Eclipse

Kapitel 5

Beispielapplikation

In diesem Kapitel soll die Verwendung des entwickelten Reporting Frameworks anhand einer Beispielapplikation genauer erklärt werden. Da keine „richtige“ also in Produktion befindliche Applikation zur Demonstration zur Verfügung stand soll die Viewer-Komponente um entsprechende Reportingfunktionalitäten erweitert werden.

5.1 Festlegen der zu loggenden Parameter

Um das Reporting Framework effektiv einsetzen zu können muss man sich zunächst genaue Gedanken darüber machen welche Informationen interessant sind und welche Werte dafür aufgezeichnet werden sollen.

Um den Überblick über die nächtlich laufende Collector-Komponente zu behalten soll das Reporting Framework eingesetzt werden. Dabei ist zum Einen die Anzahl der gelesenen Log-Dateien und zum Anderen die Zeit interessant, die der Collector gebraucht hat um diese Dateien zu lesen und die gelesenen Daten in die Datenbank einzutragen. Zur leichteren Identifikation sollen diese beiden Parameter benannt und mit einem Typ versehen werden. Der erste Parameter bekommt in diesem Beispiel den Bezeichner *logfilecount* und den Typ *integer*, da es sich bei dem aufzuzeichnenden Parameter um eine Ganzzahl handelt. Der Bezeichner für den zweiten Parameter ist *collecttime*. Da die Zeit die der Collector für die Erledigung seiner Aufgabe benötigt in Millisekunden gemessen werden soll ist auch dieser Parameter vom Typ *integer*.

5.2 Erstellen der Konfiguration

Im nächsten Schritt muss eine Konfiguration gemäß den Vorüberlegungen zu Anzahl und Typ der zu loggenden Parameter erstellt werden. Um diesen Vorgang zu vereinfachen und zu beschleunigen kommt der in Ab-

```
1  <logrecord name="collectLogFiles">
2    <logentry name="logFileCount" type="integer" />
3    <logentry name="collectTime" type="integer" />
4
5    <report>
6      <chart name="collectLogFiles">
7        <series column="logFileCount" caption="number of log
             files" />
8        <series column="collectTime" caption="time to collect
             " />
9      </chart>
10   </report>
11 </logrecord>
```

Codebeispiel 5.1: LogRecord Konfiguration

schnitt 4.6.1 erläuterte Assistent zum Erstellen der Konfigurationsdatei zum Einsatz (siehe Abbildung 4.17 auf Seite 84).

Um die in Abschnitt 4.6 beschriebenen Werkzeuge und den in Abschnitt 4.2 entwickelten Report Design Generator nutzen zu können muss zunächst einmal das BIRT Plugin in der verwendeten Eclipse Entwicklungsumgebung installiert sein. Zusätzlich ist natürlich das im Rahmen dieser Arbeit entwickelte Plugin mit den oben genannten Erweiterungen ebenfalls zu installieren.

Der Dialog mit den Assistenten für neue Dateien ist in Eclipse über das Menü *File → New → Other...* zu erreichen. In der Kategorie *GenericReports* ist dort der Assistent für eine neue Konfigurationsdatei mit dem Namen *Config file* zu finden. Der Assistent ermöglicht wie bereits beschrieben die Eingabe der allgemeinen Daten zu einer Applikation wie deren Name, den Namen des Verzeichnisses in welches die Log-Dateien von der Writer-Komponente geschrieben werden sollen und die Parameter für die verwendete Datenbank (JDBC-Treibername, URL, Benutzername, Passwort und eventuell ein Präfix für die Namen der Tabellen). In der vom Assistenten generierten Konfigurationsdatei müssen nun nur noch die entsprechenden LogRecords gemäß der Überlegungen im ersten Schritt formuliert werden. Codebeispiel 5.1 zeigt die entsprechende Konfiguration für dieses Beispiel. Zu beachten ist, dass obwohl kein `<logentry>` für die Zeit konfiguriert wurde diese trotzdem standardmäßig mit gespeichert wird um die grafische Darstellung in einem Diagramm über der Zeit zu ermöglichen. Nach der Definition der einzelnen Werte dieses LogRecords folgt noch die Definition eines Diagramms, welches die Werte der beiden aufgezeichneten Parameter über der Zeit grafisch darstellt.

5.3 Generieren der benötigten Artefakte

Aus der erstellten Konfiguration können nun verschiedene Artefakte über die in dem entwickelten Eclipse Plugin zur Verfügung stehenden Werkzeuge generiert werden, die für den Einsatz des Reporting Frameworks in einer Applikation benötigt werden. Die jeweiligen Generatoren sind über Schaltflächen in der Aktionsleiste des Eclipse XML-Editors zu erreichen.

1. Für die Writer-Komponente wird wie in Abschnitt 4.3 beschrieben eine log4j-Konfiguration mit entsprechend konfigurierten Loggern und Appendern benötigt.

2. Das Datenbankschema mit den benötigten Tabellen kann ebenfalls entsprechend der Konfiguration generiert werden.

3. Die Report Designs, die BIRT für die Generierung der Reports benötigt werden durch den in Abschnitt 4.2 Report Design Generator generiert.

Abbildung 5.1 zeigt ein Eclipse Projekt im Package Explorer View mit der erstellten Konfigurationsdatei (markiert) und allen generierten Artefakten.

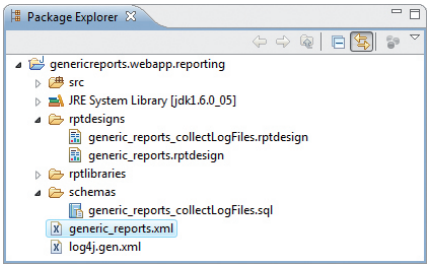

Abbildung 5.1: Eclipse Projekt mit Konfiguration (markiert) und den daraus generierten Artefakten

Die XML-Datei *log4j.gen.xml* ist die log4j-Konfigurationsdatei in der die von der Writer-Komponente benötigten Appender und Logger definiert werden. Im Order *rptdesigns* befinden sich die generierten Report Designs im Ordner *schemas* liegen die generierten Schema-Dateien.

5.4 Anpassen der Applikation

Die einzige Anpassung, die an der Applikation gemacht werden muss, ist das Einfügen der entsprechenden Aufrufe um die gewünschten Daten mit Hilfe der Writer-Komponente zu speichern.

Im ersten Schritt müssen die benötigten Bibliotheken zum Java *CLASSPATH* der Applikation hinzugefügt wären. Diese sind zum einen die log4j-Bibliothek von Apache und die für dieses Framework entwickelte Erweiterung in der Bibliothek *genericreports.log4j.jar*. Zusätzlich muss sich die generierte log4j-Konfigurationsdatei als *log4j.xml* ebenfalls im *CLASSPATH* der Applikation befinden. Da die Web-Viewer Applikation log4j noch für normales Logging benutzt wird, wird die Konfiguration der Appender und Logger aus der generierten log4j-Konfigurationsdatei in die bereits bestehende log4j-Konfigurationsdatei kopiert.

Im zweiten Schritt müssen die interessanten Daten in der Applikation gesammelt, in eine Map eingefügt und schließlich an den entsprechenden Logger übergeben werden. In Codebeispiel 5.2 ist die entsprechende Methode aus der Collector-Komponente zu sehen. In der 1. Zeile wird die Methode `getLogger()` in der Klasse `Logger` benutzt um eine Referenz auf den Logger mit dem Namen *collectLogFiles* (dies ist der Wert des Attributs *name* des LogRecords in der zuvor erstellten Konfiguration), über den die gesammelten Werte geschrieben werden sollen, zu erhalten. In Zeile 7 und den zwei folgenden Codezeilen werden zunächst die Map für die zu loggenden Werte und entsprechende Variablen initialisiert. In Zeile 20f wird die zuvor erzeugte Map mit den zu loggenden Werten gefüllt. Die Schlüssel unter denen die entsprechenden Werte in der Map abgelegt werden entsprechen den *name*-Attributen der entsprechenden `<logentry>`-Konfigurationen in der zuvor erstellten Konfigurationsdatei. Dabei ist darauf zu achten, dass für alle vorher konfigurierten `<logentry>`-Elemente auch ein Eintrag in der Map vorhanden ist, da sonst die Log-Datei nicht korrekt gelesen werden kann. Zuletzt wird in Zeile 23 eine geeignete Methode des zu Beginn geholten Logger-Objekts aufgerufen und die Map mit den zuvor ermittelten Werten an diese übergeben. Geeignet ist jede Methode zum Loggen des Logger-Objekts, da wie in Abschnitt 4.3.5 beschrieben die Logger für die Writer-Komponente so konfiguriert werden, dass sie jede Log-Anweisung egal mit welchem Level akzeptieren und entsprechend an den Appender übergeben.

```
1  private final Logger reportLogger = Logger.getLogger("
       collectLogFiles");
2
3  ...
4
5  public ErrorHandler collectLogFiles()
6  {
7    Map<String, Object> reportMap = new HashMap<String,
       Object>();
8
9    long startTime = System.currentTimeMillis();
10   int logFileCount = 0;
11
```

```
12    ErrorHandler errorHandler = new ErrorHandler(getClass());
13
14    for(Config config : configRepository.getConfigs())
15    {
16      collectLogFiles(config, errorHandler);
17      logFileCount += config.getLogRecordCount();
18    }
19
20    reportMap.put("logFileCount", logFileCount);
21    reportMap.put("collectTime", System.currentTimeMillis() -
          startTime);
22
23    reportLogger.info(reportMap);
24
25    return errorHandler;
26  }
```

Codebeispiel 5.2: Methode mit Reporting

Jedes Mal wenn nun die in Codebeispiel 5.2 beschriebene Methode aufgerufen wird, werden die aktuellen Werte für die zu loggenden Parameter ermittelt und über den Logger in die Log-Datei geschrieben.

5.5 Anlagen der Datenbanktabellen

Für die persistente Speicherung der geloggten Daten in einer Datenbank müssen die geeigneten Tabellen angelegt werden. In dem in Abschnitt 5.3 erläuterten Schritt wurde auch das Schema für die benötigte Tabelle generiert. Diese Datei enthält ein ausführbares SQL-Statement welches die entsprechende Tabelle in der Datenbank anlegt. Dieses kann entweder direkt ausgeführt oder als Orientierung beim Erstellen der Tabelle benutzt werden.

5.6 Zur Verfügung stellen des Reports über den Web Viewer

Als letztes müssen noch die Konfigurationsdatei und die generierten Report Designs in die entsprechenden Verzeichnisse der Web-Viewer Applikation kopiert werden.

Die Konfigurationsdatei wird zum Einen benötigt um in der Navigation die verfügbaren Reports anzeigen zu können, zum Anderen benötigt die Collektor-Komponente die Verbindungsparameter für die Datenbank und die Namen der LogRecords um die Namen der zu lesenden Log-Dateien zu konstruieren. Das Verzeichnis für die Konfigurationsdateien ist unter dem Schlüssel *configrep.basedir* in der Datei *genericreports.webapp.properties*

im Verzeichnis *WEB-INF/properties* der Web-Viewer Applikation definiert. Damit die neuen Reports in der Navigation auftauchen muss anschließend die Web-Viewer Applikation neu gestartet werden. Alternativ wird der Inhalt der *ConfigRepository*-Bean einmal täglich aktualisiert.

Das Verzeichnis für die Report Designs ist dagegen in der *web.xml*-Datei der BIRT Web-Viewer Applikation unter dem Schlüssel *BIRT_VIEWER_WORKING_FOLDER* definiert. Ein Neustart dieser Applikation ist nicht nötig damit der BIRT Web-Viewer auf die neuen Report Designs zugreifen kann.

5.7 Kopieren der Log-Dateien

Wie bereits in Abschnitt 4.4.6 erwähnt werden die Log-Dateien gemäß der log4j-Konfiguration der Writer-Komponente täglich um Mitternacht rotiert. Die Log-Dateien mit den Daten vom Vortag sollen anschließend von der Collector-Komponente gelesen und die darin befindlichen Daten in die Datenbank eingefügt werden. Dazu müssen diese Log-Dateien aus dem Verzeichnis für Log-Dateien der Applikation in das Verzeichniss kopiert werden, in dem die Collector-Komponente die zu lesenden Log-Dateien erwartet. Dieses Verzeichnis ist ebenfalls in der oben genannten Datei *genericreports.webapp.properties* unter dem Schlüssel *collector.logfiledir* konfiguriert. Nach dem erfolgten Lesevorgang werden die Log-Dateien in das unter dem Schlüssel *collector.archivedir* definierte Archiv-Verzeichniss kopiert.

Auch das Umkopieren der Log-Dateien wie oben beschrieben sollte möglichst in einem wiederkehrenden Prozess abgehandelt werden, der automatisch jede Nacht abläuft.

5.8 Aufrufen des Viewers zum Betrachten der Reports

Die Viewer Web-Applikation kann schließlich über den Browser unter der URL *http://[Servername]/genericreports.webapp* aufgerufen werden. Unter dem Namen der Applikation wie in der Konfigurationsdatei angegeben (*genericreports_webapp*) steht der konfigurierte Report *collectLogFiles* zur Auswahl. Sobald Daten für diesen Report in der Datenbank vorhanden sind kann er nach Eingabe der entsprechenden Zeitspanne generiert und im rechten Teil des Fensters betrachtet werden (siehe Abbildung 5.2).

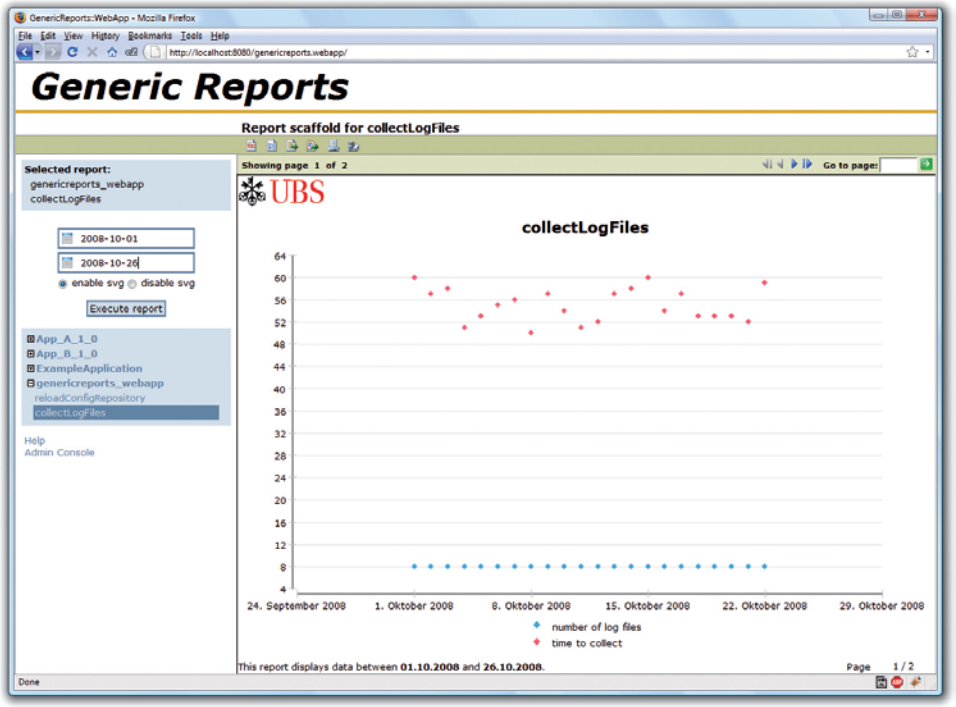

Abbildung 5.2: Komplettansicht der Viewer Web-Applikation

Kapitel 6

Fazit

In diesem Kapitel wird nun das Ergebnis dieser Diplomarbeit noch einmal zusammengefasst und die erzielten Ergebnisse anhand der gestellten Anforderungen evaluiert. Außerdem werden noch ein paar Ideen aufgeführt wie das entwickelte Framework in Zukunft weiterentwickelt werden könnte.

6.1 Zusammenfassung

Im Rahmen dieser Diplomarbeit ist es gelungen ein Framework zu entwickeln, welches Entwickler beim integrieren von Reportingfunktionalität in bereits bestehende oder in Entwicklung befindliche Applikationen unterstützt. Zusätzlich stellt das Framework Infrastrukturkomponenten zur Verfügung, die den kompletten Workflow vom Schreiben der Log-Daten in Dateien bis hin zum Generieren des fertigen Reports mit Hilfe einer Web-Applikation implementieren. Ferner wurden verschiedene Werkzeuge entwickelt, die den Entwickler beim Erstellen der Frameworkskonfiguration und der daraus abgeleiteten Artefakte unterstützen bzw. deren Generierung komplett übernehmen. Darüberhinaus sind nur noch wenige manuelle Schritte notwendig, um *einfache* Reportingfunktionalität in eine Applikation zu integrieren, die jedoch möglicherweise nicht immer den Anforderungen genügt.

Aufgrund der Entscheidung *BIRT* für die Generierung der Reports einzusetzen ist es jedoch möglich die generierten Report Designs mit Hilfe der BIRT-eigenen Werkzeuge an die eigenen Anforderungen anzupassen oder entsprechend zu erweitern indem man zum Beispiel zusätzliche Report Elemente oder Datenquellen hinzufügt. Eine Anpassung der anderen Komponenten (Writer, Collector, Viewer) ist dafür nicht notwendig.

Mit der Entwicklung der Web-Viewer Applikation wurde zudem die Möglichkeit geschaffen einfach und ohne die Installation von Zusatzsoftware auf die verfügbaren Reports zuzugreifen und diese gegebenenfalls

für weitere Zwecke abzuspeichern.

Bei der Entwicklung des Frameworks wurde großen Wert auf den Einsatz von OpenSource Software gelegt, angefangen bei den kleinen Bibliotheken wie *log4j* oder *Commons Configuration* über die beiden großen verwendeten Frameworks *BIRT* und *Spring* bis hin zur Entwicklungsumgebung *Eclipse* die nicht nur zur Entwicklung der Frameworkskomponenten sondern auch als Grundlage für die entwickelten Werkzeuge diente. Ein Vorteil ist natürlich, dass diese Bibliotheken und Frameworks jedem Entwickler kostenlos zur Verfügung stehen. Ein weiterer viel wichtiger Punkt allerdings ist, dass OpenSource Software durch diesen Umstand eine enorme Verbreitung erreicht und damit in den verschiedensten Umgebungen und unter verschiedenen Voraussetzungen getestet wird. Gefundene Probleme werden in aller Regel auch schneller behoben als bei ClosedSource Software, da jeder der über die benötigten Fähigkeiten verfügt an der Entwicklung teilhaben kann. Ohne den Einsatz von OpenSource Software wäre die Entwicklung des Frameworks im Rahmen dieser Diplomarbeit schon aus Zeitgründen nicht in diesem Umfang möglich gewesen.

6.2 Evaluierung

Dieser Abschnitt überprüft, ob das entwickelte Framework den Anforderungen, die zu Beginn der Arbeit in Abschnitt 2.2 aufgestellt wurden genügt.

Struktur der Logdaten konfigurierbar

Über die in Abschnitt 4.1 beschriebene Konfiguration ist es möglich für jede Anwendung die das Framework verwendet einige allgemeine Parameter sowie die Struktur der Logdaten festzulegen. Durch die Art und Weise wie diese Komponente entwickelt wurde ist es einfach möglich bei einem späteren Ausbau des Frameworks zusätzliche Konfigurationsoptionen zu integrieren.

API zum Loggen von Daten

Die Writer-Komponente bietet mit Hilfe von log4j ein Java-API zum einfachen Loggen der vorher festgelegten Daten. Die Änderungen die für deren Einsatz im Code vorgenommen werden müssen sind minimal und beeinträchtigen weder die Übersichtlichkeit des Codes noch die Performance der Applikation in relevanter Weise.

Persistente Speicherung der Logdaten

Die Collector-Komponente liest die geloggten Daten aus den Log-Dateien und fügt diese in zuvor erstellte Tabellen in einer Datenbank ein. Durch die Verwendung von JDBC an dieser Stelle wurde eine weitgehende Unabhängigkeit von der verwendeten Datenbank geschaffen. Diese Komponente ist als Java Serverapplikation umgesetzt, die zu einer vorher konfigurierten Zeit automatisch ihre Aufgabe erfüllt, manuelles Eingreifen ist somit nicht nötig.

Webbasierter Zugriff auf Reports

Um den Zugriff auf die gewünschten Reports möglchst einfach zu gestalten wurde eine Web-Applikation entwickelt, die den BIRT Web-Viewer integriert, mit Hilfe dessen intuitiv in einem Report navigiert werden kann und der verschiedene Optionen zum Export des Reports in verschiedene Formate zur weiteren Verarbeitung oder zur Verteilung an verschiedene Interessenten anbietet.

Generieren der Reportvorlagen

Mit dem Report Design Generator wurde ein Werkzeug entwickelt, welches basierend auf der erstellten Konfiguration ein entsprechendes Report Design erstellt. Die Implementation dieses Generators erfolgte zur einfachen Anwendung als Eclipse Plugin mit Hilfe der BIRT APIs. Auch hier wurde darauf geachtet, dass der Generator einfach zu erweitern ist um zukünftige Entwicklungen des Frameworks nicht zu behindern.

Zusätzliche Entwicklungen

Zusätzlich zu den zu Beginn gestellten Anforderungen wurden noch einige Werkzeuge entwickelt, die den Entwickler bei der Verwendung des Frameworks unterstützen und die manuellen Schritte bei dessen Anwendung auf ein Minimum reduzieren. Dabei wurde auf eine gute Integration in die bereits im Einsatz befindliche Entwicklungsumgebung großen Wert gelegt.

6.3 Ausblick

In diesem Abschnitt werden einige Möglichkeiten kurz umrissen wie das Framework in Zukunft weiterentwickelt werden könnte.

Erweiterung der generierten Reports

Aktuell werden wie beschrieben auf den generierten Reports nur eine Tabelle mit den geloggten Werten angezeigt. Optional kann in der Konfiguration angegeben werden, dass zusätzlich Diagramme die verschiedene Wertverläufe grafisch darstellen auf dem Report zu sehen sein sollen. Um die Auswertung der Daten zu vereinfachen könnten zum Beispiel aus den geloggten Werten bestimmte Angaben abgeleitet werden (z.B. Durchschnittswert) die dann auch in dem generierten Report vorhanden wären. Um dies zu erreichen müsste nur der Report Design Generator angepasst werden um die entsprechenden Report Elemente zu erzeugen und deren Werte entsprechend zu berechnen.

Zusätzlich könnte die Bibliothek zum Lesen der Konfiguration angepasst werden um zusätzliche Optionen zu ermöglichen.

Fehlerbehandlung beim Schreiben der Logdaten

In Abschnitt 5.4 wurde erwähnt, dass der Entwickler bei der Integration der Writer-Komponente in die Applikation darauf achten muss, dass er die Map mit allen vorher konfigurierten Werten füllt, da es sonst zu Problemen beim Lesen der Log-Datei kommt. Um solche Fehler bereits während der Entwicklung zu bemerken wäre es denkbar, dass die Writer-Komponente vor dem Schreiben der Daten in die Log-Datei überprüft, ob alle Werte gemäß der LogRecord-Konfiguration vorhanden sind und wenn nicht eine Warnung ausgibt, um den Entwickler darüber zu informieren.

Sinnvoll wäre an dieser Stelle, wenn die Writer-Komponente während der Entwicklung in einer Art Debug-Modus betrieben werden könnte in dem Fehler bei der Verwendung als Warnung ausgegeben werden. Ist eine Applikation einmal produktiv in Betrieb und alle Aufrufe der Writer-Komponente ohne Fehler könnte dann der Debug-Modus wieder abgeschalten werden um keine unnötige Fehlerüberprüfung vornehmen zu müssen.

Weiterentwicklung der Viewer-Komponente

Die Web-Viewer Applikation die im Rahmen dieser Diplomarbeit zum Zugriff auf die Reports entwickelt wurde bietet darüberhinaus kaum zusätzliche Möglichkeiten. So ist es durchaus denkbar, dass einige Reports möglicherweise vertrauliche Daten enthalten. Um den Zugriff auf diese Reports zu beschränken müsste die Web-Viewer Applikation zusätzlich um Benutzerauthentifizierung und Autorisierung erweitert werden. Auch für diese Anforderung bietet das verwendete Framework Spring entsprechende Möglichkeiten.

Abgesehen von der Zugriffsbeschränkung wäre es durch die Authentifizierung von Benutzern möglich die Web-Viewer Applikation zu perso-

nalisieren und zum Beispiel eine Art Favoriten-Liste anzubieten, in die ein Benutzer häufig genutzte Reports aufnehmen kann um schneller darauf zuzugreifen zu können.

Genau wie BIRT bietet auch Spring die Möglichkeit Applikationen zu internationalisieren. Durch die Auslagerung von Strings in Property-Dateien wurde dieser Schritt in der Web-Viewer Applikation bereits vollzogen. In einem weiteren Schritt könnte sie dann in weitere Sprachen lokalisiert werden.

Anhang A

Build Skripte

A.1 Ant Script für genericreports.config Bibliothek

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!-- ============================================
3       22.07.2008 10:15:45
4
5       genericreports.config
6       This project builds a jar file that contains classes
7       that allow to easily access the genericreports
           configuration.
8
9       t370888
10      ============================================ -->
11 <project name="genericreports.config" default="jar">
12
13   <property name="libs" location="${basedir}/../
        genericreports.lib" />
14   <property name="src" location="src" />
15   <property name="src-test" location="test" />
16   <property name="bin" location="bin" />
17   <property name="doc" location="doc" />
18   <property name="dist" location="dist" />
19   <property name="build.compiler" value="org.eclipse.jdt.
        core.JDTCompilerAdapter" />
20
21   <path id="classpath">
22     <pathelement path="${libs}/commons-jxpath-1.3.jar" />
23     <pathelement path="${libs}/commons-lang-2.4.jar" />
24     <pathelement path="${libs}/commons-collections-3.2.1.
          jar" />
25     <pathelement path="${libs}/commons-configuration-1.5.
          jar" />
26     <pathelement path="${libs}/commons-logging-1.1.1.jar" /
          >
```

```
27      <pathelement path="${libs}/apache-log4j-1.2.15/log4j
            -1.2.15.jar" />
28    </path>
29
30    <path id="classpath-test">
31      <pathelement path="${libs}/junit.jar" />
32      <pathelement path="${libs}/easymock.jar" />
33      <pathelement path="${libs}/easymockclassextension.jar"
            />
34      <pathelement path="${libs}/cglib-nodep-2.2.jar" />
35    </path>
36
37    <!-- =================================
38            target: clean
39           ================================= -->
40    <target name="clean" description="cleans up the output
         directory">
41      <delete dir="${bin}" />
42      <delete dir="${dist}" />
43    </target>
44
45    <!-- =================================
46            target: init
47           ================================= -->
48    <target name="init" depends="clean" description="
         initializes the environment">
49      <mkdir dir="${bin}" />
50      <mkdir dir="${dist}" />
51    </target>
52
53    <!-- =================================
54            target: build
55           ================================= -->
56    <target name="build" depends="init" description="compiles
         the java files">
57      <javac destdir="${bin}" source="1.5" target="1.5" debug
           ="true">
58        <classpath refid="classpath" />
59        <classpath refid="classpath-test" />
60        <src>
61          <pathelement path="${src}" />
62          <pathelement path="${src-test}" />
63        </src>
64      </javac>
65      <copy todir="${bin}/config">
66        <fileset dir="${src}/config" />
67      </copy>
68    </target>
69
```

```
70   <!-- ================================
71           target: test
72         ================================ -->
73     <target name="test" depends="build" description="-->
          executes the tests; stops on failure">
74       <mkdir dir="testoutput" />
75
76         <junit printsummary="yes" fork="true" haltonfailure
             ="yes">
77           <classpath refid="classpath" />
78           <classpath refid="classpath-test" />
79           <classpath location="${bin}" />
80           <formatter type="plain" />
81           <batchtest haltonfailure="no" todir="testoutput">
82             <fileset dir="${src-test}">
83             <include name="**/*Test.java" />
84         </fileset>
85           </batchtest>
86         </junit>
87     </target>
88
89
90   <!-- ================================
91           target: jar
92         ================================ -->
93   <target name="jar" depends="test" description="integrates
        the built files into a jar file">
94     <jar destfile="${dist}/genericreports.config.jar"
          compress="true">
95       <fileset dir="${bin}" excludes="**/*Test.class" />
96     </jar>
97   </target>
98
99 </project>
```

Codebeispiel A.1: Ant Script für genericreports.config Bibliothek

A.2 Ant Script für genericreports.log4j Bibliothek

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!-- ============================================
3       22.07.2008 08:53:14
4
5       genericreports.log4j
6       This project contains enhancements for the log4j
           package
7       to make it usable with the generic report generator.
8
9       t370888
10      ============================================ -->
11 <project name="genericreports.log4j" default="jar">
12     <description>
13             This project contains enhancements for the
                  log4j package to make it usable with the
                  generic report generator.
14     </description>
15
16   <property name="src" location="src" />
17   <property name="bin" location="bin" />
18   <property name="dist" location="dist" />
19   <property name="libs" location="${basedir}/../
        genericreports.lib" />
20   <property name="build.compiler" value="org.eclipse.jdt.
        core.JDTCompilerAdapter" />
21
22   <path id="classpath">
23     <pathelement path="${libs}/log4j-1.2.15.jar" />
24   </path>
25
26   <!-- =================================
27           target: clean
28          ================================= -->
29     <target name="clean" description="--> cleans the output
           directories">
30       <delete dir="${bin}" />
31       <delete dir="${dist}" />
32     </target>
33
34
35   <!-- =================================
36           target: init
37          ================================= -->
38     <target name="init" depends="clean" description="-->
          initializes all directories needed for building the
          project.">
39        <mkdir dir="${bin}" />
```

```
40        <mkdir dir="${dist}" />
41      </target>
42
43
44      <!-- ================================
45            target: build
46           ================================ -->
47      <target name="build" depends="init" description="-->
           This project contains enhancements for the log4j
           package to make it usable with the generic report
           generator.">
48          <javac destdir="${bin}" source="1.5" target="1.5"
              debug="true">
49            <classpath refid="classpath" />
50            <src>
51              <pathelement path="${src}" />
52            </src>
53          </javac>
54      </target>
55
56    <!-- ================================
57            target: jar
58           ================================ -->
59      <target name="jar" depends="build" description="-->
           packages the built class files into a jar file">
60          <jar destfile="${dist}/genericreports.log4j.jar"
              compress="true">
61          <fileset dir="${bin}" />
62          </jar>
63      </target>
64
65  </project>
```

Codebeispiel A.2: Ant Script für genericreports.log4j Bibliothek

A.3 Ant Script für Eclipse Plugin

```
1  <?xml version="1.0" encoding="UTF-8" ?>
2  <!-- ========================================
3       10.08.2008 11:23:17
4
5       genericreports.plugin
6       This project builds an eclipse plugin that contains
7       tools that ease the use of the GenericReports
            framework.
8
9       t370888
10      ======================================== -->
11 <project name="genericreports.plugin" default="
       build_product_without_source">
12
13   <!-- ================================
14           target: properties
15          ================================ -->
16   <target name="properties">
17     <property name="src" value="src" />
18     <property name="bin" value="bin" />
19     <property name="libs" value="libs" />
20
21     <property name="build.root" value="../build/
          genericreports.plugin" />
22     <property name="build.temp" value="${build.root}/temp"
          />
23     <property name="build.out" value="${build.root}/product
          " />
24
25     <!-- copy MANIFEST.MF to temp directory and modify it
          slightly to use it as a properties file -->
26     <copy file="META-INF/MANIFEST.MF" todir="${build.temp}"
           />
27     <replace file="${build.temp}/MANIFEST.MF">
28       <replacefilter token=":" value="=" />
29       <replacetoken>,
30 </replacetoken>
31       <replacevalue> </replacevalue>
32     </replace>
33     <property file="${build.temp}/MANIFEST.MF" />
34
35     <property name="plugin.filename" value="genericreports.
          plugin_${Bundle-Version}" />
36     <property name="plugin.files" location="${build.temp}/
          files" />
37     <property name="plugin.classes.jar" location="${plugin.
          files}/genericreports.plugin.jar" />
```

```
38      <property name="plugin.jars" location="${build.temp}/
           jars" />
39      <property name="plugin.jar" location="${plugin.jars}/
           plugins/${plugin.filename}.jar" />
40      <property name="product.zip" value="${build.out}/
           genericreports.plugin_v${Bundle-Version}.zip" />
41    </target>
42
43    <!-- ================================
44            target: clean
45           ================================ -->
46    <target name="clean" depends="properties">
47      <!-- only delete the temp directory to keep old built
           versions of the product -->
48      <delete dir="${build.temp}" />
49    </target>
50
51    <!-- ================================
52            target: init
53           ================================ -->
54    <target name="init" depends="clean">
55      <mkdir dir="${build.temp}" />
56      <mkdir dir="${build.out}" />
57      <mkdir dir="${plugin.files}" />
58      <mkdir dir="${plugin.jars}/plugins" />
59    </target>
60
61    <!-- ================================
62            macrodef: build_plugin
63           ================================ -->
64    <macrodef name="build_plugin" description="builds the
         plugin jar file">
65      <sequential>
66
67        <zip destfile="${plugin.classes.jar}">
68          <fileset dir="${bin}">
69            <include name="**/*.*" />
70            <exclude name="**/*Test.class" />
71          </fileset>
72        </zip>
73
74        <copy todir="${plugin.files}">
75          <fileset dir="." includes="${plugin.classes.jar}" /
               >
76
77          <fileset dir="." includes="META-INF/MANIFEST.MF" />
78          <fileset dir="." includes="plugin.xml" />
79          <fileset dir="." includes="velocity.properties" />
80
```

```
81        <fileset dir="." includes="${libs}/*.jar" />
82
83        <fileset dir="." includes="icons/*" />
84        <fileset dir="." includes="rptlibraries/*.*" />
85        <fileset dir="." includes="templates/*.*" />
86      </copy>
87
88      <zip destfile="${plugin.jar}">
89        <fileset dir="${plugin.files}">
90          <include name="**/*.*" />
91        </fileset>
92      </zip>
93
94    </sequential>
95  </macrodef>
96
97  <!-- ================================
98          macrodef: build_product
99         ================================ -->
100 <macrodef name="build_product">
101   <sequential>
102
103     <build_plugin />
104
105     <zip destfile="${product.zip}">
106       <fileset dir="${plugin.jars}" />
107     </zip>
108
109   </sequential>
110 </macrodef>
111
112 <!-- ================================
113         target: build_product_with_source
114        ================================ -->
115   <target name="build_product_with_source" depends="init"
          description="builds the product ready for
        installation with source code included">
116
117   <zip destfile="${plugin.files}/src.zip">
118     <fileset dir="${src}" />
119   </zip>
120
121     <build_product />
122
123   </target>
124
125 <!-- ================================
126         target: build_product_without_source
127        ================================ -->
```

```
128   <target name="build_product_without_source" depends="init
          " description="builds the product ready for
          installation">
129
130     <build_product />
131
132   </target>
133
134 </project>
```

Codebeispiel A.3: Ant Script für Eclipse Plugin

A.4 Ant Script für GenericReports Web-Applikation

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!-- ============================================
3       13.08.2008 14:48:06
4
5       genericreports.webapp
6       This project realizes the viewer and collector web
7       application for the generic reports toolset.
8
9       t370888
10      ================<============================ -->
11 <project name="genericreports.webapp" default="war">
12   <description>
13     This project realizes the viewer and collector web
           application for the generic reports toolset.
14     </description>
15
16   <property name="src.webcontent" location="WebContent" />
17   <property name="src.classes" location="build/classes" />
18
19   <property name="build.root" location="../build/
        genericreports.webapp" />
20   <property name="build.temp" location="${build.root}/temp"
         />
21   <property name="build.out" location="${build.root}/war" /
        >
22   <property name="build.classes" location="${build.temp}/
        WEB-INF/classes"/>
23   <property name="build.war" value="genericreports.webapp.
        war" />
24
25   <!-- - - - - - - - - - - - - - - - - - -
26           target: gather
27         - - - - - - - - - - - - - - - - - - - - -->
28   <target name="gather">
29     <mkdir dir="${build.temp}" />
30     <copy todir="${build.temp}">
31       <fileset dir="${src.webcontent}">
32         <include name="**/*.*" />
33       </fileset>
34     </copy>
35
36     <mkdir dir="${build.classes}" />
37     <copy todir="${build.classes}">
38       <fileset dir="${src.classes}">
39         <include name="**/*.*" />
40       </fileset>
41     </copy>
```

```
42    </target>
43
44
45    <!-- =================================
46           target: war
47          ================================= -->
48    <target name="war" depends="gather" description="--> This
          project simply exists to modify and repack the
        WebViewerExample that comes with BIRT.">
49      <mkdir dir="${build.out}" />
50
51      <zip destfile="${build.out}/${build.war}">
52        <fileset dir="${build.temp}">
53          <include name="**/*.*" />
54        </fileset>
55      </zip>
56    </target>
57
58  </project>
```

Codebeispiel A.4: Ant Script für GenericReports Web-Applikation

Anhang B

Velocity Templates

B.1 Template für neue Konfiguration

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE config PUBLIC "-//UBS//DTD XML GenericReports
       Config v1.0//EN" "config.dtd">
3  <config appname="${appName}">
4
5    <logfiledir>${logFileDir}</logfiledir>
6
7    <datastore>
8      <driver>${dataStoreDriver}</driver>
9      <url>${dataStoreUrl}</url>
10     <username>${dataStoreUsername}</username>
11     <password>${dataStorePassword}</password>
12 #if(${dataStorePrefix} != "")
13     <prefix>${dataStorePrefix}</prefix>
14 #end
15   </datastore>
16
17   <!-- TODO Add LogRecord configuration here -->
18
19 </config>
```

Codebeispiel B.1: Template für neue Konfiguration

B.2 Template für log4j Konfiguration (properties)

```
1  log4j.renderer.java.util.Map = genericreports.log4j.
       renderer.MapXMLRenderer
2
3  #foreach($logRecCfg in $logRecCfgs)
4  #set($name = $logRecCfg.getName())
5  log4j.appender.${name}=org.apache.log4j.
       DailyRollingFileAppender
6  log4j.appender.${name}.file=${logPath}/${name}.log
7  log4j.appender.${name}.append=true
8  log4j.appender.${name}.datePattern=${frequencyPattern}
9  log4j.appender.${name}.layout=genericreports.log4j.renderer
       .XMLReportLayout
10 log4j.appender.${name}.layout.includeLocationInfo=false
11 log4j.appender.${name}.layout.includeProperties=false
12
13 #end
14
15 #foreach($logRecCfg in $logRecCfgs)
16 #set($name = $logRecCfg.getName())
17 log4j.logger.${name}=INFO, ${name}
18 log4j.additivity.${name}=false
19
20 #end
```

Codebeispiel B.2: Template für log4j Konfiguration im .properties-Format

B.3 Template für log4j Konfiguration (XML)

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">
3
4  <log4j:configuration debug="false" xmlns:log4j="http://
       jakarta.apache.org/log4j/">
5    <renderer renderedClass="java.util.Map" renderingClass="
       genericreports.log4j.renderer.MapXMLRenderer" />
6
7  #foreach($logRecCfg in $logRecCfgs)
8  #set($name = $logRecCfg.getName())
9    <appender name="${name}" class="org.apache.log4j.
       DailyRollingFileAppender">
10     <param name="file" value="${logPath}/${name}.log" />
11     <param name="append" value="true" />
12     <param name="datePattern" value="${frequencyPattern}"
         />
13
14     <layout class="genericreports.log4j.layout.
         XMLReportLayout">
15       <param name="includeLocationInfo" value="false" />
16       <param name="includeProperties" value="false" />
17     </layout>
18   </appender>
19
20 #end
21
22 #foreach($logRecCfg in $logRecCfgs)
23 #set($name = $logRecCfg.getName())
24   <logger name="${name}" additivity="false">
25     <level value="ALL" />
26
27     <appender-ref ref="${name}" />
28   </logger>
29
30 #end
31 </log4j:configuration>
```

Codebeispiel B.3: Template für log4j Konfiguration im XML-Format

B.4 Template für Datenbankschema

```
1 CREATE TABLE ${tableName}
2 (
3 ${tableColumns}
4 )
```

Codebeispiel B.4: Template für Datenbankschema

Anhang C

Javascript Code

C.1 Javascript für Viewer Navigation

```
1  document.observe('dom:loaded', function()
2  {
3    // observe the submit event on the reportform
4    Event.observe('reportform', 'submit', validateForm);
5
6    $$('li.app').each(function(element)
7    {
8      // apply the collapsed class to all the list items that
           represent applications
9      element.addClassName('collapsed');
10
11     // observe the click event on the first child element
           to provide list expanding/collapsing
12     element.down().observe('click', function(event)
13     {
14       var treebutton = Event.element(event).up();
15
16       if(treebutton.hasClassName('collapsed'))
17       {
18         treebutton.removeClassName('collapsed').
               addClassName('expanded');
19       }
20       else
21       {
22         treebutton.removeClassName('expanded').addClassName
               ('collapsed');
23       }
24     });
25   });
26 });
27
28 function setReport(app, report, linkElement)
```

```
29 {
30   if(report == '')
31   {
32     reportPath = app + '.rptdesign';
33   }
34   else
35   {
36     reportPath = app + '_' + report + '.rptdesign';
37   }
38
39   report = report == '' ? 'summary report' : report;
40
41   $('selectedapp').update(app);
42   $('selectedreport').update(report);
43
44   $('report').writeAttribute('value', reportPath);
45
46   // remove active class before adding
47   $$('li.active').invoke('removeClassName', 'active');
48
49   linkElement = $(linkElement);
50   linkElement.up().addClassName('active');
51 }
52
53 function showCalendar(element)
54 {
55   new CalendarDateSelect(element,
56   {
57     buttons: false,
58     valid_date_check: function(date)
59     {
60       return date.stripTime() <= (new Date().stripTime());
61     }
62   });
63 }
64
65 function validateForm(submitEvent)
66 {
67   var submitForm = true;
68
69   var startDateGiven = false;
70   var startDateValid = false;
71
72   var endDateGiven = false;
73   var endDateValid = false;
74
75   var reportGiven = false;
76
77   var reDate = /^(\d{4})-(\d{1,2})-(\d{1,2})$/;
```

```
78
79    var startDateString = $('startDate').getValue();
80    if(startDateString != null && !startDateString.blank())
81    {
82      startDateGiven = true;
83
84      if(startDateString.search(reDate) != -1)
85      {
86        startDateValid = true;
87      }
88    }
89
90    var endDateString = $('endDate').getValue();
91    if(endDateString != null && !endDateString.blank())
92    {
93      endDateGiven = true;
94
95      if(endDateString.search(reDate) != -1)
96      {
97        endDateValid = true;
98      }
99    }
100
101   if(!$('report').getValue().blank())
102   {
103     reportGiven = true;
104   }
105
106   // if only start date is given conveniently set the
        endDate to the day after the startDate
107   if(startDateValid && !endDateGiven && reportGiven)
108   {
109     startDateInfo = startDateString.match(reDate);
110
111     // month basis is 0 so substract 1 from the month
112     // add one to day for endDate
113     d = new Date(startDateInfo[1], startDateInfo[2] - 1,
          parseInt(startDateInfo[3], 10) + 1);
114
115     month = d.getMonth() + 1;
116     month = month < 10 ? '0' + month : month;
117     day = d.getDate() < 10 ? '0' + d.getDate() : d.getDate
          ();
118
119     endDateString = d.getFullYear() + '-' + month + '-' +
          day
120     $('endDate').setValue(endDateString);
121
122     endDateGiven = true;
```

```
123       endDateValid = true;
124     }
125
126     var errorMessage = '';
127
128     if(!startDateGiven)
129     {
130       errorMessage = errorMessage.concat('The start date is
              missing.<br />');
131       submitForm = false;
132     }
133     else if(!startDateValid)
134     {
135       errorMessage = errorMessage.concat('The start date
              format is invalid.<br />');
136       submitForm = false;
137     }
138
139     if(!endDateGiven)
140     {
141       errorMessage = errorMessage.concat('The end date is
              missing.<br />');
142       submitForm = false;
143     }
144     else if(!endDateValid)
145     {
146       errorMessage = errorMessage.concat('The end date format
               is invalid.<br />');
147       submitForm = false;
148     }
149
150     if(startDateValid && endDateValid)
151     {
152       startDateInfo = startDateString.match(reDate);
153       endDateInfo = endDateString.match(reDate);
154
155       startDate = new Date(startDateInfo[1], startDateInfo[2]
               - 1, startDateInfo[3]).getTime();
156       endDate = new Date(endDateInfo[1], endDateInfo[2] - 1,
              endDateInfo[3]).getTime();
157
158       if(startDate > endDate)
159       {
160         errorMessage = errorMessage.concat('The start date is
                  set after the end date.<br />');
161         submitForm = false;
162       }
163       else if(startDate == endDate)
164       {
```

```
165       errorMessage = errorMessage.concat('The start date
              and the end date are identical.<br />');
166       submitForm = false;
167     }
168   }
169
170   if(!reportGiven)
171   {
172     errorMessage = errorMessage.concat('No report selected
          .');
173     submitForm = false;
174   }
175
176   $('nav-errors').update(errorMessage);
177
178   if(!submitForm)
179   {
180     Effect.Queues.get('errorscope').each(function(effect) {
            effect.cancel(); });
181
182     new Effect.Highlight('nav-errors', {
183       queue: {position: 'end', scope: 'errorscope'},
184       duration: 0.4,
185       startcolor: '#FFFFFF',
186       endcolor: '#FFF0F0',
187       restorecolor: '#FFF0F0'
188     });
189
190     new Effect.Highlight('nav-errors', {
191       queue: {position: 'end', scope: 'errorscope'},
192       duration: 1,
193       delay: 0.5,
194       startcolor: '#FFF0F0',
195       endcolor: '#FFFFFF',
196       restorecolor: '#FFFFFF'
197     });
198
199     submitEvent.stop();
200   }
201 }
```

Codebeispiel C.1: Javascript für Viewer Navigation

Anhang D

Viewer-Komponente

D.1 Deployment Descriptor

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-
       instance"
3    xmlns="http://java.sun.com/xml/ns/javaee"
4    xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.
       xsd"
5    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
       http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
6    id="WebApp_ID" version="2.5">
7
8    <display-name>WebApplication for generic reports</display
       -name>
9
10   <servlet>
11     <servlet-name>genericreports.webapp</servlet-name>
12     <servlet-class>org.springframework.web.servlet.
         DispatcherServlet</servlet-class>
13     <load-on-startup>1</load-on-startup>
14   </servlet>
15
16   <servlet-mapping>
17     <servlet-name>genericreports.webapp</servlet-name>
18     <url-pattern>*.html</url-pattern>
19   </servlet-mapping>
20
21   <welcome-file-list>
22     <welcome-file>index.jsp</welcome-file>
23   </welcome-file-list>
24
25   <jsp-config>
26     <taglib>
27       <taglib-uri>/spring</taglib-uri>
```

```
28        <taglib-location>/WEB-INF/taglibs/spring.tld</taglib-
              location>
29      </taglib>
30    </jsp-config>
31
32  </web-app>
```

Codebeispiel D.1: Deployment Descriptor der Viewer Web-Applikation

D.2 Application Context

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <beans xmlns="http://www.springframework.org/schema/beans"
3       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4       xsi:schemaLocation="http://www.springframework.org/
           schema/beans
5       http://www.springframework.org/schema/beans/spring-
           beans-2.5.xsd">
6
7    <bean id="internalResourceViewResolver" class="org.
        springframework.web.servlet.view.
        InternalResourceViewResolver">
8      <property name="viewClass" value="org.springframework.
          web.servlet.view.JstlView" />
9      <property name="prefix" value="/WEB-INF/jsp/" />
10     <property name="suffix" value=".jsp" />
11   </bean>
12
13   <bean id="propertyPlaceholderConfigurer" class="org.
        springframework.beans.factory.config.
        PropertyPlaceholderConfigurer">
14     <property name="locations" value="WEB-INF/properties/
          genericreports.webapp.properties" />
15   </bean>
16
17   <bean id="messageSource" class="org.springframework.
        context.support.ResourceBundleMessageSource">
18     <property name="basename" value="resources.messages" />
19     <property name="useCodeAsDefaultMessage" value="true" /
          >
20   </bean>
21
22   <bean id="simpleUrlMapping" class="org.springframework.
        web.servlet.handler.SimpleUrlHandlerMapping">
23     <property name="mappings">
24       <props>
25         <prop key="/title.html">titleController</prop>
26         <prop key="/navigation.html">navigationController</
              prop>
27         <prop key="/help.html">helpController</prop>
28         <prop key="/admin.html">adminController</prop>
29       </props>
30     </property>
31   </bean>
32
33   <bean id="collectorJob" class="org.springframework.
        scheduling.quartz.JobDetailBean">
```

```
34      <property name="jobClass" value="genericreports.webapp.
           jobs.LogFileCollectorJob" />
35      <property name="jobDataAsMap">
36        <map>
37          <entry key="logFileCollector" value-ref="
               logFileCollector" />
38        </map>
39      </property>
40    </bean>
41
42    <bean id="collectorJobTrigger" class="org.springframework
         .scheduling.quartz.CronTriggerBean">
43      <property name="jobDetail" ref="collectorJob" />
44      <property name="cronExpression" value="${collector.
           cronexpression}" />
45    </bean>
46
47    <bean id="reloadConfigRepositoryJob" class="org.
         springframework.scheduling.quartz.JobDetailBean">
48      <property name="jobClass" value="genericreports.webapp.
           jobs.ConfigRepositoryReloadJob" />
49      <property name="jobDataAsMap">
50        <map>
51          <entry key="configRepository" value-ref="
               configRepository" />
52        </map>
53      </property>
54    </bean>
55
56    <bean id="reloadConfigRepositoryJobTrigger" class="org.
         springframework.scheduling.quartz.CronTriggerBean">
57      <property name="jobDetail" ref="
           reloadConfigRepositoryJob" />
58      <property name="cronExpression" value="${configrep.
           reloaderjob.cronexpression}" />
59    </bean>
60
61    <bean id="schedulerFactory" class="org.springframework.
         scheduling.quartz.SchedulerFactoryBean">
62      <property name="triggers">
63        <list>
64          <ref bean="collectorJobTrigger" />
65          <ref bean="reloadConfigRepositoryJobTrigger" />
66        </list>
67      </property>
68      <property name="quartzProperties">
69        <props>
70          <prop key="org.quartz.threadPool.threadCount">1</
               prop>
```

```
71      </props>
72    </property>
73  </bean>
74
75  <bean id="titleController" class="genericreports.webapp.
       controller.TitleController" />
76
77  <bean id="helpController" class="genericreports.webapp.
       controller.HelpController" />
78
79  <bean id="navigationController" class="genericreports.
       webapp.controller.NavigationController">
80    <property name="configRepository" ref="configRepository
         " />
81  </bean>
82
83  <bean id="adminController" class="genericreports.webapp.
       controller.AdminController">
84    <property name="methodNameResolver" ref="
         parameterMethodNameResolver" />
85    <property name="configRepository" ref="configRepository
         " />
86    <property name="logFileCollector" ref="logFileCollector
         " />
87  </bean>
88
89  <bean id="parameterMethodNameResolver" class="org.
       springframework.web.servlet.mvc.multiaction.
       ParameterMethodNameResolver">
90    <property name="paramName" value="action" />
91    <property name="defaultMethodName" value="index" />
92  </bean>
93
94  <bean id="configRepository" class="genericreports.webapp.
       ConfigRepository">
95    <property name="configFileDir" value="${configrep.
         basedir}" />
96  </bean>
97
98  <bean id="logFileCollector" class="genericreports.webapp.
       collector.LogFileCollector">
99    <property name="logFileParser" ref="logFileParser" />
100   <property name="configRepository" ref="configRepository
         " />
101   <property name="logFileDir" value="${collector.
         logfiledir}" />
102   <property name="archiveDir" value="${collector.
         archivedir}" />
103 </bean>
```

```
104
105   <bean id="logFileParser" class="genericreports.webapp.
          collector.LogFileParser" />
106
107 </beans>
```

Codebeispiel D.2: Application Context der Viewer Web-Applikation

Anhang E

CD mit Quelltexten

Im hinteren Buchdeckel dieser Diplomarbeit befindet sich eine CD mit allen im Laufe dieser Arbeit entwickelten Quelltexten und dieser Diplomarbeit als PDF-Datei. Für Hinweise zur Verwendung der Quelltexte sollte die Datei *Readme.txt* auf der CD konsultiert werden.

Literaturverzeichnis

[CR06] Clayberg, Eric und Dan Rubel: *Eclipse: Building Commercial-Quality Plug-ins*. Addison Wesley Professional, 2. Auflage, 3 2006.

[ecl08] eclipse.org: *Using the BIRT Report Viewer*. `http://www.eclipse.org/birt/phoenix/deploy/viewerUsage2.2.php`, 2008. [Online; Stand 17. Oktober 2008].

[GS03] Gülcü, Ceki und Scott Stark: *The Complete Log4j Manual: The Reliable, Fast and Flexible Logging Framework for Java*. QOS.ch, 2003.

[PHH06] Peh, Diana, Alethea Hannemann und Nola Hague: *BIRT: A Field Guide to Reporting*. Addison Wesley Professional, 10 2006.

[SAX] SAXProject.org: *SAX project homepage*. `http://www.saxproject.org/`. [Online; Stand 10.September 2008].

[spr08] springframework.org: *Spring Documentation*. `http://springframework.org/documentation/`, 2008. [Online; Stand 13. Oktober 2008].

[W3Sa] W3Schools.com: *Javascript Tutorial*. `http://w3schools.com/js/`. [Online; Stand 10. September 2008].

[W3Sb] W3Schools.com: *XML Tutorial*. `http://w3schools.com/xml/`. [Online; Stand 10. September 2008].

[WB07] Walls, Craig und Ryan Breidenbach: *Spring in Action*. Manning Publications, 2. Auflage, 8 2007.

[WFB+06] Weathersby, Jason, Don French, Tom Bondur, Jane Tatchell und Iana Chatalbasheva: *Integrating and Extending BIRT*. Addison Wesley Professional, 11 2006.

[Wik08a] Wikipedia: *Framing (World Wide Web) — Wikipedia, The Free Encyclopedia*. `http://en.wikipedia.org/w/index.php?title=Framing_(World_Wide_Web)&oldid=243159829`, 2008. [Online; Stand 15 Oktober 2008].

[Wik08b] Wikipedia: *JAR (file format) — Wikipedia, The Free Encyclopedia*. http://en.wikipedia.org/w/index.php?title=JAR_(file_format)&oldid=238025153, 2008. [Online; Stand 16. September 2008].

[Wik08c] Wikipedia: *Java annotation — Wikipedia, The Free Encyclopedia*. http://en.wikipedia.org/w/index.php?title=Java_annotation&oldid=220456015, 2008. [Online; Stand 16. September 2008].

[Wik08d] Wikipedia: *Spring Framework — Wikipedia, The Free Encyclopedia*. http://en.wikipedia.org/w/index.php?title=Spring_Framework&oldid=242743370, 2008. [Online; Stand 14. Oktober 2008].