Frontend eines Agilent-Parsers erstellen

Belegarbeit

eingereicht am Fachbereich

Informatik

der Hochschule Zittau/Görlitz (HAW)

als Prüfungsleistung im Fach

Data Mining 2

vorgelegt von:

Christof Ochmann (35989)

Ingo Körner (40586)

Görlitz, 07. Februar 2013

Betreuer: Prof. ten Hagen

Abstract

In diesem Projekt wird ein Logdatei-Parser erstellt, der eine Agilent-Logdatei einlesen und in ein gegebenes Zwischenformat umwandeln kann. Das Zwischenformat wird dabei nicht ausgeben, sondern nur im Hauptspeicher zur Verfügung gestellt. Der Parser soll als eine Art Bibliothek von einem anderen Programmen genutzt werden, dass dieses Zwischenformat übernimmt und weiterverarbeitet.

Inhaltsverzeichnis

Abbildungsverzeichnis

Abkürzungsverzeichnis

JVM	Java Virtual Machine
yacc	yet another compiler compiler
JavaCC	Java Compiler Compiler
UTF-8	8-Bit Universal Character Set Transformation Format

Literaturverzeichnis

[1] Martin, Robert C. (2008): Clean Code: A Handbook of Agile Software Craftsmanship. Prentice Hall International

[2] Freeman, Eric (2007): Entwurfsmuster von Kopf bis Fuß. O'REILLY

[3] Pilone, Dan (2008): Software-Entwicklung von Kopf bis Fuß. O'REILLY

1 Allgemeines

1.1 Einleitung

Ziel dieses Projektes ist es, das Frontend für einen Agilent-Compiler zu schreiben. Dabei soll das Frontend Agilent-Logdateien einlesen und sie in ein Intermediate-Format umwandeln.

1.2 Aufgabenstellung

In diesem Projekt sollen gegebene Agilent-Logdateien geparst werden. Aus den geparsten Zeilen soll ein Baum in einem Intermediate-Format erstellt werden. Der erzeugte Baum wird im Hauptspeicher des Rechners gehalten. Die Weiterverarbeitung des Baumes im Intermediate-Format erfolgt in einem anderen Projekt und ist nicht Gegenstand dieser Arbeit. In diesem Projekt muss das Intermediate-Format nicht entworfen werden. Statt dessen wird es von Felix Deutschmann und Daniel Horbach übernommen. Ein Baum im Intermediate-Format wird immer nur aus einer Agilent-Logdatei erzeugt, d.h. es soll nicht ein Baum aus zwei oder mehreren Agilent-Logdateien erzeugt werden.

Das Frontend soll nur Knotennamen berücksichtigen, die in den zur Verfügung stehenden Agilent-Logdateien auch vorkommen. Andere Knotennamen brauchen im Frontend nicht implementiert werden. Treten bei der Verarbeitung einer Agilent-Logdatei einmal unerwartete Knotennamen auf, werden diese in der Datei UnsupportedNodes.txt weggeschrieben.

Die Reihenfolge der Kindknoten spielt bei der Erstellung des Baumes keine Rolle.

1.3 Relevanz des Forschungsgegenstandes

Der Forschungsgegenstand dieser Arbeit ist, ein Compiler-Frontend für Agilent-Logdateien zu erstellen. Der Forschungsgegenstand ist relevant, da bisher kein Compiler-Frontend für das Umwandeln von Agilent-Logdateien in das Intermediate-Format vorliegt. Darüberhinaus müssen für das Erstellen des Frontends technische Probleme gelöst werden. Ziel der Forschung ist es, einen Frontend zu entwerfen, dass das Agilent-Logformat in ein Intermediate-Format überführt.

1.4 Der aktuelle Wissensstand

Noch nicht vorhandene Kenntnisse über das Parsen von Agilent-Logdateien werden aus der Format6.pdf gewonnen. In der Format6.pdf wird das Agilent-Logformat beschrieben.

1.5 Hintergrund

Ein Compiler hat im Allgemeinen zwei Phasen: das Frontemd und das Backend. In diesem Projekt soll ein Frontend für einen Compiler erstellt werden. In diesem Frontend wird ein sogenannter Logdatei-Parser eingesetzt. Er liest eine Eingabe im Agilent-Logformat und erstellt daraus ein Syntax-Baum in einem gegebenen Zwischenformat. In einem Folgeprojekt wird ein Compiler-Backend erstellt, dass den vom Frontend erzeugten Zwischencode nimmt und daraus die gewünschte Ausgabe in einem Zielformat schreibt.

Da das Agilent-Logformat nur halbformal in einer pdf-Datei beschrieben ist und es keine formale Grammatik für das Format gibt, können keine Parsergeneratoren

wie JavaCC oder yacc eingesetzt werden, sondern es wird hier für das Agilent-Logformat ein Parser von Hand geschrieben.

2 Umsetzung

2.1 Analyse

In Abbildung 2.1 auf Seite 4 ist das Analyseklassendiagramm vom Agilent-Parser zu sehen.

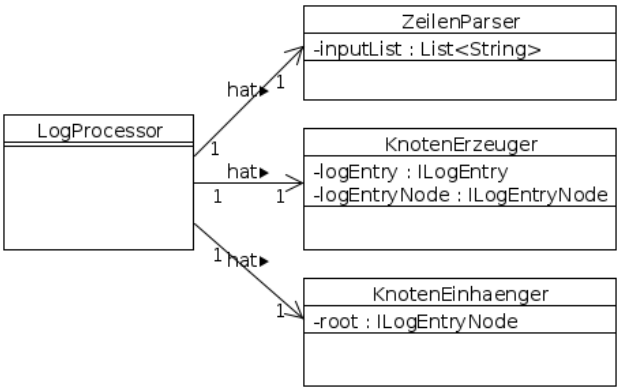

Abbildung 2.1: Analyseklassendiagramm Agilent-Parser

2.2 Funktionale Anforderungen

Der Parser soll eine Agilent-Logdatei einlesen und sie in ein gegebenes Intermediate-Format umwandeln. Dabei sollen nur die Agilent-Knotennamen berücksichtigt werden, die auch in den zur verfügung gestellten Agilent-Logdateien vorkommen. Die zur Verfügung gestellten Logdateien sind auf der Projekt-CD zu finden: 2012.zip,

20120831.zip, 20120827.zip und 040107105943solmb3t11.dat.

In diesen Logdateien kommen 21 Knoten vor:

@BATCH, @BTEST, @BLOCK, @A-JUM, @LIM2, @PF, @A-MEA, @A-RES, @A-CAP, @A-DIO, @A-FUS, @A-IND, @A-ZEN, @LIM3, @M-CL, @TJET, @TS, @D-T, @BS-CON, @BS-S, @BS-O

Diese Knoten sollen beim Parsen von Agilent-Logdateien berücksichtigt werden. Werden bei Logdateien bisher unbekannte Knoten entdeckt, werden die unbekannten Knotennamen in die Logdatei UnsupportedNodes.txt geschrieben.

2.3 Nichtfunktionale Anforderungen

Die Anwendung wird test-driven entwickelt. Für jede nach außen sichtbare Funktionalität einer Klasse werden Unit-Tests geschrieben. Es werden Funktionstests geschrieben, die den Parser funktionsübergreifend testen. Daneben dokumentieren funktionsübergreifende Tests, wie der Parser verwendet werden sollte - d.h. welche Methoden aufgerufen werden können und was diese Methoden zurückgeben. Die Anwendung soll außerdem wartungsfreundlich und damit leicht änderbar und anpassbar sein, wenn z.B. neue Knotennamen hinzugefügt werden. Es wird auf sprechende Variablennamen und Methodennamen geachtet. Es wird darauf geachtet, dass jede Methode nur eine Funktionalität implementiert und somit kurz und übersichtlich bleibt. Es wird Dependency Injection eingesetzt, um die Objektinstanziierung von der Programmlogik zu trennen. Es wird Maven eingesetzt, um die Anwendung automatisiert erstellen und testen zu können.

2.4 Entwurf

In Abbildung 2.2 auf Seite 6 ist das Entwurfsklassendiagramm vom Agilent-Parser zu sehen.

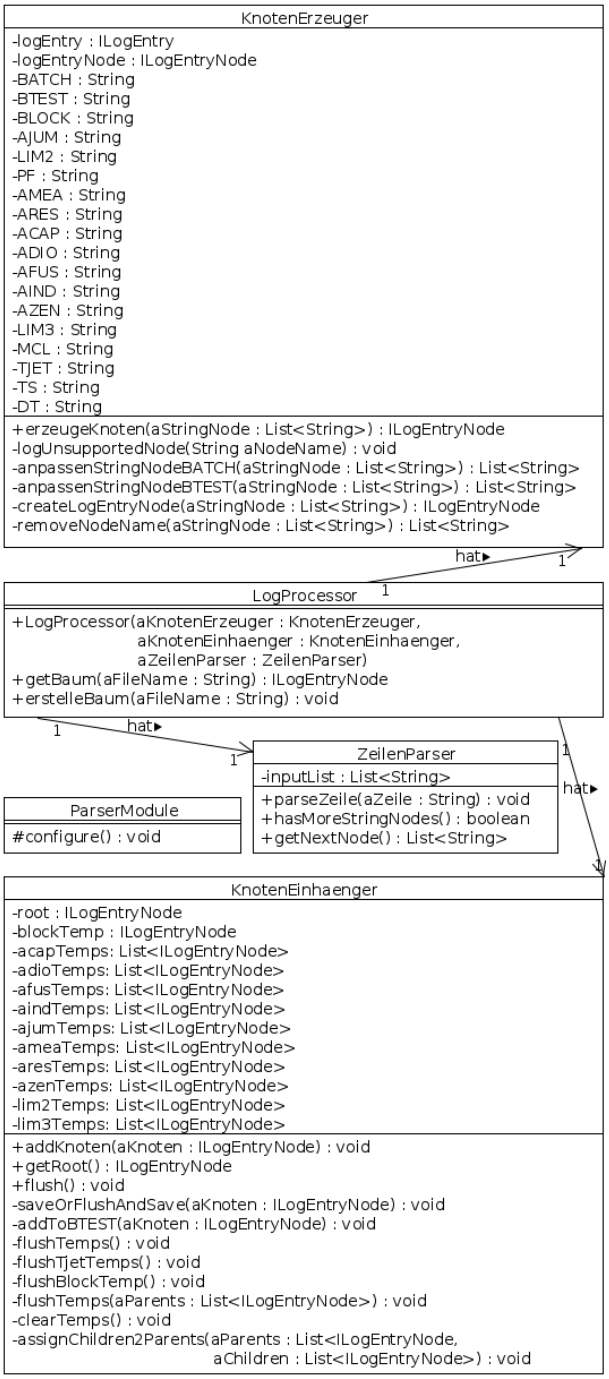

Abbildung 2.2: Entwurfsklassendiagramm Agilent-Parser

2.5 Text File Encoding

Unter Ubuntu 10.04 ist in Eclipse Juno das Text File Encoding standartmäßig auf UTF-8 gesetzt. Die kann beim Kopieren von Stings aus dem Agilent-Logformat zu Fehlern führen, da bestimmte UTF-8 Zeichen im Java-Editor unsichtbar sind. Um unsichtbare Zeichen im Java-Editor sichtbar zu machen, sollte unter Eclipse → Preference → General → Workspace das "Text File Encoding" von UTF-8 auf ISO-8859-1 umgestellt werden.

2.6 Das Agilent Format

2.6.1 Struktur und Syntax

Testdaten im Agilent-Format werden in einer Datei als eine Folge von s.g. log-records gespeichert. Jedes log-record ist durch geschweifte Klammern umschlossen und beginnt mit einen Präfix, welches aus einem @-Zeichen besteht, gefolgt von beschreibenden Zeichen, die den Typ des Records eindeutig identifizieren. Darauf folgt eine bestimmte Anzahl von Datenfeldern, die alphanumerische Messwerte im ASCII-Format enthalten und voneinander durch das |-Zeichen(Pipe) getrennt sind. Sollten einem Datenfeld keine Messwerte zugeordnet worden sein, wird dies vom Testsystem durch ein Leerzeichen zwischen den Pipes dargestellt. Die log-records sind hierarchisch angeordnet in records und subrecords. Die subrecords dienen dazu das vorhergehende record genauer zu beschreiben. Dabei ist jedem subrecord höchstens ein record unmittelbar übergeordnet und somit kann man die Log-Datei als einen Baum darstellen um zwischen den einzelnen Knoten zu navigieren und sie auszulesen.
Das Wurzelelement jeder der uns zur Verfügung stehenden Log-Datei ist das @BATCH-Record, welches erst vollständig durch das @BTEST-Record beschrieben wird und dieses als subrecord mit geschweiften Klammern umschließt. Jede Log-Datei enthält jeweils ein @BATCH-Record und ein @BTEST-Record. Dies ist die übliche Struktur und sie wurde in allen der uns zur Verfügung stehenden Log-Dateien

befolgt. Je nach Testsystem könnte es jedoch evtl. zu Abweichungen von dieser Struktur kommen. Es könnten beispielsweise mehrere @BATCH-Records in einer Log-Datei gespeichert werden. Das @BTEST-Record kann wiederum mehrere subrecords enthalten(Siehe Agilent Data Format), die auch mehrfach in einer Log-Datei vorkommen können. Ein Beispiel eines häufig vorkommenden subrecords von @BTEST wäre @BLOCK.

2.6.2 Besonderheiten und Bemerkungen

In der Dokumentation des Agilent Data Formats ist @BS-CON als Kind des @BTEST-Records als auch des @BLOCK-Records aufgeführt und taucht in den Log-Dateien an beiden in Frage kommenden Stellen auf, was den Aufwand bei der Implementierung des Parsers deutlich erhöht. Man muss herausfinden ob @BS-CON in der Hierarchie auf derselben Höhe steht wie @BLOCK oder ein Kind von @BLOCK ist.
Des Weiteren stimmt die Anzahl der Attribute für das @BATCH-Record und für das @BTEST-Record in der Dokumentation nicht mit der Legende und auch nicht mit den tatsächlichen Log-Daten überein. In der Dokumentation hat @BATCH 13 Attribute, in der Legende und in den Log-Daten jeweils 14 Attribute. Das @BTEST-Record hat 13 Attribute in der Dokumentation und in der Legende, aber in den Log-Daten sind es nur 12.

2.7 Entwicklungsumgebung

JavaSE-1.6, Maven 3.0.4, Eclipse Java EE IDE for Web Developers (Juno) Service Release 1, Ubuntu 12.04, JUnit 4.8.1, Google Guice 3.0, AgilentLogCompiler, Git 1.7.9.5

2.8 Dependency Injection mit Google Guice

Um die Abhängigkeiten zwischen den Objekten zu minimieren, wird das Entwurfsmuster Dependency Injection verwendet. Die Abhängigkeiten werden in diesem Projekt von dem Dependency Injection Framework Google Guice bereitgestellt. Da mit Hilfe von Dependency Injection die Abhängigkeiten eines Objektes reduziert werden, ist es leichter unit-testbar.

Ohne Dependency Injection hätte der LogProcessorTest drei zusätzliche Abhängigkeiten: Knotenerzeuger, KnotenEinhaenger und ZeilenParser:

LogProcessor lp = new LogProcessor(new KnotenErzeuger(), new KnotenEinhaenger(), new ZeilenParser());

Mit Dependency Injection entfallen diese zusätzlichen Abhängigkeiten:

LogProcessor lp = injector.getInstance(LogProcessor.class);

2.9 Projekt importieren

Das Projekt liegt im Repository unter folgender URL:

https://github.com/rinkdotrink/agilentParser.github

Der Projektquelltext kann über git mit folgender Befehlssequenz heruntergeladen werden:

- git init
- git remote add origin https://github.com/rinkdotrink/agilentParser.github
- git pull origin master

Um den heruntergeladenen Quelltext in Eclipse zu importieren, kann in Eclipse File → Import → Maven → Existing Maven Projects gewählt werden.

2.10 Projekt ausführen

Das Eclipse-Projekt ist nicht als eigenständiges Programm auszufühen, sondern es ist vielmehr als Bibliothek zu verstehen. Wie es verwendet wird, geht am Besten aus den funktionalen Tests hervor.

2.11 Funktionale Tests

Die funktionalen Tests zeigen u.a. wie der Parser verwendet wird. Ein Compiler-Backend erzeugt sich ein LogProcessor-Objekt und ruft darauf die Methode getBaum() auf. Der Methode getBaum() wird der Pfad zu der zu verarbeitenden Agilent-Logdatei übergeben. Als Ergebnis liefert getBaum() die Wurzel des Baumes im gewünschten Intermediate-Format:

ILogEntryNode root = logProcessor.getBaum("src/test/resources/Snapshot1");

Für die funktionalen Tests wurden exemplarisch fünf Agilent-Logdateien angelegt. In den Tests werden diese fünf Agilent-Logdateien verarbeitet. Dabei wird sichergestellt, dass aus ihnen der Baum im Intermediate-Format korrekt erstellt wird.

2.12 Unit Tests

2.12.1 ZeilenParserTest

Im ZeilenparserTest werden die Methode *parseZeile()*, *hasMoreStringNodes()* sowie *getNextNode()* getestet. Zuerst wird der Methode parseZeile() eine Zeile der Log-Datei als String übergeben z.B.:

```
{@BATCH|501-6338-02|50|12|1||btest|040107103921||solmb3t1|||||
```

Daraufhin werden alle geschweiften Klammern aufgelöst und die einzelnen Parameter in einer List<String> gespeichert. Die Methode hasMoreStringNodes() liefert true, falls ein Listenelement den Anfang eines Records darstellt und mit @-Zeichen beginnt. Wenn eine Zeile mehrere Records enthält, wird mit getNextNode() immer das komplette nächste Record als Liste zurückgegeben. Diese Funktionalität wird beispielsweise in folgendem Test geprüft:

```
@Test
public void testGetNextSecond() {
String inpu = "{@A-JUM|0|+7.630803E+06{@LIM2|+9.999999E+99|+1.000000E+04}}";
zeilenParser.parseZeile(inpu);
zeilenParser.getNextNode();
ArrayList<String> stringNode = new ArrayList<String>();
stringNode.add("@LIM2");
stringNode.add("+9.999999E+99");
stringNode.add("+1.000000E+04");
assertTrue(zeilenParser.getNextNode().equals(stringNode));
}
```

2.12.2 KnotenErzeugerTest

Im KnotenErzeugerTest wird die Methode *erzeugeKnoten()* getestet. Ihr wird eine Liste mit den Attributen eines Records übergeben, die mit Hilfe der *NodeCreatorUtil*-Klasse erzeugt wurde. Daraufhin prüft die Methode anhand des ersten Listenelements um welches Record es sich handelt und erzeugt eine Instanz der entsprechenden Record-Klasse, die eine Implementierung der *ILogEntry*-Schnittstelle ist. Jede Record-Instanz enthält die beiden Listen **headings** und **values**. Headings sind die Namen der Record-Attribute und werden im Konstruktor gesetzt. Anschließend ruft die *erzeugeKnoten()*-Methode createLogEntryNode() auf, in der die values gesetzt werden und eine Instanz der *LogEntryNode-Klasse* erzeugt wird. Im Unit-Test wird geprüft, ob der richtige Knoten erfolgreich erzeugt wurde:

```
@Test
```

```
public void testErzeugeKnotenBATCH() {
ILogEntryNode logEntryNode = testNodeCreator.createNodeBATCH();
assertEquals(BATCH.class, logEntryNode.getLogEntry().getClass());
}
```

2.12.3 KnotenEinhaengenTest

Hierbei wird die Methode *addKnoten()* getestet. Nachdem ein Knoten erzeugt wurde, wird er ihr als Parameter übergeben. Diese prüft zuerst um welchen Knoten-Typ es sich handelt. Wenn es ein BATCH-Knoten ist, wird er zum Wurzelknoten(root) des Baumes. Der BTEST-Knoten wird mit *root.getSubNodes().add(knoten)* an den Wurzelknoten eingehängt. Folgt anschließend ein BLOCK-Knoten, werden zuerst an alle seine Unterknoten deren Kinder eingehängt und zum Schluss wird der BLOCK-Knoten in *flushBlockTemp()* an den BTEST-Knoten angefügt. Im KnotenEinhaengenTest werden zuerst der Reihe nach die gewählten Knoten an das Wurzelelement mit addKnoten() eingehängt um anschließend ein bestimmtes Element herauszulesen:

```
@Test
public void testAddKnotenBATCH_BTEST_BLOCK_AJUM() {
knotenEinhaenger.addKnoten(testNodeCreator.createNodeBATCH());
knotenEinhaenger.addKnoten(testNodeCreator.createNodeBTEST());
knotenEinhaenger.addKnoten(testNodeCreator.createNodeBLOCK());
knotenEinhaenger.addKnoten(testNodeCreator.createNodeAJUM());
knotenEinhaenger.flush();
ILogEntryNode node = knotenEinhaenger.getRoot();
ILogEntryNode batch = node;
ILogEntryNode btest = batch.getSubNodes().get(0);
ILogEntryNode block0 = btest.getSubNodes().get(0);
ILogEntryNode block0_ajum = block0.getSubNodes().get(0);
assertEquals(AJUM.class, block0_ajum.getLogEntry().getClass());
}
```

2.13 Zusammenfassung

Der Parser kann Agilent-Logdateien einlesen und sie in ein Intermediate-Format umwandeln. Bisher unbekannte Knoten werden in eine Logdatei geschrieben. Die Funktionalität der Klassen wurde über Unittests sichergestellt. Das Zusammenspiel der Klassen untereinander wurde durch Funktionstests geprüft. Bis auf das Problem mit BSCON, dass unter 2.6.2 auf Seite 8 beschrieben wurde, konnten alle Anforderungen im Agilent-Parser umgesetzt werden.

A Arbeitsaufteilung

Arbeit	**C. Ochmann**	**I. Körner**
Abstract		0
Einleitung		1.1
Aufgabenstellung		1.2
Forschungsgegenstand		1.3
akt. Wissensstand		1.4
Hintergrund		1.5
Analyse		2.1
Funktionale Anforderungen		2.2
Nichtfunktionale Anforderungen		2.3
Entwurf		2.4
Entwicklungsumgebung		2.7
Dependency Injection mit Goolge Guice		2.8
Projekt importieren		2.9
Projekt ausführen		2.10
Funktionale Tests		2.11
Zusammenfassung		2.13

Tabelle A.1: Aufteilung

B Eigenständigkeitserklärung

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe. Mir ist bekannt, dass jede Form des Plagiats mit der Note 5 (Betrugsversuch) bewertet wird.

Ochmann, Christof Unterschrift:

Körner, Ingo Unterschrift: