# 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...

# Inhaltsverzeichnis

# Abbildungsverzeichnis

# Abkürzungsverzeichnis

| | |
|---|---|
| **JVM** | Java Virtual Machine |

# 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] `http://www.cs.waikato.ac.nz/ml/weka/arff.html (08.06.2012)`

# 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. $Agilent - Logdatei - > Agilent - Parser - > Baum im Intermediate - Format$.

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 UnsupportedNodeNames.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.

# 2 Umsetzung

## 2.1 Analyse

Ziel dieses Projektes ist es, ...

In Abbildung 2.1 auf Seite 3 ist das Analyseklassendiagramm vom Agilent-Parser zu sehen.

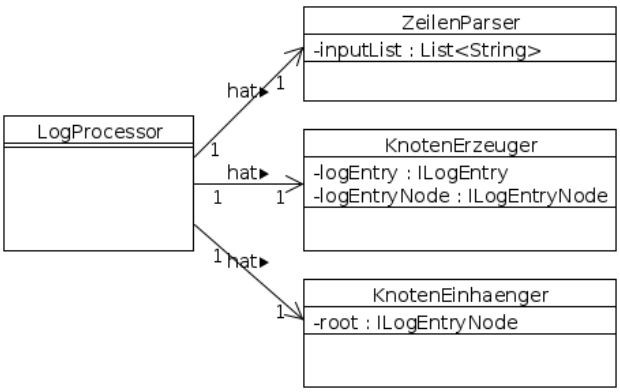


Abbildung 2.1: Analyseklassendiagramm Agilent-Parser

## 2.2 Entwurf

Ziel dieses Projektes ist es, ...

In Abbildung 2.2 auf Seite 4 ist das Entwurfsklassendiagramm vom Agilent-Parser zu sehen.

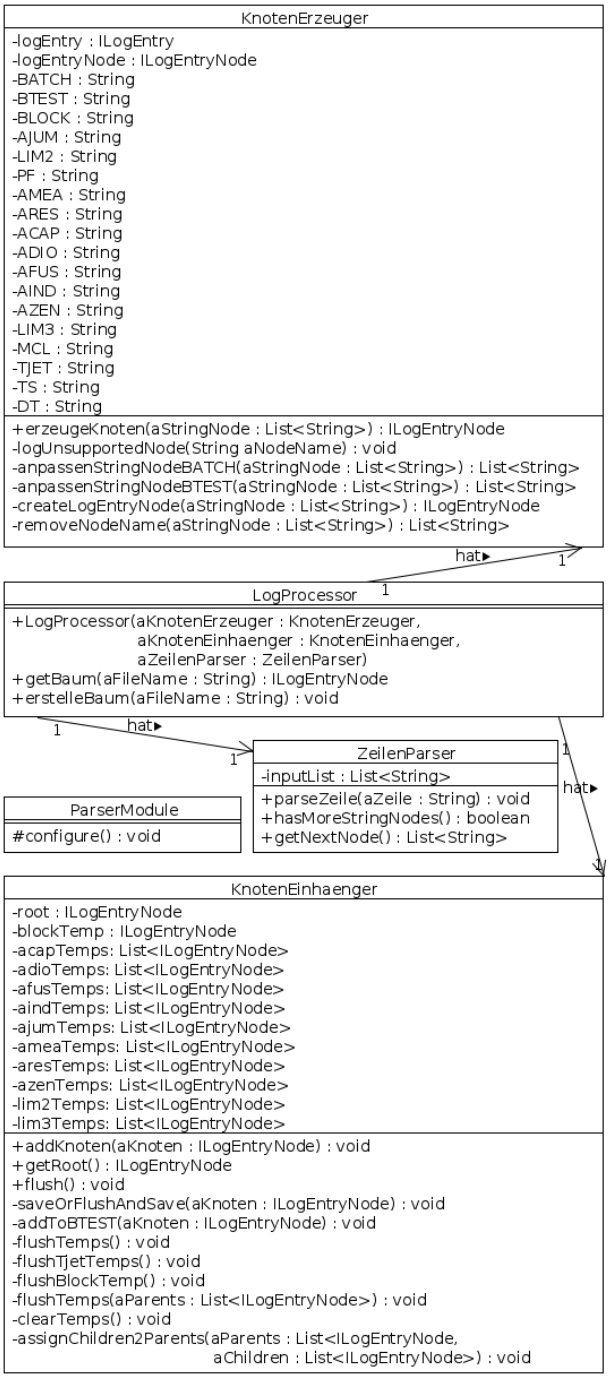


Abbildung 2.2: Entwurfsklassendiagramm Agilent-Parser

## 2.3 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.

## 2.4 Das Agilent Format

### 2.4.1 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.

### 2.4.2 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.

## 2.5 Unstimmigkeiten im Agilent-Format

In der Format06.pdf auf Seite 35 stimmt bei BATCH die Anzahl der Attribute nicht mit der Legende überein. Auch in den tatsächlichen Agilent-Logdateien stimmt die Anzahl der Attribute nicht immer mit der Anzahl der Attribute überein, wie sie in der Format06.pdf beschrieben werden.

In der Format06.pdf auf Seite 40 stimmt bei BTEST die Anzahl der Attribute nicht immer mit der tatsächlich vorkommenden Anzahl in der Agilent-Logdatei überein.

## 2.6 Unit Tests

### 2.6.1 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));
}
```

### 2.6.2 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 ent-

sprechenden 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());
}
```

### 2.6.3 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());
}
```

# A Arbeitsaufteilung

| Arbeit | C. Ochmann | I. Körner |
|---|---|---|
| Abstract | | 0 |
| Einleitung | | 1.1 |
| Aufgabenstellung | | 1.2 |
| Forschungsgegenstand | | 1.3 |
| akt. Wissensstand | | 1.4 |
| Zusammenfassung | | ?? |
| Ausblick | | ?? |

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: