

258321 DKE Projekt

**Gruppe 2**

**Teammitglieder:**

k01607605, Aistleithner Andrea

k01256561, Dusanic Maja

k01356577, Teuchtmann Alexander

k01356229, Tomic Milos

MeilenStein Implementierung

Contents

[1. Programmarchitektur 2](#_Toc97107)

[1.1. Komponenten 2](#_Toc97108)

[1.2. Schnittstellen 3](#_Toc97109)

[1.3. Umsetzung im Programm 4](#_Toc97110)

[2. Funktionalitäten einzelner Klassen 4](#_Toc97111)

[2.1. DataGeneratorCBR 5](#_Toc97112)

[2.1.1. BusinessCase 5](#_Toc97113)

[2.1.2. BusinessCaseClass 5](#_Toc97114)

[2.1.3. Context 6](#_Toc97115)

[2.1.4. ContextClass 6](#_Toc97116)

[2.1.5. GeneratorCBR 6](#_Toc97117)

[2.1.6. Module 7](#_Toc97118)

[2.1.7. Parameter 8](#_Toc97119)

[2.1.8. ParameterValue 8](#_Toc97120)

[2.2. DataGeneratorRandomString 8](#_Toc97121)

[2.3. DataGeneratorRMI 9](#_Toc97122)

[2.3.1. GeneratorRuleModelInheritance 9](#_Toc97123)

[2.3.2. NonRelationalAtom 10](#_Toc97124)

[2.3.3. RMIModule 11](#_Toc97125)

[2.3.4. Module 12](#_Toc97126)

[List<String>outputPredicate; 12](#_Toc97127)

[2.3.5. Rule 12](#_Toc97128)

[Weiters enthält diese Klasse die folgende (wichtigere) Methode: 13](#_Toc97129)

[public String generateOnlyRules(int rulesCount, Program pr); 13](#_Toc97130)

[Diese Methode dient zur Meta-Code Darstellung einer Regel. 13](#_Toc97131)

[2.3.6. Program 13](#_Toc97132)

[2.3.7. RelationalAtom 13](#_Toc97133)

[Weiters enthält diese Klasse die folgende (wichtigere) Methode: 14](#_Toc97134)

[public String generateOnlyFacts(int factsNum, Program pr); 14](#_Toc97135)

[Diese Methode dient zur Meta-Code Darstellung eine Atoms. 14](#_Toc97136)

[2.3.8. Annotation 14](#_Toc97137)

[Sie enthält folgende Attribute: 14](#_Toc97138)

[Weiters enthält diese Klasse die folgende (wichtigere) Methode: 15](#_Toc97139)

[public String generateAnnotationsModule (Program pr); 15](#_Toc97140)

[Diese Methode dient zur Meta-Code Darstellung einer Annotation. 15](#_Toc97141)

[2.3.9. Term 15](#_Toc97142)

[2.4. DB 16](#_Toc97143)

[2.5. EvaluationFramework 16](#_Toc97144)

[2.5.1. CBR Code Generierung 16](#_Toc97145)

[2.5.2. RMI Code Generierung 18](#_Toc97146)

[2.6. Models 18](#_Toc97147)

[2.6.1. CBR 18](#_Toc97148)

[2.6.2. RMI 18](#_Toc97149)

[2.7. Test 18](#_Toc97150)

[2.7.1. CBR Tests 18](#_Toc97151)

[2.7.2. RMI Tests 18](#_Toc97152)

[2.8. Vadalog 18](#_Toc97153)

[3. Tests der geforderten Funktionalitäten 18](#_Toc97154)

[4. Abhängigkeiten Bibliotheken 18](#_Toc97155)

[5. Installationsanleitung 18](#_Toc97156)

[6. Limitierungen und Verbesserungsvorschläge im praktischen Einsatz 19](#_Toc97157)

[7. Abbildungsverzeichnis 20](#_Toc97158)

[8. Tabellenverzeichnis 20](#_Toc97159)

# Programmarchitektur

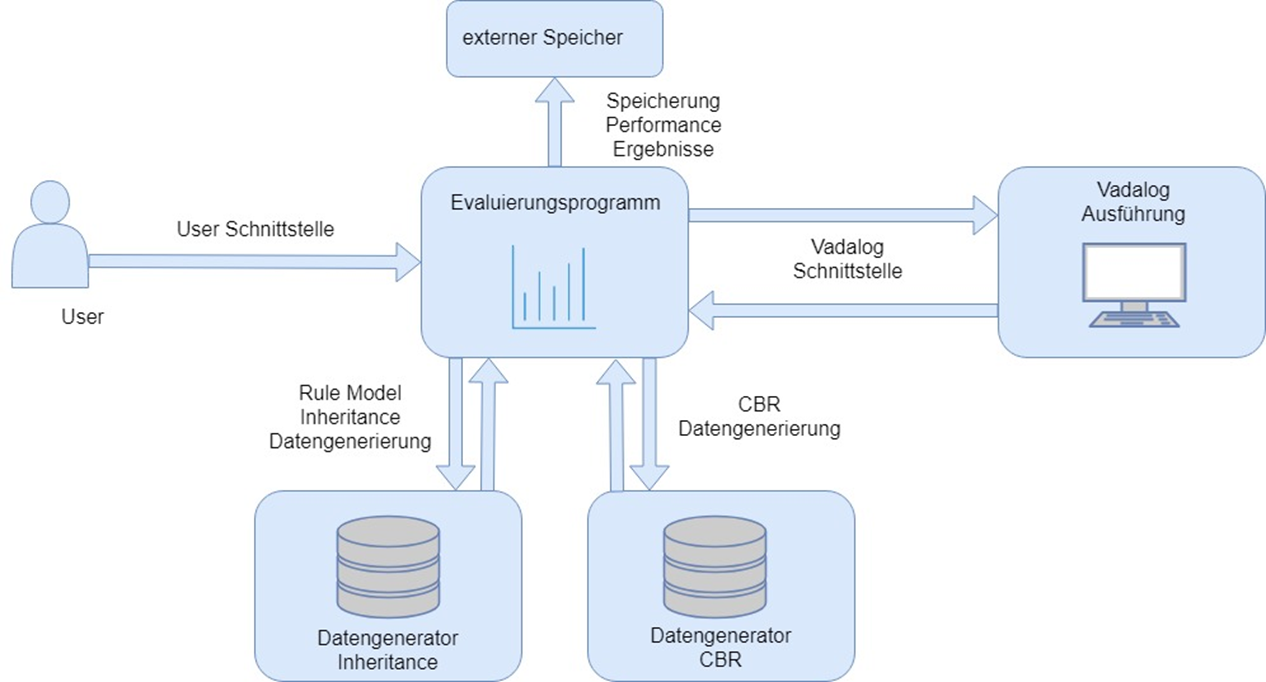


Abbildung 1: Programmarchitektur

In der oben abgebildeten Grafik, Abbildung 1, ist die Architektur des Programmes abgebildet. Diese Grafik wurde schon beim Konzeptuellen Entwurf erstellt und nurmehr leicht abgeändert, die essentiellen Komponenten blieben aber genauso erhalten.

## Komponenten

Das Programm setzt sich aus folgenden Komponenten zusammen:

**Das Evaluierungsprogramm**

Diese Komponente beinhaltet das ausführbare Hauptprogramm und ist der Mittelpunkt des Programmes, worüber die anderen Komponenten miteinander verbunden werden. Der User agiert über die User Schnittstelle mit dem Evaluierungsprogramm und übergibt die Input Werte zur Datengenerierung.

**Der Rule Model Inheritance Datengenerator**

Diese Komponente generiert den Rule Model Inheritance Code mithilfe der Input Werte des Users, welche über das Evaluierungsprogramm an den Datengenerator weitergegeben werden. Die fertig generierten Code Daten werden vom Generator zurück an die Komponente des Evaluierungsprogrammes übergeben.

**Der CBR Datengenerator**

Diese Komponente generiert den CBR Code mithilfe der Input Werte des Users, welche über das Evaluierungsprogramm an den Datengenerator weitergegeben werden. Die fertig generierten Code Daten werden vom Generator zurück an die Komponente des Evaluierungsprogrammes übergeben.

**Die Dummy Vadalog Ausführung**

Nachdem der fertig generierte Test Code vom Generator an das Evaluierungsprogramm zurückgegeben wurde, ruft die Komponente Evaluierungsprogramm die Dummy Vadalog Ausführung auf. Diese Komponente simuliert die Vadalog Ausführung und generiert Zufalls-Evaluierungswerte, die Execution Time, ob Fehler aufgetretenen sind und wie viel CPU Leistung bei der Ausführung verbraucht wurde. Diese Werte werden zurück an das Evaluierungsprogramm gegeben.

Der generierte Test Code wird in ein Text File geschrieben, welches generiert wird, da die Konsole schnell überläuft und lange Codes nicht vollständig ausgeben kann. Ebenso werden die Input Werte und die Evaluierungsergebnisse in dieses Text File geschrieben.

**Der externe Speicher, die Datenbank**

Nachdem die Testdaten generiert und die Ausführung durchgeführt werden, übergibt das Evaluierungsprogramm die zu speichernden Werte an die Datenbank. Hierbei werden die Input Werte, die Ergebnisse der Tests und das Datum und die Uhrzeit der Durchführung gespeichert. Der generierte Code wird nicht in der Datenbank gespeichert, sondern nur ausgegeben.

## Schnittstellen

Folgende Schnittstellen lassen die Komponenten des Programmes miteinander kommunizieren:

**Die User Schnittstelle**

Diese Schnittstelle lässt den User die Input Werte an das Evaluierungsprogramm übergeben.

**Die Rule Model Inheritance Schnittstelle**

Diese Schnittstelle erlaubt es, die Input Werte vom Evaluierungsprogramm an den Rule Model Inheritance Datengenerator zu übergeben und die generierte Test Codes wieder zurück an das Evaluierungsprogramm zu übergeben.

**Die CBR Schnittstelle**

Diese Schnittstelle erlaubt es, die Input Werte vom Evaluierungsprogramm an den CBR Datengenerator zu übergeben und die generierte Test Codes wieder zurück an das Evaluierungsprogramm zu übergeben.

**Die Vadalog Schnittstelle**

Die Vadalog Schnittstelle ermöglicht die Kommunikation zwischen Evaluierungsprogramm und der Dummy Vadalog Ausführung.

**Die Speicherung Schnittstelle**

Über die Speicherung Schnittstelle werden die zu speichernden Daten an die Datenbank übergeben.

## Umsetzung im Programm

Abbildung 2: Package Struktur im Programm

In Abbildung 2 ist die Struktur der Packages zu sehen und wie die geplanten Komponenten im Programm umgesetzt wurden.

Das Package **DataGeneratorCBR** setzt die Funktionalität der Komponente **CBR Generator** um.

Das Package **GeneratorRandomString** enthält die Funktionalität, dass ein zufallsgenerierter String aus zufällig zusammengestetzten Buchstaben generiert wird. Dies wird zur Unterstützung der Datengeneratoren eingesetzt.

Das Package **DataGeneratorRuleModelInheritance** setzt die Komponente **Rule Model Inheritance** **Generator** um.

Das Package **DB** erfüllt die Aufgabe der Datenspeicherung und setzt die Komponente **externer Speicher** um.

Das Package **EvaluationFramework** erfüllt die Aufgabe der Komponente des **Evaluierungs Frameworks.**

Das Package **Exceptions** dient zur Unterstützung, dass bei der Eingabe der Input Werte keine negativen Zahlen eingegeben werden können.

Das Package **Models** dient zur Unterstützung der Datenspeicherung. Darin befinden sich zwei Klassen, die ein CBR Objekt oder ein RMI Objekt erstellen können. Dies erleichtert die Einträge in die Datenbank.

In dem Package **Test** wurden während dem Implementieren die Datengeneratoren getestet.

Das Package **Vadalog** übernimmt die Aufgabe der Komponente **Vadalog Ausführung**.

Das Text File **out.txt** ist die Ausgabe des generierten Codes, der Input Werte und der Evaluierungsergebnisse.

# Funktionalitäten einzelner Klassen

Die einzelnen Packages, die in Abbildung 2 zu sehen sind, und deren dazugehörigen Klassen werden nun in diesem Abschnitt genauer beschrieben. Es werden die Funktionalität und das Zusammenspiel der Klassen beschrieben.

## DataGeneratorCBR

Abbildung 3: Package DataGeneratorCBR

In Abbildung 3 sind die Klassen des Packages **DataGeneratorCBR** zu sehen. Diese Klassen werden im Folgenden genauer hinsichtlich ihrer Funktionalität und ihrem Zusammenspiel beschrieben.

### BusinessCase

Die Klasse **BusinessCase.java** speichert die Informationen über den BusinessCase, welche für die Generierung des CBR Codes notwendig sind.

Die Klasse hat folgende Attribute:

String name;

**private** List<String> descProp = **new** ArrayList<String>();

**private** List<String> parameterValues = **new** ArrayList<String>();

In dieser Klasse gibt es zwei ArrayLists mit dem Typ String. Eine der beiden Listen speichert die describing Properties, welche zu dem Business Case gehören. Die andere Liste speichert die zugehörigen ParameterValues, die zu dem Business Case gehören.

Ansonsten finden sich in der Klasse noch die Get und Set Methoden für die Attribute sowie Methoden um neue Werte zu den Listen hinzuzufügen.

### BusinessCaseClass

Die Klasse **BusinessCaseClass.java** speichert die Informationen über die BusinessCaseClass, welche für die Generierung des CBR Codes notwendig sind.

Die Klasse hat folgende Attribute:

**private** String name;

**private** List<BusinessCase> businessCases = **new** ArrayList<BusinessCase>();

In der Klasse gibt es ein Attribut name, vom Typ String, welches den Namen der BusinessCaseClass speichert.

Ebenso existiert eine Array Liste vom Typ BusinessCase, in welcher die dazugehörigen BusinessCase Objekte zu dem BusinessCaseClass Objekt gespeichert werden.

Ansonsten finden sich in der Klasse noch die Get und Set Methoden für das Attribut name sowie eine Methode um neue Werte zu der Liste hinzuzufügen.

### Context

Die Klasse **Context.java** speichert die Informationen über den Context, welche für die Generierung des CBR Codes notwendig sind.

Die Klasse hat folgende Attribute:

**private** String ctx;

**private** String name;

**private** List<ParameterValue> parameterValues = **new** ArrayList<ParameterValue>();

**private** Module module;

**private** String contextClass;

Die Klasse verfügt über einen String ctx, welcher die ID des Context repräsentiert. Der String name speichert den Namen des Context Objekts.

Ebenso verfügt die Klasse über eine Array Liste vom Typ parameterValues, welche die zugehörigen ParameterValues zu dem Context gespeichert werden.

Zusätzlich verfügt die Klasse über ein Objekt Module, welches das zugehörige Module zu dem Context speichert. Ebenso ist ein Attribut contextClass in der Klasse vom Typ String, welches die zugehörige contextClass des Objektes speichert.

Ansonsten finden sich in der Klasse noch die Get und Set Methoden für die Attribute sowie eine Methode um neue Werte zu der Liste hinzuzufügen.

### ContextClass

Die Klasse **ContextClass.java** speichert die Informationen über die ContextClass, welche für die Generierung des CBR Codes notwendig sind.

Die Klasse hat folgende Attribute:

**private** String name;

**private** List<Parameter> parameters = **new** ArrayList<Parameter>();

**private** List<Context> contexts = **new** ArrayList<Context>();

Die Klasse verfügt über ein Attribut name vom Typ String, welches den Namen des ContextClass Objektes speichern soll.

Ebenso verfügt die Klasse über zwei Array Listen. Die eine Liste ist vom Typ Parameter und speichert alle zugehörigen Parameter zu dem ContextClass Objekt.

Die zweite Liste ist vom Typ Context und speichert alle zugehörigen Context Objekte zu dem jeweiligen ContextClass Objekt.

Ansonsten finden sich in der Klasse noch die Get und Set Methoden für die Attribute sowie Methoden um neue Werte zu den Listen hinzuzufügen.

### GeneratorCBR

Die Klasse **GeneratorCBR.java** ist die wichtigste Klasse aus diesem Package. Sie verfügt über die Funktionalität, welche den CBR Code generiert.

Die Klasse verfügt über folgende Attribute:

**private** **static** String *CBRCode*;

**private** **static** ContextClass *cc*;

**private** **static** BusinessCaseClass *bcc*;

Das Attribut CBRCode, welches vom Typ String ist, ist die Variable, in welche der genertierte Code dann gespeichert wird.

Das Attribut cc, vom Typ ContextClass, speichert das ContextClass Objekt, welches für die Generierung des CBR Codes benötigt wird.

Das Attribut bcc, vom Typ BusinessCaseClass, speichert das BusinessCaseClass Objekt, welches für die Generierugn des CBR Codes benötigt wird.

Die Klasse beinhaltet eine Hauptmethode zur Generierung des Codes. Diese Hauptmethode wird aus der ausführbaren Hauptklasse des Programms, EvaluationFrameworkApp.java, aufgerufen, um den CBR Code mit den Input Werten zu generieren.

Diese Hauptmethode sieht folgendermaßen aus:

**public** **static** String generateCBRCode(**int** parameters, **int** paramValues, **int** businessCases) {

*CBRCode* = "";

*CBRCode* += *generateContextClass*();

*CBRCode* += *generateBusinessCaseClass*();

*CBRCode* += *generateParameters*(parameters);

*CBRCode* += *generateParameterValues*(paramValues);

*CBRCode* += *generateParameterValuesHierarchies*();

*CBRCode* += *generateContexts*(paramValues);

*CBRCode* += *generateDetermineParameterValues*();

*CBRCode* += *generateBusinessCases*(businessCases);

*CBRCode* += *generateStaticCode*();

**return** *CBRCode*;

}

Die Hauptmethode des CBR Datengenerators besteht aus Hilfsmethoden, welche den jeweiligen Teil des Codes generieren. Die Aufteilung der Hilfsmethoden weicht von der geplanten Umsetzung des konzeptuellen Entwurfs leicht ab, da während der Implementierung erkannt wurde, dass eine Abänderung sinnvoll und hilfreich ist.

Die Hilfsmethoden generieren den entsprechenden Code Teil und erzeugen dabei die Objekte, welche in den anderen Klassen dieses Packages beschrieben sind. Diese Objekte werden benötigt um den CBR Code generieren zu können.

Die Hauptmethode übergibt einen String, in welchem der gesamte CBR Code gespeichert ist.

Da die Klasse EvaluationFrameworkApp.java diese Methode aufruft, befindet sich der generierte CBR Code dann in dieser Klasse.

### Module

Die Klasse **Module.java** speichert die Informationen über das Module, welche für die Generierung des CBR Codes notwendig sind.

Die Klasse hat folgendes Attribut:

**private** String name;

Das Attribut name vom Typ String beschreibt den Namen des Module Objekts.

Ansonsten finden sich in der Klasse noch die Get und Set Methoden für das Attribut.

### Parameter

Die Klasse **Parameter.java** speichert die Informationen über das Module, welche für die Generierung des CBR Codes notwendig sind.

Die Klasse hat folgende Attribute:

**private** String name;

**private** List<ParameterValue> parameterValues = **new** ArrayList<ParameterValue>();

**private** String descProp;

In der Klasse befindet sich an Attribut name, vom Typ String, welches den Namen des Parameter Objekt speichert.

Ebenso befindet sich eine Array List vom Typ ParameterValue, welche die zugehörigen parameterValues zu dem Parameter Objekt speichert.

Zusätlich gibt es noch ein Attribut descProp, vom Typ String, welches die describing Property des Parameter Objekt speichert.

Ansonsten finden sich in der Klasse noch die Get und Set Methoden für die Attribute sowie eine Methode um neue Werte zu der Liste hinzuzufügen.

### ParameterValue

Die Klasse **ParameterValue.java** speichert die Informationen über das Module, welche für die Generierung des CBR Codes notwendig sind.

Die Klasse hat folgendes Attribut:

**private** String name;

Das Attribut name vom Typ String beschreibt den Namen des ParameterValue Objekts.

Ansonsten finden sich in der Klasse noch die Get und Set Methoden für das Attribut.

## DataGeneratorRandomString



Abbildung 4: Package DataGeneratorRandomString

In dem Package DataGeneratorRandomString befindet sich nur eine Klasse, die **GeneratroRandomString.java**. Diese Klasse dient zur Unterstützung der CBR und RMI Datengeneratoren, dies ist in Abbildung 4 zu sehen.

In dieser Klasse befinden sich 2 Methoden, welche zufällig generierte Strings erzeugen.

Folgende 2 Methoden befinden sich in der Klasse:

**public** **static** String getRandomString(**int** length){…}

**public** **static** String getRandomBigChar(**int** length){…}

Die Methode getRandomString erzeugt einen String aus zufällig aneinander gereihten Kleinbuchstaben. Die Länge des zu generierenden Strings wird als Parameter der Methode mitgegeben.

Die Methode getRandomBigChar erzeugt einen String aus zufällig aneinander gereihten Großbuchstaben. Die Länge des zu generierenden Strings wird als Parameter der Methode mitgegeben.

Die beiden Methoden werden von den Datengeneratoren aufgerufen, um diese bei der Erzeugung der CBR und RMI Codes zu unterstützen.

## DataGeneratorRMI

Abbildung 5: Package DataGeneratorRuleModellInheritance (RMI)

In der Abbildung 4 sind die Klassen des Packages **DataGeneratorRMI** zu sehen. Diese Klassen werden im Folgenden genauer hinsichtlich ihrer Funktionalität und ihrem Zusammenspiel beschrieben.

### GeneratorRuleModelInheritance

Die Klasse **GeneratorRuleModellInheritance.java** wurde zum Testen vom generierten Meta-Code im Yupiter benutzt.

Die Klasse hat folgende Attribute:

**private** **static** Program *program* = **new** Program();

**private** **static** List<Program> *programs*;

**private** **static** List <String> *modules* = **new** ArrayList<>();

**public** **static** Map<Integer, String> *moduleFacts* = **new** HashMap<>();

**static** **int** *countModule* = 0;

Das Attribut program, welches vom Typ Program ist, ist die Refferenz auf ein neues Programm.

Das Attribut programs, welches eine Liste vom Typ Programm ist, speicher die neuen Programm Objekte.

Das Attribut modules, welches eine Array-Liste vom Typ String ist, speichert die generierten Module.

Das Attribut moduleFacts, welches eine HashMap-Liste vom Typ Integer-String ist, ordnet die Fakten einem Modul ein.

Das Attribut countModule, welches vom Typ int ist, zählt die generierten Module.

Weiters enthält die Klasse **GeneratorRuleModellInheritance.java** verschiedene Methoden, die einen Meta-Code ohne Logik generieren. Dementsprechend wurde damit nur die Syntax im Yupiter getestet. Es wurde die Syntax von Regeln, (nicht) relationalen Atomen, Annotationen und Terms getestet. Rund um diese Klasse wurden im Nachhinein andere Klassen erstellt, die für den logisch basierten Meta-Code zuständig sind. Die neuen Klassen haben den Code einiger Methoden kopiert und für den eigenen Gebrauch entsprechend adaptiert.

Ansonsten finden sich in der Klasse noch die Get und Set Methoden für die Attribute sowie Methoden um neue Werte zu den Listen hinzuzufügen.

### NonRelationalAtom

Die Klasse **NonRelationalAtom.java** wurde zum Testen vom generierten Meta-Code im Yupiter benutzt.

Die Klasse hat folgendes Attribut:

**private** String name;

Das Attribut name, welches vom Typ String ist, wurde für die Speicherung des Namens benutzt.

Diese Klasse wurde in Kombination mit GeneratorRuleModelInheritance benutzt, um einen Meta-Code ohne Logik zu generieren.

Ansonsten finden sich in der Klasse noch die Get und Set Methoden für die Attribute.

### RMIModule

Die Klasse **RMIModule.java** ist die zentrale Klasse des Programms. Sie vereint alle anderen Klassen, die einen Teil des Moduls darstellen, und generiert dadurch einen logisch basierten Meta-Code.

Sie enthält folgende Attribute:

**static String var;**

Module m1;

Das Attribut var, welches vom Typ String ist, wird für die Speicherung einer Variable benutzt.

Das Attribut m1, welches vom Typ Module ist, wird für die Speicherung eines Moduls benutzt.

Sie enthält u.a. folgende zentrale Methoden:

public String generateRMIModule(int rules, int facts, int in, int out);

Diese Methode generiert den Meta-Code vom RMI Modul basierend auf der eingegebenen Anzahl der Regeln, Fakten, Input- und Outputparameter. Das RMI Modul wird so erstellt, dass zuerst ein neues Objekt der Klasse Modul erstellt wird. Dieses wird anschließend mit Objekten der Anderen Klassen (Program, RelationalAtom, Annotation, Term) befüllt. Die angesprochenen Objekte werden mithilfe von verschiedenen Hilfsmethoden zur Verfügung gestellt.

public String generateRMIModule(int rules, int facts, int in, int out, Module myModule)

Diese Methode generiert den Meta-Code eines geerbten RMI Moduls basierend auf der eingegebenen Anzahl der Regeln, Fakten, Input- und Outputparameter und der Superklasse. Das RMI Modul wird so erstellt, dass zuerst ein neues Objekt der Klasse Modul erstellt wird. Dieses wird anschließend mit Objekten der Anderen Klassen (Program, RelationalAtom, Annotation, Term) befüllt. Die angesprochenen Objekte werden mithilfe von verschiedenen Hilfsmethoden zur Verfügung gestellt. Im Unterschied zur oberen Methode wird hier noch zusätzlich die Verbindung zwischen der Superklasse und der Subklasse generiert.

Die Restlichen größeren Methoden in der Klasse **RMIModule.java** haben ähnliche Algorithmen deren Bedeutung aus dem Code herausgelesen werden kann.

Ansonsten finden sich in der Klasse noch die Get und Set Methoden für die Attribute.

### Module

Die Klasse **Module.java** dient als Basis für das RMI Modul.

Diese Klasse enthält folgende Attribute:

**private** String name;

**private** **static** **int** *nrOfModule* = 1;

List<Rule> rules;

List<RelationalAtoms> facts;

List<String> inputPredicate;

### List<String>outputPredicate;

Das Attribut name, welches vom Typ String ist, wird für die Speicherung des Namens benutzt.

Das Attribut nrOfModule, welches vom Type int ist, wird für die Zählung von den erzeugten Modulen benutzt.

Das Attribut rules, welches eine Liste vom Typ Rule ist, wird für die Speicherung der Regeln benutzt.

Das Attribut facts, welches eine Liste vom Typ RelationalAtom ist, wird für die Speicherung der Fakten benutzt.

Das Attribut inputPredicate, welches eine Liste vom Typ String ist, wird für die Speicherung der Input-Prädikate benutzt.

Das Attribut outputPredicate, welches eine Liste vom Type String ist, wird für die Speicherung der Output-Prädikate benutzt.

Ansonsten finden sich in der Klasse noch die Get und Set Methoden für die Attribute sowie Methoden um neue Werte zu den Listen hinzuzufügen.

### Rule

Die Klasse **Rule.java** dient zur Erstellung von Regeln eines Moduls.

Sie enthält folgende Attribute:

**private** String name;

**private** RelationalAtoms head;

**private** List<RelationalAtoms> relationalAtomsBody;

**private** String annotation;

Das Attribut name, welches vom Typ String ist, wird für die Speicherung des Namens der Regel benutzt.

Das Attribut head, welches vom Typ RelationalAtom ist, wird für die Speicherung des Head-Atoms benutzt.

Das Attribut relationalAtomsBody, welches eine Liste vom Typ RelationalAtom ist, wird für die Speicherung der Body-Atome benutzt.

Das Attribut annotation, welches vom Typ String ist, wird für die Speicherung der Annotation einer Regel benutzt.

### Weiters enthält diese Klasse die folgende (wichtigere) Methode:

### public String generateOnlyRules(int rulesCount, Program pr);

### Diese Methode dient zur Meta-Code Darstellung einer Regel.

Ansonsten finden sich in der Klasse noch die Get und Set Methoden für die Attribute sowie Methoden um neue Werte zu den Listen hinzuzufügen.

### Program

Die Klasse **Programm.java** dient zur Erstellung vom Programm eines Moduls.

Sie enthält folgende Attribute:

**private** String name;

**private** **static** **int** *nrOfPr* = 1;

Das Attribut name, welches vom Typ String ist, wird für die Speicherung des Namens vom Programm benutzt.

Das Attribut nrOfPr, welches vom Type int ist, wird für die Zählung der Nummer eines Programms benutzt.

Ansonsten finden sich in der Klasse noch die Get und Set Methoden für die Attribute.

### RelationalAtom

Die Klasse **RelationalAtom.java** dient zur Erstellung von relationalen Atomen.

Sie enthält folgende Attribute:

List<Term> t;

**private** String name;

**private** String id;

**private** String predicate;

Das Attribut t, welches eine Liste vom Typ Term ist, wird für die Speicherung der Terms eine Atoms benutzt.

Das Attribut name, welches vom Typ String ist, wird für die Speicherung des Namens eines Atoms benutzt.

Das Attribut id, welches vom Typ String ist, wird für die Speicherung der Objekt-ID eines Atoms benutzt.

Das Attribut predicate, welches vom Typ String ist, wird für die Speicherung vom Prädikat eins Atoms benutzt.

### Weiters enthält diese Klasse die folgende (wichtigere) Methode:

### public String generateOnlyFacts(int factsNum, Program pr);

### Diese Methode dient zur Meta-Code Darstellung eine Atoms.

Ansonsten finden sich in der Klasse noch die Get und Set Methoden für die Attribute sowie Methoden um neue Werte zu den Listen hinzuzufügen.

### Annotation

Die Klasse **Annotation.java** dient zur Erstellung von Annotationen.

### Sie enthält folgende Attribute:

**private** String name;

**private** String term;

List<String> inputPredicate;

List<String> outputPredicate;

Das Attribut name, welches vom Typ String ist, wird für die Speicherung des Namens vom Term benutzt.

Das Attribut term, welches vom Type String ist, wird für die Speicherung der Objekt-ID vom Term benutzt.

Das Attribut inputPredicate, welches eine Liste vom Typ String ist, wird für die Speicherung von Input-Prädikaten benutzt.

Das Attribut outputPredicate, welches eine Liste vom Typ String ist, wird für die Speicherung von Output-Prädikaten benutzt.

### Weiters enthält diese Klasse die folgende (wichtigere) Methode:

### public String generateAnnotationsModule (Program pr);

### Diese Methode dient zur Meta-Code Darstellung einer Annotation.

Ansonsten finden sich in der Klasse noch die Get und Set Methoden für die Attribute sowie Methoden um neue Werte zu den Listen hinzuzufügen.

### Term

Die Klasse **Term.java** dient zur Erstellung von Terms.

Sie enthält folgende Attribute:

**private** String name;

**private** String serialization;

Das Attribut name, welches vom Typ String ist, wird für die Speicherung des Namens eines Terms benutzt.

Das Attribut serialization, welches vom Typ String ist, wird für die Speicherung der Serialization eines Terms benutzt.

Ansonsten finden sich in der Klasse noch die Get und Set Methoden für die Attribute.

## DB

## EvaluationFramework

Abbildung 6: Package EvaluationFramework

In dem Package EvaluationFramework befindet sich nur eine Klasse, die **EvaluationFrameworkApp.java**. Diese Klasse beinhaltet die Main Methode des Programmes und ist somit die ausführbare Hauptklasse des Programmes. Die Funktionalität des Programmes befindet sich in dieser Klasse und aus dieser Klasse werden die alle Methoden, die hinter der Funktionalität hinter den in der Systemarchitektur beschriebenen Kopomenten, aufruft und ausführt.

Die User Interaktion erfolgt ebenfalls über diese Klasse. Wenn das Programm gestartet wird, wird der User über die Konosle aufgefordert eine Test Art, CBR oder RMI, auszuwählen. Die Auswahl erfolgt über die Konsole, über die Eingabe der Zahl des gewünschten Tests.

Die nachfolgende Abbildung zeigt die Konsolenausgabe beim Start des Programmes.

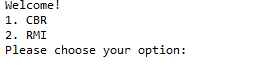


Abbildung 7: Konsolenausgabe bei Programmstart

Die Funktionalität der Klasse EvaluationFrameworkApp.java wird in den beiden nachfolgenden Abschnitten beschrieben anhand der Code Generierung von CBR und RMI.

### CBR Code Generierung

Nachdem die Testart CBR ausgewählt wurde, wird der Benutzer aufgefordert, die Input Parameter einzugeben. Dies kann anhand der folgenden Grafik, der Konsolenausgabe, gesehen werden.

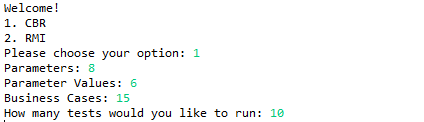


Abbildung 8: Konsolenausgabe Input Parameter

Der User gibt die Anzahl der Parameter, die Anzahl der Parameter Values und die Anzahl der Business Cases an, die generiert werden sollen. Ebenso bestimmt der User, wie oft diese Art von Test mit diesen Input Werten durchgeführt werden soll.

Nun beginnt das Programm, den Code zu generieren. Die Input Werte werden an die folgende Methode

**public** **static** String generateCBRCode(**int** parameters, **int** paramValues, **int** businessCases)

im Package DataGeneratorCBR übergeben. Diese Methode wird in der Klasse EvaluationFrameworkApp.java aufgerufen und erhält als Result einen String mit dem fertig generierten CBR Code.

Anschließend werden die Dummy Vadalog Methoden aus der Klasse VadalogExecution.java, aus dem Package Vadalog, aufgerufen, um die Evaluierungswerte zu generieren.

Da die Ausgabe über die Konsole begrenzt ist, wurde entschieden, die Ausgabe des Codes und der Evaluierungswerte in ein Text File zu schreiben. Dies ist eine Abweichung vom Konzeptuellen Entwurf.

Ausgegeben bzw. in das Text File geschrieben werden die Input Werte, der generierte Code sowie die Evaluierungsergebnisse.

Anschließend wird ein CBR Objekt erstellt, welches die Speicherung der Daten in die Datenbank erleichtern soll.

In dieses Objekt, welches dann über JDBC Funktionen an die Datenbank übergeben wird, werden das Datum und die Uhrzeit des Durchführungszeitpunktes gespeichert. Ebenso werden die Inputwerte und die Evaluierungsergebnisse gespeichert. Der generierte Code wird nicht mit in die Datenbank gespeichert.

Dieses CBR Objekt wird dann an die folgende Methode

**public** **static** **void** newCBR(CBR cbr)

aus der Klasse SaveEntry.java, aus dem Package DB, übergeben. Diese Methode wird aufgerufen und speichert die Werte des Objektes in die Datenbank.

Dies wird so oft wiederholt, wie der Benutzer die Anzahl der Tests angegeben hat.

Sobald alle Tests durchgeführt, alle Codes generiert und zusammen mit ihren Evaluierungswerten in die Datenbank gespeichert wurden, wird auf der Konsole noch ausgegeben, dass die Generierung nun abgeschlossen ist und auf die Text File verwiesen. Dies ist in der nachfolgenden Abbildung zu sehen.

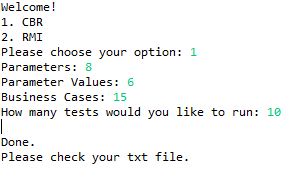


Abbildung 9: Konsolenausgabe Durchführung abgeschlossen

### RMI Code Generierung

Nachdem die Testart RMI ausgewählt wurde, wird der Benutzer nach einer Auswahl gefragt, welche der Rule Model Inheritance Testfälle durchgeführt werden soll. Dies kann anhand der folgenden Grafik, der Konsolenausgabe, gesehen werden.

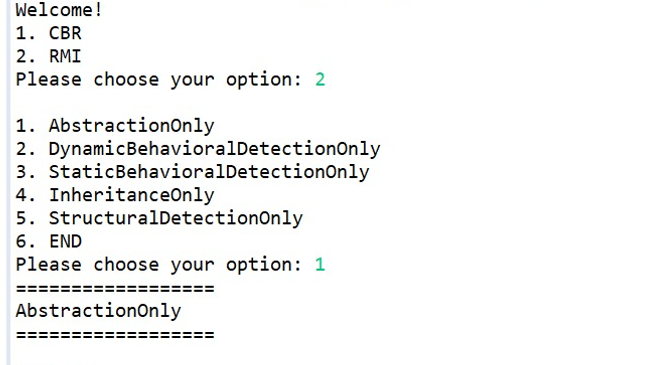


Abbildung 10: Konsolenausgabe Testart

Danach wird der Benutzer aufgefordert, die Input Parameter einzugeben. Dies kann anhand der folgenden Grafik, der Konsolenausgabe, gesehen werden.

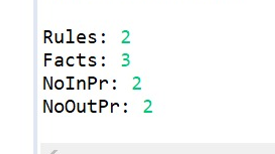


Abbildung 11: Konsolenausgabe bie Input Parameter

Es wird eine bestimmte Anzahl an Regeln und Fakten gefordert, sowie Anzahl von Output und Input Prädikaten Werte welche der User als Zahl, Integer, eingeben soll.

Nun beginnt das Programm, den Code zu generieren. Basierend auf der Auswahl des Users werden die Input Werte an die verschiedenen Methoden im Package DataGeneratorRMI übergeben. Diese Methoden werden in der Klasse EvaluationFrameworkApp.java aufgerufen und erhalten als Result einen String mit dem fertig generierten RMI Meta-Code.

Die Dummy Vadalog Methoden werden aus der Klasse VadalogExecution.java, aus dem Package Vadalog, aufgerufen, um die Evaluierungswerte zu generieren.

Ausgegeben bzw. in das Text File geschrieben werden die Input Werte, der generierte Code sowie die Evaluierungsergebnisse.

Anschließend wird ein RMI Objekt erstellt, welches die Speicherung der Daten in die Datenbank erleichtern soll.

In dieses Objekt, welches dann über JDBC Funktionen an die Datenbank übergeben wird, werden das Datum und die Uhrzeit des Durchführungszeitpunktes gespeichert. Ebenso werden die Inputwerte und die Evaluierungsergebnisse gespeichert. Der generierte Code wird nicht mit in die Datenbank gespeichert.

Dieses RMI Objekt wird dann an die folgende Methode

**public** **static** **void** newRMI(RMI rmi)

aus der Klasse SaveEntry.java, aus dem Package DB, übergeben. Diese Methode wird aufgerufen und speichert die Werte des Objektes in die Datenbank.

Sobald alle Tests durchgeführt, alle Codes generiert und zusammen mit ihren Evaluierungswerten in die Datenbank gespeichert wurden, wird auf der Konsole noch ausgegeben, dass die Generierung nun abgeschlossen ist und auf die Text File verwiesen. Dies ist in der nachfolgenden Abbildung zu sehen.

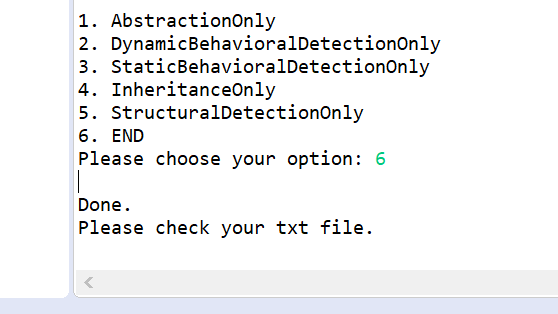


Abbildung 12: Konsolenausgabe Durchführung abgeschlossen

## Models

### CBR

### RMI

## Test

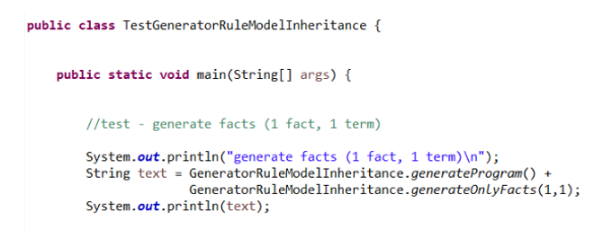
### CBR Tests

### RMI Tests

Im Folgenden wird erläutert wie der Meta-Code mit und ohne der Logik getestet wurde.

* + - 1. Testen der Syntax (ohne Logik)

Die Klasse „Test Generator Rule Modell Inheritance“ wurde benutzt, um die Syntax (ohne Logik) vom Meta-Code im Yupiter zu prüfen. Dies ist in den folgenden Abbildungen zu sehen.



### 

Abbildung 13: Ausschnitt aus der Klasse

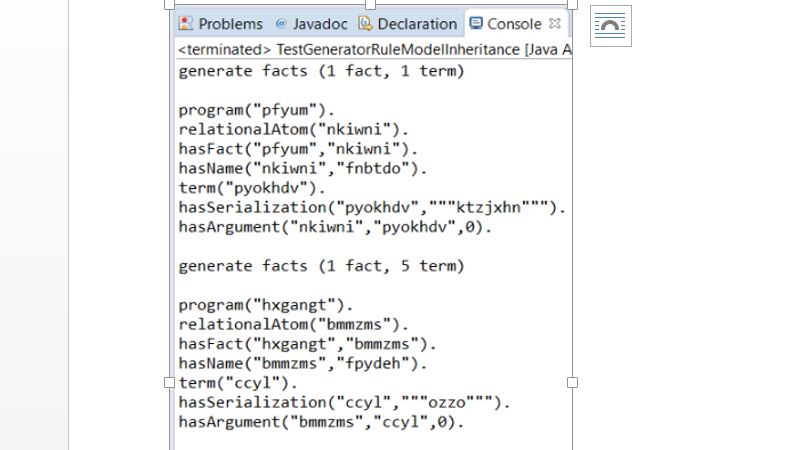


Abbildung 14: Ausschnitt aus der Konsole

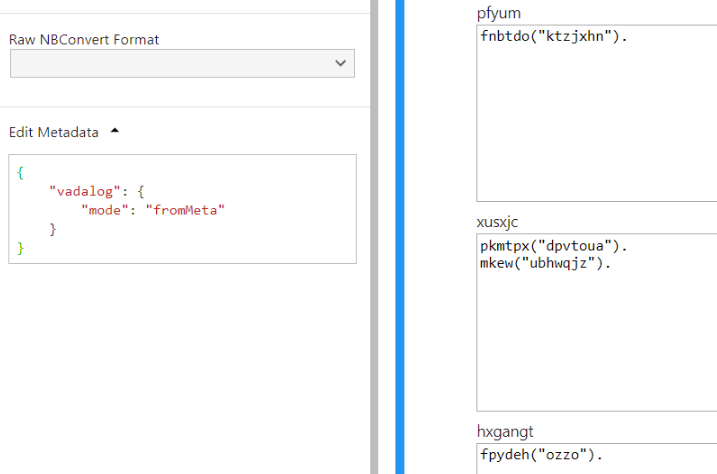


Abbildung 15: Ausschnitt aus Yupiter

* + - 1. Testen der Syntax und der Logik

Die Klasse „Evaluierung Framework App Test Generator Rule Modell Inheritance“ wurde benutzt, um die Syntax und die Logik vom Meta-Code im Yupiter zu prüfen. Im Folgenden werden Ausschnitte aus dem Testen von 1. AbstractionOnly gezeigt.

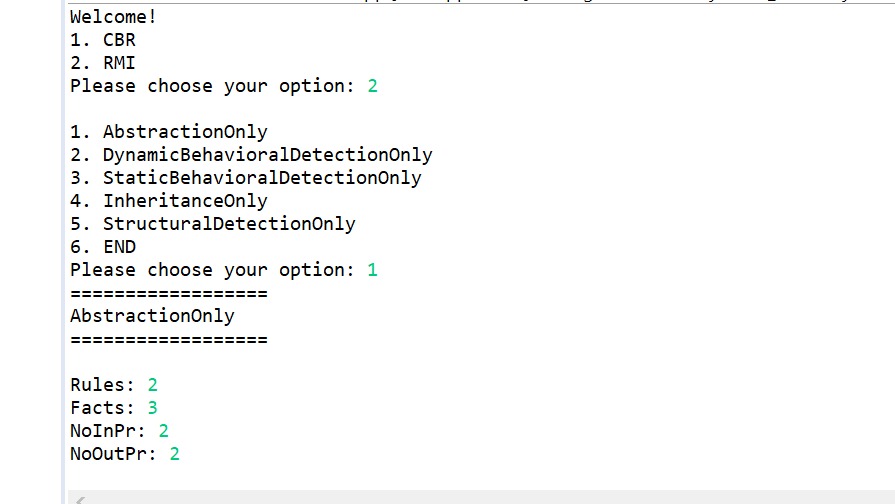


Abbildung 16: Ausschnitt aus der Konsole

Ausschnit aus der Datei, wo der Meta-Code gespeichert wird:

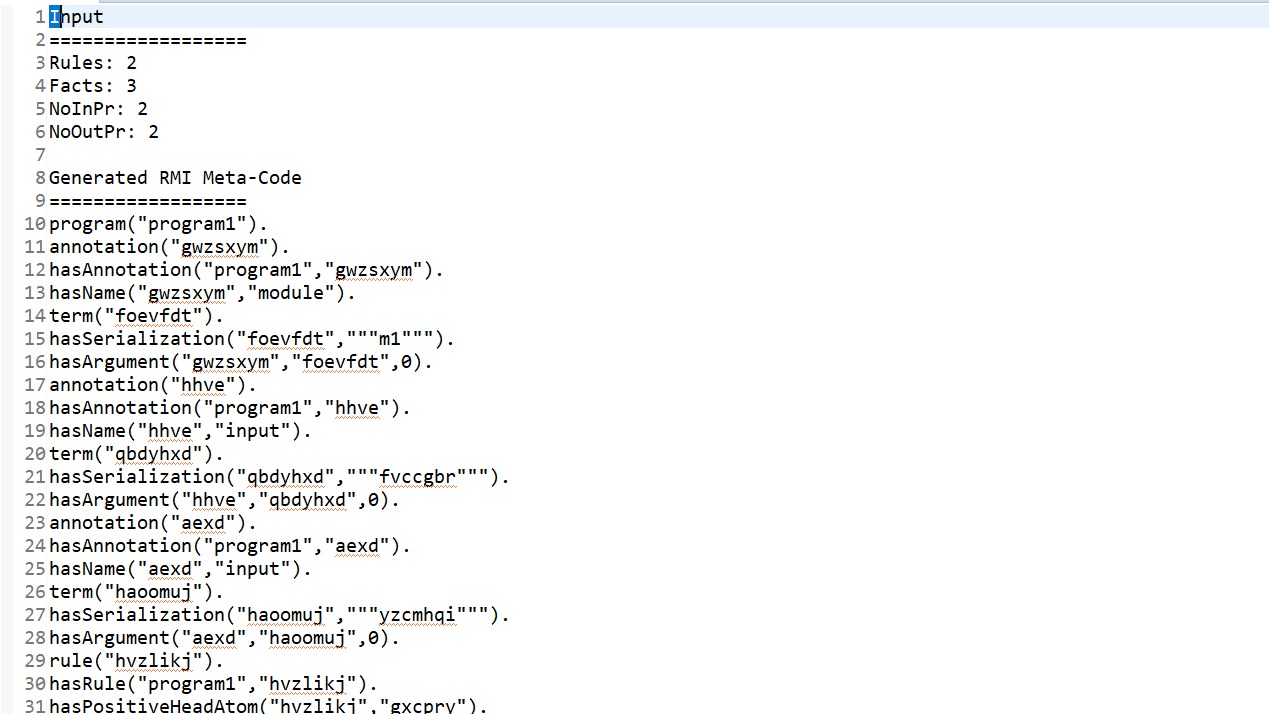


Abbildung 17: Meta-Code



Abbildung 14: Ausschnitt aus dem Yupiter

## Vadalog

# Tests der geforderten Funktionalitäten

# Abhängigkeiten Bibliotheken

# Installationsanleitung

# Limitierungen und Verbesserungsvorschläge im praktischen Einsatz

# Abbildungsverzeichnis

[Abbildung 1: Programmarchitektur 2](file:///C:\Users\Anwender\Documents\GitHub\258321_DKE_PR\Dokumente\Meilenstein%20Implementierung\Meilenstein_Implementierung.docx#_Toc25822)

[Abbildung 2: Package Struktur im Programm 4](file:///C:\Users\Anwender\Documents\GitHub\258321_DKE_PR\Dokumente\Meilenstein%20Implementierung\Meilenstein_Implementierung.docx#_Toc25823)

[Abbildung 3: Package DataGeneratorCBR 5](file:///C:\Users\Anwender\Documents\GitHub\258321_DKE_PR\Dokumente\Meilenstein%20Implementierung\Meilenstein_Implementierung.docx#_Toc25824)

[Abbildung 4: Package DataGeneratorRandomString 8](file:///C:\Users\Anwender\Documents\GitHub\258321_DKE_PR\Dokumente\Meilenstein%20Implementierung\Meilenstein_Implementierung.docx#_Toc25825)

Abbildung 5: Package Rule Model Inheritance(RMI)……………………………………………………….10

Abbildung 6: Package Evaluationframework……………………………………………………………......17

Abbildung 7: Konsolenausgabe bei Programstart………………………..…………………………………17

Abbildung 8: Konsolenausgabe bei Inputparameter…………………………………..……………………17

Abbildung 9: Konsolenausgabe Durchführung abgeschlossen……...……………………………………19

Abbildung 10: Konsolenausgabe Testart…………………………………………………………………….19

Abbildung 11: Konsolenausgabe bei Inputparameter………………………………………………………19

Abbildung 12: Konsolenausgabe Durchführung abgeschlossen………………………………………….20

Abbildung 13: Ausschnitt aus der Klasse…………………………………………………………………….21

Abbildung 14: Ausschnitt aus der Konsole…………………………………………………………………..21

Abbildung 15: Ausschnitt aus Yupiter………………………………………………………………………...22

Abbildung 16: Ausschnitt aus der Konsole…………………………………………………………………..23

Abbildung 17: Meta-Code…………………………………………………………………………………...…23

Abbildung 18: Ausschnitt aus dem Yupiter…………………………………………………………………..24

# Tabellenverzeichnis

**Es konnten keine Einträge für ein Abbildungsverzeichnis gefunden werden.**