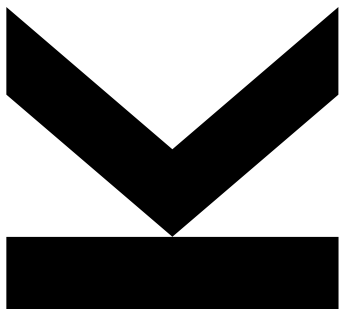


MEILENSTEIN IMPLEMENTIERUNG



258321 DKE Projekt

Gruppe 2

Teammitglieder:

k01607605, Aistleithner Andrea

k01256561, Dusanic Maja

k01356577, Teuchtmann Alexander

k01356229, Tomic Milos

Inhaltsverzeichnis

1.	Programmarchitektur	3
1.1.	Komponenten.....	3
1.1.1.	Das Evaluierungsprogramm	3
1.1.2.	Der Rule Model Inheritance Datengenerator.....	3
1.1.3.	Der CBR Datengenerator.....	4
1.1.4.	Die Dummy Vadalog Ausführung.....	4
1.1.5.	Der externe Speicher, die Datenbank	4
1.2.	Schnittstellen.....	4
1.2.1.	Die User Schnittstelle	4
1.2.2.	Die Rule Model Inheritance Schnittstelle.....	4
1.2.3.	Die CBR Schnittstelle	4
1.2.4.	Die Vadalog Schnittstelle	4
1.2.5.	Die Speicherung Schnittstelle	5
1.3.	Umsetzung im Programm.....	5
2.	Funktionalitäten einzelner Klassen.....	6
2.1.	DataGeneratorCBR	6
2.1.1.	BusinessCase.....	6
2.1.2.	BusinessCaseClass	6
2.1.3.	Context	7
2.1.4.	ContextClass	7
2.1.5.	GeneratorCBR.....	7
2.1.6.	Module.....	9
2.1.7.	Parameter.....	9
2.1.8.	ParameterValue.....	9
2.2.	DataGeneratorRandomString	9
2.3.	DataGeneratorRMI	10
2.3.1.	GeneratorRuleModelInheritance	10
2.3.2.	NonRelationalAtom.....	11
2.3.3.	RMIModule	11
2.3.4.	Module.....	12
2.3.5.	Rule	13
2.3.6.	Program.....	14
2.3.7.	RelationalAtom	14
2.3.8.	Annotation	15

2.3.9. Term	15
2.4. DB	16
2.4.1. Connector	16
2.4.2. SaveEntry	16
2.5. EvaluationFramework	17
2.5.1. CBR Code Generierung	17
2.5.2. RMI Code Generierung	19
2.6. Models	21
2.6.1. CBR	21
2.6.2. RMI	22
2.7. Test	22
2.7.1. CBR Tests	22
2.7.2. RMI Tests	23
2.8. Vadalog	26
2.8.1. VadalogExecution	26
3. Tests der geforderten Funktionalitäten	27
4. Abhängigkeiten Bibliotheken	27
4.1. mySQL Connector	27
5. Installationsanleitung	27
5.1. Allgemeines	27
5.2. Betriebssystem	27
5.3. Programme	28
5.4. Ausführung	28
6. Limitierungen und Verbesserungsvorschläge im praktischen Einsatz	31
6.1. Grafische Benutzeroberfläche	31
6.2. Speicherung Generierter Code	31
7. Abbildungsverzeichnis	32

1. Programmarchitektur

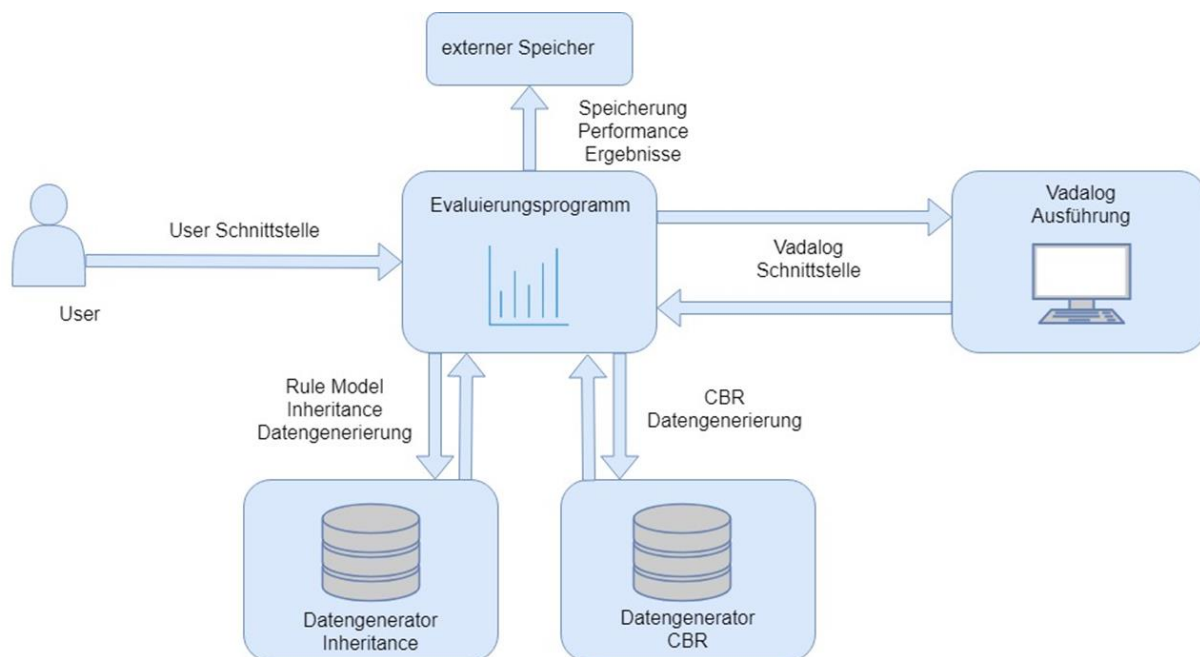


Abbildung 1: Programmarchitektur

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.

1.1. Komponenten

Das Programm setzt sich aus folgenden Komponenten zusammen:

1.1.1. 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.

1.1.2. 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.

1.1.3. 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.

1.1.4. 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 aufgetreten 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.

1.1.5. 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.

1.2. Schnittstellen

Folgende Schnittstellen lassen die Komponenten des Programmes miteinander kommunizieren:

1.2.1. Die User Schnittstelle

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

1.2.2. 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.

1.2.3. 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.

1.2.4. Die Vadalog Schnittstelle

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

1.2.5. Die Speicherung Schnittstelle

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

1.3. Umsetzung im Programm

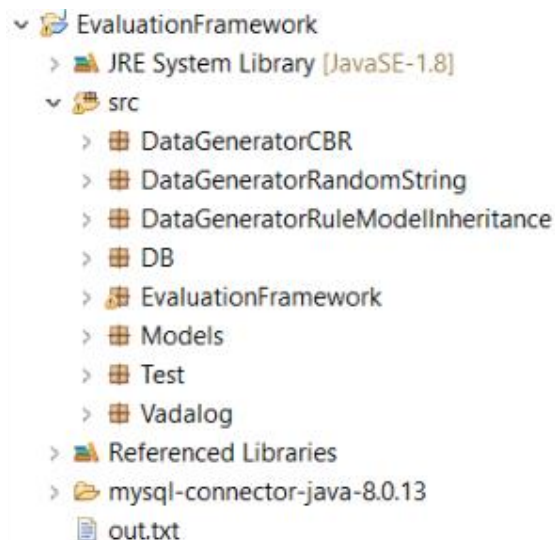


Abbildung 2: Package Struktur im Programm

In Abbildung 2 ist die Struktur des Programmes und 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 zusammengestellten 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 **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.

2. 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.

2.1. DataGeneratorCBR

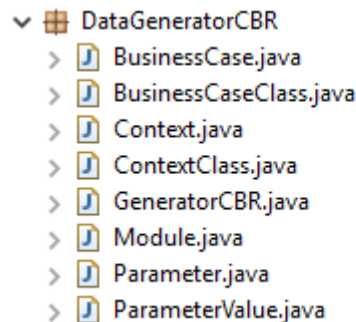


Abbildung 3: Package DataGeneratorCBR

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

2.1.1. 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.

2.1.2. 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.

2.1.3. 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.

2.1.4. 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.

2.1.5. 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 generierte 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 Generierung 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.

2.1.6. 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.

2.1.7. 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.

Zusätzlich 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.

2.1.8. 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.

2.2. DataGeneratorRandomString

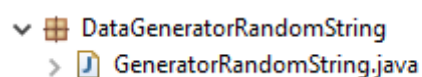


Abbildung 4: Package DataGeneratorRandomString

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.

2.3. DataGeneratorRMI

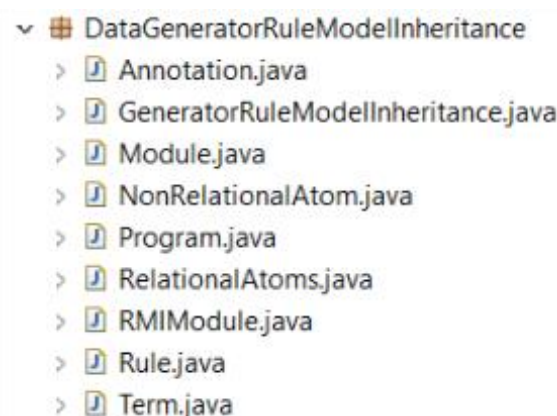


Abbildung 5: Package
DataGeneratorRuleModellInheritance (RMI)

In der Abbildung 5 sind die Klassen des Packages **DataGeneratorRMI** zu sehen. Diese Klassen werden im Folgenden genauer hinsichtlich ihrer Funktionalität und ihrem Zusammenspiel beschrieben.

2.3.1. GeneratorRuleModellInheritance

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 Referenz auf ein neues Programm.

Das Attribut programs, welches eine Liste vom Typ Programm ist, speichert 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.

2.3.2. 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 GeneratorRuleModellInheritance benutzt, um einen Meta-Code ohne Logik zu generieren.

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

2.3.3. 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.

2.3.4. 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.

2.3.5. 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.

2.3.6. Program

Die Klasse **Program.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.

2.3.7. 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 eines 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 eines 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.

2.3.8. 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.

2.3.9. 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.

2.4. DB

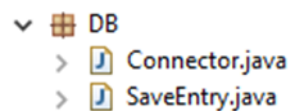


Abbildung 6: Package DB

Das Package DB umfasst zwei Klassen, welche die Verbindung zur Datenbank, sowie Operationen an der Datenbank handeln, dies ist in Abbildung 6 zu sehen.

2.4.1. Connector

Die Klasse **Connector.java** dient dem Hauptprogramm zur Erstellung einer Verbindung zur Datenbank. Sie enthält folgende Attribute:

```
private static String url = "jdbc:mysql://e42776-mysql.services.easyname.eu:3306/u48005db23?useSSL=false";
private static String username = "u48005db23";
private static String password = "prdke2018";
```

Das Attribut url, welches vom Typ String ist, beinhaltet die für die Verbindung notwendige Server-URL. Es wird, aufgrund des fehlenden SSL-Zertifikats seitens des Serveranbieters, eine ungesicherte Verbindung hergestellt (useSSL=false).

Das Attribut username, welches vom Typ String ist, beinhaltet den Benutzername für den Login der Datenbank.

Das Attribut password, welches vom Typ String ist, beinhaltet das Passwort für den Login der Datenbank.

2.4.2. SaveEntry

Die Klasse **SaveEntry.java** dient dem Hauptprogramm, unter Zuhilfenahme des Connectors, zur Speicherung der relevanten Daten. Sie enthält folgende Attribute, in diesem Fall anhand von CBR. Der Aufbau für RMI ist analog gestaltet:

```
String queryInsertCBR = "INSERT INTO cbr (date, time, noParam, noParamVal, noBusCase, exTime, errors, cpuUsage) VALUES (?, ?, ?, ?, ?, ?, ?)";
PreparedStatement insertCBR = null;
Connection connection = null;
```

Das Attribut queryInsertCBR, welches vom Typ String ist, beinhaltet das Insert Statement, wodurch die richtigen Spalten in der Datenbank angesprochen werden, sowie Platzhalter für die Inhalte bereitgestellt werden.

Das Attribut insertCBR, vom Typ PreparedStatement, dient im weiteren Verlauf als Prepared Statement für die Speicherung der Daten.

Das Attribut connection, vom Typ Connection, dient zum Verbindungsaufbau mit der Datenbank.

Im Verlauf der Klasse werden die konkreten Daten den Platzhaltern zugeordnet, sodass diese dann in die richtige Spalte der Tabelle gespeichert werden. Abschließend wird noch die Fehlerbehandlung durchgeführt.

2.5. EvaluationFramework

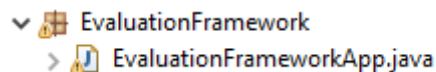


Abbildung 7: Package Evaluation Framework

In dem Package EvaluationFramework befindet sich nur eine Klasse, die **EvaluationFrameworkApp.java**, zu sehen in Abbildung 7.

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 Komponenten, aufruft und ausführt.

Die User Interaktion erfolgt ebenfalls über diese Klasse. Wenn das Programm gestartet wird, wird der User über die Konsole 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 8 zeigt die Konsolenausgabe beim Start des Programmes.

```
Welcome!
1. CBR
2. RMI
Please choose your option:
```

Abbildung 8: Konsolenausgabe Programmstart

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

2.5.1. CBR Code Generierung

Nachdem die Testart CBR ausgewählt wurde, wird der Benutzer aufgefordert, die Input Parameter einzugeben. Dies kann anhand der folgenden Grafik, Abbildung 9, der Konsolenausgabe, gesehen werden.

```
Welcome!
1. CBR
2. RMI
Please choose your option: 1
Parameters: 8
Parameter Values: 6
Business Cases: 15
How many tests would you like to run: 10
```

Abbildung 9: 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 10 zu sehen.

```
Welcome!
1. CBR
2. RMI
Please choose your option: 1
Parameters: 8
Parameter Values: 6
Business Cases: 15
How many tests would you like to run: 10
|
Done.
Please check your txt file.
```

Abbildung 10: Konsolenausgabe Durchführung abgeschlossen

2.5.2. 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 bei Start, Abbildung 11, gesehen werden.

```
welcome:
1. CBR
2. RMI
Please choose your option: 2

1. AbstractionOnly
2. DynamicBehavioralDetectionOnly
3. StaticBehavioralDetectionOnly
4. InheritanceOnly
5. StructuralDetectionOnly
6. END
Please choose your option: 1
=====
AbstractionOnly
=====
```

Abbildung 11: Konsolenausgabe Teststart

Danach wird der Benutzer aufgefordert, die Input Parameter einzugeben. Dies kann anhand der folgenden Grafik, der Konsolenausgabe, Abbildung 12, gesehen werden.

```
Rules: 2
Facts: 3
NoInPr: 2
NoOutPr: 2
```

Abbildung 12: Konsolenausgabe bei 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.

```

1. AbstractionOnly
2. DynamicBehavioralDetectionOnly
3. StaticBehavioralDetectionOnly
4. InheritanceOnly
5. StructuralDetectionOnly
6. END
Please choose your option: 6
|
Done.
Please check your txt file.

```

Abbildung 13: Konsolenausgabe Durchführung abgeschlossen

2.6. Models

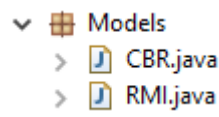


Abbildung 14:
Package Models

Das Package Models umfasst zwei Klassen, welche die Models zu CBR und RMI bereitstellen. Die beinhalten alle relevanten Informationen für die spätere Datenspeicherung.

2.6.1. CBR

Die Klasse **CBR.java** dient dem Hauptprogramm zur Erstellung der CBR Objekte, welches alle Informationen über den jeweiligen CBR-Test enthält. Sie enthält folgende Attribute:

```

private Date date;
private Time time;
private int noParm;
private int noParmVal;
private int noCont;
private int noBusCase;
private double exTime;
private boolean errors;
private double cpuUsage;

```

Die Attribute date und time, vom Typ Date und Time, enthalten jeweils das Datum und die Uhrzeit, an welchen der Test durchgeführt wurde.

Die Attribute noParm, noParmVal, noCont und noBusCase, vom Typ int, sind CBR-spezifische Informationen, welche bei jedem Test benötigt werden.

Die Attribute exTime, errors und cpuUsage, vom Typ double und boolean, sind die Testspezifischen, zufallsgenerierten, Ergebnisse.

Weiters beinhaltet diese Klasse nur Getter und Setter Methoden für die einzelnen Attribute.

2.6.2. RMI

Die Klasse **RMI.java** dient dem Hauptprogramm zur Erstellung der RMI Objekte, welches alle Informationen über den jeweiligen RMITest enthält. Sie enthält folgende Attribute:

```
private Date date;
private Time time;
private int testType;
private int noRules;
private int noFacts;
private int noInPr;
private int noOutPr;
private double exTime;
private boolean errors;
private double cpuUsage;
```

Die Attribute date und time, vom Typ Date und Time, enthalten jeweils das Datum und die Uhrzeit, an welchen der Test durchgeführt wurde.

Die Attribute testType, noRules, noFacts, noInPr, noOutPr, vom Typ int, sind CBR-spezifische Informationen, welche bei jedem Test benötigt werden.

Die Attribute exTime, errors und cpuUsage, vom Typ double und boolean, sind die Testspezifischen, zufallsgenerierten, Ergebnisse.

Weiters beinhaltet diese Klasse nur Getter und Setter Methoden für die einzelnen Attribute.

2.7. Test

Das Package Test besteht aus folgenden zwei Klassen, welche in der nachfolgenden Abbildung zu sehen sind.

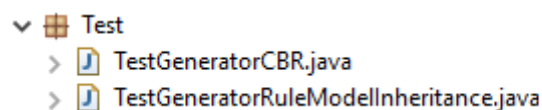


Abbildung 15: Package Test

In diesen beiden Klassen wurde der jeweilige Datengenerator getestet. Die Tests der Datengeneratoren werden in den beiden nachfolgenden Abschnitten beschrieben.

2.7.1. CBR Tests

Während des Implementierens des CBR Datengenerators wurde laufend der generierte Code Teil, auch wenn der gesamte Code noch nicht fertig generiert wurde, ins Jupyter Notebook kopiert und ausgeführt. Dies wurde gemacht, um festzustellen, ob Syntax Fehler im generierten Code vorhanden sind. So konnten während des Implementierens laufend die Fehler ausgebessert werden.

Als es möglich war, den gesamten Code zu generieren, wurde natürlich auch auf Funktionalität im Jupyter Notebook getestet.

Dieser Test war ebenfalls erfolgreich.

In der folgenden Abbildung ist der Code Ausschnitt aus der Test Klasse zu sehen. Die Hauptmethode des CBR Datengenerators wurde aufgerufen und das Ergebnis auf der Konsole ausgegeben, um den generierten Code zu kopieren und im Jupyter Notebook testen zu können.

```
public class TestGeneratorCBR {

    public static void main(String[] args) {

        // this class was used to test and print the Generated CBR Code

        System.out.println(GeneratorCBR.generateCBRCode(3, 4, 5));

    } //main
}
```

Abbildung 16: Ausschnitt CBR Test

2.7.2. RMI Tests

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

Testen der Syntax (ohne Logik)

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

```
public class TestGeneratorRuleModelInheritance {

    public static void main(String[] args) {

        //test - generate facts (1 fact, 1 term)

        System.out.println("generate facts (1 fact, 1 term)\n");
        String text = GeneratorRuleModelInheritance.generateProgram() +
                    GeneratorRuleModelInheritance.generateOnlyFacts(1,1);
        System.out.println(text);

    }
}
```

Abbildung 17: Ausschnitt aus der Klasse


```

<terminated> TestGeneratorRuleModellInheritance [Java A
generate facts (1 fact, 1 term)

program("pfyum").
relationalAtom("nkiwni").
hasFact("pfyum", "nkiwni").
hasName("nkiwni", "fnbtdo").
term("pyokhdv").
hasSerialization("pyokhdv", ""ktzjxhn").
hasArgument("nkiwni", "pyokhdv", 0).

generate facts (1 fact, 5 term)

program("hxgangt").
relationalAtom("bmmzms").
hasFact("hxgangt", "bmmzms").
hasName("bmmzms", "fpydeh").
term("ccyl").
hasSerialization("ccyl", ""ozzo").
hasArgument("bmmzms", "ccyl", 0).

```

Abbildung 18: Ausschnitt aus der Konsole

Raw NBConvert Format

Edit Metadata ^

```

{
  "vadalog": {
    "mode": "fromMeta"
  }
}

```

pfyum

```
fnbtdo("ktzjxhn").
```

xusxjc

```
pkmtpx("dpvtoua").
mkew("ubhwqjz").
```

hxgangt

```
fpydeh("ozzo").
```

Abbildung 19: Ausschnitt aus Jupyter Test

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 Jupyter zu prüfen. Im Folgenden werden Ausschnitte aus dem Testen von 1. AbstractionOnly gezeigt.

```
Welcome!
1. CBR
2. RMI
Please choose your option: 2

1. AbstractionOnly
2. DynamicBehavioralDetectionOnly
3. StaticBehavioralDetectionOnly
4. InheritanceOnly
5. StructuralDetectionOnly
6. END
Please choose your option: 1
=====
AbstractionOnly
=====

Rules: 2
Facts: 3
NoInPr: 2
NoOutPr: 2
```

Abbildung 20: Ausschnitt aus der Konsole

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

```
1 Input
2 =====
3 Rules: 2
4 Facts: 3
5 NoInPr: 2
6 NoOutPr: 2
7
8 Generated RMI Meta-Code
9 =====
10 program("program1").
11 annotation("gwzsxym").
12 hasAnnotation("program1", "gwzsxym").
13 hasName("gwzsxym", "module").
14 term("foevfdt").
15 hasSerialization("foevfdt", ""m1").
16 hasArgument("gwzsxym", "foevfdt", 0).
17 annotation("hhve").
18 hasAnnotation("program1", "hhve").
19 hasName("hhve", "input").
20 term("qbdyhd").
21 hasSerialization("qbdyhd", ""fvccgbr").
22 hasArgument("hhve", "qbdyhd", 0).
23 annotation("aexd").
24 hasAnnotation("program1", "aexd").
25 hasName("aexd", "input").
26 term("haoomuj").
27 hasSerialization("haoomuj", ""yzcmhqi").
28 hasArgument("aexd", "haoomuj", 0).
29 rule("hvzlikj").
30 hasRule("program1", "hvzlikj").
31 hasPositiveHeadAtom("hvzlikj", "gxcprv").
```

Abbildung 21: Screenshot Meta-Code

```
hasConcreteDependency(M,X) :- predicateDependency(M,X,Y), concretePredicate(M,Y).
hasAbstractDependency(M,X) :- predicateDependency(M,X,Y), abstractPredicate(M,Y).

abstractPredicate(M,P) :- predicate(M,P), predicate(M,P2), not concretePredicate(M,P2), P2 = P. % P2 necessary as o
abstractModule(M) :- abstractPredicate(M,_).

@output("abstractPredicate").
@output("abstractModule").
```

abstractModule	abstractPredicate
M	M P
program1	program1 luxjg

```
]:
```

```
]:
```

Abbildung 22: Ausschnitt aus dem Jupyter

2.8. Vadalog



▼  Vadalog
>  VadalogExecution.java

Abbildung 23: Package Vadalog

Das Package Vadalog umfasst eine Klasse, welche die Generierung der Performedaten durch die Vadalogschnittstelle imitieren soll, dies ist in der oben stehenden Abbildung zu sehen.

2.8.1. VadalogExecution

Die Klasse **VadalogExecution.java** dient dem Hauptprogramm Generierung der benötigten Performedaten. Diese Schnittstelle wird lediglich initiiert, die erzeugten Daten werden zufallsgeneriert und orientieren sich in keinsten Weise an den Eingabedaten. Sie enthält folgende Methoden:

```
public static double calcExTime()
public static boolean calcNoErrors()
public static double calcCpuUsage()
```

Die Methode calcExTime() liefert einen double Wert als Ergebnis und generiert eine zufällige Ausführungszeit für die Tests.

Die Methode calcNoErrors() liefert einen boolean Wert als Ergebnis und generiert entweder eine positive oder negative Aussage über entstandene Fehler während der Tests.

Die Methode calcCpuUsage() liefert einen double Wert als Ergebnis und generiert eine zufällige CPU-Auslastung, welche während der Tests herrschte.

3. Tests der geforderten Funktionalitäten

Um die geforderten Funktionalitäten zu testen, wurden eigene Test Klassen erstellt. Dies ist im Abschnitt 2.7 Test genau erläutert.

4. Abhängigkeiten Bibliotheken

Um alle notwendigen Funktionalitäten zu bekommen, wurde eine Bibliothek verwendet. Diese wurde im Programm hinterlegt. Die Bibliothek wird im nachfolgenden Abschnitt genauer erläutert.

4.1. mySQL Connector

Um eine Verbindung mit der verwendeten mySQL Datenbank herzustellen, wird der mySQL Connector benötigt. Dieser stellt die Möglichkeit für die Verbindungsherstellung, sowie für das Übermitteln der Daten dar.

Diese Bibliothek wird dafür verwendet und ist auch im Programm hinterlegt:

- mysql-connector-java-8.0.13.jar

5. Installationsanleitung

Nachfolgend werden die notwendigen Schritte beschrieben, um das Programm zu installieren und ausführen zu können.

5.1. Allgemeines

Zum Testen der Evaluierungssoftware wurde eine durch die LVA-Leitung vorgegebene Softwareumgebung erschaffen. Die Tests beziehen sich nur auf die nachfolgend angegebenen Betriebssysteme und Programme. Andere Versionen oder Ausführungen dieser Softwareprogramme könnten den Betrieb der Evaluierungssoftware beeinflussen und den stabilen Betrieb möglicherweise nicht garantieren.

5.2. Betriebssystem

Auf einer virtuellen Maschine unter VMWare Workstation 15 Player wurde das Betriebssystem Fedora 28 installiert. Folgende Eckdaten der verwendeten Software konnten festgehalten werden:

VMWare Version:	15.0.2 build-10952284
Fedora Version:	v28; 32-Bit; GNOME 3.28.1

Die zum Login in das Betriebssystem notwendigen Daten lauten:

Benutzername: DKE PR
Passwort: dkepr

Die virtuelle Maschine ist bei Bedarf unter folgendem Link herunterladbar:

https://www.ywopikaf.com/Fedora_28.rar

5.3. Programme

Folgende zur Ausführung benötigte Programme wurden nachträglich installiert. Diese lauten wie folgt:

- Java (Javaws implementation from OpenJDK)
Version: 1.7.1-11.fc28

5.4. Ausführung

Zuerst wird die ausführbare Evaluierungssoftware (EvaluationFramework.jar) an einem leicht erreichbaren Ort abgelegt. In diesem Fall liegt sie auf „/home/dkepr/Downloads“.

In der folgenden Abbildung ist das Speicherverzeichnis der Evaluierungssoftware zu sehen.

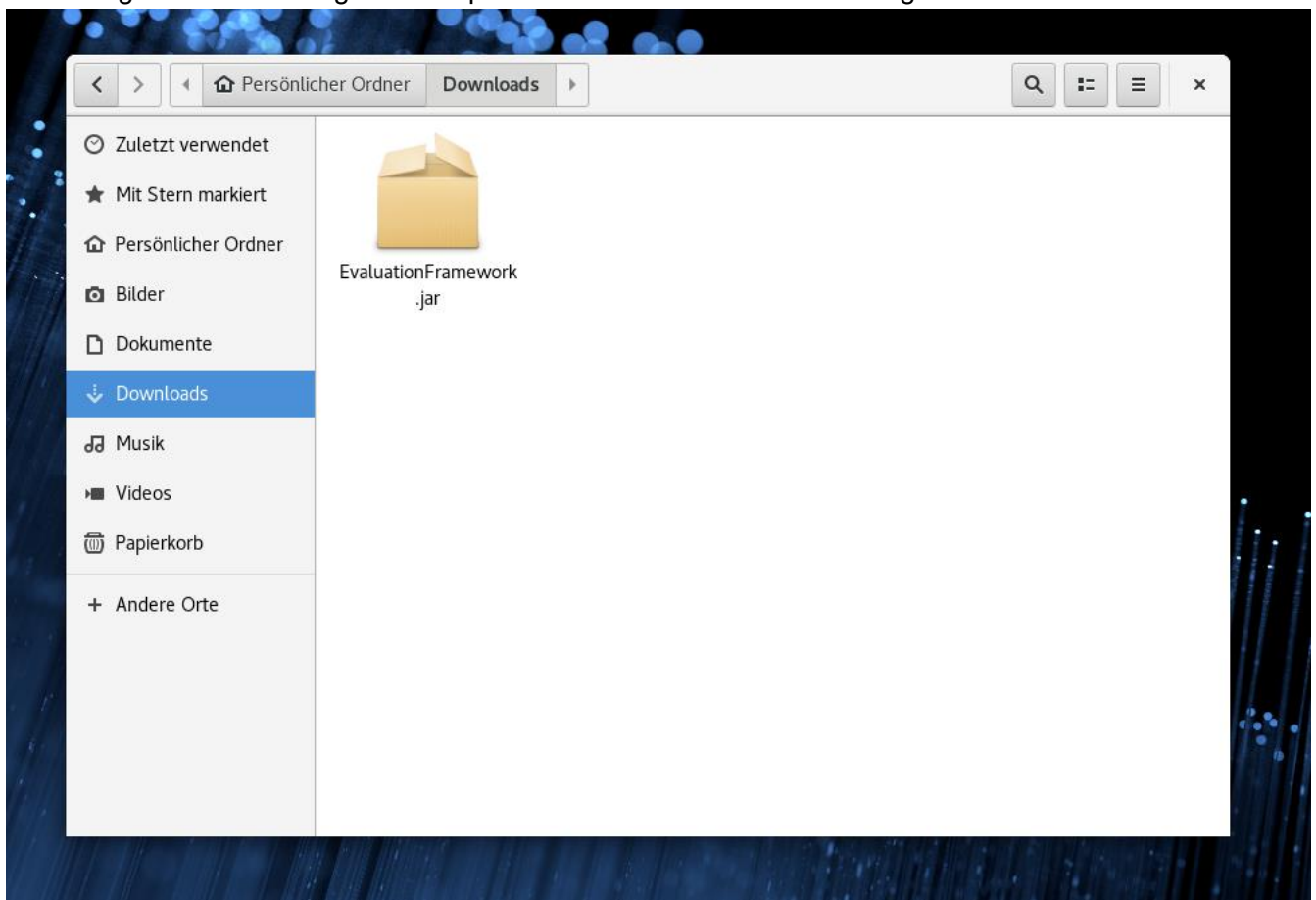


Abbildung 24: Speicherverzeichnis der Evaluierungssoftware

Im nächsten Schritt wird das Terminal geöffnet, und in das betreffende Verzeichnis navigiert. Dies ist in dem folgenden Screenshot zu sehen.

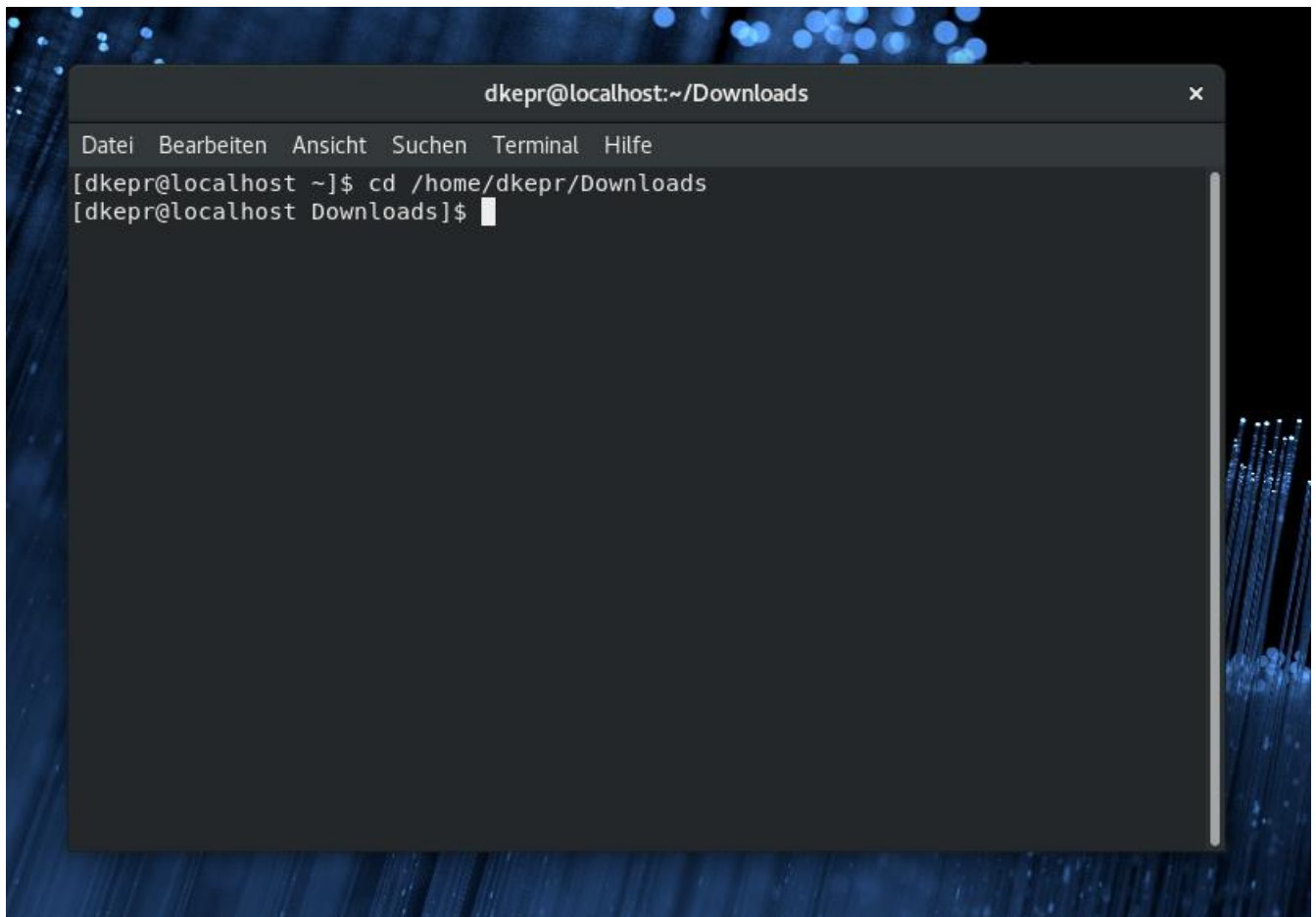


Abbildung 25: Terminal mit richtigem Pfad

Hier wird nun folgender Befehl eingegeben, wodurch das Programm dann startet:

„java -jar EvaluationFramework.jar“

Dies ist in dem nachfolgenden Screenshot zu sehen.

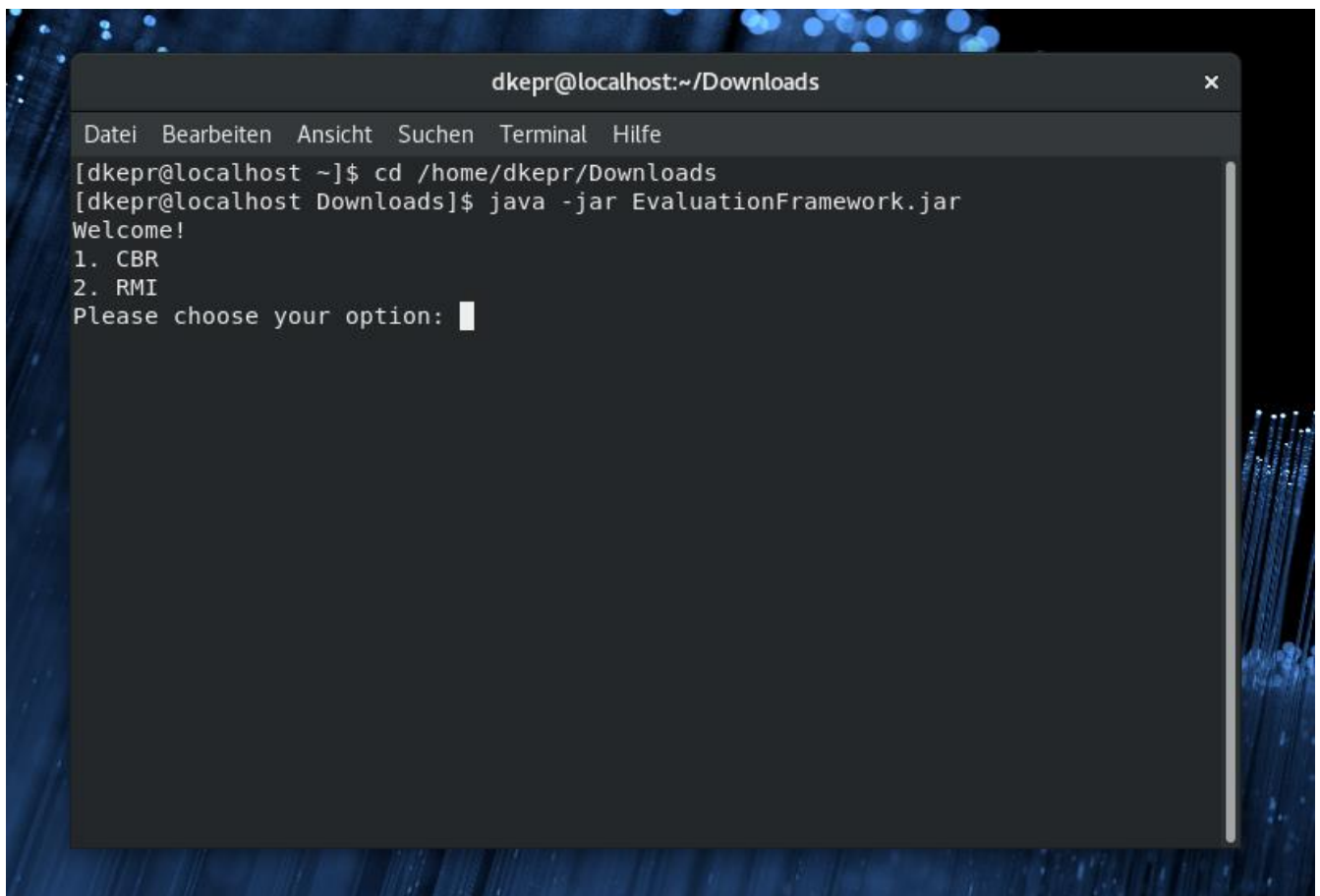


Abbildung 26: Terminal mit laufendem Programm

Nun kann der Benutzer je nach Bedarf einen Testlauf starten. Ist dieser beendet, so erhält der Benutzer eine Erfolgsmeldung, die Performancedaten werden in der Datenbank abgelegt und ein Textdokument mit den generierten Daten wird im gleichen Verzeichnis erzeugt, zu sehen in folgendem Screenshot.

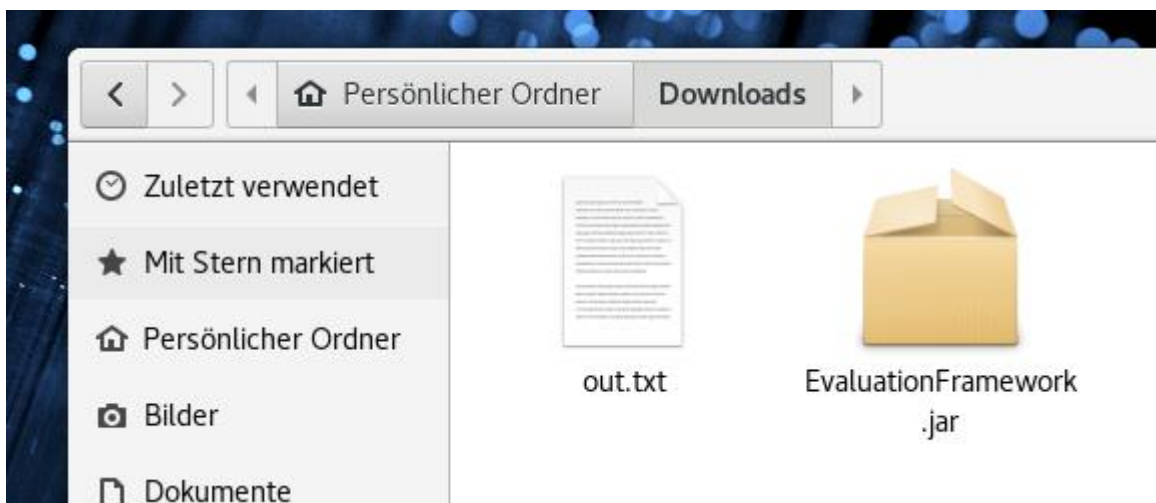


Abbildung 27: Endprodukt nach Ausführung des Programms

6. Limitierungen und Verbesserungsvorschläge im praktischen Einsatz

Das Projektteam schlägt im folgenden Abschnitt Verbesserungsvorschläge für den praktischen Einsatz vor.

6.1. Grafische Benutzeroberfläche

Zurzeit erfolgt die Bedienung des Evaluation Frameworks über die Konsole. Der Einsatz einer Grafischen Benutzeroberfläche würde die User Interaktionen erleichtern.

6.2. Speicherung Generierter Code

Um die Evaluierungsergebnisse, insbesondere aufgetretene Fehler, im Nachhinein besser nachvollziehen zu können, wäre der Vorschlag, auch den generierten Code in der Datenbank zu speichern. Dies erfolgt bei aktuellen Stand des Programmes nicht.

7. Abbildungsverzeichnis

Abbildung 1: Programmarchitektur	3
Abbildung 2: Package Struktur im Programm.....	5
Abbildung 3: Package DataGeneratorCBR	6
Abbildung 4: Package DataGeneratorRandomString	9
Abbildung 5: Package DataGeneratorRuleModellInheritance (RMI)	10
Abbildung 6: Package DB	16
Abbildung 7: Package Evaluation Framework	17
Abbildung 8: Konsolenausgabe Programmstart	17
Abbildung 9: Konsolenausgabe Input Parameter	18
Abbildung 10: Konsolenausgabe Durchführung abgeschlossen.....	19
Abbildung 11: Konsolenausgabe Teststart.....	19
Abbildung 12: Konsolenausgabe bei Input Parameter.....	20
Abbildung 13: Konsolenausgabe Durchführung abgeschlossen.....	21
Abbildung 14: Package Models.....	21
Abbildung 15: Package Test	22
Abbildung 16: Ausschnitt CBR Test	23
Abbildung 17: Ausschnitt aus der Klasse	23
Abbildung 18: Ausschnitt aus der Konsole	24
Abbildung 19: Ausschnitt aus Jupyter Test.....	24
Abbildung 20: Ausschnitt aus der Konsole	25
Abbildung 21: Screenshot Meta-Code.....	25
Abbildung 22: Ausschnitt aus dem Jupyter.....	26
Abbildung 23: Package Vatalog	26
Abbildung 24: Speicherverzeichnis der Evaluierungssoftware.....	28
Abbildung 25: Terminal mit richtigem Pfad	29
Abbildung 26: Terminal mit laufendem Programm.....	30
Abbildung 27: Endprodukt nach Ausführung des Programms	30