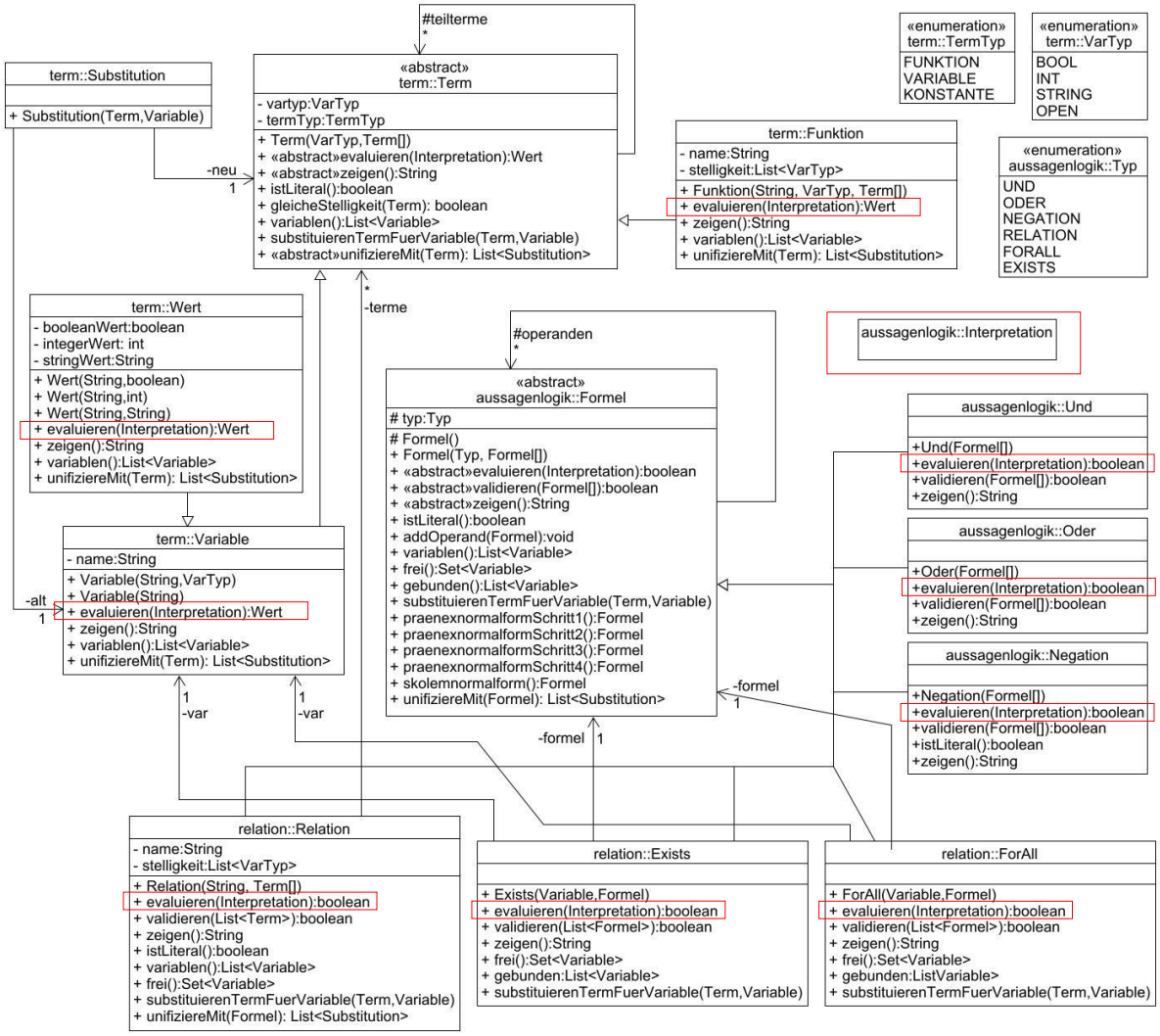
Prof. Dr. Stephan Kleuker
Hochschule Osnabrück
Fakultät Ing.-Wissenschaften und Informatik
- Software-Entwicklung -

Logik

Sommersemester 2022
5. Aufgabenblatt



Das Klassendiagramm zeigt den vollständigen Aufbau nach drei Aufgabenblättern, aktuell zu erstellende Teile sind mit einem roten Kasten markiert. Für Freunde präziser Klassendiagramme: <<nutzt>>-Beziehungen sind nicht eingezeichnet, weiterhin werden existierende und nutzbare get- und set-Methoden weggelassen.

Aufgabe 6 (2 Punkte) [Grundverständnis Syntax und Semantik]

i) $P(x, z) \wedge P(y, z)$
ii) $\exists z\ \forall x\ \forall y\ (P(x, z) \wedge P(y, z))$
iii) $\forall x\ (P(x) \Rightarrow \neg(P(f(x)))\ )$
iv) $P(f(x, y, z), f(y, z, x), f(z, x, y))$

Gegeben seien obige Formeln, geben Sie zu jeder Formel folgendes an:

a) Darstellung als Ableitungsbaum
b) freie und gebundene Variablen
c) handelt es sich um eine Aussage?
d) Nutzen Sie als Universum die ganzen Zahlen. Geben Sie jeweils, wenn möglich, eine nicht triviale Interpretation an, so dass die Formel einmal nach wahr und einmal nach falsch ausgewertet wird. Der informelle Begriff „nicht trivial“ bezieht sich darauf, dass z. B. als Relation nicht die leere Menge genutzt wird. (Sie könnten die Semantik formal

ausrechnen; eine kurze Begründung für das Ergebnis würde auch reichen. Zu jedem Aufgabenteil müssen damit zwei Interpretationen angegeben werden.)

Aufgabe 7 (7 Punkte)

Das vorherige Klassendiagramm basiert auf dem der letzten Aufgaben, ist aber stark erweitert und für (noch fehlende) Interpretationen deutlich modifiziert. Die Klasse Atom fällt weg und wird mit *Relation* als Basisstruktur ersetzt. Neu für die Syntax und Semantik sind *Term*e, die als Design-Entscheidung als eigene Klasse umsetzt werden. Aus Sicht der Implementierung sind Relationen und Funktionen eng verwandt, so dass hier auch über eine gemeinsame Basisklasse nachgedacht werden könnte. Da sich das Klassenmodell eng an der Syntax orientiert, ist das hier nicht der Fall, was zu sehr ähnlichen Methoden bei *Term*en und *Relation*en führen kann. Konstanten, also formal nullstellige Funktionen werden explizit durch die Klasse *Wert* repräsentiert, die weiterhin auch als Ergebnistyp von *Funktion*en genutzt wird. Da Java typisiert ist, ist das auch für das Modell ein Ansatz, dabei werden für *Term*e nur die in der Aufzählung *VarTyp* genannten Typen unterstützt, dabei steht OPEN für einen beliebigen oder noch unbekannten Typ. Objekte der Klasse *Wert* können einen Wert des in ihrem Konstruktoraufruf definierten Typen aufnehmen. Grundsätzlich gilt wieder, dass ein gleicher Name die gleiche Variable bzw. Funktion bzw. Relation bezeichnet, was bei der Erstellung berücksichtigt wird. Formal etwas einschränkend darf es immer nur eine Funktion und eine Relation pro Namen geben, allgemeiner wären gleiche Namen mit unterschiedlichen Stelligkeiten erlaubt. Die Aufzählung *TermTyp* übernimmt für *Term*e die gleiche Rolle, wie *Typ* für *Formel*n und gibt den konkreten Typen an. Für *Funktion*en und *Relation*en werden in der Variablen *stelligkeit* die Typen der einzelnen Teilterme festgehalten. Weitere Beziehungen und das Konzept der Klassen *Exists* und *ForAll* sollten Sie sich selbst aus den formalen Syntaxregeln und dem Code erarbeiten können. Die in der vorherigen Aufgabe genutzte clone()-Methode wurde in die *default*-Methode des Interfaces *util.DeepClone* ausgelagert, heißt jetzt *deepClone()* und kann zur Erzeugung tiefer Kopien der hier gegebenen Klassen genutzt werden. Ein kleines bereits funktionierendes Beispielprogramm test.Beispiel ist:

```
public class Beispiel {
  public static void main(String[] args) {
    Variable v1 = new Variable("v1", VarTyp.INT);
    Variable v2 = new Variable("v2", VarTyp.INT);
    Wert w = new Wert("egal", 42);
    Funktion fu = new Funktion("f", VarTyp.INT, v1, w);
    Relation r = new Relation("P", fu, v2);
    Formel f = new Exists(new Variable("v1", VarTyp.INT), r);
    System.out.println(f.zeigen() + "\n" + f);
  }
}
```

Die Ausgabe lautet (es ist gelb angedeutet, dass es entgegen der Objekt-Erstellung automatisch nur eine Variable v1 gibt):

```
(∃v1 P(f(v1, 42), v2))
Exists [var=Variable [name=v1, typ=INT, teilterme=[], hashCode()=718231523],
typ=EXISTS, operanden=[Relation [name=P, terme=[Funktion [name=f,
stelligkeit=[INT, INT], typ=INT, teilterme=[Variable [name=v1, typ=INT,
teilterme=[], hashCode()=718231523], 42], hashCode()=1349414238], Variable
[name=v2, typ=INT, teilterme=[], hashCode()=672320506]], stelligkeit=[INT,
INT], typ=RELATION, operanden=[]]]]
```

a) Schreiben bzw. ergänzen Sie die Klasse *Interpretation*, deren Objekte jeweils eine konkrete Interpretation für eine prädikatenlogische Formel sind. Überlegen Sie, dass eine Interpretation Belegungen für *Variable*n, *Funktion*en und *Relation*en beinhalten

muss. Da Java einfache Funktionen nicht als Objekte hat, müssen Sie dies über ein einfaches Interface mit nur einer Methode realisieren, dabei kann die Nutzung des Typs *Wert* für Parameter und Ergebnisse hilfreich sein. (Dieser Tipp sollte reichen, Sie können sich in dem Zusammenhang optional auch genauer mit Functional Interfaces beschäftigen. Ein Umsetzungsansatz befindet sich mit den Interfaces term.IFunktion und relation.IRelation im Code)

b) Realisieren Sie für die gezeigten Klassen dann die Methode *evaluieren(.)* für alle Klassen zunächst bis auf *Exists* und *ForAll*. Sollte eine *Interpretation* nicht passen, es fehlt z. B. ein Wert für eine Variable, brechen Sie einfach die Ausführung mit einer Exception mit passender Meldung ab. Evaluationen von Formeln ergeben einen Booleschen Wert, Evaluationen von Termen ein Objekt vom gegebenen Typ *Wert*. Schreiben Sie dann einige Beispiele mit denen Sie zeigen können, dass Ihr Ansatz funktioniert.
Orientieren Sie sich dabei an der formalen Semantik und überlegen Sie im zweiten Schritt, wie Sie *evaluieren(.)* z. B. in der Klasse Funktion umsetzen wollen. Sie müssen dazu zuerst alle Teilterme interpretieren, d. h. deren *evaluieren()* aufrufen, so dass Sie eine Liste von Wert-Objekten als Ergebnisse der Terme erhalten. Dann muss das Interpretationsobjekt die konkrete Funktion zum Funktionsobjekt liefern, dabei ist auszunutzen, dass Funktionsnamen eindeutig sein sollen (stellt die gegebene Implementierung schon sicher). Dann muss die erhaltene konkrete Funktion auf die Liste von Wert-Objekten angewandt werden, um dann den konkreten Wert der Funktion unter der gegebenen Interpretation zu berechnen [so steht es in der Semantik-Definition]. Der folgende Code zeigt ein Beispiel, bei dem einer zweistelligen Funktion fun/2 eine einfache Addition zugeordnet wird. Die Methoden- und Interface-Namen könnten Sie auch nutzen, können aber auch andere Ideen verfolgen. Es muss ein Programm test.Interpretationsbeispiel der folgenden Form möglich sein, bei dem direkt einer Funktion fun und einer Relation rel eine konkrete Semantik mit ip.setFunktion(.) und ip.setRelation(.) zugeordnet wird.

```
package test;

import java.util.List;

import aussagenlogik.Interpretation;
import relation.IRelation;
import relation.Relation;
import term.Funktion;
import term.IFunktion;
import term.VarTyp;
import term.Variable;
import term.Wert;

public class Interpretationsbeispiel {
  public static void main(String[] args) {
    Funktion fun = new Funktion("fun", VarTyp.INT
        , new Variable("x",VarTyp.INT), new Variable("y", VarTyp.INT));
    Interpretation ip = new Interpretation();
    // uebergebener Parameter kann Objekt einer Klasse sein, die das
    // Interface realisiert, geht kuerzer als Lambda-Ausdruck
    ip.setFunktion("fun", new IFunktion() {
      @Override
      public Wert funktion(List<Wert> parameter) {
        if (parameter.size() != 2
            || parameter.get(0) == null
            || parameter.get(1) == null
```

```
              || parameter.get(0).getTyp() != VarTyp.INT
              || parameter.get(1).getTyp() != VarTyp.INT) {
            throw new IllegalStateException("Addition nur auf 2 INT moeglich"
                + ", gefunden: " + parameter);
          }
          return new Wert("egal"
              ,   parameter.get(0).getIntegerWert()
                + parameter.get(1).getIntegerWert());
        }
      });
      ip.setVariable("x", new Wert("x", 1));
      ip.setVariable("y", new Wert("y", 2));
      System.out.println(fun.zeigen() + "\n" + fun.evaluieren(ip));

      Variable vi = new Variable("a", VarTyp.INT);
      Relation rel = new relation.Relation("rel", fun, vi);
      ip.setRelation("rel", new IRelation() {
        @Override
        public boolean relation(List<Wert> parameter) {
          if (parameter.size() != 2
              || parameter.get(0) == null
              || parameter.get(1) == null
              || parameter.get(0).getTyp() != VarTyp.INT
              || parameter.get(1).getTyp() != VarTyp.INT) {
            throw new IllegalStateException("Kleiner nur auf 2 INT moeglich"
                + ", gefunden: " + parameter);
          }
          return parameter.get(0).getIntegerWert()
              < parameter.get(1).getIntegerWert();
        }
      });
      ip.setVariable("a", new Wert("a", 4));
      System.out.println(rel.zeigen() + "\n" + rel.evaluieren(ip));
      ip.setVariable("a", new Wert("a", 1));
      System.out.println(rel.zeigen() + "\n" + rel.evaluieren(ip));
    }
  }
```

Die Ausgabe lautet:

```
fun(x, y)
3
rel(fun(x, y), a)
true
rel(fun(x, y), a)
false
```

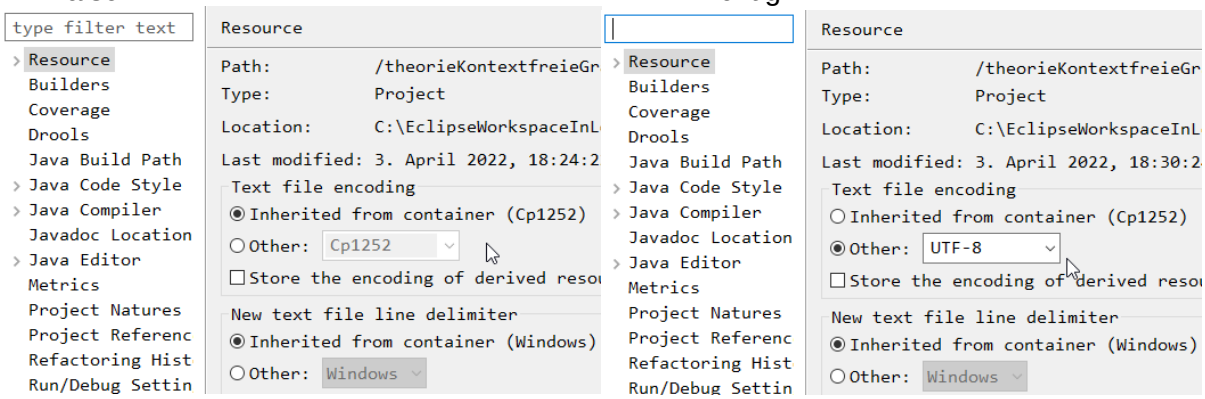
c) [optional, aber hier wird es spannend] Realisieren Sie die Methode *evaluieren(.)* für die Klassen *Exists* und *ForAll*. Dies macht auf den ersten Blick hoffentlich für Sie keinen Sinn, da es sich um unendlich viele Werte handelt (richtig). Der Ansatz ist hier, dass Sie endlich viele Werte ausprobieren, die Sie sich aussuchen können, und so bei *Exists* im Erfolgsfall zeigen können, dass die gesamte Formel wahr ist und wenn Sie keinen passenden Wert finden, dann vereinfacht false zurückgeben und zusätzlich einen Warntext ausgeben wird, dass kein passender Wert gefunden wurde, er aber existieren könnte. Dual dazu, können Sie zeigen, dass die gesamte *ForAll*-Formel false ist. Passen alle Beispielwerte wird dann vereinfacht true zurückgeben und zusätzlich einen Text ausgegeben, dass kein nicht passender Wert gefunden wurde, er aber existieren

könnte. Vereinfacht ausgedrückt, nehmen Sie z. B. eine endliche Menge von Zahlen und tun Sie so, als ob das alle Zahlen sind, die es gibt.

d) [optional] Setzen Sie Ihre Lösungen aus Aufgabe 6 zu Prädikaten, die keine freie Variablen enthalten, jeweils für die Ergebnisse true und false mit Ihrer Klasse *Interpretation* als Java-Programm um.

Hinweis: Sollten Sie in der Ausgabe keine Quantoren sehen, ist wahrscheinlich ein falscher Schriftsatz eingestellt.

falsch: richtig:



Die Einstellung sollte generell für alle Projekte unter Window > Preferences > General > Workspace unten unter „Text file encoding“ eingestellt werden.