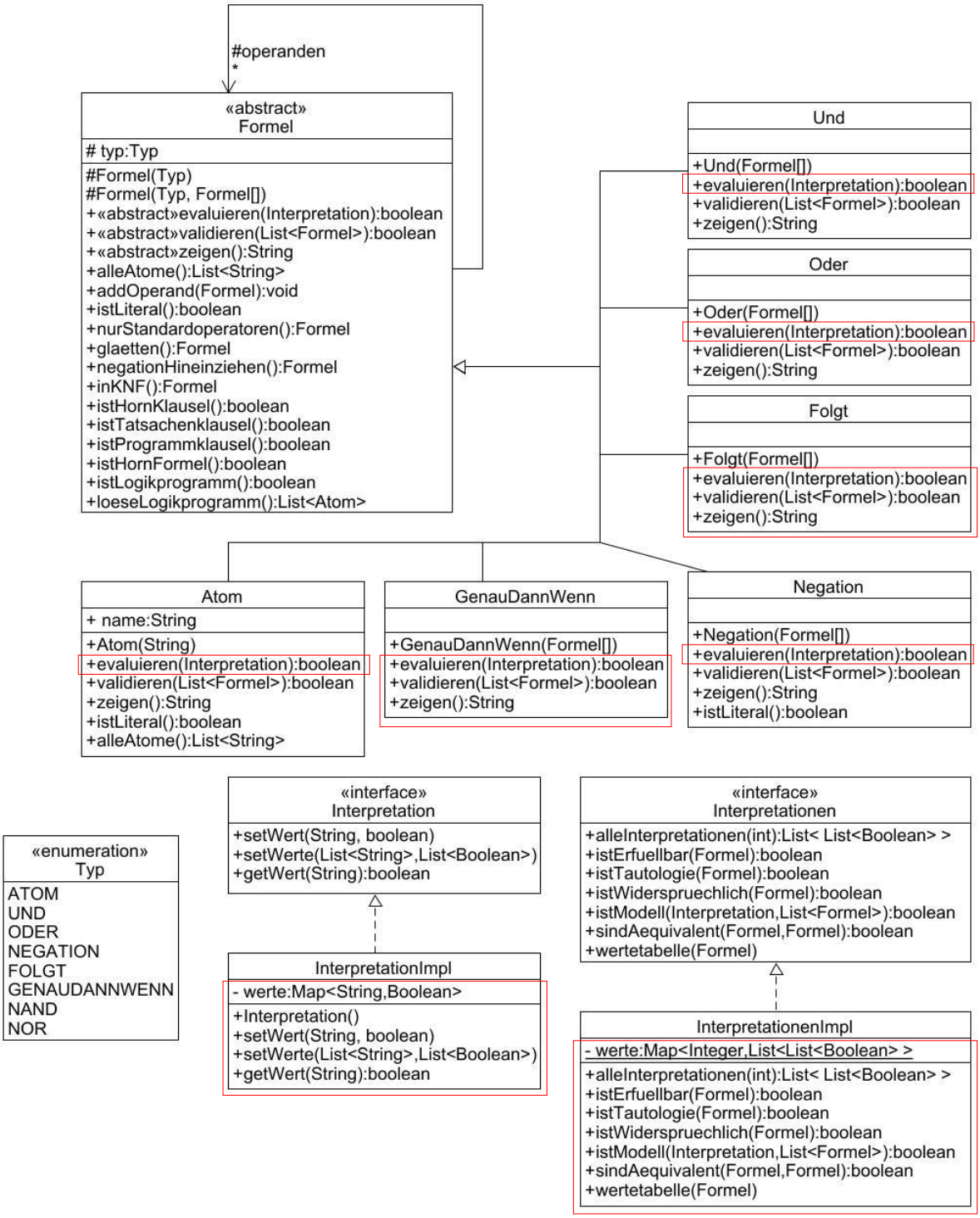
Prof. Dr. Stephan Kleuker
Hochschule Osnabrück
Fakultät Ing.-Wissenschaften und Informatik
- Software-Entwicklung -

Logik

Sommersemester 2022
2. 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.

Aufgabe 3 (9 Punkte)

Eine Möglichkeit formale Definitionen genauer zu verstehen ist eine Umsetzung in einer Programmiersprache. Eine solche Umsetzung wurde in dem von der Veranstaltungsseite

erhältlichem Projekt logikAufgabeAussagenlogik begonnen. Lesen Sie zuerst die Informationen zum Modell und schauen Sie vor der Bearbeitung der Aufgaben den existierenden Code durch.

Alle Formeln leiten sich aus der Klasse *Formel* ab, dabei kann jede Formel zunächst grundsätzlich beliebig viele *operanden* haben, die wieder Formeln sind. Die Basisform sind Aussagenvariablen, die hier (und später in der Veranstaltung) auch *Atom*e genannt werden. Eine einfache Formel kann z. B. wie folgt konstruiert werden, dabei hat der Konstruktor der Klasse *Formel* eine dynamische Parameterliste mit beliebig vielen Formeln als Parameter (im Klassendiagramm als zwei Konstruktoren dargestellt, da die UML programmiersprachenunabhängig ist).

```
Formel formel = new Und(new Atom("a"), new Atom("b"));
```

Jede Formel hat eine Methode *validieren*, der eine Liste von Formeln, typischerweise die Operanden übergeben werden und die prüft, ob es sich um sinnvolle Operanden handelt. Bei einem „UND“ reicht dazu die Forderung, dass es zumindest ein Operand (genau ein Operand ist dann die Identitätsfunktion) ist, bei der *Negation* muss es genau ein Operand sein. Als Design-Entscheidung wird bei Fehlern, also auch gescheiterten Validierungen eine IllegalArgumentException mit einer Begründung geworfen.
Es gibt zwei Methoden, um Formeln als Text darzustellen. Die *toString*-Methode gibt wie üblich den Aufbau des Objektes zurück, die *zeigen*-Methode eine lesbare Version der Formel.

```
System.out.println(formel);
System.out.println(formel.zeigen());
```

Die zugehörige Ausgabe sieht wie folgt aus.

```
Formel [typ=UND, operanden=[Atom [name=a, typ=ATOM,
operanden=[]], Atom [name=b, typ=ATOM, operanden=[]]]]
(a AND b)
```

Jede Klasse hat zusätzlich eine Variable *typ* der Aufzählung *Typ*, die den genauen Typen der Formel beinhaltet. Das ist in Java eigentlich unnötig, macht aber die *toString*-Ausgabe etwas lesbarer und vereinfacht auch die Umsetzung dieses Programms in einer anderen Programmiersprache (was z. B. in Ihrer Hausarbeit als Erweiterung dieser Aufgabe in Java oder einer anderen Sprache erfolgen könnte).

a) Realisieren Sie die Details der Klassen *Folgt* (Ausgabe: `(a => b)`) und *GenauDannWenn* (Ausgabe: `(a <=> b)`), dabei können Sie für die Methode evaluieren zunächst die gegebene falsche Implementierung stehen lassen. Die Tests der Klasse *test.ImplikationsTest* sollen laufen.

Als Design-Entscheidung wird festgelegt, dass alle Atome mit dem gleichen Namen die identische Variable bezeichnen, Variablen sind damit über ihren Namen eindeutig. Mit der Methode *alleAtome()* wird eine Liste der Namen der Atome ohne Doppelte berechnet.
Mit einer Interpretation werden Variablen Boolesche Werte zugeordnet. Die geschieht durch Realisierungen des Interfaces *Interpretation*.

b) Realisieren Sie das Interface *Interpretation* in der ohne sinnvolle Implementierung gegebenen Klasse *InterpretationImpl*. Setzen Sie im nächsten Schritt die Methoden *evaluieren(Interpretation)* in allen von Formel abgeleiteten Klassen um. Da dies induktiv über den Aufbau passiert, ist es sinnvoll mit *Atom* zu beginnen. Die Tests der Klasse *test.InterpretationTest* sollen laufen.
c) Statt nur einzelne Interpretation zu betrachten, kann es hilfreich sein die Menge aller Interpretationen zu betrachten. Erinnern Sie sich, dass es zu n Variablen genau 2^n mögliche Interpretationen gibt. Durch das Ausprobieren aller möglichen

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

Interpretationen für eine Formel kann z. B. berechnet werden, ob es sich um eine Tautologie handelt.
Realisieren Sie zur Umsetzung der Ideen das Interface *Interpretation**en*** in der bereits vorhandenen Klasse *Interpretation**en**Impl*. Am Ende sollen alle Tests der Klasse *test.Interpretation**en**Test* laufen.
(Hinweis: Die Umsetzung basiert darauf, dass mit der Methode *alleInterpretationen(anzahl)* eine Liste mit allen Interpretationen für anzahl Variablen berechnet werden. Dabei ist eine Interpretation als eine Liste von anzahl Booleschen Werten (*List<Boolean>*) zu realisieren. Das Gesamtergebnis des Aufrufs ist dann eine Liste mit allen möglichen Interpretationen, die Liste hat dann den Typ *List<List<Boolean>>* und enthält 2^{anzahl} Elemente. Ein etwas formatierte *toString*-Ausgabe von *ienimpl.alleInterpretationen(3)* sieht wie folgt aus und zweigt alle 8 möglichen Interpretationen von drei Variablen:

```
[[false, false, false], [false, false, true], [false, true, false],
 [false, true, true],   [true, false, false], [true, false, true],
 [true, true, false],   [true, true, true]]
```

Um die Berechnung dieser Mengen nicht immer wieder neu durchführen zu müssen, ist es sinnvoll sich diese Ergebnisse, z. B. in einer Map zu merken.
Natürlich können Sie Ihre Klasse Interpretation bei Berechnungen nutzen.
Die Methode *wertetabelle(Formel)* wird nicht getestet, soll aber für eine übergebene Formel eine ordentlich formatierte Wertetabelle ausgeben. Zu dem in der *main*-Methode von *InterpretationenImpl* Code:

```
Formel f = new Folgt(new Atom("a1")
                   , new Folgt(new Atom("a2"), new Atom("a3")));
ienimpl.wertetabelle(f);
```

Sollte das Ergebnis in etwa wie folgt aussehen.

```
(a1 => (a2 => a3))
a1|a2|a3|ergebnis
--+--+--+--------
0 |0 |0 | 1
0 |0 |1 | 1
0 |1 |0 | 1
0 |1 |1 | 1
1 |0 |0 | 1
1 |0 |1 | 1
1 |1 |0 | 0
1 |1 |1 | 1
```