UNIVERSITÄT ZU LÜBECK
INSTITUTE FOR SOFTWARE ENGINEERING
AND PROGRAMMING LANGUAGES

isp

Software Engineering im Wintersemester 2021/2022
Prof. Dr. Martin Leucker, Malte Schmitz, Stefan Benox, Julian Schulz, Benedikt Stepanek, Friederike Weilbeer, Tom Wetterich

Übungszettel 9 (Lösungsvorschlag)

12.01.2022

Abgabe bis Donnerstag, 13. Januar um 23:59 Uhr online im Moodle.

Dieser Übungszettel hat wegen der Weihnachtspause eine längere Deadline und entsprechend mehr Aufgaben. Nehmen Sie sich ausreichend Zeit und bearbeiten Sie diesen Zettel gründlich!

Aufgabe 9.1: Model-View-Controller

Machen Sie diese Aufgabe gründlich und stellen Sie sicher, dass der Code auf Ihrem Computer funktioniert. Die Themen dieser Aufgabe sind eine zentrale Vorbereitung auf das Praktikum im nächsten Semester! Wir werden im Laufe des Semesters auf dieses Beispiel zurückkommen.

8 Punkte, schwer

1. Strukturieren Sie Ihr Programm aus Aufgabe 7.3 so um, dass die grafischer Nutzerschnittstelle (GUI) und das Kommandozeileninterface (CLI) parallel genutzt werden können. Zur Erinnerung: Das Programm nimmt drei Integer-Zahlen entgegen. Jede Zahl soll dabei als Seitenlänge eines Dreiecks verstanden werden. Das Programm gibt aus, ob das Dreieck mit diesen drei Seitenlängen gleichschenklig, gleichseitig oder ungleichseitig ist.

   Die GUI soll die drei Zahlen in drei mit *a*, *b* und *c* beschrifteten Eingabefeldern entgegen nehmen und die Ausgabe in einem Label anzeigen.

   Die CLI nimmt Befehle zum setzen einer Kantenlänge entgegen. So setzt zum Beispiel der Befehl `a = 7` die Länge der Kante *a* auf 7.

   GUI und CLI können parallel verwendet werden und zeigen die gleichen Daten stets synchron an, unabhängig davon, ob diese über die GUI oder das CLI geändert wurden. Bei jeder Änderung der Daten gibt die CLI die Ausgabe des Programms erneut aus.

   Halten Sie sich bei der Realisierung an das MVC-Pattern, bestehend aus Observer, Strategy und Composite.

   Nutzen Sie Maven als Build-System und Java-Version 11 als Source- und Target-Version, sowie das Maven plugin for JavaFX, sodass sich ihr Projekt mit `mvn javafx:run` starten und mit `mvn javafx:jlink` bauen lässt.

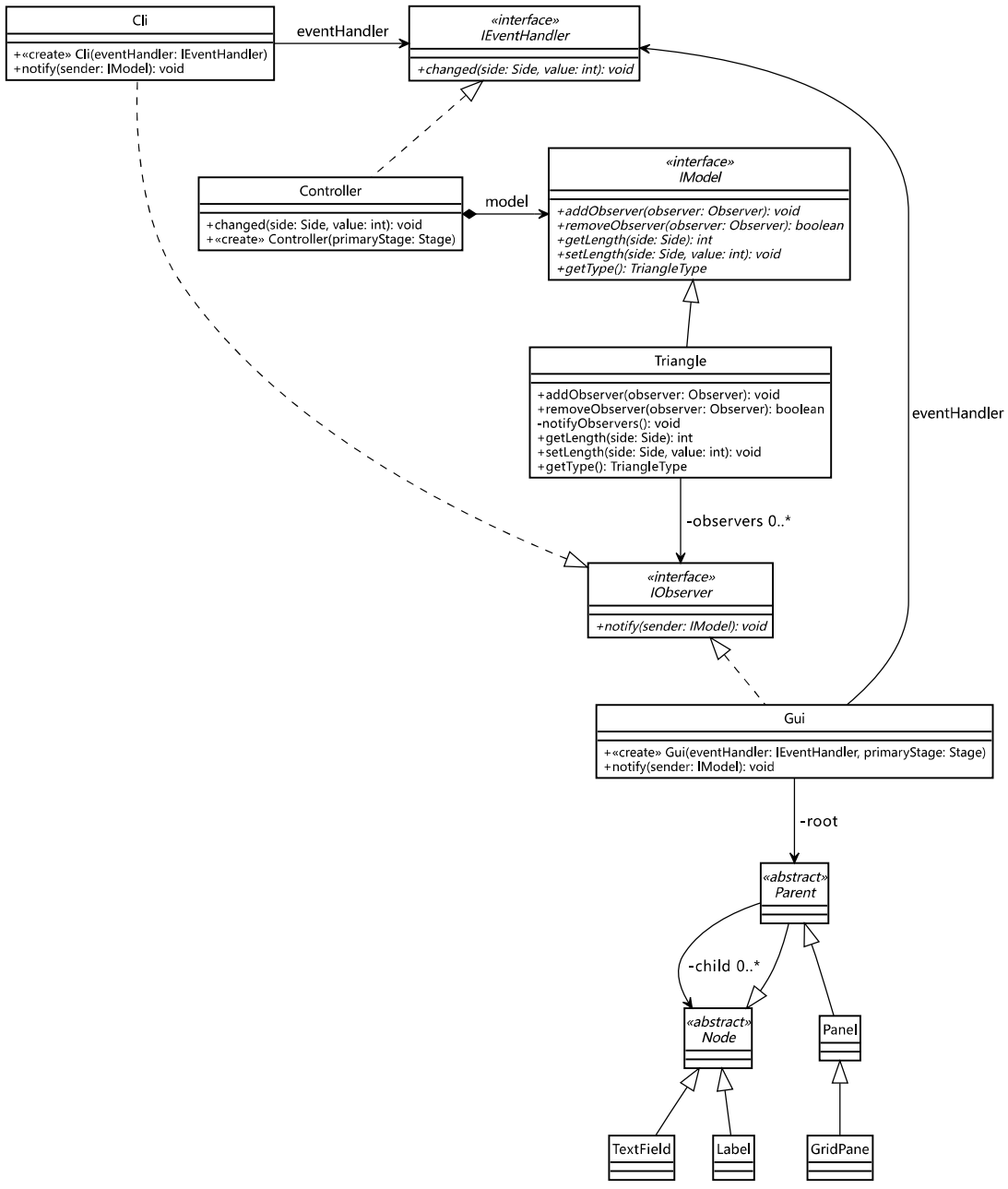
Reichen Sie Ihren Code als Zip-Archiv im Moodle ein. Achten Sie dabei darauf, tatsächlich nur den Quellcode und keine gebauten Artefakte mit abzugeben.

▼ **Lösungsvorschlag**

Siehe `triangle-mvc.zip`.

2. Erläutern Sie unter Verwendung eines Klassendiagramms, wie die einzelnen Bestandteile des MVC-Patterns angewandt wurden.

▼ **Lösungsvorschlag**



▶ Quelltext des Diagramms

Das Observer-Pattern wird für die Kommunikation von Model (also der Klasse `Triangle`) und View (also den Klassen `Gui` und `Cli`) verwendet: Die Views implementieren das Interface `IObserver`, das eine Methode `notify` zur Verfügung stellt. Hierüber benachrichtigt das Model die Views, wenn

sich Werte im Model geändert haben. Die Views werden bei der Initialisierung über die Methode `addObserver` am Model registriert.

Das Strategy-Pattern wird für die restliche Kommunikation verwendet: Die Views rufen Event-Handler über das Interface `IEventHandler` im Controller auf. Die Views kennen den Controller nur über dieses Interface. Die Views kennen das Model nur über das Interface `IModel` und lesen die Werte des Models über dieses Interface aus. Der Controller interagiert mit dem Model ebenfalls durch dieses Interface. Durch die Verwendung von klaren Schnittstellen, bleiben die Verantwortlichkeiten sauber getrennt und die einfache Austauschbarkeit der einzelnen Elemente wird gewährleistet.

Die GUI verwendet das Composite-Pattern für verschachtelte View-Komponenten: Eine sehr vereinfachte Hierarchie der JavaFX-Komponenten wird am unteren Ende des Klassendiagramms dargestellt. Die View kennt ein Root-Element, das ein Parent ist. Dieser Parent kann beliebig viele Nodes enthalten, ist aber selber ebenfalls ein Node.

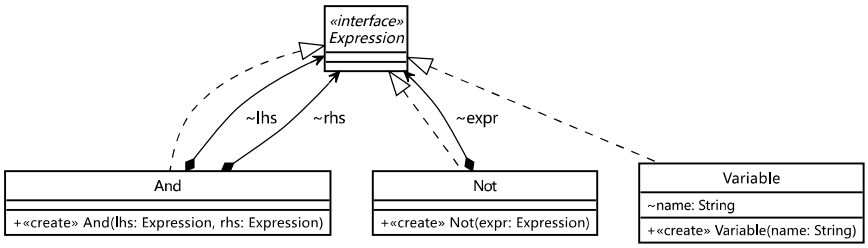
Aufgabe 9.2: Visitor-Pattern

9 Punkte, mittel

Wir bauen in dieser Aufgabe eine Datenstruktur für aussagenlogische Formeln so um, dass diese das Visitor-Pattern verwendet, um die Formeln auszuwerten.

Es geht in dieser Aufgabe nur um die Architektur der Datenstruktur. *Sie geben also keinen Quellcode ab.* Es kann allerdings durchaus zum Verständnis hilfreich sein, die beschriebenen Datenstrukturen und Methoden tatsächlich umzusetzen.

Die Datenstruktur wird durch folgendes Klassendiagramm beschrieben:



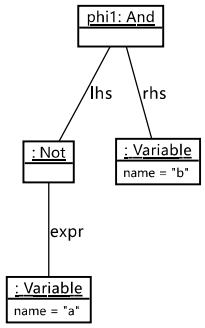
▶ Quelltext des Diagramms

1. Die Formel $\varphi_1 = \neg a \wedge b$ kann in dieser Datenstruktur dargestellt werden als:

```
Expression phi1 = new And(new Not(new Variable("a")), new Variable("b"));
```

   Zeichnen Sie ein Objektdiagramm von `phi1`. (1 Punkt)

▼ Lösungsvorschlag



▶ Quelltext des Diagramms

2. Formeln, die weitere aussagenlogische Operatoren enthalten, können unter Anwendung der De-morgansche Gesetze in äquivalente Formeln überführt werden, die in dieser Datenstruktur repräsentiert werden können. Formen Sie die folgenden beiden Formeln entsprechend um und geben Sie Konstruktor-Aufrufe in Java an, die diese Formeln in der Datenstruktur darstellen (1 Punkt):

$$\varphi_2 = (p \wedge q) \vee r$$
$$\varphi_3 = (a \wedge b) \vee \neg a \vee \neg b$$

▼ **Lösungsvorschlag**

$$\varphi_2 \equiv \neg(\neg(p \wedge q) \wedge \neg r)$$
$$\varphi_3 \equiv \neg(\neg(a \wedge b) \wedge a \wedge b)$$

```
Expression phi2 = new Not(new And(
    new Not(new And(
        new Variable("p"),
        new Variable("q"))),
    new Not(new Variable("r"))));
Expression phi3 = new Not(new And(
    new And(
        new Not(new And(
            new Variable("a"),
            new Variable("b"))),
        new Variable("a")),
    new Variable("b")));
```

3. Wir wollen diese Datenstruktur nun zur Anwendung des Visitor-Patterns vorbereiten. Wir definieren dazu ein generisches Interface `ExpressionVisitor` mit je einer eigenen Methode für die Klassen `And`, `Not` und `Variable`. Diese Methoden haben alle einen vollständig generischen Rückgabetyp, der durch Spezialisierung des Interfaces konkretisiert wird. Für die Klasse `And` sieht die Signatur der Methode zum Beispiel so aus: `T visitAnd(And and)`.

   Wir ergänzen im Interface `Expression` die generische Methode `<T> T accept(ExpressionVisitor<T> visitor)` und implementieren diese als generische Methode in den Klassen `And`, `Not` und `Variable`.

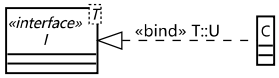
   Die String-Serialisierung für die Datenstruktur funktioniert wie folgt: Für Variablen wird der Name der Variable ausgegeben. Der Operator `And` wird als Serialisierung des linken Operanden gefolgt von `&&` gefolgt von der Serialisierung des rechten Operanden ausgegeben. Die Negation wird als `!(` gefolgt von der Serialisierung des Operanden und `)` ausgegeben. Die Serialisierung von `phi1` ist demnach `!(a) && b`.

Die String-Serialisierung ist in einer Klasse `Printer` realisiert, die das Interface `ExpressionVisitor<String>` implementiert. Darüber hinaus hat die Klasse eine öffentliche statische Methode `String print(Expression expr)`, die eine Instanz der Klasse `Printer` anlegt und diese der Methode `accept` von `expr` übergibt.

Die Auswertung für die Datenstruktur funktioniert ähnlich: Eine aussagenlogische Formel wird für eine Variablenbelegung ausgewertet. Eine Variablenbelegung ist eine Menge von Variablen, die zu true auswerten. Eine solche Menge wird auch Welt genannt. Eine Variable wertet zu true aus, wenn sie in der gegebenen Welt vorhanden ist und sonst zu false. Eine Negation, also `Not`, negiert die Auswertung ihres Operanden. Eine Konjunktion, also `And`, wertet zu true aus, wenn beide Operanden zu true auswerten und sonst zu false. Die Auswertung von φ_1 für die Welt $\{a, b\}$ ist false und für die Welt $\{b\}$ true. Wir notieren dies als $\{a, b\} \not\models \varphi_1$ und $\{b\} \models \varphi_1$.

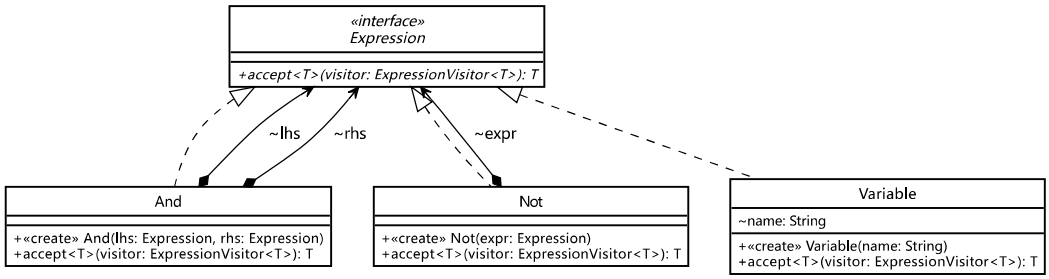
Die Auswertung ist in einer Klasse `Evaluator` realisiert, die das Interface `ExpressionVisitor<Boolean>` implementiert. Der Konstruktor nimmt eine Welt, also eine Menge von Variablen, als `Set<String>` und speichert diese im privaten lokalen Feld `world`. Darüber hinaus hat die Klasse eine öffentliche statische Methode `evaluate(Expression expr, Set<String> world)`, die eine Instanz der Klasse `Evaluator` mit der gegebenen `world` anlegt und diese der Methode `accept` von `expr` übergibt.

Zeichnen Sie ein aktualisiertes Klassendiagramm der Klassen und Schnittstellen `Expression`, `And`, `Not`, `Variable`, `ExpressionVisitor`, `Evaluator` und `Printer`. Sie können dabei wahlweise ignorieren, dass `ExpressionVisitor` generisch ist, oder folgende Notation verwenden, um einen generische Typparameter `T` einer Schnittstelle `I` sowie die Konkretisierung des Typparameters auf einen konkreten Typ `U` bei einer von `I` erbenden Klasse `C` darzustellen (4 Punkte):

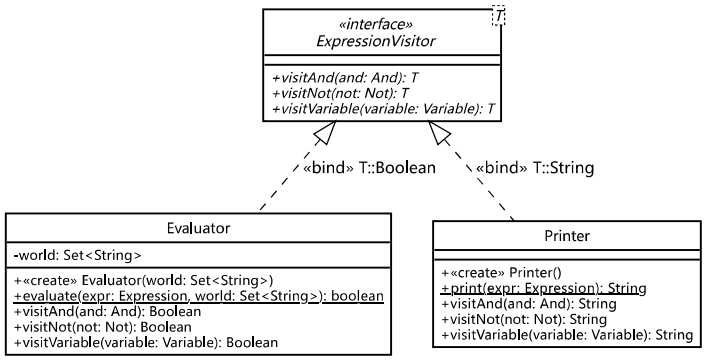


▶ Quelltext des Diagramms

▼ **Lösungsvorschlag**



▶ Quelltext des Diagramms



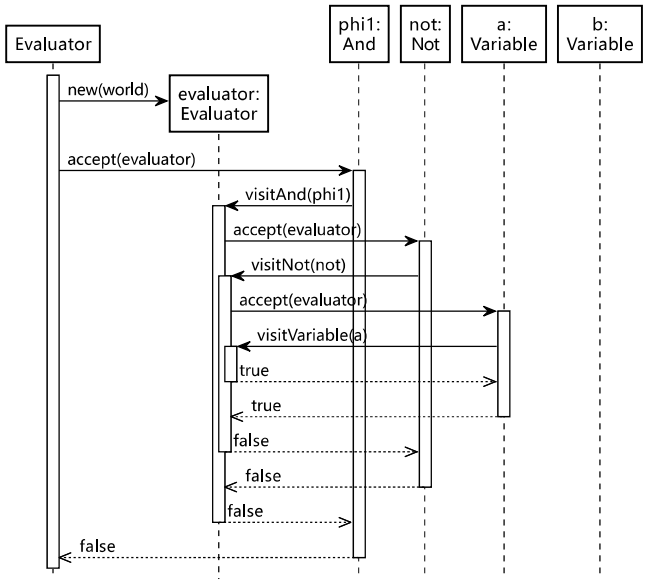
▶ Quelltext des Diagramms

4. Zeichnen Sie ein Sequenzdiagramm und erläutern Sie in Ihren eigenen Worten, wie das Visitor-Pattern funktioniert an folgendem Beispiel (3 Punkte):

```
Expression phi1 = new And(new Not(new Variable("a")), new Variable("b"));
Set<String> world = Set.of("a", "b");
Evaluator.evaluate(phi1, world);
```

Das Sequenzdiagramm soll die Lebenslinien der statischen Klasse `Evaluator`, des neu erzeugte Objekts `evaluator: Evaluator` und der Objekte `phi1: And`, `not: Not`, `a: Variable` und `b: Variable` darstellen.

▼ Lösungsvorschlag

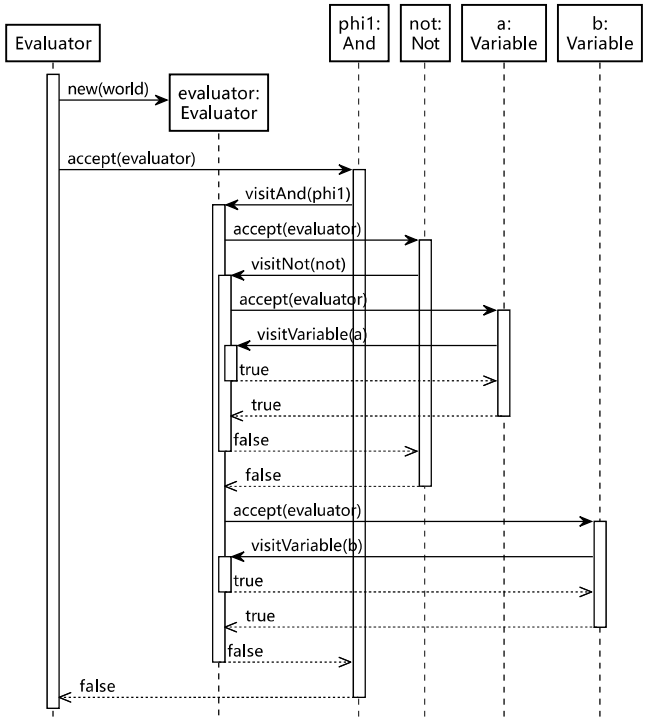


▶ Quelltext des Diagramms

Man erkennt im Sequenzdiagramm, wie der Aufruf der Methode `accept` auf einem Objekt mit dem Visitor `evaluator` als Argument jeweils dazu führt, dass die entsprechende Methode `visit...` auf dem `evaluator` mit dem passenden Objekt als Argument aufgerufen wird. Das Visitor-Patter führt also dazu, dass die eigentliche Arbeit immer in entsprechenden Methoden des Visitors passiert, die allerdings jeweils indirekt durch die Objekte der Datenstruktur aufgerufen werden.

Durch die Kurzschlussauswertung ist die Auswertung von `phi1: And` bereits nach der Auswertung des ersten Operanden zu `false` beendet, da die Konjunktion nur zu `false` auswerten kann, wenn einer der Operanden

zu `false` auswertet. Der zweite Operand `b: Variable` wird also nie besucht. Bei anderen Implementierungen ohne Kurzschlussauswertung würde sich folgendes Sequenzdiagramm ergeben:

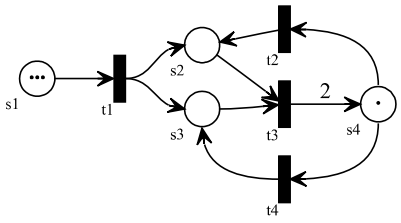


▶ Quelltext des Diagramms

Aufgabe 9.3: Eigenschaften eines Petri-Netzes

3 Punkte, mittel

Es sei folgendes Petri-Netz gegeben:



▶ Quelltext des Diagramms

1. Ist das Petri-Netz lebendig? Begründen Sie die Antwort.

▼ **Lösungsvorschlag**

Nein, dieses Petri-Netz ist nicht lebendig. Ein Petri-Netz ist lebendig, wenn für jede initial erreichbare Markierung, und für jede Transition t immer eine Markierung erreichbar ist unter der t aktiviert wird. In diesem Petri-Netz kann t1 ausgehend von der initialen Markierung genau drei mal schalten. Danach ist keine Markierung mehr erreichbar, unter der t1 schalten kann.

2. Besitzt das Petri-Netz eine Verklemmung? Begründen Sie die Antwort.

▼ **Lösungsvorschlag**

Ja, dieses Petri-Netz besitzt eine Verklemmung. Eine Verklemmung ist eine initial erreichbare Konfiguration, unter der keine Transition aktiviert ist. Wenn in diesem Petri-Netz ausgehend von der initialen Markierung drei mal t1, dann drei mal t3 und schließlich sieben mal t4 schaltet, dann ist eine Markierung erreicht, unter der keine Transition aktiviert ist.

Die Existenz einer Verklemmung ist ein weiteres Argument, wieso dieses Petri-Netz nicht lebendig ist. Umgekehrt wäre das kein Argument: Aus der Abwesenheit einer Verklemmung folgt nicht zwingend die Lebendigkeit des Petri-Netzes, denn die Abwesenheit der Verklemmung bedeutet lediglich, dass immer mindestens *eine Transition* aktiviert sein muss, während die Lebendigkeit verlangt, dass für *jede Transition* eine Markierung erreichbar sein muss, unter der diese Transition aktiviert ist.

3. Ist der Erreichbarkeitsgraph des Petri-Netzes endlich groß? Begründen Sie die Antwort.

▼ **Lösungsvorschlag**

Ja, der Erreichbarkeitsgraph dieses Petri-Netzes ist endlich, da insgesamt maximal sieben Token im Petri-Netz existieren können. Die Transition t1 ist die einzige Transition, welche mehr Token generiert als konsumiert. Die Transition t1 kann maximal drei mal schalten. Danach existieren sieben Token, deren Anzahl durch die übrigen Transitionen nicht verändert wird. Entsprechend gibt es nur endlich viele mögliche Markierungen. Die Markierungen entsprechen den Knoten im Erreichbarkeitsgraphen, sodass dieser ebenfalls endlich ist.