SE1, Aufgabenblatt 3

Softwareentwicklung I – Wintersemester 2011/12

Syntaktische Struktur von Java, Entwurf eigener Klassen, Fallunterscheidung

MIN-CommSy-URL: ..... https://www.mincommsy.uni-hamburg.de/

Projektraum: ..... SE-1 CommSy WiSe 11/12

Ausgabedatum: ..... 03. November 2011

Kernbegriffe

Der Zustand eines Exemplars ist durch seine *Zustandsfelder* (kurz: *Felder*, engl.: fields) definiert. Mit Hilfe einer speziellen Operation – die *Konstruktor* (engl.: constructor) genannt wird - werden die Felder eines frisch erzeugten Exemplars in einen definierten Anfangszustand gebracht.

Die Methoden einer Klasse definieren das *Verhalten* ihrer Exemplare. Methoden werden unterschieden in *verändernde* und *sondierende Methoden* (engl.: mutator and accessor methods). Verändernde Methoden ändern den Zustand eines Exemplars, sondierende Methoden fragen den Zustand lediglich ab, ohne ihn zu verändern.

Die *Syntax* einer Programmiersprache gibt vor, wie der Quelltext aufgebaut sein darf. Die Syntax wird üblicherweise mit einer sogenannten *kontextfreien Grammatik* (engl.: context free grammar) definiert. Wie die Grammatik einer natürlichen Sprache gibt sie vor, wie Wörter zu Sätzen kombiniert werden dürfen. Die Java-Syntax legt u.a. die Struktur von Klassendefinitionen, Methoden und Anweisungen fest, beispielsweise, dass eine einfache Zuweisung in Java mit einem Semikolon abgeschlossen wird (siehe Kapitel 18, http://java.sun.com/docs/books/jls/). Eine kontextfreie Grammatik besteht aus einer Menge von *Ableitungsregeln*, die immer eine linke und eine rechte Seite haben. Eine Ableitungsregel besagt, dass das Element auf der linken Seite (ein *Nichtterminal,* engl.: non-terminal) ersetzt werden kann durch die rechte Seite. Die rechte Seite einer Ableitungsregel kann aus Nichtterminalen und *Terminalen* (engl.: terminal) bestehen. Aus Sicht der Grammatik sind Terminale Basisbausteine (konkrete Zeichenfolgen, die nicht weiter abgeleitet werden müssen), während für jedes Nichtterminal eine Regel existieren muss. Jede Grammatik hat ein festgelegtes Nichtterminal, das immer als Startpunkt für Ableitungen dient (das *Startsymbol*). Am Ende einer Ableitung steht immer eine Folge von Terminalen. Eine weit verbreitete Notation für die Syntax von Programmiersprachen ist die *EBNF* (erweiterte Backus-Naur-Form).

Damit sich ein Java-Quelltext übersetzen lässt, muss er der Java-Syntax entsprechen. Zusätzlich muss er weitere Regeln befolgen, die nicht in der Syntax ausgedrückt werden. Eine solche Regel besagt z.B., dass Variablen zuerst deklariert werden müssen, bevor man sie benutzen kann.

In Java gibt es den Basisdatentyp `boolean`, der nur die *booleschen Werte* `true` (wahr) und `false` (falsch) definiert, sowie *logische Operationen* auf diesen Werten. Ausdrücke, die als Ergebnis einen booleschen Wert liefern, heißen *boolesche Ausdrücke*.

Sprachmechanismen, die die Reihenfolge der Ausführung von Anweisungen eines Programms steuern, heißen *Kontrollstrukturen* (engl.: control structures). Neben der trivialen Kontrollstruktur *Sequenz* (Anweisungen werden zur Laufzeit in der Reihenfolge ausgeführt, in der sie im Quelltext stehen) gibt es die *Fallunterscheidung*. Bei der Fallunterscheidung mit einer *if-Anweisung* wird abhängig von einer *Bedingung* (dem Ergebnis eines booleschen Ausdrucks) eine Anweisung ausgeführt oder nicht. Ein Beispiel für eine if-Anweisung findet ihr in den Quelltextkonventionen.

Neben Variablen mit änderbaren Werten gibt es auch *Konstanten*. Einer Konstanten wird genau einmal bei ihrer Initialisierung ein Wert zugewiesen, der danach unveränderlich ist. Um Variablen als Konstanten zu deklarieren, wird in Java das Schlüsselwort `final` verwendet. Felder, die als Konstanten deklariert werden, nennt man *Exemplarkonstanten*. Exemplarkonstanten können im Konstruktor initialisiert werden, danach lassen sie sich nicht mehr ändern.

Ein *Block* (engl.: block) fasst eine Sequenz von Anweisungen mit geschweiften Klammern zusammen. Ein Block kann überall dort verwendet werden, wo auch eine einzelne Anweisung stehen kann. In einem Block können lokale Variablen deklariert werden.

Lernziele

Syntaxbeschreibungen in EBNF lesen können; neue Klasse definieren können; Exemplare im Konstruktor initialisieren können; Objektzustände mit Feldern modellieren können; Zustände über Methoden verändern und sondieren können; Fallunterscheidungen programmieren können.

Aufgabe 3.1 Der Klassiker: das Konto

3.1.1 Erzeugt ein neues Projekt mit dem Titel *Bank* und speichert es in eurem Verzeichnis. Erzeugt darin eine Klasse `Konto`, die Bankkonten modellieren soll.

3.1.2 Der Zustand eines Kontos soll durch einen *Saldo* definiert sein, der den aktuellen Kontostand angibt. Definiert ein Feld vom Typ `int` für den Saldo. Verändert dazu die Definition des von BlueJ vorgegebenen Beispielfeldes und sorgt für eine korrekte Initialisierung.

3.1.3 Definiert zwei Methoden, `void zahleEin(int)` und `void hebeAb(int)`, die den richtigen Kontostand berechnen und den Zustand entsprechend verändern. Überprüft, ob eure Methoden funktionieren, indem ihr in BlueJ den Zustand der Exemplare vor und nach der Ausführung der Methoden betrachtet.

3.1.4 Der Kontostand soll nun von außen abgefragt werden können (z.B. interaktiv in BlueJ). Implementiert eine Methode, die die Belegung des Feldes zurückliefert. Hinweis: Verwendet als Anhaltspunkt die Beispielmethode.

3.1.5 Welche Methoden des Kontos sind sondierende Methoden, welche verändernde?

3.1.6 Damit auf den ersten Blick erkennbar ist, wer die Klasse `Konto` programmiert hat und was modelliert wird, sollt ihr nun den Kommentar am Beginn des Quelltextes verändern und eine Beschreibung der Klasse einfügen. Wechselt im Editor die Ansicht von „Implementierung“ auf „Schnittstelle“. Wird euer Kommentar sichtbar? Wenn nein, warum nicht? Wenn ja, warum?

3.1.7 Die Bank will neue Kunden werben und spendiert jedem Kontoeröffner 10 Euro. Implementiert einen Konstruktor, der den Saldo eines neu erzeugten Kontos auf diesen Betrag setzt. Hinweis: Der vorgegebene Konstruktor muss nur verändert werden.

3.1.8 Jedes Konto soll eine Kontonummer erhalten, die nur beim Anlegen des Kontos gesetzt werden kann. Implementiert dies mit Hilfe einer Exemplarkonstanten. Warum muss es für die Kontonummer auch eine sondierende Methode geben?

3.1.9 Bereitet euren Quelltext für die Abnahme vor. Im CommSy liegen hierfür die Quelltextkonventionen in einem PDF-Dokument bereit, das ihr euch durchlesen solltet. Diese Quelltextkonventionen beschreiben sinnvolle Regeln zur Vereinheitlichung und guten Lesbarkeit von Programmtext**. Ab jetzt gilt: Nur Quelltexte, die diesen Regeln entsprechen, werden von Betreuern und Betreuerinnen abgenommen!**

Aufgabe 3.2 Digitale Waagen

Heutzutage gibt es digitale Körpergewicht-Waagen zum Schleuderpreis. Um in diesem Markt etwas Besonderes bieten zu können, wollen wir eine Java-Klasse schreiben, deren Exemplare in einer modernen Waage zum Einsatz kommen sollen und den Besitzer über seinen Fortschritt bei der Gewichtskontrolle informieren.

3.2.1 Legt ein neues Projekt *Fitness* an und darin eine neue Klasse `Waage`. Diese soll über einen Konstruktor verfügen, der das aktuelle Körpergewicht der Person als Parameter mit dem Typ `int` entgegennimmt und in einer Exemplarvariablen `_letztesGewicht` hinterlegt. Macht Euch Gedanken darüber, in welcher Einheit ihr das Gewicht speichern wollt, beispielsweise Gramm oder Kilogramm. Macht dies für Klienten deutlich, indem ihr **wie immer entsprechende Schnittstellen-Kommentare** schreibt!

3.2.2 Schreibt eine Methode `void registriere(int neuesGewicht)`. Diese wird jedes Mal aufgerufen, wenn der Besitzer sich erneut wiegt. Als Parameter bekommt sie das Ergebnis der physischen Messung der Waage übergeben.

3.2.3 In der Methode `registriere` soll weiterhin festgestellt werden, ob sich das Gewicht seit der letzten Messung verändert hat. Diesen Trend sollt ihr im Zustand des Exemplars festhalten. Implementiert anschließend eine parameterlose Methode `gibTrend` mit dem Ergebnistyp `int`, welche folgendes zurückgeben soll:

-1, falls der Besitzer leichter geworden ist

+1, falls er schwerer geworden ist

0 sonst

3.2.4 Implementiert zwei weitere parameterlose Methoden `gibMinimalgewicht` und `gibMaximalgewicht`, die als Ergebnis vom Typ `int` die extremen Messwerte der bisherigen Messreihe eines Objekts zurückgeben.

3.2.5 *Zusatzaufgabe*: Implementiert eine parameterlose Methode `gibDurchschnittsgewicht`, die das durchschnittliche Gewicht über alle Messungen eines Objekts bildet und diesen als Ergebnistyp `int` zurückgibt. **Herausforderung**: Verwendet zur Berechnung nur bisher in SE1 bekannte Konzepte.

Aufgabe 3.3 Syntax definiert mit der EBNF

Die folgende Grammatik in der EBNF mit dem Startsymbol *Ausdruck* definiert die Syntax für einfache arithmetische Ausdrücke mit ganzen Zahlen (da es hier farbig wird, solltet ihr euch das Blatt besser im CommSy ansehen):

Ausdruck:
IntegerLiteral **{** *InfixOp IntegerLiteral* **}**

InfixOp:
+
-
*****
/

IntegerLiteral:
[*Vorzeichen* **]** *Ziffern*

Vorzeichen:
-

Ziffern:
Ziffer
Ziffern Ziffer

Ziffer:
0
1
2
3
4
5
6
7
8
9

3.3.1 Was bedeutet es, wenn es zu einem Nichtterminal mehrere Zeilen gibt (beispielsweise für das Nichtterminal *Ziffern*)? Welche Semantik haben die geschweiften und eckigen Klammern? Erläutert dies mündlich bei der Abnahme.

3.3.2 Überprüft die folgenden arithmetischen Ausdrücke **schriftlich** auf ihre syntaktische Korrektheit bezüglich dieser Grammatik, indem ihr versucht, die Ausdrücke ausgehend vom Startsymbol *Ausdruck* zu konstruieren. Dazu müsst ihr mehrfach die links stehenden Nichtterminale durch eine der auf den Doppelpunkt folgenden Zeilen ersetzt. Diesen Vorgang nennt man Ableitung.

Beispiel **30:**

Ausdruck -> *IntegerLiteral {InfixOp IntegerLiteral}*
-> *[Vorzeichen] Ziffern*
-> Ziffern Ziffer
-> Ziffer 0
-> 30 **Lässt sich ableiten.**

Hinweis: Ein einfacher Text-Editor ist ein sehr gutes Hilfsmittel hierfür.

Einige Ausdrücke lassen sich nicht ableiten. Erläutert anhand der Grammatik, warum nicht.

(20 – 1) * 2

2 – 7 + 8

–500

3 + –8

+42

– –17

Aufgabe 3.4 Grammatik für Java Level 1

3.4.1 Die auf dem Zusatzblatt gegebenen EBNF-Regeln (siehe auch *JavaLevel01-Syntax.pdf* im CommSy) beschreiben die Syntax für eine vereinfachte Variante von Java (Java Level 1). Auf den ersten Aufgabenblättern beschränken wir uns auf diese Untermenge des Java-Sprachumfangs. Die Notation entspricht der üblichen Notation für die Java-Syntax in der „offiziellen“ Sprachdefinition (siehe Kapitel 18, http://java.sun.com/docs/books/jls/).

Überprüft den folgenden Quelltext auf seine syntaktische Korrektheit, indem ihr ihn, ausgehend von der Regel für das Startsymbol *CompilationUnit*, ableitet. Eine solche Ableitung wird in ähnlicher Weise implizit durchgeführt, wenn der Compiler einen Quelltext übersetzt.
Beispiel für den Beginn einer Ableitung (die Terminale des abgeleiteten Quelltextes sind zur Übersichtlichkeit unterstrichen):

CompilationUnit -> NormalClassDeclaration

-> class *Identifier ClassBody*

-> class Vertrag *ClassBody*

-> class Vertrag { *{ ClassBodyDeclaration }* }

-> class Vertrag { *ClassBodyDeclaration*

ClassBodyDeclaration }

Hinweis: Ein *Identifier* ist ein *Bezeichner* (ein Name wie „Vertrag“ in diesem Beispiel) und ist aus Sicht der Grammatik ein terminales Symbol (nicht weiter ableitbar). Das gleiche gilt für Literale wie den int-Wert `512` oder den String `"Banane"`.

Leitet den folgenden Quelltext vollständig in der angegebenen Weise ab:

```
class Vertrag
{
    private int _vertragsnummer;

    public int gibVertragsnummer()
    {
        return _vertragsnummer;
    }
}
```

Benutzt einen beliebigen Texteditor, damit ihr bei einem Fehler leichter korrigieren könnt. Bei jedem Ableitungsschritt solltet ihr nur genau eine Ersetzung vornehmen: Entweder ersetzt ihr ein Nichtterminal durch eine seiner möglichen Alternativen oder ihr ersetzt Metasymbole wie die geschweiften Klammern der EBNF.

3.4.2 Leitet auf Basis der gegebenen Grammatik eine syntaktisch minimale Java-Klasse ab. Hinweis: Wir bezeichnen eine Klasse als syntaktisch minimal, wenn die Ableitung den kürzesten Pfad durch die Grammatik beschreibt.

3.4.3 *Zusatzaufgabe*: Lässt sich folgende Klassendefinition ableiten?

```
class B
{
    int i = true;
}
```

Wenn nein, warum nicht? Wenn ja, wie lässt sich die Variablendeklaration ableiten?