Prof. Dr. Thomas Breuel
Dipl.-Inf. Patrick Michel (AG ST)
Dipl.-Ing. Ludwig Schmidt-Hackenberg

TU Kaiserslautern
Fachbereich Informatik
AG Bildverstehen und Mustererkennung

Übungsblatt 13: Software-Entwicklung 1 (WS 2010/11)

Ausgabe: in der Woche vom 31.01. bis zum 04.02.11
Abgabe: in der Woche vom 07.02. bis zum 11.02.11
Abnahme: max. zwei Tage nach der Übung

Aufgabe 1 Dynamisches Binden und Casts (Präsenzaufgabe)

Betrachten Sie folgendes Programm.

```
1  interface I1 {
2    public char mi (A a);
3  }
4
5  interface I2 extends I1 {
6    public char mii (I2 a);
7  }
8
9  interface A {
10   public char m();
11 }
12
13 interface B extends A, I1 {
14   public char n(A a);
15 }
16
17 class C implements B, I2 {
18   public char m() { return 'C'; };
19   public void o(I1 i) { ... };
20   ...
21 }
```

```
23 class E implements A {
24   public char m() { return 'E'; };
25   public char n(A a) { return a.m(); };
26   public void o(I1 i) { ... };
27
28   public static void main (String[] args) {
29     A a1 = new E();
30     A a2 = new C();
31     ...
32     char r1 = a1.m();
33     char r2 = a2.m();
34     char r3 = ((E) a1).n(a2);
35     char r4 = ((B) a2).m();
36     ... // beliebiger anderer Code
37     a1.o(a2);
38     a1.o(new C());
39     ...
40   }
41 }
```

a) Welche Methoden müssen die Klassen `C` und `E` implementieren?

b) Von welchem Typ können `s` und `t` in den folgenden Ausdrücken sein?
   `new E().n(s)`
   `new C().o(t)`

c) Welchen Wert enthalten die Variablen `r1` bis `r4` nach der Ausführung von Zeile 35?

d) Zeile 36 kann beliebigen Code enthalten. Ist der Aufruf `a1.o(a2)` typkorrekt? Falls nein, verändern Sie ihn so, dass er statisch typkorrekt ist und zur Laufzeit keine Casts fehlschlagen.

e) Betrachten Sie nun Zeile 38. Fügen Sie der angegebenen Klassen- und Interface-Hierarchie einen Typ hinzu, so dass Sie den Aufruf `a1.o(new C())` einfacher korrigieren können als in der vorhergehenden Teilaufgabe.

   Hinweis: Passen Sie die Deklaration von `a1` geeignet an.

f) Welche Methoden können in Zeile 31 auf den Referenzen `a1` und `a2` aufgerufen werden, wenn keine Casts erlaubt sind? Welcher Code wird jeweils ausgeführt? Welche Casts sind statisch erlaubt und wie ändert sich die Menge der aufrufbaren Methoden?

Aufgabe 2 Klassenattribute und -methoden (Einreichaufgabe)

a) Machen Sie sich mit der Schnittstelle der Klasse Math vertraut (Beschreibung auf http://java.sun.com/javase/6/docs/api/java/lang/Math.html. Begründen Sie, warum statische Attribute / Methoden verwendet wurden.

b) Warum kann folgendes Programm nicht kompiliert werden? Schreiben Sie es so um, dass es kompiliert.

```
class Main {
  int wert;
  public static void main(String[] args) {
    Main m = new Main();
    Main.wert = 7;
    System.out.println(wert);
  }
}
```

c) Implementieren Sie eine Klasse `Count`, die eine statische Methode `print()` besitzt, welche ausgibt, wie viele Objekte der Klasse `Count` bis dato instanziiert worden sind.

Aufgabe 3 Parametrische Typen (Einreichaufgabe)

Implementieren Sie eine *objektbasierte*, *parametrische* Fassung von einfach verketteten Listen. Definieren Sie dazu eine Klasse `LinkedList` – deren Instanzen je eine komplette Liste repräsentieren – die Methoden analog zu denen aus Aufgabe 2 von Blatt 9 in ihrer Schnittstelle enthält.[1] Verwenden Sie eine weitere Klasse `Entry`, um die einzelnen Elemente zu speichern und zu verketteten. Ihre Implementierung soll parametrisch im Elementtyp sein, so dass man die Listen in vielen verschiedenen Kontexten wiederverwenden kann. Stellen Sie außerdem sicher, dass Ihre Implementierung ohne Warnungen übersetzt!

Hinweis: Um alle Warnungen angezeigt zu bekommen müssen Sie den Java Compiler mit der Option `-Xlint` aufrufen, also z.B. `javac -Xlint LinkedList.java`.

Aufgabe 4 Mensapläne in Java (Präsenzaufgabe)

Wir betrachten noch einmal die Datentypdefinition für die Mensapläne von Übungsblatt 5.

```
data Tag = MO | DI | MI | DO | FR                deriving (Eq, Ord, Show)
data Ausgabe = Essen1 | Essen2 | Grill | Wok     deriving (Eq, Ord, Show)
data Angebot = Angebot Tag Ausgabe String        deriving (Eq, Ord, Show)

type Mensaplan = [Angebot]
```

a) Implementieren Sie eine Interface- und Klassenhierarchie für den Datentyp `Mensaplan`.

b) Erweitern Sie die Klasse `Mensaplan` um eine Methode `Mensaplan suche(String text)`, die zu einem Suchtext einen gefilterten Plan zurückgibt, der nur noch Angebote beinhaltet, die den Suchtext enthalten.

Hinweis: Sie können die Methode `contains` auf einem `String` verwenden, um zu testen, ob ein String in einem anderen enthalten ist: Bsp. `titel.contains(text)`

[1] Die Methode `create` wird in Ihrer objektbasierten Fassung natürlich – wenn überhaupt – zu einem Konstruktor. Die relevanten Methoden sind also `append`, `size`, `get`, `put`, `remove` und `contains`.

Aufgabe 5 The Expression Problem (Einreichaufgabe)

In dieser Aufgabe wollen wir einen Mini-Interpreter sowohl in Haskell als auch in Java entwickeln, sowie Vor- und Nachteile der Implementierungen diskutieren. Wir betrachten die Sprache, die durch folgende Datentyp-Deklaration implizit definiert wird:

```
data Expr = Const Integer | Var String | Add Expr Expr
  deriving (Eq, Ord, Show)
```

a) Realisieren Sie die beiden Haskell-Funktionen `depth :: Expr -> Integer` und `eval :: Env -> Expr -> Integer` , wobei `depth` die Tiefe eines Ausdrucksbaums bestimmt und `eval` einen Ausdruck in einer Umgebung auswertet. Für die Umgebung realisieren Sie ein Modul `Environment` mit der folgenden Schnittstelle. Sie können Ihre Lösung für das Modul `Dictionary` dafür so gut wie übernehmen.

```
module Environment (Env, emptyEnv, insert, lookUp) where

emptyEnv :: Env
insert :: String -> Integer -> Env -> Env
lookUp :: String -> Env -> Maybe Integer
```

b) Im Vergleich zur Implementierung in Haskell wollen wir nun eine alternative Java-Implementierung erstellen. Dazu verwenden wir die Eigenschaft, dass die *ist-ein*-Relation sich sehr gut durch Subtypenbildung in Java beschreiben lässt.

```
interface Expr {
    int depth();
    int eval(Environment e);
}


class Const implements Expr {
    int value;
    public int depth() { ... }
    public int eval(Environment e) { ... }
}

class Var ...
class Add ...
```

Die Klasse `Environment` ist auf der Vorlesungsseite verfügbar und besitzt folgende Signatur:

```
class Environment {
    int lookup(String name);
    void insert(String name, int value);
}
```

Implementieren Sie nun nach dem oben angegebenen Schema die Methoden `depth` und `eval`.

c) Wir wollen nun die Vor- und Nachteile der Haskell bzw. Java-Implementierung diskutieren. Überlegen Sie sich dazu, welche Anpassungen Sie am Code durchführen müssen, wenn Sie

- ihre Sprache um ein Multiplikationskonstrukt erweitern: `data Expr = ... | Mult Expr Expr`
- eine neue Funktion `countConst :: Expr -> Integer` hinzufügen, welche die Anzahl an Konstanten in einem Ausdruck zählt.

d) Wir wollen nun eine alternative Lösung in Java entwickeln, die es erlaubt, elegant neue Methoden zu definieren, ohne die unterschiedlichen Klassen (`Const`, `Var`, `Add`, ...) anzupassen. Wir wollen dazu das Entwurfsmuster *Visitor* verwenden. Über das Entwurfsmuster können Sie sich z.B. auf Wikipedia (http://de.wikipedia.org/wiki/Visitor) informieren.

Wir definieren nun erneut unsere Typen, ohne jedoch die Methoden schon anzugeben. Unsere Operationen können dann als einzelne Klassen realisiert werden.

```
interface Expr {
  void accept(ExprVisitor v);
}

class Const implements Expr {
  int value;

  public void accept(ExprVisitor v) {
    v.visit(this);
  }
}

class Var ...
class Add ...
```

```
interface ExprVisitor {
  void visit(Const c);
  void visit(Add a);
  void visit(Var v);
}

class DepthVisitor implements ExprVisitor {
  int i;

  public void visit(Const c) { i = 1; }
  public void visit(Var v) { ... }
  public void visit(Add a) { ... }
}

class EvalVisitor ...
```

Ihre `main` Methode hat nun z.B. folgende Form:

```
Expr e = ... // irgendein sinnvoller Ausdruck
DepthVisitor dv = new DepthVisitor();
e.accept(dv);
System.out.println(dv.i); // gibt das Ergebnis (Tiefe von Ausdruck e) aus
```

Implementieren Sie nun noch einmal Aufgabenteil b) und c) nach dem vorgestellten Muster. Welche Vor- und Nachteile besitzt diese Implementierung?

e) (*freiwillige Zusatzaufgabe*) Es wäre praktisch, wenn die Methode `accept` das Ergebnis sofort liefern würde, anstatt auf den internen Zustand des Visitors zugreifen zu müssen. Definieren Sie `ExprVisitor` so, dass die Methode `accept` einen parametrischen Rückgabetypen hat. Implementieren Sie Ihre Lösung noch einmal. Wie sieht es mit einem zusätzlichen parametrischen Parameter- und Exceptiontyp aus?

Aufgabe 6 Umsetzung von UML Diagrammen (Einreichaufgabe)

Im Folgenden modellieren Sie eine Stadt bestehend aus Stadtteilen, Straßen, Häusern und einem Park. Stadtteile können direkt an andere Stadtteile angrenzen. Straßen führen durch Stadtteile und eventuell auch am Park entlang. Häuser liegen in einem Stadtteil und liegen entweder an einer Straße oder einer Kreuzung von zwei Straßen. Der Park liegt in einem Stadtteil.

a) Modellieren Sie diese Stadt als Klassendiagramm mit Beziehungen. Wählen Sie gute Namen für die Beziehungen und legen Sie sinnvolle Multiplizitäten fest.

b) Nun verfeinern wir den Entwurf. Schreiben Sie Java-Klassen die zeigen, wie Sie die Beziehung zwischen Stadtteil und Haus mit Attributen realisieren würden.

Entwerfen Sie außerdem vier unterschiedliche Realisierungen der Beziehung zwischen Stadtteil und Straße. Was sind die Unterschiedlichen Vor- und Nachteile der Varianten. Wovon würden Sie die Entscheidung für eine der Varianten abhängig machen?