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

TU Kaiserslautern
Fachbereich Informatik
AG Bildverstehen und Mustererkennung

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

Die Hinweise und Vorschläge in diesem Dokument sollen der Lösungsfindung dienen und erheben demnach weder Anspruch auf Vollständigkeit noch Korrektheit. Sollten Sie Fehler finden, würden wir uns freuen, wenn Sie uns diese mitteilen. (Kontaktdaten finden Sie auf unserer Webpräsenz.)

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?

Es sind keine weiteren Methoden für die Klasse `E` erforderlich.

```
class C implements B, I2 {
  // bereits vorhanden
  public char m() { return 'C'; };  // aus A
  public void o(I1 i) { ... }

  // es fehlen noch
  public char mi (A a) { ... };     // aus I1
  public char mii (I2 a) { ... };   // aus I2
  public char n(A a) { ... };       // aus B
}
```

b) Von welchem Typ können `s` und `t` in den folgenden Ausdrücken sein?

```
new E().n(s)
new C().o(t)
```

Der Typ von `s` kann `A` oder ein Subtyp von `A` sein: `A, B, C, E`. Der Typ von `t` kann `I1` oder ein Subtyp davon sein: `I1, I2, B, C`.

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

```
r1 == 'E';
r2 == 'C';
r3 == 'C';
r4 == 'C';
```

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.

Der Ausdruck ist nicht typkorrekt, da `a1` vom Typ `A` ist, für diesen Typ jedoch keine Methode `o` deklariert ist und der Parameter von `o` in beiden möglichen Implementierungen von Typ `I1` ist und nicht von `A` wie im angegebenen Aufruf.

Es gibt Subtypen von `A`, für die `o` deklariert ist, in diesem Fall kann man `a1` entsprechend casten und `o` aufrufen. Außerdem muss der Parameter `a2` zu `I1` oder einem Subtyp davon gecastet werden.

Damit auch zur Laufzeit keine Fehler auftreten, muss die Gültigkeit der Casts vorher überprüft werden.

```
if (a1 instanceof C && a2 instanceof I1) {
  ((C) a1).o((I1) a2);
} else if (a1 instanceof E) && a2 instanceof I1) {
  ((E) a1).o((I1) a2);
}
```

Obwohl es in diesem Fall einen gemeinsamen Subtyp von `A` und `I1` gibt, nämlich `B`, kann es noch weitere Klassen mit dieser Eigenschaft geben, die nicht Subtyp von `B` sind! Deshalb sollte man nicht auf `B` casten, obwohl dies in dem hier dargestellten Code unproblematisch wäre.

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.

Ein zusätzliches Interface `D` als Subtyp von `A` mit der Methodendeklaration `void o(I1 i)`, welches von `C` und `E` implementiert wird. Dadurch kann `a1` den Typ `D` erhalten und braucht nicht mehr auf den Typ geprüft und gecastet zu werden.

`D` muss ein Subtyp von `A` sein, damit `a1` sowohl als `A`, als auch als `D` verwendet werden kann.

```
interface D extends A {
  void o(I1 i);
}
class C implements B, I2, D { ... }
class E implements A, D {

  ...
  D a1 = new E();
  ...

}
```

Da `D` ein Subtyp von `A` ist, braucht `E` nun nicht mehr explizit `A` zu implementieren.

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?

Referenz	Cast auf	Methode	Code in Zeilen
a1	-	m()	24
	Downcast auf E	m()	24
		n(A a)	25
		o(I1 i)	26
a2	-	m()	19
a2	Downcast auf C	m()	19
		n(A a)	Implementierung von n in C
		o(I1 i)	20
		mii(I2 a)	Implementierung von mii in C
		mi(A a)	Implementierung von mi in C
a2	Downcast auf B	m()	19
		n(A a)	Implementierung von n in C
		mi(A a)	Implementierung von mi in C
a2	Cast auf I1	mi(A a)	Implementierung von mi in C
a2	Cast auf I2	mi(A a)	Implementierung von mi in C
		mii(I2 a)	Implementierung von mii in C
a2	Downcast auf E	m()	Exception zur Laufzeit
		n(A a)	Exception zur Laufzeit
		o(I1 i)	Exception zur Laufzeit

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.

Beim Design der Java-API wurde unter anderem die Entscheidung getroffen, typische mathematische Funktionen (`abs`, `cos`, `sin` usw.) und Konstanten (π und e) in der Klasse `Math` zusammen zu fassen. Mathematische Funktionen sind natürlich zustandslos, weshalb die Implementierung in Form *statischer Methoden* erfolgen kann. Da sich auch Konstanten prinzipbedingt nicht verändern, ist die einmalige Definition vollkommen ausreichend, weshalb diese in der Klasse `Math` mit Hilfe *statischer Attribute* realisiert wurden. Angenehme Folge dieser Entscheidungen ist, dass man kein `Math`-Objekt erzeugen muss, um Funktionen oder Konstanten zu verwenden.

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);
  }
}
```

Die Variable `wert` ist ein normales Attribut, kein Klassenattribut, d.h. es existiert einmal pro Objekt der Klasse. Trotzdem wird versucht aus einem statischen Kontext, d.h. ohne implizite Referenz auf ein Objekt, auf dieses Attribut zuzugreifen. Es wird ebenfalls keine explizite Referenz verwendet.

Eine Möglichkeit ist das Attribut zu einem Klassenattribut zu machen:

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

Eine zweite Möglichkeit ist das zuvor erzeugte Objekt `m` zu benutzen:

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

```
class Count {
    static int i = 0;

    Count() { i++; }

    static void print() {
        System.out.println(i + " Objekte der Klasse Count wurden instanziiert");
    }
}
```

Aufgabe 3 Parametrische Typen (Einreichaufgabe)

Implementieren Sie eine *objektbasierte*, *parametrische* Fassung von einfach verketteten Listen.

```
import java.util.NoSuchElementException;

class Entry<T> {
  T element;
  Entry<T> next;

  Entry() {
    next = this;
  }
```

```
  Entry(T element, Entry<T> next) {
    this.element = element;
    this.next = next;
  }
}

public class LinkedList<T> {
  Entry<T> header = new Entry<T>();
  int size = 0;

  void append(T elem) {
    Entry<T> cur = header;
    while (cur.next != header) cur = cur.next;
    cur.next = new Entry<T>(elem, header);
    size++;
  }

  int size() {
    return size;
  }

  T get(int i) {
    if (i < 0) throw new NoSuchElementException();
    Entry<T> cur = header;
    do {
      cur = cur.next;
      if (cur == header) throw new NoSuchElementException();
    } while (i-- > 0);
    return cur.element;
  }

  void put(int i, T elem) {
    if (i < 0) throw new NoSuchElementException();
    Entry<T> cur = header;
    while (i-- > 0) {
      cur = cur.next;
      if (cur == header) throw new NoSuchElementException();
    }
    cur.next = new Entry<T>(elem, cur.next);
    size++;
  }

  void remove(int i) {
    if (i < 0) throw new NoSuchElementException();
    Entry<T> cur = header;
    while (i-- > 0) {
      cur = cur.next;
      if (cur == header) throw new NoSuchElementException();
    }
    if (cur.next == header) throw new NoSuchElementException();
    cur.next = cur.next.next;
    size--;
  }

  boolean contains(T elem) {
    Entry<T> cur = header;
    while (cur.next != header) {
      cur = cur.next;
      if (cur.element.equals(elem)) return true;
    }
    return false;
  }
}
```

Aufgabe 4 Mensapläne in Java (Präsenzaufgabe)

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.

```
// Datentyp Tag und Ausgabe als Interface, mit einer implementierenden Klasse
// pro Konstruktor. Die Methode 'name' ist notwendig, um (ohne Reflection)
// die Ausgabe sinnvoll implementieren zu koennen.
interface Tag { String name(); }

class MO implements Tag { public String name() { return "Montag";     }}
class DI implements Tag { public String name() { return "Dienstag";   }}
class MI implements Tag { public String name() { return "Mittwoch";   }}
class DO implements Tag { public String name() { return "Donnerstag"; }}
class FR implements Tag { public String name() { return "Freitag";    }}

interface Ausgabe { String name(); }

class Essen1 implements Ausgabe { public String name() { return "Essen1"; }}
class Essen2 implements Ausgabe { public String name() { return "Essen2"; }}
class Grill  implements Ausgabe { public String name() { return "Grill";  }}
class Wok    implements Ausgabe { public String name() { return "Wok";    }}

// Der Datentyp Angebot hat nur einen Konstruktor, ist also eine einfache
// Struktur. Damit reicht es hier eine Klasse mit entsprechenden Attributen
// zu verwenden.
class Angebot {
  Tag tag;
  Ausgabe ausgabe;
  String titel;

  // Um die Initialisierung zu vereinfachen geben wir explizit einen
  // Konstruktor an.
  public Angebot(Tag tag, Ausgabe ausgabe, String titel) {
    this.tag = tag;
    this.ausgabe = ausgabe;
    this.titel = titel;
  }

  // Die 'print' Methode verwendet 'name' der beiden Enumerationstypen.
  public void print() {
    System.out.println(tag.name() + "/" + ausgabe.name() + ": " + titel);
  }
}

// Ein Mensaplan wird hier jetzt als verlinkte Liste umgesetzt. Ebenfalls
// moeglich waeren hier Felder oder eine der Collection Klassen der API.
class Mensaplan {
  Angebot angebot;
  Mensaplan next;

  // Auch hier fuer einfache Initialisierung ein passender Konstruktor.
  public Mensaplan(Angebot angebot, Mensaplan next) {
    this.angebot = angebot;
    this.next = next;
  }

  // Die Ausgabe erfolgt rekursiv entlang der Liste von Angeboten.
  public void print() {
    angebot.print();
    if (next != null) next.print();
  }

  // Die Suche ist hier jetzt auch rekursiv definiert. Bei jedem Element
  // fallen zwei Unterscheidungen an; Ist das Angebot eins der gesuchten und
  // gibt es noch einen Nachfolger.
  public Mensaplan suche(String text) {
    if (angebot.titel.contains(text)) {
      if (next != null) {
        return new Mensaplan(angebot, next.suche(text));
      } else {
        return new Mensaplan(angebot, null);
```

```
      }
    } else if (next != null) {
      return next.suche(text);
    } else {
      return null;
    }
  }
}


public class MensaNormal {
  public static void main(String... args) {
    Mensaplan plan = null;

    plan = new Mensaplan(new Angebot(new MO(), new Essen1(), "SchniPoSa"), plan);
    plan = new Mensaplan(new Angebot(new DI(), new Grill(), "Schnitzel"), plan);

    plan.suche("Schnitzel").print();
  }
}
```

In Java gibt es die Möglichkeit simple Enumerationstypen wesentlich einfacher zu definieren. Dafür ersetzen wir die Schnittstellen `Tag` und `Ausgabe`, zusammen mit ihren implementierenden Klassen, durch zwei Enumerationstypen. Es ändert sich lediglich die Erzeugung im Hauptprogramm und die Ausgabe in `Angebot`:

```
enum Tag {
  MO, DI, MI, DO, FR
}

enum Ausgabe {
  Essen1, Essen2, Grill, Wok
}

// in Angebot:
public void print() {
  System.out.println(tag + "/" + ausgabe + ": " + titel);
}

// in main:
plan = new Mensaplan(new Angebot(Tag.Montag, Ausgabe.Essen1, "SchniPoSa"), plan);
plan = new Mensaplan(new Angebot(Tag.Dienstag, Ausgabe.Grill, "Schnitzel"), plan);
```

Aufgabe 5 The Expression Problem (Einreichaufgabe)

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.

`Environment` Implementierung:

```
type Env = [(String, Integer)]

emptyEnv = []
lookUp = lookup
insert s i e = (s, i) : e
```

Implementierung der beiden Funktionen:

```
depth :: Expr -> Integer
depth (Add a b) = 1 + max (depth a) (depth b)
depth _ = 1

eval :: Env -> Expr -> Integer
eval _ (Const i) = i
eval e (Add a b) = eval e a + eval e b
eval e (Var s) = case lookUp s e of
  Nothing -> error $ "unbound variable " ++ s
  Just i -> i
```

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`.

```
class Const implements Expr {
  int value;

  public Const(int value) {
    this.value = value;
  }

  public int depth() {
    return 1;
  }
```

```
  public int eval(Environment e) {
    return value;
  }
}

class Var implements Expr {
  String name;

  public Var(String name) {
    this.name = name;
  }

  public int depth() {
    return 1;
  }

  public int eval(Environment e) {
    return e.lookup(name);
  }
}

class Add implements Expr {
  Expr e1;
  Expr e2;

  public Add(Expr e1, Expr e2) {
    this.e1 = e1;
    this.e2 = e2;
  }

  public int depth() {
    return 1 + Math.max(e1.depth(), e2.depth());
  }

  public int eval(Environment e) {
    return e1.eval(e) + e2.eval(e);
  }
}
```

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`
  - Haskell: Alle existierenden Funktionen (`depth`, `eval`) müssen angepasst werden.
  - Java: Neue Klasse `Mult` mit entsprechenden Methoden muss angelegt werden, sonst keine Anpassungen nötig.
- eine neue Funktion `countConst :: Expr -> Integer` hinzufügen, welche die Anzahl an Konstanten in einem Ausdruck zählt.
  - Haskell: Neue Funktion definieren, ansonsten muss nichts angepasst werden.
  - Java: In jeder Klasse muss die neue Methode implementiert werden.

Zur Erklärung ein Auszug aus (http://www.daimi.au.dk/~madst/tool/papers/expression.txt):

The Expression Problem (by Philip Wadler, 12 November 1998)

The Expression Problem is a new name for an old problem. The goal is to define a datatype by cases, where one can add new cases to the datatype and new functions over the datatype, without recompiling existing code, and while retaining static type safety (e.g., no casts). For the concrete example, we take expressions as the data type, begin with one case (constants) and one function (evaluators), then add one more construct (plus) and one more function (conversion to a string).

Whether a language can solve the Expression Problem is a salient indicator of its capacity for expression. One can

think of cases as rows and functions as columns in a table. In a functional language, the rows are fixed (cases in a datatype declaration) but it is easy to add new columns (functions). In an object-oriented language, the columns are fixed (methods in a class declaration) but it is easy to add new rows (subclasses). We want to make it easy to add either rows or columns.

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?

```
class DepthVisitor implements ExprVisitor {
  int i;

  public void visit(Const c) {
    i = 1;
  }

  public void visit(Add a) {
    a.e1.accept(this);
    int temp = i;
    a.e2.accept(this);
    i = 1 + Math.max(temp, i);
  }

  public void visit(Var v) {
    i = 1;
  }
}

class EvalVisitor implements ExprVisitor {
  int i;
  Environment e;

  public EvalVisitor(Environment e) {
    this.e = e;
```

```
  }

  public void visit(Const c) {
    i = c.value;
  }

  public void visit(Add a) {
    a.e1.accept(this);
    int temp = i;
    a.e2.accept(this);
    i += temp;
  }

  public void visit(Var v) {
    i = e.lookup(v.name);
  }
}
```

Vorteile und Nachteile wie die der Haskell Lösung.

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?

```
interface Expr {
  <T> T accept(ExprVisitor<T> v);
}

interface ExprVisitor<T> {
  T visit(Const c);
  T visit(Add a);
  T visit(Var v);
}
```

```
class Const implements Expr {
  int value;

  public Const(int value) {
    this.value = value;
  }

  public <T> T accept(ExprVisitor<T> v) {
    return v.visit(this);
  }
}

class Var implements Expr {

  ...

...

class DepthVisitor implements ExprVisitor<Integer> {
  public Integer visit(Const c) {
    return 1;
  }

  public Integer visit(Add a) {
    return 1 + Math.max(a.e1.accept(this), a.e2.accept(this));
  }

  public Integer visit(Var v) {
```

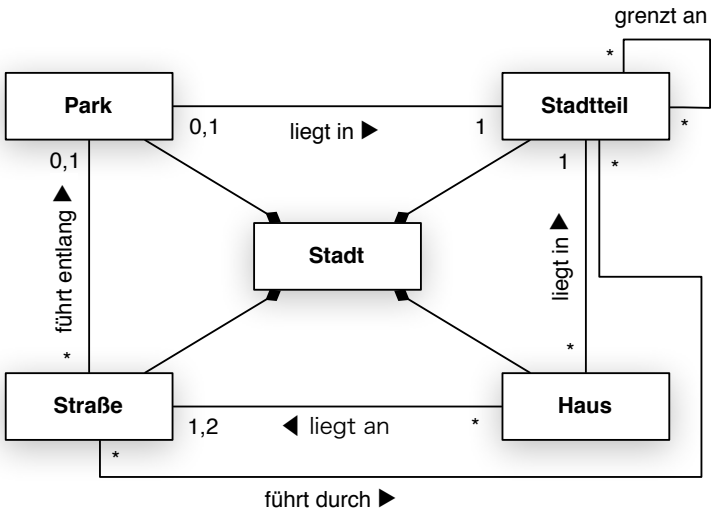
```
    return 1;
  }
}

class EvalVisitor implements ExprVisitor<Integer> {
  Environment e;

  public EvalVisitor(Environment e) {
    this.e = e;
  }

  public Integer visit(Const c) {
    return c.value;
  }

  public Integer visit(Add a) {
    return a.e1.accept(this) + a.e2.accept(this);
  }

  public Integer visit(Var v) {
    return e.lookup(v.name);
  }
}

public class ExprVisitGeneric {
  public static void main(String... args) {
    Expr test = new Add(new Add(new Const(1), new Const(2)), new Const(3));

    System.out.println(test.accept(new DepthVisitor()));
    System.out.println(test.accept(new EvalVisitor(new Environment())));
  }
}
```

Parameter- und Exception-Typ des Visitors können auch parametrisiert werden.

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.

Bei allen Varianten wurden hier Arrays verwendet, natürlich sind Listen oder andere *Collections* genauso zulässig. Realisierung der Beziehung zwischen Haus und Stadtteil:

```
class Stadteil { .. }
class Haus { Stadtteil liegtIn; .. }
```

1. Variante (Redundant):

   ```
   class Stadtteil { Strasse[] enthaelt;  ..  }
   class Strasse { Stadtteil[] fuehrtDurch; .. }
   ```

   Vorteile: In beide Richtungen einfacher Test, ob eine Beziehung besteht.

   Nachteile: Aufwändig bei Änderungen; höherer Speicherverbrauch; Redundanz.

2. Variante (Asymmetrisch):

   ```
   class Stadtteil { Strasse[] enthaelt; .. }
   class Strasse { .. }
   ```

   Vorteile: Einfache Abfrage, welche Straßen ein fester Stadtteil enthält, keine Redundanz.

   Nachteile: Schwierig abzufragen, durch welche Stadtteile eine Straße führt.

3. Variante (Asymmetrisch 2):

   ```
   class Stadtteil { .. }
   class Strasse { Stadtteil[] fuehrtDurch; .. }
   ```

   Vorteile: Einfache Abfrage, durch welche Stadtteile eine Straße führt, keine Redundanz.

   Nachteile: Schwierig abzufragen, welche Straßen ein fester Stadtteil enthält.

4. Variante (Extern):

   Hier werden die Verbindungen als Liste von Paaren dargestellt – eine andere Art der Graphdarstellung, beispielsweise durch Adjazenzmatrizen, kann natürlich auch gewählt werden. Wichtig ist zu erkennen, dass Assoziationen auch an anderer Stelle gespeichert werden können.

   ```
   class Stadtteil { .. }
   class Strasse { .. }
   class Paar { Strasse strasse; Stadtteil teil; }
   class Stadt { Paar[] fuehrtDurch; .. }
   ```

   Vorteile: Beide Richtungen sind einfach abzufragen und zu ändern, keine Redundanz.

   Nachteile: Zugriff auf `Stadt` notwendig.