Prof. Dr. J. Giesl
D. Cloerkes, S. Dollase, N. Lommen, D. Meier, F. Meyer

Tutoraufgabe 1 (Überblickswissen):

a) Wie kann die Nutzung von Interfaces dabei helfen, die Entwicklung eines größeren Programms auf mehrere Entwickler*innen zu verteilen?

b) Welches Problem kann auftreten, wenn man zu viele `default`-Implementierungen in Interfaces nutzt?

c) Warum sind `default`-Implementierungen in Interfaces manchmal dennoch sinnvoll?

Lösung:

a) Angenommen, Entwicklerin Olga ist dafür zuständig, die Klasse `O` zu entwickeln, während John die Klasse `J` entwickeln soll. Beide Klassen sollten gleichzeitig entwickelt werden, damit das Projekt nicht unnötig in die Länge gezogen wird. Nun ist es aber so, dass die Klasse `O` Objekte und Methoden der Klasse `J` nutzt. Eigentlich muss also `J` vor `O` fertiggestellt sein, damit `O` `J` nutzen kann. Dies lässt sich umgehen, indem für `J` ein Interface `IJ` entwickelt wird, welches alle von `O` genutzten Methoden von `J` enthält. So kann die Implementierung von `IJ`, nämlich `J`, gleichzeitig zum Nutzer von `IJ`, nämlich `O`, entwickelt werden.

Außerdem kann `J` nun weitere öffentliche Methoden enthalten, welche für `O` jedoch unsichtbar sind, da diese Methoden nicht in `IJ` enthalten sind. Die Nutzung eines Interfaces hilft also auch dabei, unwichtige Implementierungsdetails zu verstecken und trägt so dazu bei, "verschiedene Programmteile" voneinander zu entkoppeln.

b) Betrachten Sie den folgenden Code:

```
public interface A1 {
    default String greet() {
        return "Hallo";
    }
}

public interface A2 {
    default String greet() {
        return "Moin";
    }
}

public class B implements A1, A2 {
}
```

Hier ist vollkommen unklar, ob der Ausdruck `new B().greet()` den `String`-Wert `"Hallo"` oder den `String`-Wert `"Moin"` zurückgeben soll. Tatsächlich generiert Java für diesen Code einen Compilerfehler, welcher explizit vom Entwickler behoben werden muss, etwa durch folgenden Code, welcher dafür sorgt, dass der Ausdruck `new B().greet()` den `String`-Wert `"Hallo"` zurückgibt.

```
public class B implements A1, A2 {
    @Override
    public String greet() {
        return A1.super.greet();
    }
}
```

Dieses Problem kann auftreten, da in Java die Mehrfachvererbung von Interfaces zulässig ist. Eine Klasse kann also mehrere Interfaces implementieren. Die `default`-Implementierungen in Interfaces sollten also möglichst sparsam genutzt werden.

c) Ein Problem bei der Nutzung von Interfaces ist, dass diese nur schwer geändert werden können, sobald andere Entwicklergruppen dieses Interface in ihrem Code nutzen oder implementieren. Wenn sie es nutzen, so können keine Methoden mehr entfernt werden. Wenn Sie es implementieren, so können keine Methoden mehr hinzugefügt werden. Die Signatur einer Methode zu ändern, ist in beiden Fällen nicht mehr möglich.

Manchmal ist es jedoch bei veröffentlichten Interfaces so, dass sich erst nach der Veröffentlichung herausstellt, dass viele Entwickler*innen eine bestimmte Funktionalität häufig nutzen und diese daher selbst manuell immer wieder nachprogrammieren. Diese Funktionalität ist zwar vollständig mit den angebotenen Methoden des Interfaces umsetzbar, erfordert jedoch einige Zeilen Code, in denen mehrere verschiedene Methodenaufrufe des Interfaces richtig miteinander kombiniert werden. Dieser Code muss jedes mal erneut geschrieben werden. Viel einfacher wäre es, diese Funktionalität durch eine Methode direkt am Interface anzubieten, sodass diese einfach aufgerufen werden kann. Diese Methode würde dann die Arbeit übernehmen, die gewünschte Funktionalität mithilfe der richtig miteinander kombinierten Methodenaufrufe umzusetzen. Dies würde den Entwickler*innen viel Arbeit sparen und unnötige Copy and Paste Fehler vermeiden. Dies nennt man auch *interface evolution*.

Genau für diesen Fall wurden `default`-Implementierungen in Interfaces eingeführt[1]. Es kann nach der Veröffentlichung eines Interfaces eine Methode hinzugefügt werden. Da es bereits Implementierungen des Interfaces bei anderen Entwicklergruppen geben könnte, in denen die neue Methode nicht überschrieben wird, muss eine `default`-Implementierung der neuen Methode mitgeliefert werden, welche ausschließlich auf bereits bestehende Methoden des Interfaces zurückgreift. Bei Bedarf können die Klassen, welche das Interface implementieren, später die `default`-Implementierung des Interfaces durch Überschreiben ersetzen.

Tutoraufgabe 2 (Überschreiben, Überladen und Verdecken (Video)):

Betrachten Sie die folgenden Klassen:

Listing 1: X.java

```
1  public class X {
2      public int a = 23;
3
4      public X(int a) {                          // Signatur: X(I)
5          this.a = a;
6      }
7
8      public X(float x) {                        // Signatur: X(F)
9          this((int) (x + 1));
10     }
11
12     public void f(int  i, X o) { }             // Signatur: X.f(IX)
13     public void f(long lo, Y o) { }            // Signatur: X.f(LY)
14     public void f(long lo, X o) { }            // Signatur: X.f(LX)
15 }
```

Listing 2: Y.java

```
1  public class Y extends X {
2      public float a = 42;
3
4      public Y(double a) {                       // Signatur: Y(D)
5          this((float) (a - 1));
6      }
7
8      public Y(float a) {                        // Signatur: Y(F)
9          super(a);
10         this.a = a;
11     }
12
13     public void f(int i, X o) { }              // Signatur: Y.f(IX)
14     public void f(int i, Y o) { }              // Signatur: Y.f(IY)
15     public void f(long lo, X o) { }            // Signatur: Y.f(LX)
16 }
```

[1] `https://stackoverflow.com/a/28684917/3888450`

Listing 3: Z.java

```
1  public class Z {
2      public static void main(String [] args) {
3                                                   // a)
4          X xx1 = new X(42);                       // (1)
5          System.out.println("X.a: " + xx1.a);
6          X xx2 = new X(22.99f);                   // (2)
7          System.out.println("X.a: " + xx2.a);
8          X xy = new Y(7.5);                       // (3)
9          System.out.println("X.a: " + ((X) xy).a);
10         System.out.println("Y.a: " + ((Y) xy).a);
11         Y yy = new Y(7);                         // (4)
12         System.out.println("X.a: " + ((X) yy).a);
13         System.out.println("Y.a: " + ((Y) yy).a);
14                                                  // b)
15         int i = 1;
16         long lo = 2;
17         xx1.f(i, xy);                            // (1)
18         xx1.f(lo, xx1);                          // (2)
19         xx1.f(lo, yy);                           // (3)
20         yy.f(i, yy);                             // (4)
21         yy.f(i, xy);                             // (5)
22         yy.f(lo, yy);                            // (6)
23         xy.f(i, xx1);                            // (7)
24         xy.f(lo, yy);                            // (8)
25         //xy.f(i, yy);                           // (9)
26     }
27 }
```

In dieser Aufgabe sollen Sie angeben, welche Methoden- und Konstruktoraufrufe stattfinden, wenn die `main`-Methode der Klasse `Z` ausgeführt wird. Verwenden Sie hierzu keinen Computer, sondern nur die aus der Vorlesung bekannten Angaben zum Verhalten von Java. Benutzen Sie zur eindeutigen Bezeichnung die Funktionssignatur, die jeweils als Kommentar hinter jeder Funktionsdefinition steht. Begründen Sie Ihre Antwort kurz.

a) Geben Sie für die mit `(1)`-`(4)` markierten Konstruktoraufrufe in der Klasse `Z` jeweils an, welche Konstruktoren in welcher Reihenfolge von Java aufgerufen werden. Notieren Sie auch die von Java implizit aufgerufenen Konstruktoren. Bedenken Sie, dass die Oberklasse von `X` die Klasse `Object` ist. Erklären Sie außerdem, welche Attribute mit welchen Werten belegt werden und welche Werte durch die `println`-Anweisungen ausgegeben werden.

b) Geben Sie für die mit `(1)`-`(9)` markierten Aufrufe der Methode `f` in der Klasse `Z` jeweils an, welche Variante der Funktion von Java verwendet wird. Geben Sie hierzu die jeweilige Signatur an.

Lösung:

a) (1) `X(I)` durch expliziten Konstruktoraufruf, dann

`Object()` durch implizites `super()` in `X(I)`.

Das Attribut `a` wird durch den Konstruktor auf den übergegebenen Wert `42` gesetzt, daher wird `X.a: 42` ausgegeben.

(2) `X(F)` durch expliziten Konstruktoraufruf, dann

`X(I)` durch expliziten Aufruf durch `this((int) ...)`, dann

`Object()` durch implizites `super()` in `X(I)`.

Das Attribut `a` wird auf den Wert `(int) (22.99f + 1) = (int) 23.99f = 23` gesetzt, daher wird `X.a: 23` ausgegeben.

(3) `Y(D)` durch expliziten Konstruktoraufruf, dann

`Y(F)` durch expliziten Aufruf `this((float) ...)`, dann

`X(F)` durch expliziten Aufruf `super(a)`, dann

`X(I)` durch expliziten Aufruf durch `this((int) ...)`, dann

`Object()` durch implizites `super()` in `X(I)`.

Das Attribut `X.a` wird auf den Wert `(int) (((float) (7.5 - 1)) + 1) = (int) (((float) 6.5) + 1) = (int) (6.5 + 1) = (int) 7.5 = 7` gesetzt.

Das Attribut `Y.a` wird auf den Wert `(float) (7.5 - 1) = (float) (6.5) = 6.5` gesetzt.

Es wird daher zuerst `X.a: 7` und dann `Y.a = 6.5` ausgegeben.

(4) `Y(F)` durch Aufruf von `Y(7)`, wobei `7` implizit von `int` nach `float` konvertiert wird, dann

`X(F)` durch expliziten Aufruf `super(a)`, dann

`X(I)` durch expliziten Aufruf durch `this((int) ...)`, dann

`Object()` durch implizites `super()` in `X(I)`.

Das Attribut `X.a` wird auf den Wert `(int) (7.0 + 1) = (int) 8.0 = 8` gesetzt.

Das Attribut `Y.a` wird auf den Wert `7.0` gesetzt. Es wird daher zuerst `X.a: 8` und dann `Y.a = 7.0` ausgegeben.

b) (1) `xx1.f(i, xy)`
Die Variable `xx1` hat den Typ `X` und die Variable `xy` ist als `X` deklariert. Daher ruft Java die (genau passende) Methode `X.f(IX)` auf.

(2) `xx1.f(lo, xx1)`
Die Variable `xx1` ist als `X` deklariert. Daher ruft Java die (genau passende) Methode `X.f(LX)` auf.

(3) `xx1.f(lo, yy)`
Die Variable `yy` ist als `Y` deklariert. Daher ruft Java die (genau passende) Methode `X.f(LY)` auf.

(4) `yy.f(i, yy)`
Die Variable `yy` ist als `Y` deklariert. Daher ruft Java die (genau passende) Methode `Y.f(IY)` auf.

(5) `yy.f(i, xy)`
Die Variable `xy` ist als `X` deklariert (dass hier eine Instanz vom Typ `Y` referenziert wird, spielt dabei keine Rolle!). Daher wird von Java die (genau passende) Methode `Y.f(IX)` aufgerufen.

(6) `yy.f(lo, yy)`
Die Variable `yy` ist als `Y` deklariert. Es wird die Methode `X.f(LY)` aufgerufen, da diese spezifischer ist als `Y.f(LX)`.

(7) `xy.f(i, xx1)`
Die Variablen `xy` und `xx1` sind als `X` deklariert. Daher wird zur Compile-Zeit festgelegt, dass die Methode mit der Signatur `(IX)` aufgerufen wird. Zur Laufzeit wird dann aber die Implementierung in `Y` verwendet, da `xy` ein Objekt vom Typ `Y` referenziert. Es wird also `Y.f(IX)` aufgerufen.

(8) `xy.f(lo, yy)`
Die Variable `xy` ist als `X` deklariert, die Variable `yy` als `Y`. Daher wird zur Compile-Zeit festgelegt, dass die Methode mit der Signatur `(LY)` aufgerufen wird (hierbei spielt keine Rolle, dass `xy` zur Laufzeit eine Instanz vom Typ `Y` referenziert). Da diese in `Y` nicht überschrieben wird, wird also `X.f(LY)` aufgerufen.

(9) `xy.f(i, yy)`
Da `f(IY)` in `X` nicht deklariert ist, existiert keine genau passende Methode. Da zur Compile-Zeit nicht entschieden werden kann, welche der (passenden) Methoden `X.f(IX)` oder `X.f(LY)` aufgerufen werden sollte, würde dieser Aufruf zu einem Compile-Fehler führen.

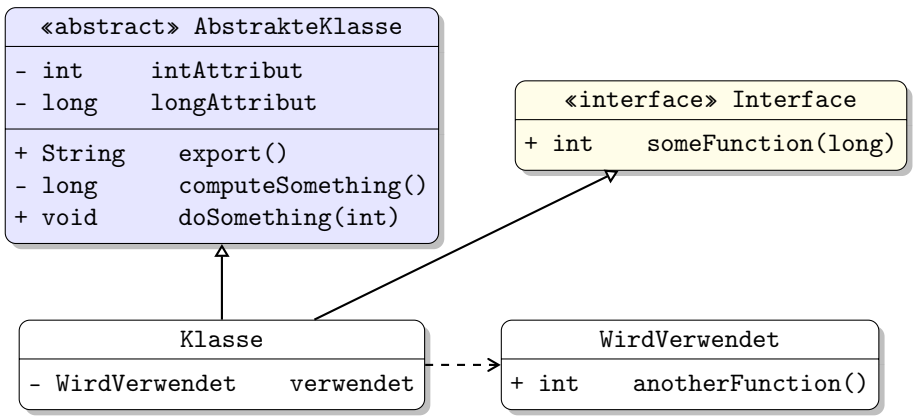
Tutoraufgabe 4 (Klassenhierarchie (Video)):

In dieser Aufgabe soll ein Weihnachtsmarkt modelliert werden.

- Ein Weihnachtsmarkt besteht aus verschiedenen Ständen. Ein Weihnachtsmarkt verfügt außerdem über eine Methode `run()`, die kein Ergebnis zurückgibt.

- Ein Stand kann entweder ein Weihnachtsartikelstand oder ein Lebensmittelstand sein. Jeder Stand hat einen Verkäufer, dessen Name von Interesse ist, und eine Anzahl von Besuchern pro Stunde. Hierfür existiert sowohl ein Attribut `besucherProStunde` als auch eine Methode `berechneBesucherProStunde()`, um diese Anzahl neu zu berechnen. Ein Stand bietet außerdem die Methode `einzelkauf()`, welche den zu bezahlenden Preis (centgenau in Euro) zurückgibt.
- Ein Weihnachtsartikelstand hat eine Reihe an Artikeln.
- Ein Artikel hat einen Namen und einen Preis (centgenau in Euro).
- Ein Lebensmittelstand verkauft ein bestimmtes Lebensmittel.
- Ein Lebensmittel ist entweder ein Flammkuchen oder eine Süßware. Es bietet die Möglichkeit, über die Methoden `getPreisPro100g()` und `getName()`, den festen Preis pro 100 Gramm (centgenau in Euro) und den Namen abzurufen.
- Bei einer Süßware ist der Preis pro 100 Gramm (centgenau in Euro) und die Süßwarenart als String von Interesse.
- Bei einem Flammkuchen ist der Preis pro 100 Gramm (centgenau in Euro) von Interesse.
- Ein Süßwarenstand ist ein Lebensmittelstand.
- Im Gegensatz zu Flammkuchenständen, die einen festen Wasseranschluss benötigen, lassen sich Weihnachtsartikelstände und Süßwarenstände mit einer Methode `verschiebe(int)` ohne Rückgabe verschieben. Dies wird regelmäßig ausgenutzt, falls die Anzahl der Besucher erhöht werden soll.

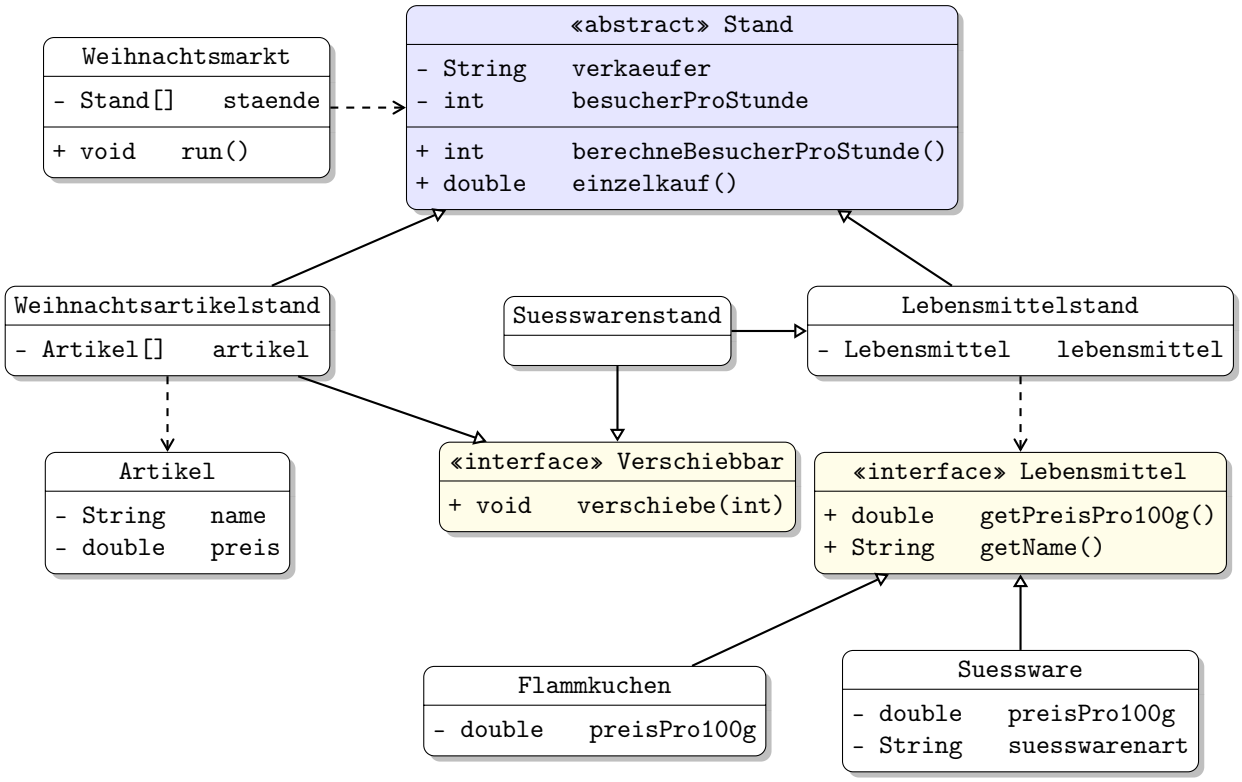
Entwerfen Sie unter Berücksichtigung der Prinzipien der Datenkapselung eine geeignete Klassenhierarchie für einen Weihnachtsmarkt. Notieren Sie keine Konstruktoren, Getter oder Setter (bis auf `getPreisPro100g` und `getName`). Sie müssen nicht markieren, ob Attribute `final` sein sollen. Achten Sie darauf, dass gemeinsame Merkmale in Oberklassen bzw. Interfaces zusammengefasst werden.
Verwenden Sie hierbei die folgende Notation:



Eine Klasse wird hier durch einen Kasten beschrieben, in dem der Name der Klasse sowie Attribute und Methoden in einzelnen Abschnitten beschrieben werden. Weiterhin bedeutet der Pfeil $B \dashrightarrow A$, dass A die Oberklasse von B ist (also `class B extends A` bzw. `class B implements A`, falls A ein Interface ist) und A `-->` B, dass A den Typ B verwendet (z.B. als Typ eines Attributs oder in der Signatur einer Methode). Benutzen sie `+` und `-` um `public` und `private` abzukürzen.
Tragen Sie keine vordefinierten Klassen (`String`, etc.) oder Pfeile dorthin in ihr Diagramm ein.

Lösung:

Die Zusammenhänge können wie folgt modelliert werden:



Tutoraufgabe 6 (Programmieren mit Klassenhierarchien):

In dieser Aufgabe soll es um Softwaretests gehen. Nehmen Sie folgende Situation an: bei einem Programmierwettbewerb soll ein Programm geschrieben werden, welches zwei Zahlen miteinander multipliziert. Sie haben eine Vielzahl an Einreichungen, und möchten diese automatisch auf Korrektheit testen. Um das Entwickeln einer dafür geeigneten Software und die Rahmenbedingungen des Wettbewerbs soll es in dieser Aufgabe gehen. Die Wettbewerbsbeiträge haben die Form einer Klasse, die denen von Ihnen gegebenen Einschränkungen entspricht.
Im Folgenden werden wir wieder teilweise von Abstraktion sprechen, und damit eine (evtl. abstrakte) Klasse, ein Interface oder einen Record meinen. Wählen Sie die jeweils geeignetste Variante. Ebenso sprechen wir nur von Vererbung, auch dann, wenn die Implementierung eines Interfaces möglich wäre. Machen Sie sich auch über sinnvolle Modifikatoren (wie z.B. `public`, `private`, `protected`, `final`, `sealed`, `non-sealed`, etc.) Gedanken.

a) Erstellen Sie zunächst eine Abstraktion `Identifiable`, bei der jede Klasse, die von ihr erbt, die Methode `String getName()` bereitstellt. Diese wird von den eingereichten Programmen und von den Tests genutzt werden, um einen menschlich lesbaren Namen anzugeben.

b) Jedes eingereichte Programm muss die Methode `int calculate(int x, int y)` implementieren, die das Ergebnis einer Multiplikation zweier Zahlen zurückliefern soll. Außerdem soll, wie bereits erwähnt, jedes Programm auch die Methoden von `Identifiable` bereitstellen.

Erstellen Sie eine Abstraktion `Program`, von dem jede Einreichungen erben sollte, um diese Bedingungen zu erfüllen.

c) Wir wollen verschiedene Arten von Tests auf den Programmen laufen lassen. Schreiben Sie eine Abstraktion `Test`, von der alle zukünftigen Tests erben werden. Die Abstraktion `Test` stellt die Methode `TestResult runTest(Program p)` bereit. `TestResult` ist dabei eine weitere Abstraktion, die Sie erstellen sollen. `TestResult` besitzt ein Boolesches Attribut `error` und ein Attribut `message` vom Typ `String`. Sobald ein `TestResult` erstellt worden ist, wird keine Änderung mehr an den Attributen vorgenommen, lediglich ein Zugriff auf die gespeicherten Werte ist nötig.

Weiterhin stellt `Test` die Methode von `Identifiable` bereit. Jeder `Test` hat auch ein `String`-Attribut `identifier`, das bei der Erstellung eines `Test` gesetzt werden muss und das als Standardrückgabewert von `getName()` dienen soll.

d) Wir möchten uns beim Behandeln von Tests auf zwei Arten von Tests beschränken: `PerformanceTest` und `FunctionalTest`. Andere Arten von Tests möchten wir ausschließen, modifizieren Sie `Test` entsprechend.

`PerformanceTest` ist eine konkrete Klasse, die bereits implementiert ist. In der `runTest`-Methode eines `PerformanceTest` wird lediglich die Zeit gemessen, die das übergebene Programm benötigt. Diese wird als `message`-Teil eines `TestResult`s zurückgegeben, der `error`-Wert ist bei einem `PerformanceTest` immer `false`. Die Eingabe, für die das Programm getestet wird, wird bei Erstellung des Tests festgelegt und danach nicht mehr geändert. Außerdem soll `PerformanceTest` auf eine eindeutige Weise durchgeführt werden, es soll daher keine Unterklassen geben. Es fehlen noch sinnvolle Modifikatoren, ergänzen Sie diese.

`FunctionalTest` ist lediglich als eine Überkategorie für die Tests gedacht, die die Programme auf Korrektheit überprüfen werden. Diese ist ebenfalls, bis auf Modifikatoren, bereits implementiert.

e) Nun geht es darum, konkrete Tests zu schreiben. Versuchen Sie, sinnvolle Kategorien von Eingaben zu finden, und für jede dieser Kategorien eine Klasse zu erstellen, die von `FunctionalTest` erbt. Jeder diese Testklassen soll dann in der `runTest`-Methode für mindestens eine Eingabe das Programm auswerten und das Ergebnis auf Richtigkeit überprüfen. Beispielhafte Einreichungen für den Wettbewerb finden Sie in den Klassen `MultiplyX` mit $X \in \{1, \ldots, 5\}$.

Würde es bei dem Wettbewerb um das Halbieren einer Zahl gehen, wären bspw. gerade und ungerade Zahlen mögliche Eingabekategorien und die Test-Klasse, die für gerade Zahlen zuständig ist, könnte das Programm mit der Eingabe 4 laufen lassen und überprüfen, ob 2 zurückgeliefert wird

Bei einem Fehler soll ein `TestResult` mit einem wahren `error`-Wert und einer aussagekräftigen Nachricht zurückgegeben werden, andernfalls mit einem unwahren `error`-Wert und beliebiger Nachricht.

f) Die Klasse `TestManager` ist vorgegeben, in deren `main`-Methode die `Program`-Objekte erzeugt werden und in einem Array gespeichert werden. Danach wird ein Array von Testobjekten erstellt. Momentan befinden sich in dem Array nur die Performance Tests, ergänzen Sie die von Ihnen implementierten Tests.

Schließlich müssen Sie noch die markierten Zeilen in der Methode `runTests` der Klasse `TestManager` ergänzen und können dann die Programme testen lassen. Drei der fünf Programme sind fehlerhaft, finden Ihre Tests die problematischen Programme?

g) Aus der Vorlesung kennen Sie formale Methoden, wie den Hoare-Kalkül, um ein Programm zu verifizieren. Wie unterscheidet sich eine solche Herangehensweise von Tests?

Lösung:

a - f)

Listing 4: Identifiable.java

```
1
2 public interface Identifiable {
3     public String getName();
4 }
```

Listing 5: Program.java

```
1
2 public interface Program extends Identifiable {
3     public int calculate(int x, int y);
4
5 }
```

Listing 6: TestResult.java

```
1 public record TestResult(boolean error, String message) {
2 }
```

Listing 7: Test.java

```
1
2 public abstract sealed class Test
3         implements Identifiable permits PerformanceTest, FunctionalTest {
4     private final String identifier;
5     public abstract TestResult runTest(Program p);
6
7     protected Test(String identifier) {
8         this.identifier=identifier;
9     }
10
11    public String getName() {
12        return identifier;
13    }
14 }
```

Listing 8: PerformanceTest.java

```
1
2 public final class PerformanceTest extends Test {
3     private final int x;
4     private final int y;
5
6     public int[] getInput() {
7         return new int[] {x, y};
8     }
9
10    public PerformanceTest(String category, int inx, int iny) {
11        super(category);
12        x = inx;
13        y = iny;
14    }
15
16    @Override
17    public TestResult runTest(Program p) {
18        long startTime = System.nanoTime();
19        p.calculate(x, y);
20        long estimatedTime = System.nanoTime() - startTime;
21        String msg = "Time elapsed: " + estimatedTime;
22        TestResult t = new TestResult(false, msg);
23        return t;
24    }
25
26 }
```

Listing 9: FunctionalTest.java

```
1
2 public abstract non-sealed class FunctionalTest extends Test {
3     protected FunctionalTest(String identifier) {
4         super(identifier);
5     }
6 }
```

Listing 10: NullTest.java

```
1
```

```
2  public class NullTest extends FunctionalTest {
3
4      public NullTest() {
5          super("NullTest");
6      }
7
8      @Override
9      public TestResult runTest(Program p) {
10         int x=0;
11         int y=5;
12         int r=p.calculate(x, y);
13         boolean error= (r!=0);
14         x=5;
15         y=0;
16         r=p.calculate(x, y);
17         error= error || (r!=0);
18         return new TestResult(error, "Multiplying with 0 does not result in 0");
19     }
20
21 }
```

Listing 11: PositiveTest.java

```
1
2  public class PositiveTest extends FunctionalTest {
3
4      public PositiveTest() {
5          super("Positive Test");
6      }
7
8      @Override
9      public TestResult runTest(Program p) {
10         int x=4;
11         int y=5;
12         int r=p.calculate(x, y);
13         boolean error= (r!=20);
14         return new TestResult(error,"Error when multiplying positive numbers");
15     }
16
17 }
```

Listing 12: NegativeTest.java

```
1
2  public class NegativeTest extends FunctionalTest {
3
4      public NegativeTest() {
5          super("Negative Test");
6      }
7
8      @Override
9      public TestResult runTest(Program p) {
10         int x=-1;
11         int y=-5;
12         int r=p.calculate(x, y);
13         boolean error= (r!=5);
14         return new TestResult(error,"Error when multiplying negative numbers");
15     }
16
17 }
```

Listing 13: PositiveNegativeTest.java

```
1
2  public class PositiveNegativeTest extends FunctionalTest {
3
4      public PositiveNegativeTest() {
5          super("PositiveNegative Test");
6      }
7
8      @Override
9      public TestResult runTest(Program p) {
10         int x=-2;
11         int y=5;
12         int r=p.calculate(x, y);
13         boolean error= (r!=-10);
14         x=5;
15         y=-2;
16         r=p.calculate(x, y);
17         error= error || (r!=-10);
18         return new TestResult(error,"Error when multiplying a negative and positive number");
19     }
20
21 }
```

Listing 14: TestManager.java

```
1  public class TestManager {
2
3      public static void main(String[] args) {
4          Program[] programs = {new Multiply1(), new Multiply2(), new Multiply3(),
5                  new Multiply4(), new Multiply5()};
6          Test t1 = new NullTest();
7          Test t2 = new PositiveNegativeTest();
8          Test t3 = new PositiveTest();
9          Test t4 = new NegativeTest();
10         Test t5 = new PerformanceTest("Small numbers", 10, 9);
11         Test t6 = new PerformanceTest("Big numbers", 4324324, 3132);
12         Test[] tests = {t1, t2, t3, t4, t5, t6};
13         runTests(programs, tests);
14     }
15
16     public static void runTests(Program[] ps, Test[] tests) {
17         for (Program p : ps) {
18             System.out.println("Run tests on program: " + p.getName());
19             for (Test t : tests) {
20                 System.out.print("\t");
21                 TestResult res = t.runTest(p);
22                 if (t instanceof PerformanceTest pt) {
23                     int[] input = pt.getInput();
24                     System.out.println("Executed performance test: " + pt.getName()
25                             + " with inputs " + input[0]
26                             + " " + input[1] + " and " + res.message());
27                 } else { //Tests is sealed, so this will be a (Subclass of) FunctionalTest
28                     if (res.error()) {
29                         System.out.println(t.getName()
30                                 + " failed with message: " + res.message());
31                     } else {
32                         System.out.println(t.getName() + ": OK");
33                     }
34
35                 }
36
```

```
37                  }
38              }
39
40          }
41
42  }
```

g) Wenn man ein Programm formal verifiziert hat, kann man sicher sein, dass es korrekt ist bzw. sich entsprechend der Spezifikation verhält. Tests können dies nicht leisten. Tests zeigen lediglich die Anwesenheit von Fehlern, nicht aber die Abwesenheit derselbigen auf. Gleichzeitig ist es ungleich schwerer, ein Program formal zu verifizieren, was die vorherrschende Verwendung von Tests in der Praxis erklärt.