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.

Aufgabe 3 (Überschreiben, Überladen und Verdecken): (6 + 5 = 11 Punkte)

Betrachten Sie die folgenden Klassen:

Listing 4: A.java

```
1  public class A {
2      public final String x;
3
4      public A() {                                    // Signatur: A()
5          this("written in A()");
6      }
```

```
7
8       public A(int p1) {                                  // Signatur: A(int)
9           this("written in A(int)");
10      }
11
12      public A(String x) {                                // Signatur: A(String)
13          this.x = x;
14      }
15
16      public void f(A p1) {                               // Signatur: A.f(A)
17          System.out.println("called A.f(A)");
18      }
19  }
```

Listing 5: B.java

```
1   public class B extends A {
2       public final String x;
3
4       public B() {                                        // Signatur: B()
5           this("written in B()");
6       }
7
8       public B(int p1) {                                  // Signatur: B(int)
9           this("written in B(int)");
10      }
11
12      public B(A p1) {                                    // Signatur: B(A)
13          this("written in B(A)");
14      }
15
16      public B(B p1) {                                    // Signatur: B(B)
17          this("written in B(B)");
18      }
19
20      public B(String x) {                                // Signatur: B(String)
21          super("written in B(String)");
22          this.x = x;
23      }
24
25      public void f(A p1) {                               // Signatur: B.f(A)
26          System.out.println("called B.f(A)");
27      }
28
29      public void f(B p1) {                               // Signatur: B.f(B)
30          System.out.println("called B.f(B)");
31      }
32  }
```

Listing 6: C.java

```
1   public class C {
2       public static void main(String[] args) {
3                                                               // a)
4           A v1 = new A(100);                                  // (1)
5           System.out.println("v1.x: " + v1.x);
6
7           A v2 = new B(100);                                  // (2)
8           System.out.println("v2.x: " + v2.x);
9           System.out.println("((B) v2).x: " + ((B) v2).x);
10
11          B v3 = new B(v2);                                   // (3)
12          System.out.println("((A) v3).x: " + ((A) v3).x);
13          System.out.println("v3.x: " + v3.x);
14
15          B v4 = new B();                                     // (4)
16          System.out.println("((A) v4).x: " + ((A) v4).x);
17          System.out.println("v4.x: " + v4.x);
18
19                                                              // b)
20          v1.f(v1);                                           // (1)
21          v1.f(v2);                                           // (2)
22          v1.f(v3);                                           // (3)
23          v2.f(v1);                                           // (4)
24          v2.f(v2);                                           // (5)
25          v2.f(v3);                                           // (6)
26          v3.f(v1);                                           // (7)
27          v3.f(v2);                                           // (8)
28          v3.f(v3);                                           // (9)
29      }
30  }
```

In dieser Aufgabe sollen Sie angeben, welche Methoden- und Konstruktoraufrufe stattfinden, wenn die `main`-Methode der Klasse `C` ausgeführt wird. Benutzen Sie hierzu keinen Computer, sondern nur die aus der Vorlesung bekannten Angaben zum Verhalten von Java. Verwenden 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 `C` 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 `A` die Klasse `Object` ist.

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

Lösung:

a) (1) Ausgeführte Konstruktoren:

`A(int)` explizit
`A(String)` explizit
`Object()` implizit

Ausgabe:

```
v1.x: written in A(int)
```

(2) Ausgeführte Konstruktoren:

`B(int)` explizit
`B(String)` explizit
`A(String)` explizit
`Object()` implizit

Ausgabe:

```
v2.x: written in B(String)
((B) v2).x: written in B(int)
```

(3) Ausgeführte Konstruktoren:

`B(A)` explizit
`B(String)` explizit
`A(String)` explizit
`Object()` implizit

Ausgabe:

```
((A) v3).x: written in B(String)
v3.x: written in B(A)
```

(4) Ausgeführte Konstruktoren:

`B()` explizit
`B(String)` explizit
`A(String)` explizit
`Object()` implizit

Ausgabe:

```
((A) v4).x: written in B(String)
v4.x: written in B()
```

b) (1) `v1.f(v1);`
Typen:

- `v1`: deklarierter Typ `A`, Laufzeittyp `A`
- `v1`: deklarierter Typ `A`

Passende Methoden: `A.f(A)`
Aufgerufene Methode: `A.f(A)`

(2) `v1.f(v2);`
Typen:

- `v1`: deklarierter Typ `A`, Laufzeittyp `A`
- `v2`: deklarierter Typ `A`

Passende Methoden: `A.f(A)`
Aufgerufene Methode: `A.f(A)`

(3) `v1.f(v3);`
Typen:

- `v1`: deklarierter Typ `A`, Laufzeittyp `A`
- `v3`: deklarierter Typ `B`

Passende Methoden: `A.f(A)`
Aufgerufene Methode: `A.f(A)`

(4) `v2.f(v1);`
Typen:

- `v2`: deklarierter Typ `A`, Laufzeittyp `B`
- `v1`: deklarierter Typ `A`

Passende Methoden: `A.f(A)`, `B.f(A)`
Aufgerufene Methode: `B.f(A)`

(5) `v2.f(v2);`
Typen:

- `v2`: deklarierter Typ `A`, Laufzeittyp `B`
- `v2`: deklarierter Typ `A`

Passende Methoden: `A.f(A)`, `B.f(A)`
Aufgerufene Methode: `B.f(A)`

(6) `v2.f(v3);`
Typen:

- `v2`: deklarierter Typ `A`, Laufzeittyp `B`
- `v3`: deklarierter Typ `B`

Passende Methoden: `A.f(A)`, `B.f(A)`, `B.f(B)`
Aufgerufene Methode: `B.f(A)`

Da `v2` als `A` deklariert ist, wird die Methode `A.f(A)` ausgewählt. Diese kann vom Laufzeittyp `B` von `v2` nur durch die Methode `B.f(A)` überschrieben werden, nicht aber durch `B.f(B)`. Somit wird, da `A.f(B)` nicht existiert, in der Tat `B.f(A)` aufgerufen, obwohl `v3` als `B` deklariert ist.

(7) `v3.f(v1);`
Typen:

- `v3`: deklarierter Typ `B`, Laufzeittyp `B`
- `v1`: deklarierter Typ `A`

Passende Methoden: `A.f(A)`, `B.f(A)`
Aufgerufene Methode: `B.f(A)`

(8) `v3.f(v2);`
Typen:

- `v3`: deklarierter Typ `B`, Laufzeittyp `B`
- `v2`: deklarierter Typ `A`

Passende Methoden: `A.f(A)`, `B.f(A)`
Aufgerufene Methode: `B.f(A)`

(9) `v3.f(v3);`
Typen:

- `v3`: deklarierter Typ `B`, Laufzeittyp `B`
- `v3`: deklarierter Typ `B`

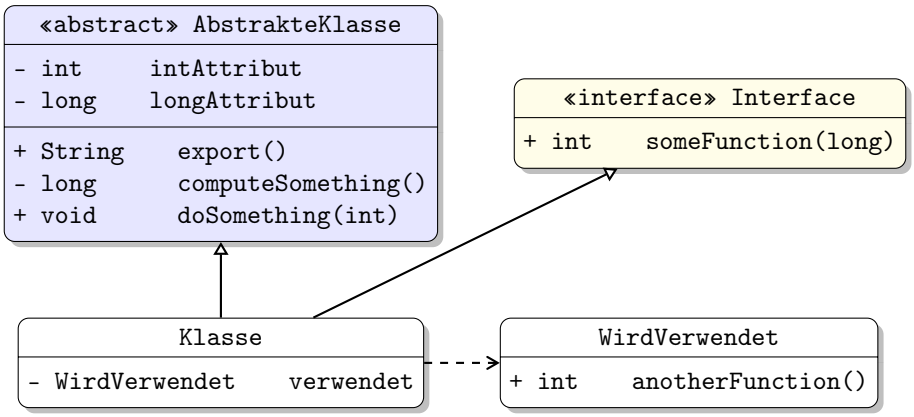
Passende Methoden: `A.f(A)`, `B.f(A)`, `B.f(B)`
Aufgerufene Methode: `B.f(B)`

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.

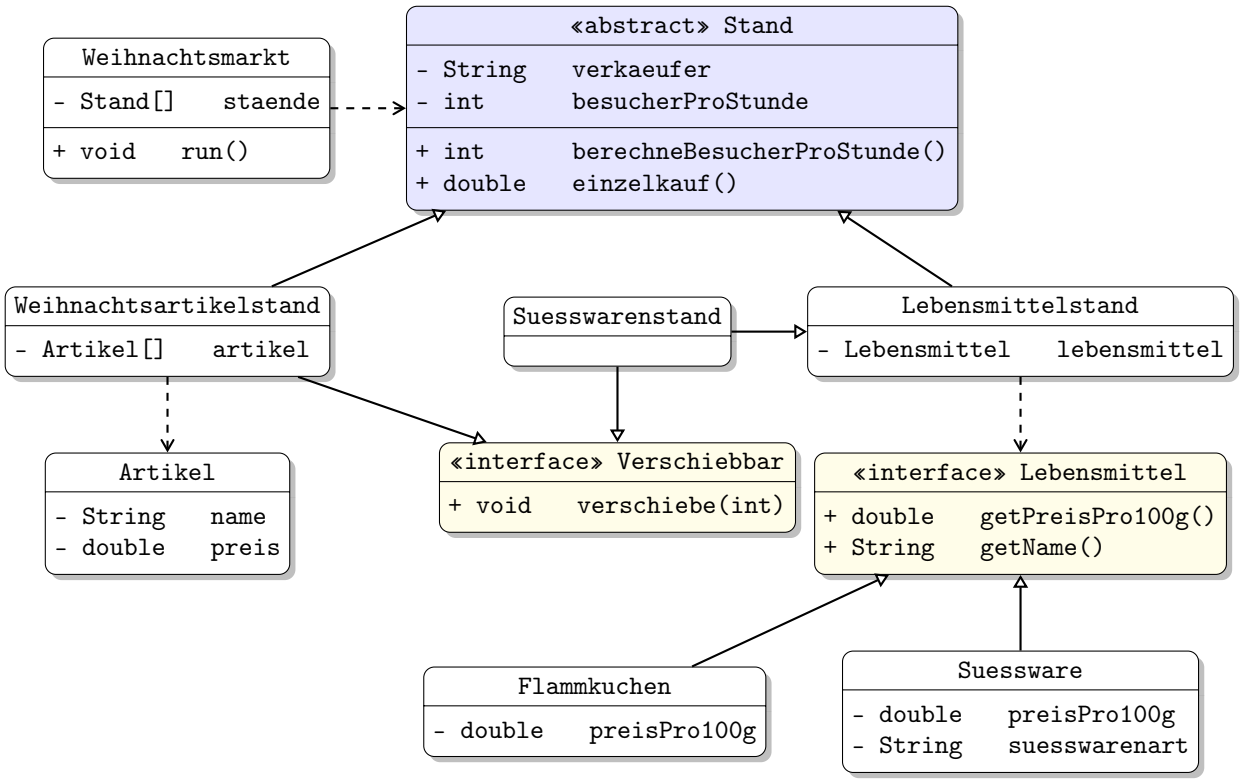
Entwerfen Sie unter Berücksichtigung der Prinzipien der Datenkapselung eine geeignete Klassenhierarchie für einen Weihnachtsmarkt. Notieren Sie keine Konstruktoren, Getter oder Setter (bis auf `getPreisPro100g` und `getName`). Sie müssen nicht markieren, ob Attribute `final` sein sollen. Achten Sie darauf, 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 $B \longrightarrow\!\!\triangleright A$, 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:



Aufgabe 5 (Klassenhierarchie): (13 Punkte)

Eine der grundlegenden Funktionalitäten des Betriebssystems ist es, einen einfachen und einheitlichen Zugriff auf gespeicherte Daten zu liefern. Dabei muss es der Benutzer*in Ordner (Directories) und Dateien (Files) präsentieren. Im Folgenden sehen Sie ein Beispiel für einen Teil eines typischen Linux Dateisystems.

```
/
/boot/
/boot/kernel
/etc/
/etc/motd
```

Wir sehen den Wurzelordner `/`, die beiden Unterordner `boot` und `etc` sowie die beiden Dateien `kernel` und `motd`[2].
Intern wird der Inhalt einer Datei oder eines Unterordners nicht unter ihrem Namen abgelegt, sondern unter einem sogenannten inode, einem Integer. Der Eintrag im Ordner enthält dann nur die Information, unter welchem inode der Inhalt zu finden ist. Falls beispielsweise der Inhalt der Datei `motd` unter dem inode 5 abgelegt ist, dann steht im Unterordner nur, dass hier eine Datei mit dem Namen `motd` existiert und dass deren Inhalt unter dem inode 5 zu finden ist.
Dies ist ein nützlicher Mechanismus, denn er erlaubt es, auf einfache Art und Weise den Inhalt zweier Dateien gleich zu halten. Dazu wird ein zweiter Eintrag im Ordner angelegt, welcher auf denselben inode verweist wie ein bereits existierender Eintrag. So ist es möglich, einen zweiten Eintrag `friendly-message.txt` im Ordner `etc` zu erstellen, welcher ebenfalls auf den inode 5 verweist. Wird nun der Inhalt von `motd` verändert, so ändert sich automatisch auch der Inhalt von `friendly-message.txt`, und umgekehrt. Man sagt, `friendly-message.txt` ist ein *Hardlink* auf den Inhalt von `motd`. Auch `motd` ist ein Hardlink auf seinen Inhalt. In der Tat sind alle Einträge im Ordner gleichberechtigte Hardlinks auf ihren Inhalt. Ein inode wird erst dann gelöscht, wenn der letzte auf ihn verweisende Hardlink gelöscht wurde.
Im Folgenden werden wir von Abstraktion sprechen, und damit eine (evtl. abstrakte) Klasse oder ein Interface meinen. Wählen Sie die jeweils geeignetste Variante.
Modellieren Sie ein Dateisystem wie folgt:

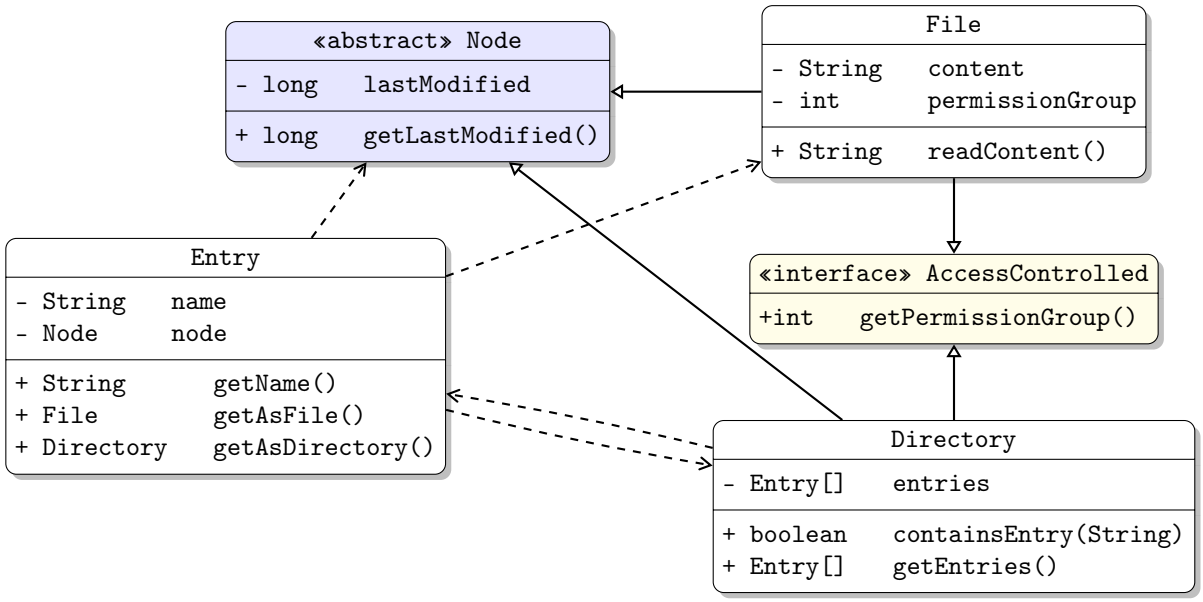
- Jeder Hardlink, also jeder Eintrag im Ordner, wird durch eine Abstraktion `Entry` dargestellt. Jedes `Entry`-Objekt hat einen `name` sowie eine Referenz auf einen `Node`.
- Jeder inode, also jeder Inhalt, wird hier nicht durch einen Integer, sondern durch ein Objekt der Abstraktion `Node` dargestellt. Jedes `Node`-Objekt hat das Attribut `lastModified`, welches den Zeitpunkt der letzten Änderung enthält und über die Methode `long getLastModified()` abgerufen werden kann.
- Ein Dateiinhalt ist ein `Node`, welcher durch ein Objekt der Abstraktion `File` dargestellt wird. Jedes `File`-Objekt hält seinen Inhalt in einem `String content`, welcher über die Methoden `String readContent()` gelesen werden kann und ein `int`-Attribut `permissionGroup`, zu dem später mehr erklärt wird.
- Ein Ordnerinhalt ist ein `Node`, welcher durch ein Objekt der Abstraktion `Directory` dargestellt wird. Jedes `Directory`-Objekt hält seine Dateien und Unterordner in einem Array von `Entry`s, welches über die Methode `Entry[] getEntries()` abgerufen werden kann. Über die Methode `boolean containsEntry(String name)` kann geprüft werden, ob der Ordner einen gegebenen Eintrag enthält.
- Die Abstraktion `Entry` bietet ebenfalls Methoden. Die Methode `String getName()` gibt das `name`-Attribut zurück. Die Methode `File getAsFile()` liefert ihren `Node` als `File`. Die Methode `Directory getAsDirectory()` arbeitet analog dazu.
- Sowohl `File` als auch `Directory` sollen Informationen über die Zugriffserlaubnis bieten. Daher sollen beide eine Methode `int getPermissionGroup()` bereitstellen. Bei `File` ergibt sich dies aus der `permissionGroup`, bei `Directory` wird die zugriffsberechtigte Gruppe aus den Ordnerinhalten berechnet. Ergänzen Sie hier ggf. eine passende Abstraktion.

Entwerfen Sie unter Berücksichtigung der Prinzipien der Datenkapselung eine geeignete Klassenhierarchie. Notieren Sie keine Konstruktoren. Die in der Aufgabenstellung erwähnten Getter und Setter sollen notiert werden, aber andere nicht. 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 gleiche Notation bzw. Darstellungsweise wie in Tutoraufgabe 4.

Lösung:

Die Zusammenhänge können wie folgt modelliert werden:

[2] message of the day



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 `TestResults` 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 7: Identifiable.java

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

Listing 8: Program.java

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

Listing 9: TestResult.java

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

Listing 10: 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 11: 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 12: FunctionalTest.java

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

Listing 13: 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 14: 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 15: 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 16: 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 17: 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.

Aufgabe 7 (Programmieren mit Klassenhierarchien): (1 + 2 + 6 + 7 + 4 + 2 + 4 = 26 Punkte)

Ziel dieser Aufgabe ist es, eine einfache Versionsverwaltung[3] zu implementieren. Diese verwaltet beliebig viele Versionen beliebig vieler Text-Dateien, die in demselben Verzeichnis liegen. Dieses bezeichnen wir im Folgenden als Wurzelverzeichnis. Alle von der Versionsverwaltung erfassten Versionen werden in dem Unterverzeichnis `vcs` des Wurzelverzeichnisses gespeichert. Dieses bezeichnen wir im Folgenden als Backupverzeichnis. Die neueste in die Versionsverwaltung eingecheckte Version ist stets direkt im Backupverzeichnis gespeichert. Wenn eine neue Version eingecheckt wird, wird im Backupverzeichnis ein neues Unterverzeichnis erstellt, in das die letzte eingecheckte Version verschoben wird. Die Versionsverwaltung erlaubt das Ausführen folgender Kommandos:

- `listfiles`: Gibt die Namen aller Dateien im Wurzelverzeichnis (aber nicht im Backupverzeichnis oder weiteren Unterverzeichnissen) aus.
- `commit`: Checkt eine neue Version in die Versionsverwaltung ein, d.h.,
  - es wird ein neues Unterverzeichnis im Backupverzeichnis erstellt,
  - alle Dateien im Backupverzeichnis werden in das neue Unterverzeichnis verschoben und
  - alle Dateien im Wurzelverzeichnis werden in das Backupverzeichnis kopiert.
- `exit`: Beendet die Anwendung.

Die Interaktion mit der Versionsverwaltung kann also z.B. wir folgt aussehen, wobei `./repository` das Wurzelverzeichnis ist. Die grauen Zahlen am Ende der Zeile gehören dabei nicht zur Aus-/Eingabe, sie dienen als Referenz, um nach der Ausgabe die jeweils resultierende Ordnerstruktur zu zeigen.

```
java Main ./repository/ 1
initialized empty repository
> listfiles 2
test2
test
> commit 3
Committed the following files:
test2
test
>commit 4
Committed the following files:
test2
test
> exit
```

Vor dem Starten des Programms ist `./repository` ein leerer Ordner. Nach 1 existiert ein neuer Unterordner `./repository/vcs`, ansonsten gibt es keine Dateien. Nehmen wir an, dass nach 1 außerhalb des Programms von einer Nutzer*in zwei Dateien `test` und `test2` erstellt und in `./repository/` abgelegt wurden. Es existieren also nur die Dateien `./repository/test1` und `./repository/test2` und diese werden ohne eine Änderung der Ordnerstruktur von `listfiles` nach Ausführen von 2 ausgegeben. Bei 3 werden nun diese beiden Dateien commited, d.h. sie werden in `./repository/vcs` kopiert und es wird ein neuer Unterordner `./repository/vcs/123456789` erstellt, wobei `123456789` stellvertretend für einen Unix-Timestamp steht. Dies wird später näher erläutert. In diesen Ordner werden alle vorigen Dateien aus `./repository/vcs` verschoben, in unserem Beispiel bleibt der Ordner `./repository/vsc/123456789` leer, da zuvor keine Dateien committed wurden. Bei Befehl 4 werden nun erneut zwei, potentiell geänderte, Versionen von `test2` und `test` commited.

[3] `https://de.wikipedia.org/wiki/Versionsverwaltung`

Es wird nun also ein neuer Unterordner `./repository/vcs/4242424242` erstellt, in den nun die alten Versionen von `test` und `test2` verschoben werden. Die aktuellen Versionen von `test` und `test2` werden wiederum in `./repository/vcs` kopiert.
Zum Lösen der folgenden Aufgaben müssen Sie die Klassen `Util, VCS`[4]`, Command` und `Main` aus dem Moodle herunterladen. Die Klasse `VCS` repräsentiert eine Versionsverwaltung und stellt die beiden Methoden `getRootDir` und `getBackupDir` zur Verfügung, die den Pfad zum Wurzelverzeichnis bzw. zum Backupverzeichnis zurückliefern. Die Klasse `Main` enthält die `main`-Methode der Versionsverwaltung. Die Klasse `Command` repräsentiert die oben genannten Kommandos. Die Klasse `Util` stellt einige Hilfsmethoden zur Verfügung, die zum Lösen der folgenden Aufgaben benötigt werden. Beachten Sie beim Lösen der folgenden Aufgaben die Prinzipien der Datenkapselung. Sie dürfen beliebig viele zusätzliche Methoden zur Klasse `Command` und den von Ihnen implementierten Klassen (aber nicht zu den anderen vorgegebenen Klassen) hinzufügen.

a) Ergänzen Sie die Implementierung der abstrakten Klasse `Command` um ein Attribut `VCS vcs`. Dieses soll dem Konstruktor als einziges Argument übergeben werden.

b) Implementieren Sie eine Klasse `Exit`, die von `Command` erbt (d.h., `Exit` ist eine Unterklasse von `Command`). Ihre `execute` Methode soll die Anwendung beenden.

Hinweise:

- Die Methode `exit` der Klasse `Util` ist zum Lösen dieser Aufgabe nützlich.

c) Implementieren Sie eine Klasse `ListFiles`, die von `Command` erbt. Die `execute` Methode dieser Klasse soll die Namen aller Dateien im Wurzelverzeichnis der Versionsverwaltung `vcs` ausgeben.

Hinweise:

- Die Methode `listFiles` der Klasse `Util` ist zum Lösen dieser Aufgabe hilfreich.

d) Implementieren Sie eine Klasse `Commit`, die von `ListFiles` erbt. Ihre `execute` Methode soll gemäß der Beschreibung des Kommandos "`commit`" eine neue Version in die Versionsverwaltung einchecken. Anschließend soll sie die Meldung "`Committed the following files:`" gefolgt von einer Liste aller Dateien, die aus dem Wurzelverzeichnis in das Backupverzeichnis kopiert wurden, ausgeben. Verwenden Sie hierzu die Funktionalität, die bereits in `ListFiles.execute` implementiert wurde, wieder. Der Name des neuen Unterverzeichnisses des Backupverzeichnisses soll der aktuelle Unix-Timestamp[5] sein. Diesen liefert die Methode `getTimestamp` der Klasse `Util`.

Hinweise:

- Die Methoden `appendFileOrDirName, mkdir, listFiles, copyFile` und `moveFile` der Klasse `Util` sind zum Lösen dieser Aufgabe nützlich.

e) Implementieren Sie die Methode `Command parse(String cmdName, VCS vcs)` in der Klasse `Command`. Diese soll eine geeignete Instanz der Klasse `Exit` zurückgeben, falls `cmdName` der String "`exit`" ist, sie soll eine geeignete Instanz der Klasse `ListFiles` zurückgeben, falls `cmdName` der String "`listfiles`" ist und sie soll eine geeignete Instanz der Klasse `Commit` zurückgeben, falls `cmdName` der String "`commit`" ist. Andernfalls soll sie eine geeignete Fehlermeldung ausgeben und `null` zurückgeben.

f) Da in Zukunft jede neue `Command`-Art den Status der Dateien nach Ausführung angeben soll und das auf eindeutige Weise, wollen Sie lediglich Vererbung vom Typ `ListFiles` erlauben (d.h. `ListFiles` darf beliebige Unterklassen haben). `Exit` und `Command` sollen, bis auf den bisherigen Stand, nicht weiter erbbar sein (d.h., sie sollen in Zukunft keine weiteren Unterklassen mehr bekommen können). Ändern Sie die vorgegebene Klassendefinitionen von `Command` und passen Sie auch die von Ihnen geschriebenen Klassen `Exit` und `ListFiles` an.

g) Erstellen Sie eine Abstraktion `Modifying`, die von `Commit` implementiert werden soll. Diese soll für alle Kommandos, die das Dateisystem oder Dateien verändern, eine Methodensignatur `String getInformation()` vorgeben. Diese Methode soll bei den entsprechenden Kommandos (in dieser einfachen Version einer Versionsverwaltung ist dies nur `Commit`) einen `String` zurückgeben, der angibt, welche Art von Dateioperationen durchgeführt werden. Ergänzen Sie an der angegebenen Stelle in der `main`-Methode Anweisungen,

[4] "Version Control System"
[5] `https://de.wikipedia.org/wiki/Unixzeit`

die dafür sorgen, dass immer wenn ein `Command` ausgeführt werden soll, der diese Methode bereitstellt, diese Informationen vorher ausgegeben werden. Passen Sie außerdem `Commit` so an, dass die neu erstellte Abstraktion genutzt wird. Beispielsweise könnte eine Ausgabe so aussehen:

```
> commit
This command does the following modifying operations:
Files: Copy and Move
Directory: create
Committed the following files:
text1.text
text2.text
```

Wenn Sie alle Teilaufgaben gelöst haben, können Sie die Versionsverwaltung ausfüllen, indem Sie `java Main root_directory` ausführen, wobei `root_directory` der Pfad zum Wurzelverzeichnis ist.

Hinweise:

- Da die Versionsverwaltung schreibend auf das Dateisystem zugreift, sollte sie nicht mit einem Wurzelverzeichnis getestet werden, das wichtige Daten enthält.

Lösung:

Listing 18: Command.java

```
1  public sealed abstract class Command permits ListFiles,Exit {
2
3      private VCS vcs;
4
5      public Command(VCS vcs) {
6          this.vcs = vcs;
7      }
8
9      public abstract void execute();
10
11     public VCS getVCS() {
12         return vcs;
13     }
14
15     public static Command parse(String cmdName, VCS vcs) {
16         return switch (cmdName) {
17             case "commit" ->  new  Commit(vcs);
18             case "listfiles" ->  new ListFiles(vcs);
19             case "exit" ->  new Exit(vcs);
20             default->{ System.out.println("unknown command " + cmdName);
21                 yield null;}
22         };
23     }
24 }
```

Listing 19: Exit.java

```
1  public final class Exit extends Command {
2
3      public Exit(VCS vcs) {
4          super(vcs);
5      }
6
```

```
7       @Override
8       public void execute() {
9           Util.exit();
10      }
11
12  }
```

Listing 20: ListFiles.java

```
1   public non-sealed class ListFiles extends Command {
2
3       public ListFiles(VCS vcs) {
4           super(vcs);
5       }
6
7       @Override
8       public void execute() {
9           for (String file: Util.listFiles(getVCS().getRootDir())) {
10              System.out.println(file);
11          }
12      }
13
14  }
```

Listing 21: Commit.java

```
1   public class Commit extends ListFiles implements Modifying {
2
3       public Commit(VCS vcs) {
4           super(vcs);
5       }
6
7       @Override
8       public void execute() {
9           String ts = Util.getTimestamp();
10          String backupDir = getVCS().getBackupDir();
11          String rootDir = getVCS().getRootDir();
12          String newDir = Util.appendFileOrDirname(backupDir, ts);
13          Util.mkdir(newDir);
14          for (String file: Util.listFiles(backupDir)) {
15              String src = Util.appendFileOrDirname(backupDir, file);
16              String dest = Util.appendFileOrDirname(newDir, file);
17              Util.moveFile(src, dest);
18          }
19          for (String file: Util.listFiles(rootDir)) {
20              String src = Util.appendFileOrDirname(rootDir, file);
21              String dest = Util.appendFileOrDirname(backupDir, file);
22              Util.copyFile(src, dest);
23          }
24          System.out.println("Committed the following files:");
25          super.execute();
26      }
27
28      @Override
29      public String getInformation() {
30          return "\nFiles: Copy and Move\nDirectory: create";
31      }
32  }
```

Listing 22: VCS.java

```
1  import java.io.*;
2  import java.util.*;
3
4  public class VCS {
5
6      private String rootDir;
7
8      public VCS(String dir) {
9          this.rootDir = dir;
10         if (!new File(getBackupDir()).exists()) {
11             Util.mkdir(getBackupDir());
12             System.out.println("initialized empty repository");
13         }
14     }
15
16     /**
17      * @return the backup directory of the version control system
18      */
19     public String getBackupDir() {
20         return Util.appendFileOrDirname(rootDir, "vcs");
21     }
22
23     /**
24      * @return the root directory of the version control system
25      */
26     public String getRootDir() {
27         return rootDir;
28     }
29
30 }
```

Listing 23: Util.java

```
1  import java.io.*;
2  import java.nio.file.*;
3  import java.util.*;
4
5  public class Util {
6
7      /**
8       * Creates a new file- or directory name by appending "fileOrDirname"
9       * to "dirname".Takes care of the operating-system specific
10      * file-separator which has to be in between
11      * (e.g., "/" on Linux, but "\" on Windows)
12      * @return a new file- or directory name, pointing to
13      * the "fileOrDirname" in the directory "dirname"
14      */
15     public static String appendFileOrDirname(String dirname, String fileOrDirname) {
16         return new File(dirname + File.separator + fileOrDirname).getAbsolutePath();
17     }
18
19     /**
20      * Creates the directory "dirname".
21      * Fails (and enforces termination) if creating "dirname" fails.
22      */
23     public static void mkdir(String dirname) {
```

```
24          if (!new File(dirname).mkdir()) {
25              System.err.println("error creating directory " + dirname);
26              System.exit(-1);
27          }
28      }
29
30      /**
31       * Moves the file "srcFilename" to "destFilename".
32       * Fails (and enforces termination) if moving fails
33       * (e.g., if "destFilename" already exists).
34       */
35      public static void moveFile(String srcFilename, String destFilename) {
36          try {
37              Files.move(new File(srcFilename).toPath(),
38                      new File(destFilename).toPath());
39          } catch (IOException e) {
40              System.err.println("error writing " + srcFilename
41                      + " to " + destFilename);
42              System.exit(-1);
43          }
44      }
45
46      /**
47       * Copies the file "srcFilename" to "destFilename".
48       * Fails (and enforces termination) if copying fails
49       * (e.g., if "destFilename" already exists).
50       */
51      public static void copyFile(String srcFilename, String destFilename) {
52          try {
53              Files.copy(new File(srcFilename).toPath(),
54                      new File(destFilename).toPath());
55          } catch (IOException e) {
56              System.err.println("error writing " + srcFilename
57                      + " to " + destFilename);
58              System.exit(-1);
59          }
60      }
61
62      /**
63       * @return a string which uniquely identifies a point in time
64       * (search for "Unix timestamp" for further information)
65       */
66      public static String getTimestamp() {
67          return Long.toString(System.currentTimeMillis());
68      }
69
70      /**
71       * @return the names of all files (excluding directories)
72       * contained in the directory "dirname"
73       */
74      public static String[] listFiles(String dirname) {
75          List<String> filenames = new ArrayList<>();
76          for (File f : new File(dirname).listFiles()) {
77              if (!f.isDirectory()) {
78                  filenames.add(f.getName());
79              }
```

```
80          }
81          return filenames.toArray(new String[filenames.size()]);
82      }
83
84      /**
85       * Terminates the program.
86       */
87      public static void exit() {
88          System.exit(0);
89      }
90
91  }
```

Listing 24: Main.java

```
1   import java.io.*;
2   import java.util.*;
3
4   public class Main {
5
6       /**
7        * @param args the only expected argument is the path to the root directory
8        */
9       public static void main(String[] args) {
10          if (args.length != 1) {
11              System.out.println("wrong number of arguments");
12          } else {
13              File f = new File(args[0]);
14              if (!f.exists()) {
15                  System.out.println(args[0] + " does not exist");
16              } else if (!f.isDirectory()) {
17                  System.out.println(args[0] + " is not a directory");
18              } else if (!f.canWrite()) {
19                  System.out.println(args[0] + " is read-only");
20              } else {
21                  VCS vcs = new VCS(args[0]);
22                  Scanner scanner = new Scanner(System.in);
23                  while (true) {
24                      System.out.print("> ");
25                      Command cmd = Command.parse(scanner.nextLine(), vcs);
26                      if (cmd != null) {
27                          if( cmd instanceof Modifying m){
28                             System.out.println("This command does the " +
29                                     "following modifiying operations: "
30                                     + m.getInformation());
31                          }
32                          cmd.execute();
33                      }
34                  }
35              }
36          }
37      }
38
39  }
```

Listing 25: Main.java

```
1   public interface Modifying {
```

```
2       public String getInformation();
3   }
```

Aufgabe 8 (Codescape): (Codescape)

Schließen Sie das Spiel Codescape ab, indem Sie die letzten Missionen von Deck 7 lösen. Genießen Sie anschließend das Outro. Dieses Deck enthält keine für die Zulassung relevanten Missionen.

Hinweise:

- Es gibt drei verschiedene Möglichkeiten wie die Story endet, abhängig von Ihrer Entscheidung im finalen Raum.
- Verraten Sie Ihren Kommilitonen nicht, welche Auswirkungen Ihre Entscheidung hatte, bevor diese selbst das Spiel abgeschlossen haben.

Lösung: