Aufgabe 1 (Programmanalyse): (9 + 5 = 14 Punkte)

a) Geben Sie die Ausgabe des Programms für den Aufruf `java M` an. Tragen Sie hierzu jeweils die ausgegebenen Zeichen in die markierten Stellen hinter „`OUT:`“ ein.

Achten Sie darauf, dass Gleitkommazahlen in der Form „5.0“ und nicht als „5“ ausgegeben werden.

```
public class A {
  public Long a;
  public double b;

  public A() {
    b = 1.5f;
    a = (long)b;
  }

  public A(long x) {
    a = x;
    b = x;
  }

  public long f(Integer y) {
    return y;
  }

  public float f(double x) {
    return 2.0f;
  }
}
```

```
public class B extends A {
  public double b = 7.2;
  public Float c;

  public B(Integer f) {
    super(f);
    a = 8L;
  }

  public B(Long y) {
    super(y - y);
    c = 9f;
  }

  public long f(Integer y) {
    return 5L;
  }

  public double f(float x) {
    return 6.0;
  }
}
```

b) Wir schreiben zusätzlich zu `A` und `B` eine neue Klasse `C`. Welche drei Fehler treten beim Compilieren auf? Begründen Sie Ihre Antwort kurz.

```
1  public class C extends B {
2    public int z;
3    public C(Short x) {
4      super(x);
5      b = 5;
6    }
7
8    private double f(Float x) {
9      return b;
10   }
11
12   private long f(Integer y) {
13     return z;
14   }
15
16   public float f(double x) {
17     return b + 1.0;
18   }
19 }
```

Lösung:

a)
```
public class M {
    public static void main(String[] args) {

      A a = new A();
      System.out.println(a.a + " " + a.b);           // OUT: [ 1 ] [1.5]

      B b = new B(Integer.valueOf(3));
      System.out.println(b.b + " " + b.c);           // OUT: [7.2] [null]

      A ab = new B(4);
      System.out.println(ab.a + " " + ab.b);         // OUT: [ 8 ] [4.0]

      B b2 = new B(a.a);
      System.out.println(b2.a + " " + b2.c);         // OUT: [ 0 ] [9.0]

      System.out.println(ab instanceof B);           // OUT: [true]

      System.out.println(ab.f(Integer.valueOf(1))); // OUT: [ 5 ]

      System.out.println(b.f(4));                    // OUT: [6.0]
    }
}
```

b)
- Im ersten Konstruktor existiert kein passender Konstruktor für `B`, da `x` den Typ `Short` hat und nicht implizit zu `Integer` oder `Long` umgewandelt werden kann.
- Die Methode in Zeile 12 überschreibt die entsprechende Methode aus `B`, setzt aber den Zugriffsmodifikator verbotenerweise von `public` auf `private`.
- In Zeile 22 soll ein Wert vom Typ `float` zurückgegeben werden, es wird jedoch in der Addition ein Wert vom Typ `double` berechnet.

Aufgabe 2 (Hoare-Kalkül): (13 + 3 = 16 Punkte)

Gegeben sei folgendes Java-Programm P über den `int`-Variablen `n`, `res`, `i` und `y`:

$\langle\, 0 \leq \texttt{n} \,\rangle$ (Vorbedingung)

```
res = 0;
i = 0;
y = 1;
while (i < 2 * n + 1) {
    if (i % 2 == 0) {
        res = res + y;
    }
    y = y * 2;
    i = i + 1;
}
```

$\langle\, \texttt{res} = \sum_{j=0}^{\texttt{n}} 4^j \,\rangle$ (Nachbedingung)

a) Vervollständigen Sie die folgende Verifikation des Algorithmus P im Hoare-Kalkül, indem Sie die unterstrichenen Teile ergänzen. Hierbei dürfen zwei Zusicherungen nur dann direkt untereinander stehen, wenn die untere aus der oberen folgt. Hinter einer Programmanweisung darf nur eine Zusicherung stehen, wenn dies aus einer Regel des Hoare-Kalküls folgt.

Hinweise:

- Sie dürfen beliebig viele Zusicherungs-Zeilen ergänzen oder streichen. In der Musterlösung werden allerdings genau die angegebenen Zusicherungen benutzt.
- Bedenken Sie, dass die Regeln des Kalküls syntaktisch sind, weshalb Sie semantische Änderungen (beispielsweise von $x+1 = y+1$ zu $x = y$) nur unter Zuhilfenahme der Konsequenzregeln vornehmen dürfen.
- Begründen Sie bei Anwendung der Bedingungsregel, warum Sie die Zusicherung direkt *nach* dem `if`-Block schreiben dürfen.
- Für $q \in \mathbb{Q}$, $q \geq 0$ bezeichnet $\lfloor q \rfloor$ die größte natürliche Zahl mit $q \geq \lfloor q \rfloor$, z.B. $\lfloor 2 \rfloor = 2$ und $\lfloor \frac{5}{2} \rfloor = 2$.
- Der Programmlauf für $\texttt{n} = 2$ ist wie folgt:

n	res	i	y
2	0	0	1
2	1	1	2
2	1	2	4
2	5	3	8
2	5	4	16
2	21	5	32

Es ist $21 = \sum_{j=0}^{2} 4^j = 4^0 + 4^1 + 4^2$.

b) Untersuchen Sie den Algorithmus P auf seine Terminierung. Für einen Beweis der Terminierung muss eine Variante angegeben werden und unter Verwendung des Hoare-Kalküls die Terminierung bewiesen werden.

Lösung:

a)

$\langle \mathtt{n} \geq 0 \rangle$
$\langle \mathtt{n} \geq 0 \wedge 0 = 0 \wedge 0 = 0 \wedge 1 = 1 \rangle$

```
res = 0;
```

$\langle \mathtt{n} \geq 0 \wedge \mathtt{res} = 0 \wedge 0 = 0 \wedge 1 = 1 \rangle$

```
i = 0;
```

$\langle \mathtt{n} \geq 0 \wedge \mathtt{res} = 0 \wedge \mathtt{i} = 0 \wedge 1 = 1 \rangle$

```
y = 1;
```

$\langle \mathtt{n} \geq 0 \wedge \mathtt{res} = 0 \wedge \mathtt{i} = 0 \wedge \mathtt{y} = 1 \rangle$
$\langle \mathtt{res} = \sum_{j=0}^{\lfloor \frac{\mathtt{i}+1}{2} \rfloor - 1} 4^j \wedge \mathtt{y} = 2^{\mathtt{i}} \wedge \mathtt{i} \leq 2 \cdot \mathtt{n} + 1 \rangle$

```
while (i < 2 * n + 1) {
```

$\langle \mathtt{res} = \sum_{j=0}^{\lfloor \frac{\mathtt{i}+1}{2} \rfloor - 1} 4^j \wedge \mathtt{y} = 2^{\mathtt{i}} \wedge \mathtt{i} \leq 2 \cdot \mathtt{n} + 1 \wedge \mathtt{i} < 2 \cdot \mathtt{n} + 1 \rangle$

```
    if (i % 2 == 0) {
```

$\langle \mathtt{res} = \sum_{j=0}^{\lfloor \frac{\mathtt{i}+1}{2} \rfloor - 1} 4^j \wedge \mathtt{y} = 2^{\mathtt{i}} \wedge \mathtt{i} \leq 2 \cdot \mathtt{n} + 1 \wedge \mathtt{i} < 2 \cdot \mathtt{n} + 1 \wedge \mathtt{i} \mod 2 = 0 \rangle$
$\langle (\mathtt{res} + \mathtt{y}) = \sum_{j=0}^{\lfloor \frac{(\mathtt{i}+1)+1}{2} \rfloor - 1} 4^j \wedge \mathtt{y} \cdot 2 = 2^{(\mathtt{i}+1)} \wedge (\mathtt{i}+1) \leq 2 \cdot \mathtt{n} + 1 \rangle$

```
        res = res + y;
```

$\langle \mathtt{res} = \sum_{j=0}^{\lfloor \frac{(\mathtt{i}+1)+1}{2} \rfloor - 1} 4^j \wedge \mathtt{y} \cdot 2 = 2^{(\mathtt{i}+1)} \wedge (\mathtt{i}+1) \leq 2 \cdot \mathtt{n} + 1 \rangle$

```
    }
```

$\langle \mathtt{res} = \sum_{j=0}^{\lfloor \frac{(\mathtt{i}+1)+1}{2} \rfloor - 1} 4^j \wedge \mathtt{y} \cdot 2 = 2^{(\mathtt{i}+1)} \wedge (\mathtt{i}+1) \leq 2 \cdot \mathtt{n} + 1 \rangle$

```
    y = y * 2 ;
```

$\langle \mathtt{res} = \sum_{j=0}^{\lfloor \frac{(\mathtt{i}+1)+1}{2} \rfloor - 1} 4^j \wedge \mathtt{y} = 2^{(\mathtt{i}+1)} \wedge (\mathtt{i}+1) \leq 2 \cdot \mathtt{n} + 1 \rangle$

```
    i = i + 1;
```

$\langle \mathtt{res} = \sum_{j=0}^{\lfloor \frac{\mathtt{i}+1}{2} \rfloor - 1} 4^j \wedge \mathtt{y} = 2^{\mathtt{i}} \wedge \mathtt{i} \leq 2 \cdot \mathtt{n} + 1 \rangle$

```
}
```

$\langle \mathtt{res} = \sum_{j=0}^{\lfloor \frac{\mathtt{i}+1}{2} \rfloor - 1} 4^j \wedge \mathtt{y} = 2^{\mathtt{i}} \wedge \mathtt{i} \leq 2 \cdot \mathtt{n} + 1 \wedge \neg(\mathtt{i} < 2 \cdot \mathtt{n} + 1) \rangle$
$\langle \mathtt{res} = \sum_{j=0}^{\mathtt{n}} 4^j \rangle$

Zur Vervollständigung der Bedingungsregel 1 müssen wir noch zeigen

$$\mathtt{res} = \sum_{j=0}^{\lfloor \frac{\mathtt{i}+1}{2} \rfloor - 1} 4^j \wedge \mathtt{y} = 2^{\mathtt{i}} \wedge \mathtt{i} \leq 2 \cdot \mathtt{n} + 1 \wedge \mathtt{i} < 2 \cdot \mathtt{n} + 1 \wedge \neg(\mathtt{i} \mod 2 = 0)$$

$$\implies \mathtt{res} = \sum_{j=0}^{\lfloor \frac{(\mathtt{i}+1)+1}{2} \rfloor - 1} 4^j \wedge \mathtt{y} \cdot 2 = 2^{(\mathtt{i}+1)} \wedge (\mathtt{i}+1) \leq 2 \cdot \mathtt{n} + 1.$$

Aber $\neg(\mathtt{i} \mod 2 = 0)$ bedeutet nichts anderes, als dass $\mathtt{i}$ ungerade ist. Also gilt

$$\left\lfloor \frac{(\mathtt{i}+1)+1}{2} \right\rfloor - 1 = \left\lfloor \frac{\mathtt{i}}{2} \right\rfloor = \left\lfloor \frac{\mathtt{i}-1}{2} \right\rfloor,$$

d.h. die Implikation ist korrekt.

b) Wir wählen als Variante $V = 2 \cdot \mathtt{n} + 1 - \mathtt{i}$. Hiermit lässt sich die Terminierung von P beweisen, denn für die einzige Schleife im Programm (mit Schleifenbedingung $B = \mathtt{i} < 2 \cdot \mathtt{n} + 1$) gilt:

- $B \Rightarrow V \geq 0$, denn $B = \mathtt{i} < 2 \cdot \mathtt{n} + 1$ und

- die folgende Ableitung ist korrekt:

$\langle 2 \cdot \texttt{n} + 1 - \texttt{i} = m \wedge \texttt{i} < 2 \cdot \texttt{n} + 1 \rangle$

```
if (i % 2 == 0) {
```

$\langle 2 \cdot \texttt{n} + 1 - \texttt{i} = m \wedge \texttt{i} < 2 \cdot \texttt{n} + 1 \wedge \texttt{i} \mod 2 = 0 \rangle$

$\langle 2 \cdot \texttt{n} + 1 - (\texttt{i} + 1) < m \rangle$

```
    res = res + y;
```

$\langle 2 \cdot \texttt{n} + 1 - (\texttt{i} + 1) < m \rangle$

```
}
```

$\langle 2 \cdot \texttt{n} + 1 - (\texttt{i} + 1) < m \rangle$

```
y = y * 2 ;
```

$\langle 2 \cdot \texttt{n} + 1 - (\texttt{i} + 1) < m \rangle$

```
i = i + 1;
```

$\langle 2 \cdot \texttt{n} + 1 - \texttt{i} < m \rangle$

Zur Vervollständigung der Bedingungsregel 1 müssen wir noch folgende Implikation zeigen.

$$\begin{aligned} & 2 \cdot \texttt{n} + 1 - \texttt{i} = m \wedge \texttt{i} < 2 \cdot \texttt{n} + 1 \wedge \neg(\texttt{i} \mod 2 = 0) \\ \Longrightarrow\ & 2 \cdot \texttt{n} + 1 - (\texttt{i} + 1) < m \end{aligned}$$

Aber diese gilt für jede ganze Zahl $\texttt{i}$.

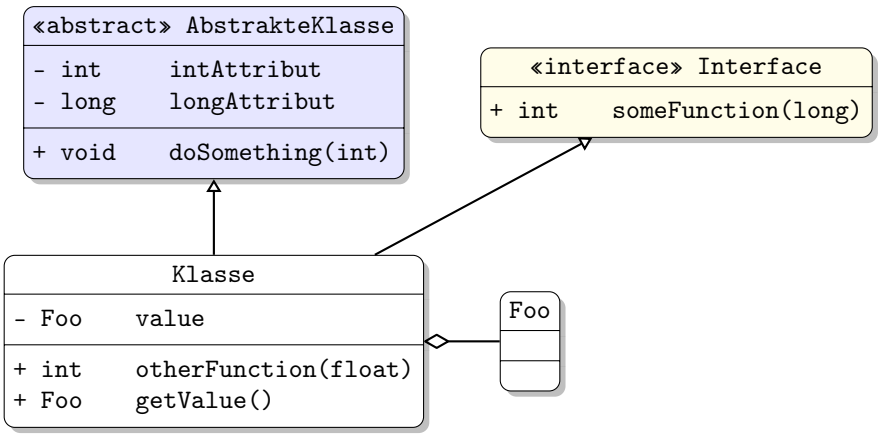
Aufgabe 3 (Klassenhierarchie): (6 + 7 = 13 Punkte)

Ziel dieser Aufgabe ist die Erstellung einer Hierarchie zur Verwaltung eines Übungsbetriebs der fiktiven Universität *Öcher Wirklich Tolle Hochschule*. Ähnlichkeiten zu real existierenden Übungsbetrieben sind rein zufällig.

- Aufgaben haben einen Namen, der als `String` gespeichert wird.
- Eine Punkteaufgabe ist eine Aufgabe. Sie hat eine ganze Zahl von maximal zu erreichenden Punkten. Außerdem enthält sie die Angabe, ob die Abgabe per VPL zu tätigen ist oder nicht.
- Eine Tutoriumsaufgabe ist eine Aufgabe. Sie hat ein Attribut `hausaufgabe` vom Typ `Punkteaufgabe`. Auf diese Hausaufgabe bereitet die Tutoriumsaufgabe vor.
- Ein Codescape-Deck ist eine Aufgabe. Es hat ein Array von Codescape-Missionen.
- Jede Aufgabe ist entweder Tutoriumsaufgabe, Punkteaufgabe oder Codecape-Deck. Es soll keine Aufgaben geben, die nicht einem dieser Typen zugeordnet sind.
- Ein Übungsblatt hat ein Array von Aufgaben und eine Nummer. Außerdem kann über die Methode `void addAufgabe(Aufgabe aufgabe)` eine Aufgabe zum aktuellen Übungsblatt hinzugefügt werden.
- Eine Präsenzübung hat ein Array von Punkteaufgaben.
- Manchmal unterlaufen den Assistenten, die den Übungsbetrieb organisieren, Fehler bei bestimmten Punkteaufgaben und Codescape-Missionen. Da die Personen, die am Übungsbetrieb teilnehmen, nichts dafür können, sollen sie für diese Punkteaufgaben und Codescape-Missionen automatisch das bestmögliche Abschneiden angerechnet bekommen: Punkteaufgaben und Codescape-Missionen sind *verschenkbar*. Die Methode `void schenken()` sorgt dafür, dass diese als vollständig erledigt gelten. Diese Methode muss von beiden Klassen implementiert werden.

a) Entwerfen Sie unter Berücksichtigung der Prinzipien der Datenkapselung eine geeignete Klassenhierarchie für die oben aufgelisteten Klassen des Übungsbetriebs. Notieren Sie keine Konstruktoren oder Selektoren. Sie müssen nicht markieren, ob Attribute `final` sein sollen. Achten Sie darauf, dass gemeinsame Merkmale in Oberklassen bzw. Interfaces zusammengefasst werden und markieren Sie alle Klassen als abstrakt, bei denen dies sinnvoll ist.

Verwenden Sie hierbei die folgende Notation:



Eine Klasse wird hier durch einen Kasten dargestellt, in dem der Name der Klasse sowie alle in der Klasse definierten Attribute und Methoden in einzelnen Abschnitten beschrieben werden. Überschriebene Methoden müssen nicht angegeben 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). Der Pfeil $B \diamond\!\!-\!\!> A$ bedeutet, dass A ein Objekt vom Typ B benutzt. Benutzen Sie `-`, um `private` abzukürzen, und `+` für alle anderen Sichtbarkeiten (wie z. B. `public`). Fügen Sie Ihrem Diagramm keine Kästen für vordefinierte Klassen wie `String` hinzu.

b) Für den Fall, dass die Abgabe über VPL für bestimmte Übungsblätter fehlschlägt, soll eine Methode `verschenkeVPL` erstellt werden. Beim Aufruf dieser Methode werden die entsprechenden Aufgaben automatisch als bestmöglich bestanden gewertet.

Schreiben Sie eine Java-Methode `verschenkeVPL`. Auf den übergebenen Übungsblättern soll diese Methode auf allen Punkteaufgaben `schenken()` aufrufen, bei denen die Abgabe per VPL zu tätigen war.

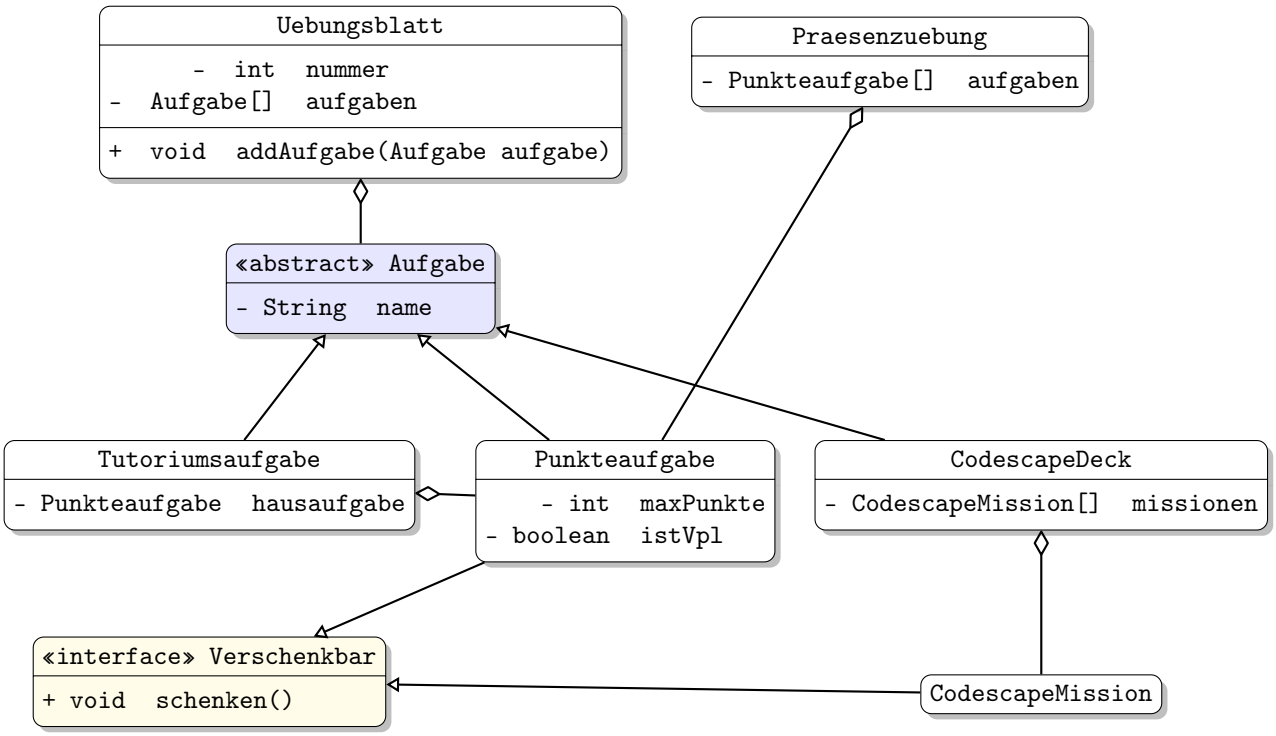
Hinweise:

- Nehmen Sie dazu an, dass der übergebene Parameter `uebungen` nicht `null` ist und dass bei jedem Übungsblatt das Array von Aufgaben nicht `null` ist.
- Gehen Sie davon aus, dass es für alle Attribute geeignete Selektoren gibt.

```
public static void verschenkeVPL(Uebungsblatt... uebungen) {
```

Lösung:

a) Die Zusammenhänge können wie folgt modelliert werden:



b)

```
public static void verschenkeVPL(Uebungsblatt... uebungen) {
    for (Uebungsblatt u : uebungen) {
        for (Aufgabe a : u.getAufgaben()) {
            if (a instanceof Punkteaufgabe) {
                Punkteaufgabe pa = (Punkteaufgabe)a;
                if (pa.istVpl()) {
                    pa.schenken();
                }
            }
        }
    }
}
```

Aufgabe 4 (Programmierung in Java): (2 + 3 + 10 + 5 + 9 = 29 Punkte)

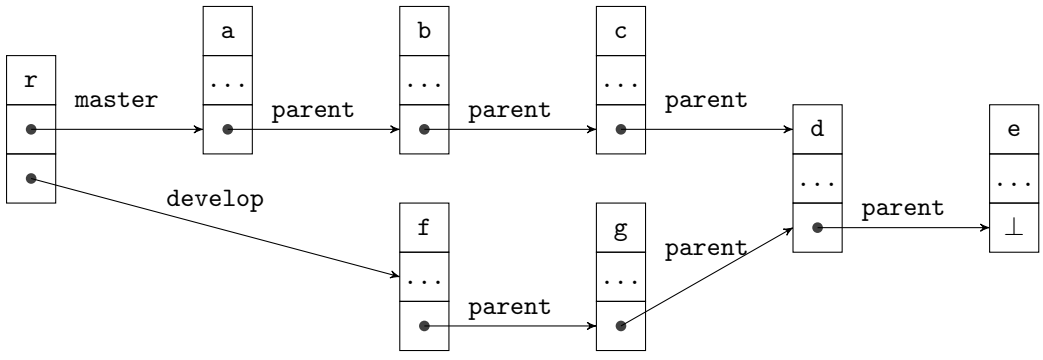
In dieser Aufgabe wollen wir die Klassen `Repository` und `Commit` zur Realisierung eines Versionskontrollsystems implementieren. Ein Versionskontrollsystem dient dazu, verschiedene Versionen eines Texts zu verwalten. Ein `Commit`-Objekt stellt eine Version dar. Es enthält den Text im Attribut `content` und hat im Attribut `parent` eine Referenz auf den `Commit`, welcher die ältere *Vorgänger*-Version darstellt. Für einen `Commit`, welcher eine Version ohne Vorgänger darstellt, hat `parent` den Wert `null`.

```
public class Commit {
  private final String content;
  private final Commit parent;

  public Commit(String content, Commit parent) {
    this.content = content;
    this.parent = parent;
  }

  public String getContent() {
    return content;
  }

  public Commit getParent() {
    return parent;
  }
}
```

Ein `Repository`-Objekt stellt eine Versionshistorie dar, welche zwei Änderungsabfolgen (`master` und `develop`) verwalten kann. Diese beiden Änderungsabfolgen werden wir im folgenden als *Branches* bezeichnen.

```
public class Repository {
  private Commit master;
  private Commit develop;
}
```

In der folgenden Grafik ist beispielhaft dargestellt, wie eine Versionshistorie `r` (vom Typ `Repository`) mit den Versionen `a` bis `g` (vom Typ `Commit`) repräsentiert wird.



Wir sehen acht Objekte, dargestellt durch je drei übereinander liegende Quadrate. Das oberste Quadrat enthält dabei immer einen Bezeichner, sodass wir uns im Laufe dieser Aufgabe auf konkrete Objekte aus diesem Beispiel beziehen können. Das Objekt `r` ist ein `Repository`-Objekt, dessen `master`-Attribut auf das Objekt `a` zeigt und dessen `develop`-Attribut auf das Objekt `f` zeigt. Die Objekte `a` bis `g` sind `Commit`-Objekte, deren `parent`-Attribut je auf ein anderes `Commit`-Objekt zeigt. Das `parent`-Attribut des Objekts `e` hat den Wert `null` (⊥). Außerdem enthält jedes `Commit`-Objekt im zweiten Quadrat einen nicht näher angegebenen `content`, dargestellt durch drei Punkte.

Hinweise:

- Sie dürfen in allen Teilaufgaben davon ausgehen, dass nur auf azyklischen Versionshistorien gearbeitet wird (d.h. man erreicht keinen Zyklus, wenn man nur den `parent`-Referenzen folgt).

- Sie dürfen außerdem in allen Teilaufgaben davon ausgehen, dass kein `content`-Attribut eines `Commit`-Objekts den Wert `null` hat.

a) Implementieren Sie in der Klasse `Repository` die Methode `commitToMaster`, welche in den `master`-Branch einen neuen `Commit` mit dem gegebenen `content` einfügt.

Sei `r` das `Repository`-Objekt aus der Aufgabenstellung. Wenn beispielsweise

```
r.commitToMaster("new content")
```

aufgerufen wird, so soll ein neues `Commit`-Objekt `h` erstellt werden, dessen `content`-Attribut auf den String `"new content"` zeigt und dessen `parent`-Attribut auf das Objekt `a` zeigt. Außerdem soll das `master`-Attribut des Objekts `r` nun auf das neu erstellte Objekt `h` zeigen.

Hinweise:

- Sie dürfen davon ausgehen, dass der Parameter `content` der Methode `commitToMaster` nicht `null` ist.
- Es ist nicht notwendig, die Methode `commitToDevelop` zu implementieren.

```
public void commitToMaster(String content) {
```

b) Implementieren Sie in der Klasse `Commit` die Methode `getRootCommit`, welche so lange den Verweisen der `parent`-Attribute folgt, bis sie ein `Commit`-Objekt findet, dessen `parent`-Attribut den Wert `null` hat. Die Methode soll dieses gefundene `Commit`-Objekt zurückgeben.

Sei `a` das `Commit`-Objekt aus der Aufgabenstellung. Wenn beispielsweise

```
a.getRootCommit()
```

aufgerufen wird, so soll eine Referenz auf das Objekt `e` zurückgegeben werden.

Verwenden Sie in dieser Teilaufgabe keine Schleifen, sondern **ausschließlich Rekursion**.

```
public Commit getRootCommit() {
```

c) Implementieren Sie in der Klasse `Commit` die Methode `findFirstCommonPredecessorWith`, welche zu den beiden gegebenen `Commit`-Objekten (`this` und der Parameter `other`) den ersten gemeinsamen Vorgänger findet. Der erste gemeinsame Vorgänger ist das jüngste `Commit`-Objekt, welches von beiden gegebenen `Commit`-Objekten erreichbar ist. Insbesondere wird nie `null` zurückgegeben. Existiert ein solcher gemeinsamer Vorgänger nicht, so wirft die Methode eine `NoCommonPredecessorException`, welche wie folgt definiert ist:

```
public class NoCommonPredecessorException extends Exception {}
```

Seien `a` und `f` die `Commit`-Objekte aus der Aufgabenstellung. Wenn beispielsweise

```
a.findFirstCommonPredecessorWith(f)
```

aufgerufen wird, so soll eine Referenz auf das Objekt `d` zurückgegeben werden. Wenn hingegen

```
a.findFirstCommonPredecessorWith(a)
```

aufgerufen wird, so soll eine Referenz auf das Objekt `a` zurückgegeben werden. Wenn

```
a.findFirstCommonPredecessorWith(new Commit("", null))
```

aufgerufen wird, so soll eine `NoCommonPredecessorException` geworfen werden.

Verwenden Sie in dieser Teilaufgabe keine Rekursion, sondern **ausschließlich Schleifen**.

Hinweise:

- Sie dürfen davon ausgehen, dass der Parameter `other` der Methode `findFirstCommonPredecessorWith` nicht `null` ist.

```
public Commit findFirstCommonPredecessorWith(Commit other)
    throws NoCommonPredecessorException {
```

d) Implementieren Sie die Klasse `FilterImpl`, welche das nachfolgende Interface `Filter` implementiert. Dabei soll die Methode `matches` überprüfen, ob das `content`-Attribut des Parameters `commit` einen `String` enthält, dessen Länge kleiner oder gleich der Zahl 1000 ist und einen entsprechenden `boolean`-Wert zurückgeben.

```
public interface Filter {
  boolean matches(Commit commit);
}
```

Wenn beispielsweise

```
new FilterImpl().matches(new Commit("asdf", null))
```

aufgerufen wird, so soll `true` zurückgegeben werden. Wenn hingegen

```
new FilterImpl().matches(new Commit(s, null))
```

aufgerufen wird, wobei `s` ein `String` ist, der mehr als 1000 Zeichen umfasst, so soll `false` zurückgegeben werden.

Hinweise:

- Sie dürfen davon ausgehen, dass der Parameter `commit` der Methode `matches` nicht `null` ist.
- Die Methode `length` gibt die Länge eines `String`s zurück. So wird etwa `"asdf".length()` zu dem `int`-Wert `4` ausgewertet.

e) Implementieren Sie in der Klasse `Commit` die Methode `toList`, welche ein neues Objekt vom Typ `LinkedList<String>` erstellt und zurückgibt. Der erste Eintrag der zurückgegebenen Liste ist der Wert des `content`-Attributs des aktuellen `Commit`-Objekts. Die weiteren Einträge sind die Werte der `content`-Attribute all seiner Vorgänger. Der letzte Eintrag der zurückgegebenen Liste soll den Wert des `content`-Attributs des `Commit`-Objekts enthalten, welches die älteste Version darstellt. Dabei werden `Commit`-Objekte ignoriert, für welche die `matches`-Methode des `filter` Parameters den Wert `false` zurückgibt.

Seien `a` bis `e` die `Commit`-Objekte aus der Aufgabenstellung, wobei wir die `String`s in ihren `content`-Attributen im Folgenden mit `c1` bis `c5` bezeichnen. Wir nehmen an, dass `c2` und `c3` eine Länge größer als 1000 haben und dass `c1`, `c4` und `c5` eine Länge kleiner oder gleich 1000 haben. Wenn beispielsweise

```
a.toList(new FilterImpl())
```

aufgerufen wird, so soll eine Liste zurückgegeben werden, welche die Elemente `c1`, `c4` und `c5` in dieser Reihenfolge enthält.

Verwenden Sie in dieser Teilaufgabe keine Rekursion, sondern **ausschließlich Schleifen**.

Hinweise:

- `LinkedList<T>` und `List<T>` bezeichnen Typen, welche sich im Paket `java.util` befinden.
- Sie dürfen die Methode `boolean add(T t)` aus dem Interface `List<T>` benutzen, um hinten an eine Liste weitere Elemente anzuhängen.
- Sie dürfen davon ausgehen, dass der Parameter `filter` der Methode `toList` nicht `null` ist.

```
public List<String> toList(Filter filter) {
```

Lösung:

a)
```
public void commitToMaster(String content) {
  master = new Commit(content, master);
}
```

b)
```
public Commit getRootCommit() {
  if (parent == null) {
    return this;
  } else {
    return parent.getRootCommit();
  }
}
```

c)
```
public Commit findFirstCommonPredecessorWith(Commit other)
    throws NoCommonPredecessorException {
  Commit current1 = this;
  while (current1 != null) {
    Commit current2 = other;
    while (current2 != null) {
      if (current1 == current2) {
        return current1;
      }
      current2 = current2.getParent();
    }
    current1 = current1.getParent();
  }
  throw new NoCommonPredecessorException();
}
```

d)
```
public class FilterImpl implements Filter {
  @Override
  public boolean matches(Commit commit) {
    return commit.getContent().length() <= 1000;
  }
}
```

e)
```
public List<String> toList(Filter filter) {
  List<String> result = new LinkedList<>();
  Commit current = this;
  while (current != null) {
    if (filter.matches(current)) {
      result.add(current.getContent());
    }
    current = current.getParent();
  }
  return result;
}
```