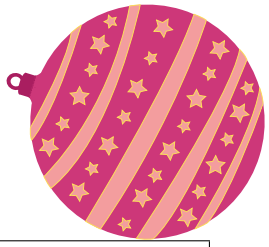
Praktikum Grundlagen der Programmierung

Prof. Dr. Harald Räcke, Prof. Dr. Felix Brandt, J. Kranz, A. Reuss WS 2017/18

Übungsblatt 9 Abgabefrist: 8.1.2018 um 05:00 Uhr

Σ Punkte: 20 | Σ Bonuspunkte: 3

Aufgabe 9.1 (P) Polymorphie



Betrachten Sie die Aufrufe 1 bis 4 in dem folgenden Programm.

```
1  public class A {
2     public int  min(C c, B b) { return 0; }     // Methode 1
3     public void min(A a, B b) { }               // Methode 2
4  }
5  public class B extends A {
6     public void min(A a1, A a2) { }             // Methode 3
7  }
8  public class C extends B {
9     public B min(A a, C c) { return new B(); } // Methode 4
10 }
11 public class Poly {
12    public static void main (String[] args) {
13       A a = (B)(new C());
```

```
14          B b = new B();
15          C c = new C();
16          c.min(a, c);          // Aufruf 1
17          b.min(a, (B)c);       // Aufruf 2
18          ((B)c).min(c, (B)c); // Aufruf 3
19          ((A)b).min((B)a, b); // Aufruf 4
20      }
21  }
```

Bestimmen Sie für jeden der vier Aufrufe $e_0.f(e_1, \ldots, e_k)$

a) die statischen Typen $T_0, \ldots, T_k$ der Ausdrücke $e_0, \ldots, e_k$.

b) die Signatur der Methode f an dieser Aufrufstelle.

c) den dynamischen Typ des Objekts, zu dem sich e_0 auswertet.

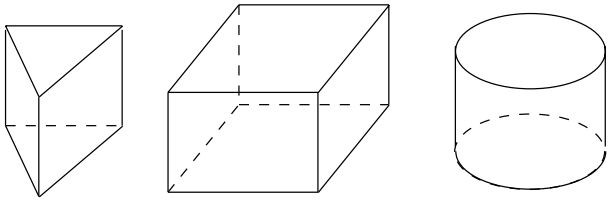
d) die aufgerufene Methode (1, 2, 3 oder 4).

Schreiben Sie die Lösung für a) bis c) in die folgende Tabelle. Geben Sie dabei $T_0, \ldots, T_k$ für a) als mit Komma getrennte *geordnete* Liste an. Bestimmen Sie die aufgerufene Methode für c) durch Angabe der entsprechenden Methodennummer.

Aufruf	a) (als Liste)	b)	c)	d) Methode
1	`C, A, C`	`min(A, C)`	`C`	`4`
2	`B, A, B`	`min(A, B)`	`B`	`2`
3	`B, C, B`	`min(C, B)`	`C`	`1`
4	`A, B, B`	`min(B, B)`	`B`	`2`

Aufgabe 9.2 (P) Alle Jahre wieder (Modellierung)

Prismen oder *Zylinder* sind geometrische Körper, die durch Parallelverschiebung ihrer Grundfläche im Raum entstehen. Die folgende Abbildung zeigt als Beispiele für Prismen ein Dreiecksprisma, einen Quader und einen Zylinder, die durch Verschiebung eines Dreiecks, eines Rechtecks bzw. eines Kreises entstehen:



Prismen sind durch ihre Höhe und ihre jeweilige Grundfläche bestimmt. Als Grundflächen in Frage kommen (zum Beispiel):

- *regelmäßige n-Ecke*; bestimmt durch die Anzahl der Ecken und die Seitenlänge
- *Kreise*; bestimmt durch ihren Radius
- *Rechtecke*; bestimmt durch Breite und Länge

Eine Grundfläche muss die Berechnung von Umfang und Flächeninhalt zur Verfügung stellen; ein Prisma die Berechnung von Oberfläche und Volumen.

In dieser Aufgabe soll eine Klassenhierarchie für diese geometrischen Körper in Java implementiert werden.

1. Erstellen Sie die Klasse `Grundflaeche.java` als Oberklasse für alle Grundflächen. Legen Sie Basisversionen der Methoden `umfang()` und `flaeche()` sowie eine geeignete `toString()`-Methode an.
2. Erstellen Sie Klassen `Kreis`, `Rechteck` und `NEck` zur Repräsentation von Kreisen, Rechtecken und n-Ecken. Diese Klassen haben `Grundflaeche` als Oberklasse und erweitern die Oberklasse um die benötigten Objektvariablen und überschreiben die Methoden zur Flächen- und Umfangberechnung. Ergänzen Sie die `toString()`-Methode sinnvoll. Verwenden Sie nur Integer bei den Objektvariablen, um Ungenauigkeiten bei den arithmetischen Operationen zu vermeiden.

3. Erstellen Sie die Klasse `Prisma` mit den benötigten Objektvariablen und den Methoden `volumen()` und `oberflaeche()` sowie einer geeigneten `toString()`-Methode. Verwenden Sie wie zuvor bei Objektvariablen nur Integer für Größenangaben.
4. Ergänzen Sie alle Grundflächen um die Methode `istQuadrat()`, die zurückgibt, ob es sich bei der aktuellen Instanz um ein Quadrat handelt. Z.B. ist ein Rechteck ein Quadrat, falls Länge und Breite gleich sind.
5. Erstellen Sie die Klasse `Quadrat`, welche durch ihre Seitenlänge eindeutig bestimmt ist, als weitere Unterklasse von `Grundflaeche` und ergänzen Sie alle Grundflächen um eine Methode `zuQuadrat()`, die, falls es sich um ein Quadrat (auch passende Rechtecke und n-Ecke) handelt, ein `Quadrat`-Objekt mit der entsprechenden Seitenlänge zurückgibt. Handelt es sich nicht um ein Quadrat, dann wird `null` zurückgegeben.
6. Ergänzen Sie die Klasse `Prisma` um eine Methode `istWuerfel()`, die zurückgibt, ob es sich bei dem aktuellen Prisma um einen Würfel handelt.
7. Testen Sie Ihren Code.

Hinweise:

- Planen Sie vor dem Programmieren die Klassenhierarchie.
- Fassen Sie gleichartige Attribute und Methoden in einer geeigneten Oberklasse zusammen.
- Verwenden Sie Getter-Methoden.
- Greifen Sie, falls möglich, auf bereits implementierte Methoden zurück.
- Die Fläche eines regelmäßigen n-Ecks mit Seitenlänge a ist $\frac{n*a^2}{4*\tan(\frac{\pi}{n})}$.
- Die Java-Klasse `Math` stellt in der Klassenvariablen `Math.PI` einen Wert für π sowie in der Klassenmethode `Math.tan()` die Berechnung des Tangens zur Verfügung.
- Testen Sie ihre Implementierung mit einer geeigneten Test-Klasse.

Lösungsvorschlag 9.2

Die Lösung befindet sich im Ordner `modellierung_loesung`.

Aufgabe 9.3 (P) Geometrische Figuren mit Visitor-Pattern

In der vorherigen Aufgabe haben wir virtuelle Methoden verwendet, um Funktionalität in einer Basisklasse anzubieten, die dann von jeder Unterklasse verschieden implementiert wurde. Dieser Ansatz ist in vielen Fällen auch genau richtig, hat allerdings in manchen Situationen Nachteile:

- Die Implementierung wird über mehrere Klassen verteilt. Dies kann wünschenswert sein, kann das Verständis von Code aber auch schwieriger machen, wenn sehr viel zwischen verschiedenen Klassen gesprungen werden muss, um eine bestimmte Methode nachzuvollziehen.
- Die Klassen in der Hierarchie müssen angepasst werden. Stellen Sie sich vor, Sie verwenden eine Hierarchie von Klassen an, die an unterschiedlichen Stellen in Ihrem Code verwendet wird. Wann immer Sie Funktionalität benötigen, die nicht für alle Elemente in der Hierarchie auf die gleiche Weise implementierbar ist, müssen Sie alle Klassen um virtuelle Methoden erweitern. Dies kann dazu führen, dass Ihre Klassen viele Methoden enthalten, die nur für Spezialzwecke in „Ecken“ Ihres Programms gebraucht werden. Es kann außerdem sein, dass Sie eine Bibliothek anbieten möchten, die jemand anderes in seinem Code verwenden kann. Dieser *Client* kann dann keine eigenen Methoden zu Ihrer Hierarchie hinzufügen.

Die oben genannten Probleme löst das sog. *Visitor-Pattern*[1], indem es eine allgemeine virtuelle Methode `public void accept(Visitor visitor` jeder Klasse der Hierarchie hinzufügt, die dann von verschiedenen *Besuchern* verwendet werden kann, um jeweils eine bestimmte Funktionalität außerhalb der Hierarchie zu implementieren. Dazu implementieren die Besucher für jede Unterklasse `U` eine Methode `public void visit(U u)`, die jeweils von `accept()` aufgerufen wird.

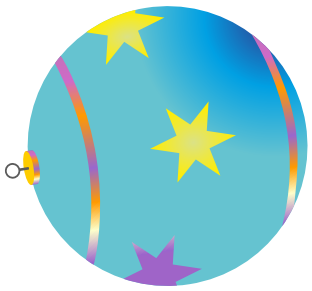
Wir gehen in dieser Aufgabe davon aus, dass wir die Berechnung der Fläche einer geometrischen Figur mit einem Visitor durchführen wollen, statt durch Implementierung in den jeweiligen Unterklassen. Wir arbeiten daher nun mit folgender Hierarchie:

```java
public class Grundflaeche {
  public void accept(Visitor visitor) {
  }
}

public class Kreis extends Grundflaeche {
  private int radius;

  public int getRadius() {
    return radius;
  }

  public Kreis(int radius) {
    this.radius = radius;
  }
```



[1] https://en.wikipedia.org/wiki/Visitor_pattern

```
16  }
17
18  public class NEck extends Grundflaeche {
19    private int laenge;
20
21    public int getLaenge() {
22      return laenge;
23    }
24
25    private int n;
26
27    public int getN() {
28      return n;
29    }
30
31    public NEck(int n, int laenge) {
32      this.n = n;
33      this.laenge = laenge;
34    }
35  }
36
37  public class Quadrat extends Grundflaeche {
38    private int laenge;
39
40    public int getLaenge() {
41      return laenge;
42    }
43
44    public Quadrat(int laenge) {
45      this.laenge = laenge;
46    }
47  }
48
49  public class Rechteck extends Grundflaeche {
50    private int breite;
51
52    public int getBreite() {
53      return breite;
54    }
55
56    private int laenge;
57
58    public int getLaenge() {
59      return laenge;
60    }
61  }
```

Implementieren Sie das Visitor-Pattern und die geforderte Funktionalität zur Flächenberechnung in den folgenden Schritten.

1. Implementieren Sie die `accept()`-Methode in jeder Klasse sinnvoll.
2. Implementieren Sie die Klasse `Visitor`, die die Basisklasse für alle Besucher in dieser Aufgabe darstellt. Sie bietet eine `visit()`-Methode für jede Unterklasse von `Grundflaeche` an.
3. Implementieren Sie schließlich die Klasse `FlaechenVisitor`, der die Fläche einer beliebigen geometrischen Figur der Hierarchie berechnet.

Hinweis: Sie dürfen den oben gegebenen Klassen keine weiteren Methoden (außer `accept()`) hinzufügen.

Lösungsvorschlag 9.3

Die Lösung befindet sich im Ordner `visitor_loesung`.

Aufgabe 9.4 (P) Doppelt verlistete Kettung

In dieser Aufgabe soll eine doppelt verkettete Liste implementiert werden. In einer doppelt verketteten Liste hat jedes Listenelement neben einer Referenz auf das nächste Element auch eine Referenz auf das vorherige Element. Gibt es kein nächstes oder vorheriges Element, ist die Referenz `null`.

Verwenden Sie das vorgegebene Programmgerüst in `DoublyLinkedList.java`. Der vorgegebene Code darf nicht verändert werden. Sie dürfen jedoch weitere Membervariablen oder Methoden in der Klasse `DoublyLinkedList` hinzufügen (nicht notwendig!). In der Klasse `Entry` darf kein weiterer Code hinzugefügt werden. Die Klasse `class Entry` ist eine innere Klasse der Klasse `DoublyLinkedList`. D. h. aus der Klasse `DoublyLinkedList` kann auf die Membervariablen der Klasse `Entry` zugegriffen werden. Die Klasse `DoublyLinkedList` repräsentiert die doppelt verkettete Liste. Die Klasse `Entry` repräsentiert ein Listenelement in dieser Liste. Implementieren Sie die folgenden Methoden und den Konstruktor:

1. Der Konstruktor `public DoublyLinkedList()` erzeugt eine leere `DoublyLinkedList`. Eine `DoublyLinkedList` ist leer, wenn die Referenz `head` `null` ist.
2. Die Methode `public int size()` liefert als Rückgabewert die Anzahl der Elemente der `DoublyLinkedList`.
3. Die Methode `public void add(int info)` fügt am Ende der `DoublyLinkedList` ein neues `Entry`-Element hinzu. Die Membervariable `elem` dieses `Entry`-Elements hat den Wert des Methodenparameters `info`. Implementieren Sie diese Methode *rekursiv*. Dazu benötigen Sie eine zusätzliche Hilfsmethode. Überlegen Sie sich, welche Parameter diese Hilfsmethode benötigt, und implementieren Sie sie entsprechend. Deklarieren Sie Ihre Hilfsmethode als `private`.
4. Die Methode `public void add(int index, int info)` fügt an der Position `index` der `DoublyLinkedList` ein `Entry`-Element ein. Die Membervariable `elem` dieses `Entry`-Elements hat den Wert des Methodenparameters `info`. Der Index beginnt bei 0, d. h. ist `index == 0`, wird der `head` der `DoublyLinkedList` verändert, ist `index == 1`, wird das neue Element als zweites Element eingefügt.

5. Die Methode `public int remove(int index)` entfernt das `Entry`-Element an Position `index` und liefert als Rückgabewert die Membervariable `elem` des entfernten `Entry`-Elements. Der Index beginnt bei 0, d. h. bei `index == 0` wird das erste `Entry`-Element der `DoublyLinkedList` entfernt usw. Ist der Index ungültig (z. B. negativ), wird kein Element entfernt und der Rückgabewert ist `Integer.MIN_VALUE`.

6. Die Methode `public void shiftLeft(int index)` verschiebt die Elemente der `DoublyLinkedList` um `index` Positionen nach links. D. h. das Element an Position `index` wird das erste Element. Die Elemente, die nach links „aus“ der Liste geschoben würden, werden am Ende der Liste entsprechend der Reihenfolge angehangen. Die Verschiebung einer Liste mit n Elementen erfolgt nur, wenn `index` im Bereich 0 bis $n - 1$ liegt. Beispiele: $[0, 1, 2, 4]$ wird bei `shiftLeft(2)` zu $[2, 4, 0, 1]$, $[0, 1]$ wird bei `shiftLeft(1)` zu $[1, 0]$, bei `shiftLeft(0)` bleibt die Liste unverändert. Es dürfen nur die `next`-, `prev`- und `head`-Referenzen der `Entry`-Elemente sowie die Objektvariable `head` der `DoublyLinkedList` verändert werden. D. h. insbesondere, dass teilweise oder vollständige Kopien der `DoublyLinkedList` nicht verwendet werden dürfen.

Lösungsvorschlag 9.4

Die Lösung befindet sich in `DoublyLinkedList.java`.

Die Hausaufgabenabgabe erfolgt über Moodle. Bitte geben Sie Ihren Code als UTF8-kodierte (ohne BOM) Textdatei(en) mit der Dateiendung `.java` ab. Geben Sie **keine** Projektdateien Ihrer Entwicklungsumgebung ab. Geben Sie **keinen** kompilierten Code ab (`.class`-Dateien). Wenn Sie Ihre Dateien als Archiv abgeben möchten, verwenden Sie bitte ausschließlich `.zip`-Dateien. Sie dürfen Packages verwenden; erstellen Sie jedoch nicht mehr als ein Package pro Aufgabe. Achten Sie darauf, dass Ihr Code kompiliert. Bitte vermerken Sie aus Datenschützgründen nicht Ihren Namen oder Ihre Matrikelnummer im Code. Hausaufgaben, die sich nicht im vorgegebenen Format befinden, werden nur mit Punktabzug oder gar nicht bewertet.

Aufgabe 9.5 (H) Weihnachtsbaum schmücken [10 Punkte + 1 Bonuspunkte]

In dieser Aufgabe sollen Sie ein Spiel schreiben, in dessen Verlauf der Spieler einen Weihnachtsbaum baut und schmückt. Wir haben Ihnen bereits ein Grundgerüst erstellt, damit Sie möglichst schnell den interessanten Teil implementieren können. Das Bauen und

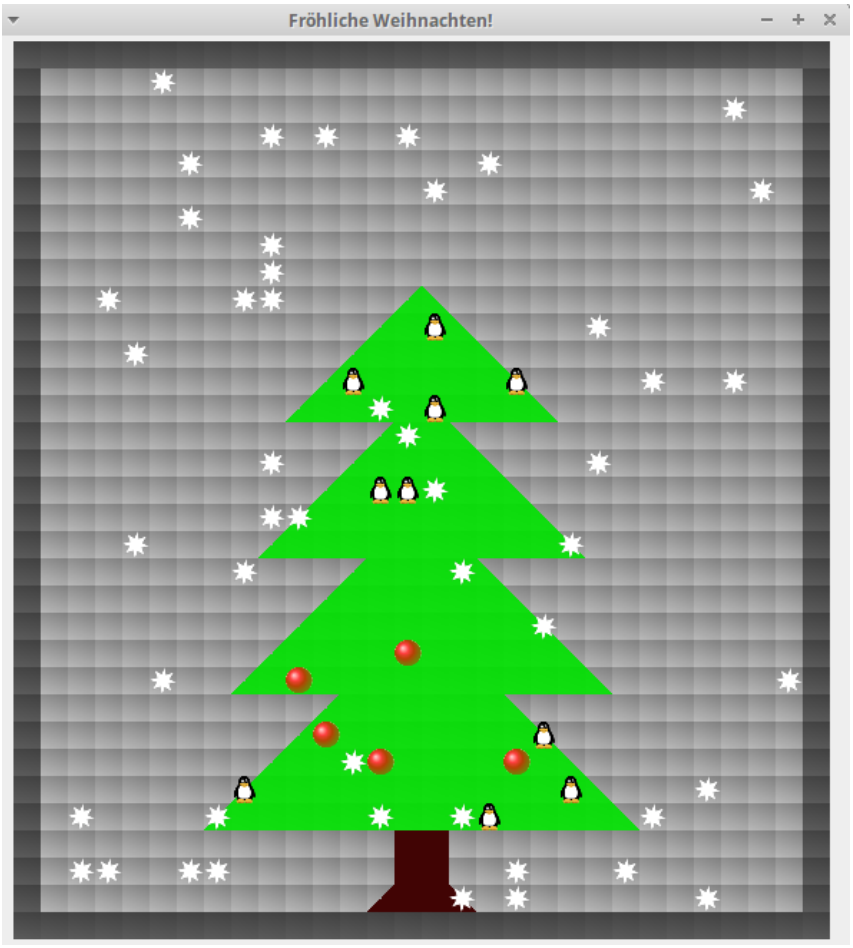


Abbildung 1: GUI: Weihnachtsbaum schmücken

Schmücken des Weihnachtsbaumes soll dabei Tetris-ähnlich funktionieren. Der Baum setzt sich aus `Weihnachtsobjekt`en zusammen, welche am oberen Rand des Spielfeldes erscheinen und im Spielverlauf um stets eine Zeile herunterfallen, bis sie auf entweder den Boden oder ein anderes `Weihnachtsobjekt` treffen. Ähnlich wie bei Tetris kann der Spieler die fallenden `Weihnachtsobjekte` nach links und rechts verschieben, um den schönsten Weihnachtsbaum zu schmücken. Damit Ihr Weihnachtsbaum eine einem echten Weihnachtsbaum angemessen ähnliche Form hat, muss die Spielerin eventuell fallende `Weihnachtsobjekte` zerstören. Dies kann der Spieler durch Bewegen der `Weihnachtsobjekte` an den linken oder rechten Rand des Spielfeldes erreichen.

Jedes `Weihnachtsobjekt` ist entweder ein `SingleObject`, das genau eine Zelle des Spielfeldes einnimmt, oder ein `MultiObject`, das mehrere Zellen des Spielfeldes einnimmt. Ein `MultiObject` setzt sich aus einer `Liste` von `SingleObjects` zusammen.

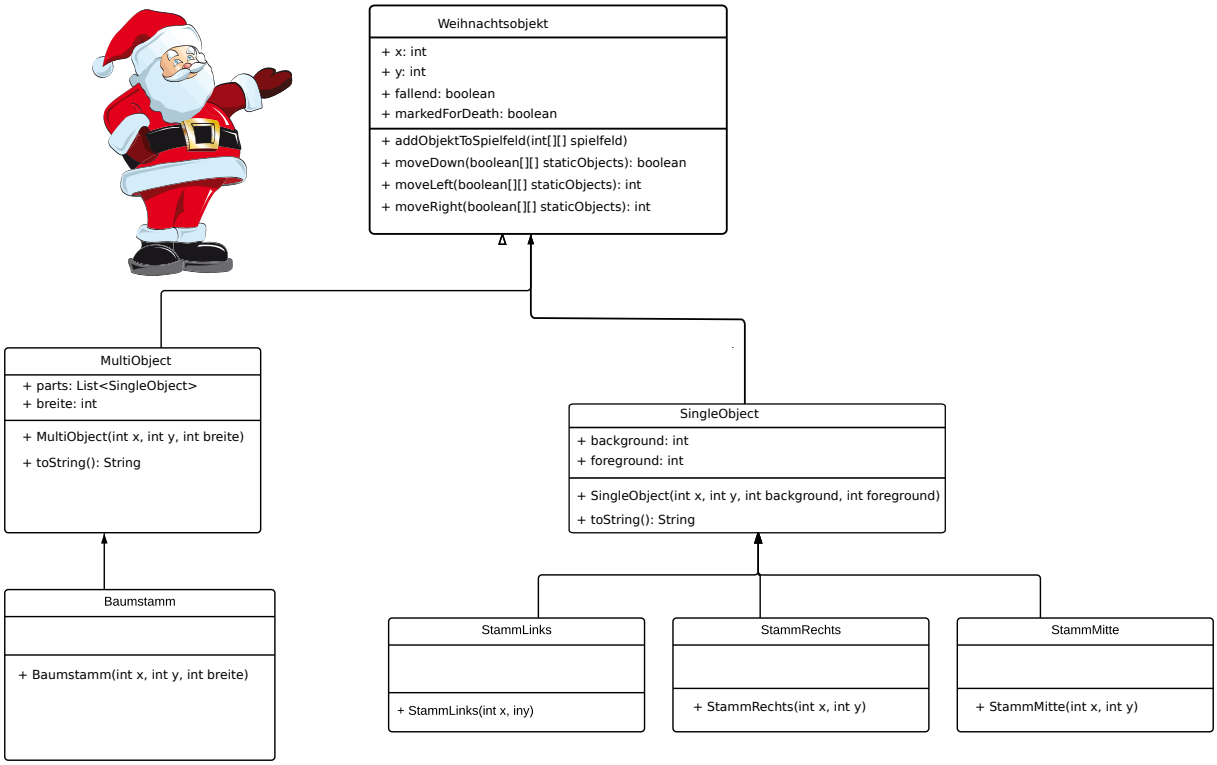


Abbildung 2: Einige Klassen

Um die Implementierung des Spiels zu vereinfachen, nutzen wir *Vererbung*. Schauen Sie sich Abbildung 2 an, um sich mit der Klassenhierarchie des Spiels vertraut zu machen. Dieses Diagramm stellt **nicht alle** Klassen dar, sondern verdeutlicht nur die Hierarchie der Klassen.

Im Folgenden werden die `Weihnachtsobjekte` und ihre korrespondierenden Klassen beschrieben. Sie dürfen diese Klassen beliebig erweitern, aber die hier aufgeführten Variablen und Methoden müssen existieren und wie beschrieben implementiert sein.

- `public class Weihnachtsobjekt`
  - `protected int x, y;` Die aktuelle Position des Objektes.
  - `protected boolean markedForDeath = false;` Ist das Objekt bzw. ein Teil davon über den linken oder rechten Rand verschoben worden, dann soll es gelöscht werden. D. h. es soll vom Spielfeld verschwinden (und nicht mehr angezeigt werden).
  - `public void addObjektToSpielfeld(int[][] spielfeld);` Fügt dieses Objekt dem Spielfeld hinzu.
  - `public boolean moveDown(boolean[][] staticObjects);` Das Objekt wird um eine Zeile nach unten verschoben, wenn es möglich ist. Dazu bekommt es die Liste bereits *gelandeter* Objekte übergeben, also derjenigen Objekte, die sich nicht weiter nach unten bewegen.

  - `public int moveLeft(boolean[][] staticObjects);` Das Objekt wird um eine Spalte nach links verschoben, wenn es möglich ist. Dazu bekommt es die Liste bereits gelandeter Objekte übergeben.
  - `public int moveRight(boolean[][] staticObjects);` Das Objekt wird um eine Spalte nach rechts verschoben, wenn es möglich ist. Dazu bekommt es die Liste bereits gelandeter Objekte übergeben.

- `public class SingleObject extends Weihnachtsobjekt`

  Diese Klasse stellt ein Objekt dar, welches genau 1 Zelle einnimmt.

- `public class MultiObject extends Weihnachtsobjekt;` Stellt ein Objekt dar, welches mehr als 1 Zelle einnimmt. Dafür hat es mindestens die folgenden zusätzlichen Variablen. Alle Teile übernehmen das Verhalten des `MultiObjects`, d. h. sie fallen z. B. alle nach unten oder sie fallen alle nicht.

  - `protected List<SingleObject> parts;` Das Objekt besteht aus mehreren Objekten, die jeweils genau 1 Zelle einnehmen. Die Liste verwaltet diese Objekte.

    Hinweis: Schauen Sie sich die Javadoc für die Klasse List an.
  - `protected int breite;` Die Breite des Objektes: Ein MultiObject besteht immer aus einem linken `SingleObject`, einem rechten SingleObject und dazwischen `2*breite` mittleren `SingleObjects`.

  Für die folgenden Klassen überlegen Sie selbst, ob und wenn ja, was Sie an den jeweiligen Oberklassen ändern müssen.

- `public class Ast extends MultiObject;` Ein Ast besteht aus einem linken, einem rechten und mehreren mittleren Aststücken. Äste fallen so lange nach unten, bis ein Teil (oder mehrere) von ihnen auf eine durch einen `Ast` oder `Baumstamm` belegte Zelle treffen.

- `public class Baumstamm extends MultiObject;` Stämme fallen ebenfalls so lange nach unten, bis ein Teil (oder mehrere) von ihnen auf eine durch einen `Ast` oder `Baumstamm` belegte Zelle treffen.

- `public class AstLinks extends SingleObject`
- `public class AstRechts extends SingleObject`
- `public class AstMitte extends SingleObject`
- `public class StammLinks extends SingleObject`
- `public class StammRechts extends SingleObject`
- `public class StammMitte extends SingleObject`

- `public class Weihnachtskugel extends SingleObject;`
  `public class Pinguin extends SingleObject;`
  `public class Schneeflocke extends SingleObject;`

  Diese Objekte werden **im Vordergrund** gezeichnet. Sie fallen so lange nach unten, bis entweder sich ein anderes solches Objekt in der Zelle unter ihnen befindet, das sich nicht weiter nach unten bewegt, oder in der selben Zelle ein Ast-Teil und in

keiner Zelle darunter ein Ast-Teil ist. Schneeflocken fallen immer in gerader Linie herunter und bleiben, falls möglich, unabhängig von Eingaben des Spielers immer in der gleichen Spalte. Falls nicht möglich, wechseln sie die Spalte entsprechend der Eingabe der Spielerin.

- **Hinweis**: Es ist Ihnen erlaubt, die Klassenhierarchie auch so zu erweitern, dass Sie z. B. eine Klasse zwischen `SingleObject` und `Pinguin` einfügen. Dies kann Ihre Implementierung eventuell vereinfachen, muss es aber nicht.

Nachdem Sie diese Klassen implementiert haben, vervollständigen Sie nun die Implementierung der Klasse `Weihnachtsbaum`. Das Spielfeld hat eine feste Größe: **30 x 33**. Implementieren Sie die folgenden Methoden und eventuell genutzte Hilfsmethoden der Klasse `Weihnachtsbaum`:

- `public static Weihnachtsobjekt createRandomObjekt()`
  Diese Methode erzeugt ein zufälliges Weihnachtsobjekt, wobei das Spiel *2 Phasen* hat. In der ersten Phase werden nur Baumstämme und Äste erzeugt, der Baum somit gebaut. In der zweiten Phase werden Schneeflocken, Weihnachtskugeln und Pinguine erzeugt. Die Phasen können Sie zum Beispiel über eine statische Variable der Klasse lösen. Zur Auswahl der herabfallenden Teile kann eine Methode der Klasse `WeihnachtsElfen` (s. u.) genutzt oder selbst eine entsprechende Methode geschrieben werden.
- Vervollständigen Sie die Implementierung von `public static void keyPressed (int key)`. Diese Methode wird bei der Eingabe einer der Pfeiltasten aufgerufen. Der Parameter `key` gibt an, welche Taste gedrückt wurde. Folgende Aktionen werden bei der jeweiligen Ausgabe ausgeführt:
  - *Pfeil-Unten* Alle noch fallenden Objekte bewegen sich, wenn möglich, um eine Zeile nach unten.
  - *Pfeil-Rechts/Links* Alle noch fallenden Objekte bewegen sich zuerst, wenn möglich, um eine Spalte nach links bzw. rechts und im Anschluss zusätzlich um eine Zeile nach unten, wenn möglich. Wenn Objekte links bzw. rechts über den Rand hinaus verschoben werden, wird das **gesamte** Objekt gelöscht.
  - *Pfeil-Oben* Ändert die Phase des Spiels, die kontrolliert, welche Objekte zufällig erzeugt werden.

Am Ende einer jeden solchen Aktion wird zudem eventuell ein neues `Weihnachtsobjekt` mit Hilfe der bereits implementierten Methode erzeugt.

Die zahlreichen `WeihnachtsElfen` der Informatik Fakultät möchten Ihnen ein möglichst angenehmes Weihnachtsfest bescheren und haben daher eine Menge nützliche Funktionen für Sie vorbereitet, die Ihnen beim Lösen der Aufgabe hoffentlich hilfreich sein werden.

- `static void sortWeihnachtsbjectsByYCoordinate(List<Weihnachtsobjekt> liste)`
  Sortiert die Liste nach den Y-Werten in absteigender Reihenfolge.
- `static void sortMultiObjectsByYCoordinate(List<MultiObject> liste)`
  Sortiert die Liste nach den Y-Werten in absteigender Reihenfolge.
- `static void removeMarkedForDeath(List<Weihnachtsobjekt> objekte)` Löscht alle Elemente aus der Liste, die `true` als Wert für `markedForDeath` haben.

- `static List<Baumstamm> getAllBaumstaemme(ArrayList<Weihnachtsobjekt> liste)`
  Gibt eine Liste mit nur den Baumstämmen zurück.
- `static List<Ast> getAllAeste(ArrayList<Weihnachtsobjekt> liste)`
  Gibt eine Liste mit nur den Ästen zurück.
- Zudem beinhaltet die Klasse die Konstanten, die die Weihnachtsobjekte als Wert in das `int[][] spielfeld` eintragen sollen, unterschieden nach Hintergrund- und Vordergrundobjekten.

  ```
  public static final int BACKGROUND_EMPTY = 0;
  public static final int BACKGROUND_TRUNK_MIDDLE = 1 << 8;
  public static final int BACKGROUND_TRUNK_LEFT   = 2 << 8;
  public static final int BACKGROUND_TRUNK_RIGHT  = 3 << 8;
  public static final int BACKGROUND_GREEN_MIDDLE = 4 << 8;
  public static final int BACKGROUND_GREEN_LEFT   = 5 << 8;
  public static final int BACKGROUND_GREEN_RIGHT  = 6 << 8;

  public static final int FOREGROUND_EMPTY = 0;
  public static final int FOREGROUND_SNOWFLAKE = 1;
  public static final int FOREGROUND_BAUBLE = 2;
  public static final int FOREGROUND_PENGUIN = 3;
  ```

  Um Hintergrund und Vordergrund zusammen zeichnen zu lassen, addiert man die jeweiligen Konstanten.
- `public static void newRandomObject()`
  Erzeugt beim Aufruf einen Vorschlag für das herabfallende Ast- oder Baumstamm-Objekt beim nächsten Spielschritt und hinterlegt den Vorschlag in der statischen Variablen `randomObjects` am Index `currentRandomObject`. Am ersten Index steht dabei der Vorschlag (`FALLING_NONE`, `FALLING_TRUNK` oder `FALLING_GREEN` für kein, ein Baumstamm- bzw. ein Ast-Objekt), am zweiten die vorgeschlagene Größe.

Bonus: Änderen Sie Ihre Implementierung so ab, so dass bei jedem Spielschritt Schnee an einer zufälligen Stelle erstellt wird. Machen Sie sich Gedanken darüber, ob dies zu Problemen führen kann, und beheben Sie diese, falls dies der Fall ist.

Lösungsvorschlag 9.5

Siehe Moodle.

Korrekturhinweise:

- Unwesentliche Änderungen der Spielregeln sind akzeptabel, sofern gut dokumentiert (Konsolenausgabe, Kommentar)
  Beispiel: Diagonales Herunterfallen statt erst zur Seite, dann nach unten

Bewertungsschema: Es gibt 6 Punkte für Funktionalität, 4 Punkte für den Rest (Einhalten von Vorgaben, Code-Qualität usw. - s. u.) und einen Bonuspunkt für Schnee.

- 2 Punkte: Objekte werden angezeigt und bewegen sich

- 2 Punkte: Spiel kann nach den Regeln (ggf. mit kleinen Abweichungen) gespielt und der Baum (Stämme, Äste) gebaut und geschmückt (Kugeln, Pinguine) werden.
- 2 Punkte: Reibungsloses und nachvollziehbares Spielgeschehen (ggf. mit Ausgabe auf der Konsole, insbesondere im Fall von kleinen Abweichungen)
- Abzüge Funktionalität (6 Punkte):
  - Fehler (insbesondere eintretende Exceptions)
  - Fehlende oder mangelhafte Funktionalität
- Abzüge aus den restlichen 4 Punkten gibt es für:
  - Spielregeln: größere oder grobe Abweichungen
    Beispiel: Spiel beendet sich, wenn Stammstück den Rand erreicht.
  - Vorlage: nicht oder nur teilweise genutzt oder unzulässig verändert
    Beispiele: Klassenhierarchie umgeändert (statt erweitert), vorgegebene Methodensignatur verändert (bzw. vorgegebene Methode nicht implementiert)
  - OO: Grobe Verstöße gegen Prinzipien der Objektorientierung
    Beispiel: selber Code in allen Unterklassen einer Klasse (statt dort)

- 1 Bonuspunkt für Schnee:
  + 0.5 für Flockenfall
  + 0.5 für sinnvolle Flockenfall-Regeln
    (Interaktion mit Kugeln/Pinguinen, Liegenbleiben)

Aufgabe 9.6 (H) In der Weihnachtsassemblybäckerei

[10 Punkte + 2 Bonuspunkte]

Auf dem vorletzten Blatt haben wir einen Interpreter für eine einfache Assembler-Sprache, unser sog. *Steck-Assembly*, implementiert. In dieser Aufgabe soll ein Code-Generator implementiert werden, der aus einem Syntaxbaum Assembly-Instruktionen generiert, die anschließend von der Steckmaschine ausgeführt werden können. Bevor Sie mit der Aufgabe fortfahren, sollten Sie sich die Assemblersprache und die Implementierung des Interpreters ins Gedächtnis rufen. Sie dürfen für Ihre Lösung entweder Ihre eigene Lösung zur Steckmaschine verwenden oder Ihre Lösung auf der Musterlösung aufbauen, welche ab Montag auf Moodle zur Verfügung steht.

Im ersten Schritt erweitern bzw. verändern wir die Steck-Sprache, damit die Code-Generierung für *Minijava mit Funktionen* möglich wird. In der folgenden Tabelle werden nur Instruktionen aufgeführt, die sich gegenüber der ursprünglichen Implementierung geändert haben oder neu hinzugekommen sind.

fsdf

Instruktion	Immediate	Beschreibung
`DIV`	keins	Berechnet die Integer-Division o_1 / o_2
`AND`	keins	Berechnet das bitweise UND, also o_1 & o_2
`OR`	keins	Berechnet das bitweise ODER, also o_1 | o_2
`NOT`	keins	Berechnet das bitweise NOT, also ~o_1
`JUMP`	i (16 Bit)	Springt zur Instruktion mit dem Index i, falls $o_1 = -1$ gilt
`EQ`	keins	Legt -1 auf den Stack, falls $o_1 = o_2$ gilt; sonst legt die Instruktion 0 auf den Stack.
`LT`	keins	Legt -1 auf den Stack, falls $o_1 < o_2$ gilt; sonst legt die Instruktion 0 auf den Stack.
`LE`	keins	Legt -1 auf den Stack, falls $o_1 \leq o_2$ gilt; sonst legt die Instruktion 0 auf den Stack.
`JE`	—	Entfällt, ersetzt durch EQ und JUMP
`JNE`	—	Entfällt, ersetzt durch NOT, EQ und JUMP
`JLT`	—	Entfällt, ersetzt durch LT und JUMP

Es sei hier wiederum o_1 das oberste Stack-Element, o_2 das zweit-oberste Stack-Element. Schreiben Sie außerdem eine Methode `public static String programToString(int[] program)` in der Klasse `Interpreter` (die Implementierung der Steckmaschine), die ein Programm in einen String umwandelt. Der Rückgabewert der Methode könnte z.B. wie folgt aussehen:

```
0: LDI 33
1: CALL 0
2: HALT
```

Die Methode ist im Folgenden sehr nützlich, Ihre Implementierung zu debuggen. Wir werden nun eine Hierarchie von Klassen implementieren, die je einen Knoten im Syntaxbaum repräsentieren. Die Klassen sind stark an die Grammatik von MiniJava angelehnt, erweitern diese allerdings um *Funktionen* und *Funktionsaufrufe*. Die gesamte Hierarchie soll das *Visitor-Pattern* umsetzen, welches bereits aus der Präsenzaufgabe bekannt ist. Sollten Sie die Präsenzaufgabe noch nicht bearbeitet haben, sollten Sie dies zuerst tun. Im Folgenden sind alle nötigen Klassen mit je einer kurzen Beschreibung aufgelistet, wobei Unterklassen jeweils in Unterpunkten zu finden sind. Die Klassen enthalten für jede *getter*-Methode ein privates Attribut, welches im Konstruktor initialisiert wird. Die Konstruktoren sollen die Attribute entsprechend der gezeigten Reihenfolge erhalten. Die Beschreibung lässt ferner alle Methoden, die für das Visitor-Pattern implementiert werden müssen, aus.

- Klasse `Program`: Repräsentiert ein Programm, welches aus Funktionen besteht
  - Methode `public Function[] getFunctions()`: Liefert die Funktionen des Programms zurück
- Klasse `Function`: Repräsentiert eine Funktion des Programms
  - Methode `public String getName()`: Liefert den Namen der Funktion
  - Methode `public String[] getParameters()`: Liefert die Parameter der Funktion

  - Methode `public Declaration[] getDeclarations()`: Liefert die Deklarationen am Anfang der Funktion
  - Methode `public Statement[] getStatements()`: Liefert die Statements innerhalb der Funktion

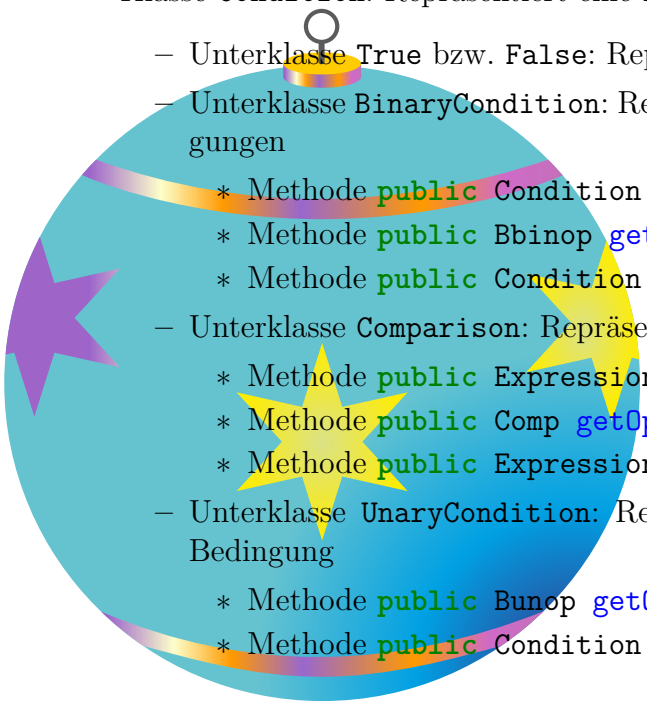
- Klasse `Declaration`: Repräsentiert eine Liste von Variabledeklarationen von `int`-Variablen
  - Methode `public String[] getNames()`: Liefert die Namen der deklarierten Variablen

- Klasse `Statement`: Repräsentiert ein Statement des Programms
  - Unterklasse `Assignment`: Repräsentiert eine Zuweisung der Form `variable = expression;`
    * Methode `public String getName()`: Liefert den Namen der zugewiesenen Variablen
    * Methode `public Expression getExpression()`: Liefert den zugewiesenen Ausdruck
  - Unterklasse `Composite`: Repräsentiert eine Menge von folgenden Statements
    * Methode `public Statement[] getStatements()`: Liefert die enthaltenen Statements zurück
  - Unterklasse `IfThen` bzw. `IfThenElse`: Repräsentiert ein If-Statement
    * Methode `public Condition getCond()`: Liefert die Bedingung
    * Methode `public Statement getThenBranch()`: Liefert das Statement im Then-Zweig zurück
    * Methode `public Statement getElseBranch()`: Liefert das Statement im Else-Zweig zurück (nur in der Klasse `IfThenElse`)
  - Unterklasse `While`: Repräsentiert ein While-Statement
    * Methode `public Condition getCond()`: Liefert die Bedingung
    * Methode `public Statement getBody()`: Liefert den Körper der Schleife zurück
  - Unterklasse `Read`: Repräsentiert ein MiniJava-Read-Statement der Form `variable = read();`
    * Methode `public String getName()`: Liefert den Namen der zugewiesenen Variablen
  - Unterklasse `Write`: Repräsentiert ein MiniJava-Write-Statement der Form `write(expression);`
    * Methode `public Expression getExpression()`: Liefert den Ausdruck, dessen Wert ausgegeben werden soll
  - Unterklasse `Return`: Repräsentiert ein `return`-Statement
    * Methode `public Expression getExpression()`: Liefert den Ausdruck, der den Rückgabewert berechnet

- Klasse `Expression`: Repräsentiert einen Ausdruck
  - Unterklasse `Variable`: Repräsentiert eine Variable

        - Methode `public String getName()`: Liefert den Namen der Variablen
    - Unterklasse `Number`: Repräsentiert eine Konstante
        - Methode `public int getValue()`: Liefert den Wert der Konstanten
    - Unterklasse `Binary`: Repräsentiert einen binären Ausdruck
        - Methode `public Expression getLhs()`: Liefert den linken Operanden
        - Methode `public Binop getOperator()`: Liefert den Operator
        - Methode `public Expression getRhs()`: Liefert den rechten Operanden
    - Unterklasse `Unary`: Repräsentiert einen unären Ausdruck
        - Methode `public Unop getOperator()`: Liefert den Operator
        - Methode `public Expression getOperand()`: Liefert den Operanden
    - Unterklasse `Call`: Repräsentiert einen Methodenaufruf
        - Methode `public String getFunctionName()`: Liefert den Namen der aufgerufenen Funktion
        - Methode `public Expression[] getArguments()`: Liefert die Argumente der aufgerufenen Funktion

- Klasse `Condition`: Repräsentiert eine Bedingung
    - Unterklasse `True` bzw. `False`: Repräsentiert `true` bzw. `false`
    - Unterklasse `BinaryCondition`: Repräsentiert eine binäre Operation auf Bedingungen
        - Methode `public Condition getLhs()`: Liefert den linken Operanden
        - Methode `public Bbinop getOperator()`: Liefert den Operator
        - Methode `public Condition getRhs()`: Liefert den rechten Operanden
    - Unterklasse `Comparison`: Repräsentiert einen Vergleich
        - Methode `public Expression getLhs()`: Liefert den linken Operanden
        - Methode `public Comp getOperator()`: Liefert den Operator
        - Methode `public Expression getRhs()`: Liefert den rechten Operanden
    - Unterklasse `UnaryCondition`: Repräsentiert eine unäre Operation auf einer Bedingung
        - Methode `public Bunop getOperator()`: Liefert den Operator
        - Methode `public Condition getOperand()`: Liefert den Operanden



Die Aufzählungstypen `Binop`, `Unop`, `Comp`, `Bbinop` und `Bunop` werden mit der Aufgabe verteilt. Implementieren Sie außerdem die Klasse `Visitor`, die einen allgemeinen Visitor entsprechend dem Visitor-Pattern darstellt. Leiten Sie von diesem eine Klasse `CodeGenerationVisitor` ab, die eine Methode `public int[] getProgram()` anbietet, welche das generierte Programm zurückliefert.

Die Code-Generierung generiert Code für alle Funktionen eines Programms. Jedes valide Programm muss über eine Funktion verfügen, welche den Namen `main` hat. Ihr Assembler-Programm muss diese Funktion als erstes aufrufen. In dem Ausschnitt von Java, den wir in dieser Aufgabe behandeln, gibt *jede* Funktion einen Integer zurück. Ein Programm, welches eine Funktion ohne `return`-Statement verlässt, ist invalide; sie müssen diesen Fall nicht beachten. Der Rückgabewert der `main()`-Funktion entspricht dem Rückgabewert der `execute()`-Funktion der Steckmaschine. Gehen Sie bei Ihrer Implementierung wie folgt vor:

1. Beginnen Sie mit Ausdrücken, erlauben Sie die Generierung von Code für Ausdrücke. Testen Sie Ihre Implementierung anhand von Beispielen.
2. Erweitern Sie Ihre Implementierung um die Unterstützung von Variablen.
3. Erweitern Sie Ihre Implementierung um die Unterstützung von Kontrollfluss, also von Bedingungen und Schleifen. Fügen Sie nun noch Ein- und Ausgabe hinzu, haben Sie MiniJava vollständig abgedeckt.
4. Implementieren Sie Unterstützung von Funktionen und Funktionsaufrufen. Erlauben Sie zunächst nur Aufrufe von Funktionen, für die **vorher** bereits Assembly generiert wurde, deren Adresse Sie also an der *Call Site* kennen.
5. Vollenden Sie schließlich Ihre Implementierung, indem Sie auch Aufrufe von Funktionen erlauben, deren Assembly erst nach dem Aufruf generiert wird.

Behandeln Sie Fehler sinnvoll. Insbesondere sollten Sie die folgenden Fehler abfangen:

- Eine unbekannte Funktion wird aufgerufen (gilt insbesondere auch für den initialen Aufruf zu `main`).
- Eine nicht deklarierte Variable wird verwendet.

Trifft eine der oben genannten *kontextbasierten Aussagen* zu, ist das Programm fehlerhaft. Derartige Bedingungen für valide Programme lassen sich schwer oder gar nicht im Syntaxbaum ausdrücken und fallen daher erst bei der Code-Generierung auf. Andere Kontextbedingungen sind schwieriger zu erkennen und müssen daher nicht behandelt werden:

- Jeder Pfad durch eine Funktion endet in einem `return`-Statement (vgl. oben).
- Variablen wird ein Wert zugewiesen, bevor diese gelesen werden.

Nutzen Sie das Java-Statement `throw new RuntimeException("Fehlertext");`, wenn während der Code-Generierung ein Fehler auftritt. Ersetzen hier *Fehlertext* durch eine verständliche Beschreibung des aufgetretenen Fehlers.

Hinweis: Mit der Angabe werden JUnit-Tests verteilt. Aus diesen Tests können Sie auch ableiten, wie der Visitor verwendet werden soll. Bauen Sie sich selbst Tests auf Basis der vorgegebenen Testes.

Zusatzaufgabe als Bonus: 32-Bit-Konstanten

In MiniJava haben Konstanten eine Größe von 32 Bit. Ohne weiteres erlaubt unsere Code-Generierung allerdings nur 16 Bit. Wir wollen abschließend die Steckmaschine und Code-Generierung erweitern, sodass 32-Bit-Konstanten ermöglicht werden. Fügen Sie dazu zunächst die SHL-Instruktion zur Steckmaschine hinzu:

Instruktion	Immediate	Beschreibung
`SHL`	n (16 Bit)	Verschiebt o_1 um n Bits nach links

Erweitern Sie dann Ihre Code-Generierung derart, dass 32-Bit-Konstanten korrekt in Assembly-Code übersetzt werden.

Lösungsvorschlag 9.6
Pukteverteilung:

1. `programToString()` und Erweiterung des Interpreters: 1.5 Punkte
2. Klassenhierarchie: 1.5 Punkte
3. CodeGenerationVisitor: 7 Punkte
    - davon Ausdrücke ohne Variablen: 2 Punkte
    - davon Variablen: 1 Punkt
    - Kontrollfluss: 1 Punkt
    - Funktionsaufrufe: 2 Punkte
    - Funktionsaufrufe zu später definierten Funktionen: 1 Punkt
4. Bonusaufgabe: 2 Punkte

Korrekturhinweise:

- Es ist nicht verboten, zunächst die Textform des Assemblers zu generieren.