Übungsblatt 1

1. **Softwarelebenszyklus**

   Zählen Sie die *Aktivitäten* der Software Entwicklung auf, beschreiben Sie kurz deren *Inhalt* und nennen Sie den jeweils dabei entstehenden *Output*.

   Lösung:

   - Requirements Engineering: Die Anforderungen des Kunden werden ermittelt und im Lastenheft festgehalten.
   - Analysis: Die Anforderungen werden modelliert und spezifiziert (ggf. formal). Die Ergebnisse werden im Pflichtenheft hinterlegt.
   - Design: Ein Lösungsansatz basierend auf den Ergebnissen der Analyse wird erstellt. Hierzu bietet es sich an, UML und ggf. ER-Diagramme zu erstellen.
   - Implementierung Die Umsetzung des Designs in ausführbaren Quellcode erfolgt.
   - Validation: Es ist zu prüfen, ob die Implementierung die Zielsetzung aus den Requirements erfüllt. Hierzu werden z.B. Testfälle erstellt, anhand derer ein Testreport zur Verfügung gestellt werden kann.
   - Maintenance: Es ist die kontinuierliche Korrektur und Wartung der Implementierung zu gewährleisten, ggf. unter Berücksichtigung geänderter oder neuer Anforderungen.

2. **Programmieraufgabe: Universitäts-Verwaltungs-Programm**

   Programmieren Sie die nötigen Java Klassen für ein Universitäts-Verwaltungs-Programm. Achten Sie dabei insbesondere auf eine einfache Erweiterbarkeit. Die Klassen sollen mindestens *Getter* und *Setter* für jede Membervariable und einen *Konstruktur* enthalten.

   - Es sind vorerst nur die Komponenten *Professoren* und *Studierende* zu berücksichtigen.
   - Beide Personengruppen haben je einen Vor- und Nachnamen, Professoren außerdem eine Personennummer, Studierende eine Matrikelnummer.

   Lösung:

```java
public class Person {
    private String firstname;
    private String lastname;

    public Person(String firstname, String lastname) {
        this.firstname = firstname;
        this.lastname = lastname;
    }

    public String getFirstname() {
        return this.firstname;
    }

    public String getLastname() {
        return this.lastname;
    }

    [...]
}
```

```java
public class Student extends Person {
    private int matriculation_number;

    public Student(
            String firstname,
            String lastname,
            int matriculation_number) {
        super(firstname, lastname);
        this.matriculation_number = matriculation_number;
            }

    public String getMatriculationNumber() {
        return this.matriculation_number;
    }
}
```

```java
public class Professor extends Person {
    private int staff_number;

    public Student(
            String firstname,
            String lastname,
            int staff_number) {
        super(firstname, lastname);
        this.staff_number = staff_number;
            }

    public String getStaffNumber() {
        return this.staff_number;
    }
}
```

3. **Anforderungsbeschreibung mit Volere Snow Cards**

   Gegeben ist folgendes Szenario:

   Die neue Bildbearbeitungssoftware intelliPhoto ist ein interaktives Tool zum Anzeigen und Bearbeiten von Bildern. Jedes Bild wird durch ein zweidimensionales Array von Bytes repräsentiert, wobei jeder Byte-Wert für einen Farbwert des Bildpunktes steht. Der Benutzer soll in der Lage sein die Bilddimensionen abzufragen. Es sollen zwei verschiedene Arten von Bildern repräsentiert werden können: *RasterImage* und *ShapedImage*, wobei letzteres eine Spezialform vom *RasterImage* ist. Ein *ShapedImage* besitzt eine nicht-rechteckige Form (Polygon), wobei die Bytes im Array angeben, ob die jeweiligen Punkte transparent oder opak dargestellt werden sollen. Darüber hinaus soll die Software einfache Manipulationen von Bildern erlauben. So soll das Drehen, als auch das Vergrößern und Verkleinern von Bildern, das Setzen neuer Farbwerte im Bild und das Zusammenfügen zweier Bilder zu einem neuen Bild innerhalb von 0,2 Sekunden möglich sein.

   Führen Sie eine Anforderungsbeschreibung nach Volere für jeweils eine funktionale und eine nicht funktionale Anforderung durch. Eine ausführliche Beschreibung zu *Volere Snow Card* können Sie hier finden:

   `http://www.cse.chalmers.se/~feldt/courses/reqeng/Volere_Template_version1_5.doc`

   Lösung:

   - Funktionale Anforderungen:
     - Anzeigen und Bearbeiten von Bildern
     - Benutzer kann Bilddimensionen Abfragen
     - Software soll das Manipulieren von Bildern erlauben

        - drehen
        - vergrößern / verkleinern
        - neue Farbwerte im Bild setzen
        - Zusammenfügen zweier Bilder
- Nicht-funktionale Anforderungen:
    - Manipulationen sollen innerhalb von 0,2 Sekunden erfolgen

Übungsblatt 2

1. **Responseability Driven Design**

   Gegeben ist folgende Requirement Spezifikation:

   Die neue Bildbearbeitungssoftware intelliPhoto ist ein interaktives Tool zum Anzeigen und Bearbeiten von Bildern. Jedes Bild wird durch ein zweidimensionales Array von Bytes repräsentiert, wobei jeder Byte-Wert für einen Farbwert des Bildpunktes steht. Der Benutzer soll in der Lage sein die Bilddimensionen abzufragen. Es sollen zwei verschiedene Arten von Bildern repräsentiert werden können: *RasterImage* und *ShapedImage*, wobei letzteres eine Spezialform vom *RasterImage* ist. Ein *ShapedImage* besitzt eine nicht-rechteckige Form (Polygon), wobei die Bytes im Array angeben, ob die jeweiligen Punkte transparent oder opak dargestellt werden sollen. Darüber hinaus soll die Software einfache Manipulationen von Bildern erlauben. So soll das Drehen, als auch das Vergrößern und Verkleinern von Bildern, das Setzen neuer Farbwerte im Bild und das Zusammenfügen zweier Bilder zu einem neuen Bild innerhalb von 0,2 Sekunden möglich sein.

   Führen Sie eine detaillierte Analyse durch und finden Sie mit ihrer Hilfe möglichst alle **Klassen**, **Verantwortlichkeiten**, **Kollaborationen** und **Beziehungen** (bzw. **Vererbungen**). Begründen Sie Ihre Entscheidung. Es müssen nur Klassen aus der Spezifikation betrachtet werden (zum Beispiel keine GUI- oder OS-Elemente).

   Lösung:

   Klassen

   - Bild/Image
   - RasterImage
   - ShapeImage
   - Form/Shape
     - Rechteck
     - Polygon
   - Punkte/Points

   Verantwortlichkeiten

   - skalieren:
     - vergrößern
     - verkleinern
   - rotieren
   - einfärben
   - zusammenfügen

   Beziehungen

   - RasterImage is kind of Image
   - ShapeImage is kind of RasterImage
   - Dimension is part of Image
   - Rectangle is kind of Shape
   - Polygon is kind of Shape
   - Manipulation has knowledge of Image

   Kollaborationen

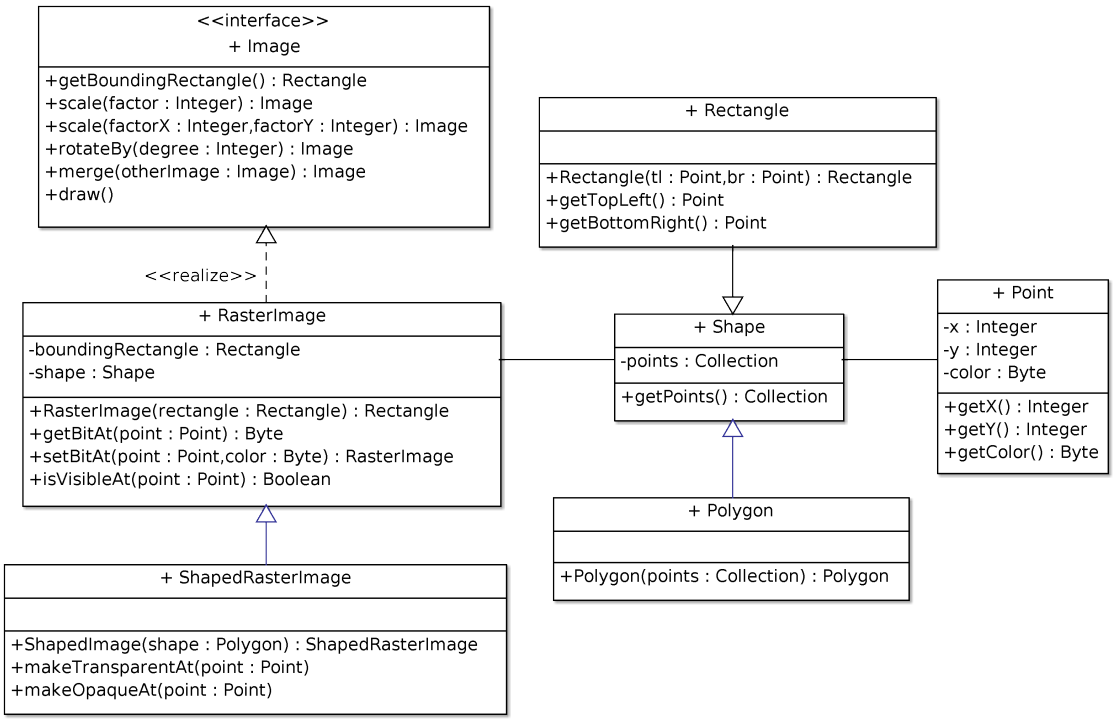
- Image kann seine Responsibilities selbst erfüllen (sehr gute Kapselung der Daten möglich)
- weitere Responsibilities sind nicht zu berücksichtigen

2. **UML-Klassendiagramm**

   Erstellen Sie auf Grundlage der Spezifikation aus Aufgabe 1 einen geeigneten Entwurf der Klassenhierarchie als *UML-Klassendiagramm*. Geben Sie dabei die genauen Interfaces eventueller Klassen an sowie die Attribute, welche die jeweiligen Klassen verwalten müssen. Begründen Sie Ihre Design-Entscheidung in wenigen Worten.

   Lösung:

   - 7 Klassen: Image, RasterImage, ShapedRasterImage, Shape, Rectangle, Polygon, Point
   - 4 Methoden: scale, rotate, merge, setBit
   - richtige Vererbungen zwischen Klassen
   - zusätzliche Methoden eingefügt (z.B. Transparenz oder Opaqueness an Point-Position hinzufügen, draw(), getTopLeft(), getBottomRight() o.ä.)



3. **Kapselung**

   Was versteht man unter dem Begriff *Kapselung* im Kontext der (objektorientierten) Programmierung? Welche Vorteile bringt dieses Konzept mit sich?

Übungsblatt 3

1. **Modelling Behavoir: Use-Case Diagramm**

   Da Sie damit beauftragt wurden die neue Bildbearbeitungssoftware IntelliPhoto zu implementieren, führten Sie eine Umfeldanalyse durch. In dieser haben Sie wertvolle Informationen über verschiedene Nutzergruppen sammeln können.

   So erfuhren Sie, dass Casual User und Einsteiger die Software hauptsächlich für kurze Aufgaben wie das Zusammenschneiden von Bildern, das Ändern der Bildauflösungen und dem Drehen von Bildern benutzen wollen. Außerdem möchten die Casual User die Software dazu benutzen um bestimmte Regionen in einem Bild zu retuschieren.

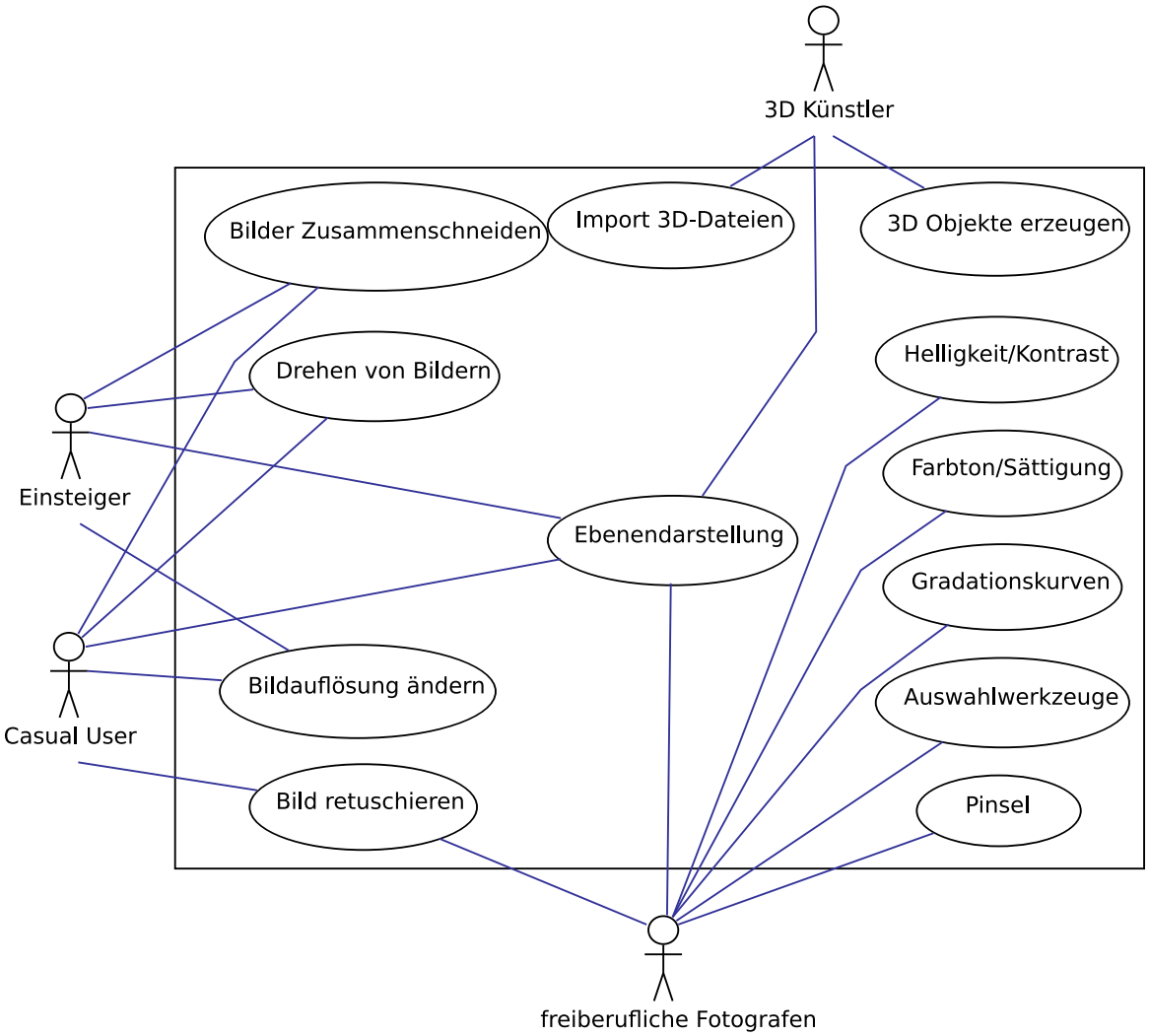
   Eine weitere Nutzergruppe, die freiberuflichen Fotografen, hingegen möchten neben der Bildretusche auch eine Reihe an Korrekturwerkzeugen, wie der *Helligkeit/Kontrast*, *Farbton/Sättigung* und den *Gradationskurven*, als auch Auswahlwerkzeuge und verschiedene Pinsel haben.

   Die letzte Gruppe von potentiellen Benutzern, die 3D Künstler, wünschen sich eine Schnittstelle für den Import von gängigen 3D-Dateien. Auch soll es für sie möglich sein, einfache geometrische 3D-Objekte direkt im Bild zu erzeugen.

   Jede Nutzergruppe gab an, dass sie sich eine Ebenendarstellung in der Software vorstellen können und benutzen würden.

   Fassen Sie die beschriebenen Ergebnisse in einem *UML-Use-Case-Diagramm* zusammen.

   Lösung:



2. **UML Structure: UML-Klassendiagramm**

   Modellieren Sie ein Unternehmen als UML-Klassendiagramm, welches weltweit beliebig viele Standorte besitzt.

   Dabei setzt sich ein Standort aus mindestens einem Gebäude inklusive Adresse zusammen.

   Ein Gebäude besitzt mehrere Büros und exakt eine Mensa. Die Büros haben Nummern sowie ein Namensschild an der Tür.

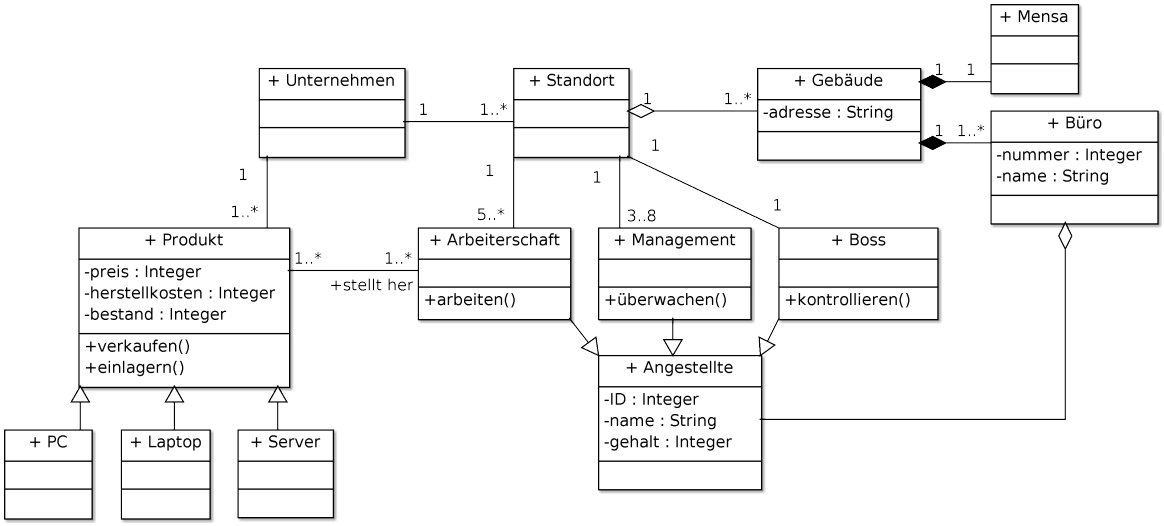
   In den Büros sitzen Angestellte, welche entweder der Chef, das Management oder der Arbeiterschaft zugeordnet sind.

   Zu beachten ist, dass einem Standort ein Chef und 3–8 Personen aus dem Management zugeordnet sind sowie mindestens 5 Arbeiter haben. Gekennzeichnet sind die Angestellten durch eine ID.

   Die Berufsgruppen haben zudem eigene Aufgabenfelder: Der Chef kontrolliert das Management, welches wiederum die Arbeiter überwacht, welche wiederum die Arbeit verrichten.

   Das Unternehmen stellt verschiedene Produkte (PCs, Laptops, Server) her.

   Lösung:



3. **Adapter Pattern: Deque**

   Implementieren Sie die Datenstruktur Deque in Java mit Hilfe des Adapter Patterns.

   Die Operationen der Deque sind:

   - `push` und `pop`: Einfügen und Entnehmen eines Elementes am hinteren Ende der Deque.
   - `put` und `get`: Einfügen und Entnehmen am vorderen Ende der Deque.
   - `first` und `last`: Lesen des ersten oder letzten Elements, ohne es zu entfernen.

   Achten Sie bei Ihrer Implementierung auf Java Generics und bauen Sie Ihre Lösung auf eine bestehende “Collection”, wie z.B. `java.util.ArrayList`, auf. Vergewissern Sie sich, dass Ihre neue Klasse *keine* weiteren Methoden als die oben genannten ausführen kann (z.B. `add()` oder `clear()`).

   Lösung:

   Interface:

```java
interface Deque<T> {
    public void push(T element);
    public T pop();

    public void put(T element);
    public T get();

    public T first();
```

```
    public T last();

    public void print();
}
```

Adapter:

```
class ArrayListToDequeAdapter<T> implements Deque<T> {
    private List<T> list = new ArrayList<T>();

    @Override
    public void push(T element) {
        list.add(element);
    }

    @Override
    public T pop() {
        T element = last();
        list.remove(list.size() - 1);

        return element;
    }

    [...]
```

4. **Visitor Pattern: Termausgabe**

   Wir betrachten Terme über die Rechenarten *op* $\in \{+, \cdot\}$, die folgendermaßen rekursiv definiert sind:

   - jedes Literal ist ein Term, z.B. 4
   - ist t ein Term, so ist (t) ein Term
   - sind t_1, t_2 Terme so ist t_1 *op* t_2 ebenso ein Term

   Beispiele für gültige Terme: $4 + 8$, $4 \cdot 8$ oder $4 + (4 \cdot 8)$.

   a) Implementieren Sie die entsprechenden Klassen `Expression`, `Literal`, `Brackets`, `BinaryExpression`, `Addition` und `Multiplication` im Sinne des Visitor Patterns.

   b) Implementieren Sie danach die Visitor Klassen `EvalVisitor` und `PrettyPrintVisitor`.

      `EvalVisitor`: Evaluiert bzw. berechnet den gegebenen Term und hält das Ergebnis

      `PrettyPrintVisitor`: Gibt einen Term in leserlicher Form aus.

      Prüfen Sie Ihre Implementierung durch geeignete Tests!

   Lösung:

   Abstract Visitable:

```
import visitors.Visitor;

public abstract class Expression {

    public abstract void accept(Visitor v);
}
```

   Concrete Visitable:

```
public class Literal extends Expression {

    private int value;

    public Literal(int v) {
        value = v;
    }
```

```

    public int getValue() {
        return value;
    }

    @Override
    public void accept(Visitor v) {
        v.visit(this);
    }
}
```

Visitor:

```
public interface Visitor {
    void visit(Expression e);
    void visit(Literal l);
    void visit(Brackets b);
    void visit(Addition a);
    void visit(Subtraction a);
    void visit(Multiplication a);
    void visit(Division a);
}
```

Concrete Visitor:

```
public class EvalVisitor implements Visitor {
    private int result = 0;

    public int getResult() {
        return result;
    }

    @Override
    public void visit(Expression e) {
        e.accept(this);
    }

    @Override
    public void visit(Literal l) {
        result = l.getValue();
    }

    [...]
```

Übungsblatt 4

1. **Kohäsion und Koppelung**

   In der Vorlesung wurden die Begriffe *Kohäsion* und *Kopplung* eingeführt.

   a) Erklären Sie mit eigenen Worten, was sich, im Kontext der objektorientierten Programmierung, hinter diesen Begriffen verbirgt.

   b) Weshalb ist es vom Vorteil, wenn ein System hohe Kohäsion und geringe Kopplung aufweist?

   Lösung:

   Kohäsion:

   - Beschreibt die Beziehungen zwischen Elementen *innerhalb einer Komponente*.
   - Idealzustand: Jede Komponente (Methode, Klasse) für genau eine Aufgabe verantwortlich, die sie eigenständig lösen kann.
   - Starke Kohäsion: Alle Elemente sind nötig für die Funktionstüchtigkeit der anderen internen Elemente, das heißt keine isolierten Elemente.
   - Schwache Kohäsion: Komponente erfüllt viele verschiedene Aufgabe oder die Elemente sind nur zusammengefasst, weil sie ähnliche Funktionalitäten anbieten.
     $\Rightarrow$ Ausgliedern in neue Komponente bietet sich an.

   Kopplung:

   - Beschreibt die Beziehungen zwischen verschiedenen Komponenten.
   - Idealzustand: Jede Komponente nur lose mit anderen verbunden.
   - Lose Kopplung: Komponenten besitzen nur wenige Abhängigkeiten untereinander.
   - Enge Kopplung: Komponenten besitzen viele Abhängigkeiten untereinander.
     $\Rightarrow$ Eine Änderung hat möglicherweise Auswirkungen auf viele andere Komponenten.

   Hohe Kohäsion und geringe Kopplung:

   - Effekte von Änderungen auf engen Kreis von Komponenten beschränkt.
     $\Rightarrow$ Fördert die Wartbarkeit und Anpassbarkeit.
   - Perfekter Zustand unerreichbar.

   Beispiel Visitor Pattern:

   - Kohäsion:
     - hoch, da jeder Visitor genau eine Funktionalität anbietet
     - hoch, da die Objektstruktur genau eine Funktionalität anbietet
   - Kopplung:
     - hoch, da Visitor Zugriff auf interne Datenstruktur benötigt
     - hoch, da Visitor durch die Objektstruktur iterieren können muss

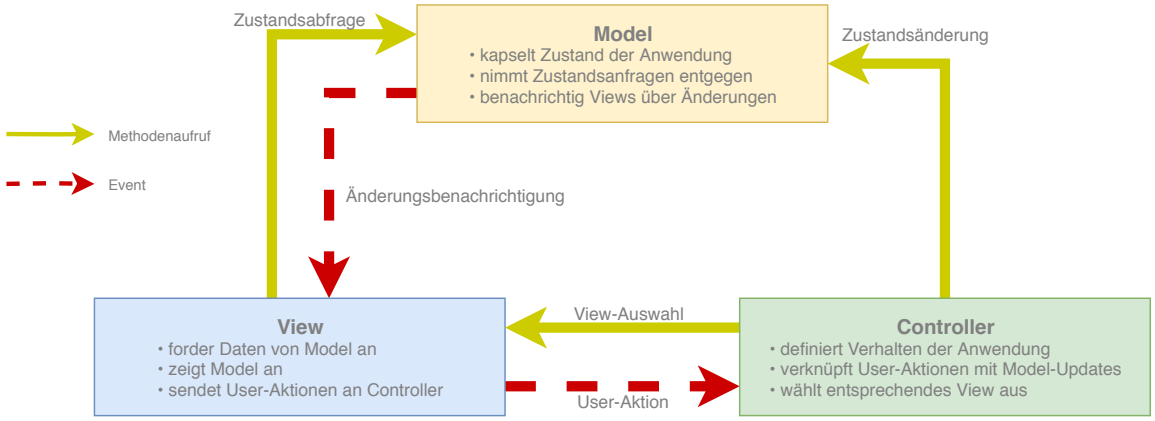
2. **Model-View-Controller (MVC)**

   a) Erstellen Sie ein Diagramm, welches die Beziehungen der einzelnen Komponenten des Model-View-Controller visualisiert.

   b) Welche Aufgaben haben die Komponenten?

   c) Welche Vor- und Nachteile hat die MVC-Architektur?

   Lösung:



Vorteile:

- höhere Kohäsion und geringere Kopplung als mit naivem, monolithischem Ansatz
- einfaches *Unit-Testing* möglich durch *Separation of Concerns*
- mehrere Views pro Model möglich
- Anpassung (zum Beispiel neuer Button) an View möglich ohne, dass das Model geändert werden muss
- einfach verschiedene Views für zum Beispiel verschiedene Anzeigegeräte (Laptop, Smartphone, etc.)

Nachteile:

- Must have strict rules on methods.

There is not much in the disadvantages part of the architecture. And the disadvantages are not so huge and are very easy to ignore in comparison with all the benefits we get.

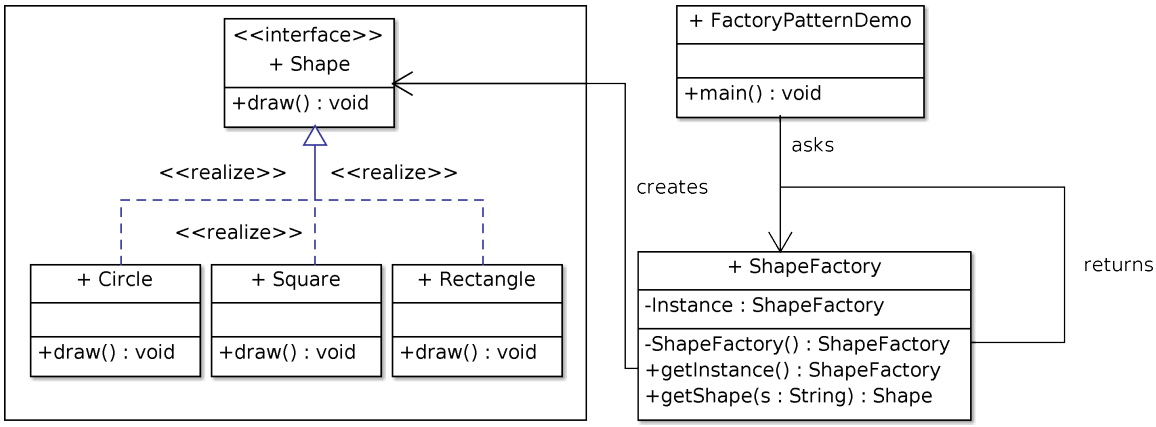
3. **Begriffsdefinitionen**

   In der Vorlesung wurden die Begriffe *Scattering*, *Tangling* und *Tyrannei der dominanten Dekomposition* eingeführt. Erläutern Sie diese Begriffe mit eigenen Worten an einem geeigneten von Ihnen selbst gewähltem Beispiel.

4. **Factory & Singleton Pattern**

   a) Erklären Sie mit eigenen Worten, wofür man das *Factory Pattern* und das *Singleton Pattern* verwendet.

   b) Gegeben ist folgendes UML-Klassendiagramm:



   Implementieren Sie auf dessen Grundlage das *Factory Pattern* mit allen angezeigten Klassen. Die `draw()`-Methoden sollen vereinfacht ausgeben, um welches Objekt es sich handelt. Die Factory-Klasse soll zudem das *Singleton Pattern* implementieren. Testen Sie Ihre Lösung.

Übungsblatt 5

1. **Begriffsdefinitionen**

   a) Erklären Sie den Unterschied zwischen *Software Validierung* und *Software Verifikation*!

   Lösung:

   Software Validierung: *Wird das richtige Produkt entwickelt*

   - Eignung der Software für den Einsatzzweck
   - Vorher aufgestelltes Anforderungsprofil

   Software Verifikation: *Ist das System richtig gebaut?*

   - Entspricht die Software der Spezifikation?
   - Fehler in der Spezifikation werden *nicht* berücksichtigt

   b) Worin besteht der Unterschied zwischen *Software-Failure* und *Software-Fault*?

   Lösung:

   Software-Failure:

   - Nicht erwartetes Ergebnis der Software
   - Wird durch Tests gefunden

   Software-Fault:

   - Grund für einen Software-Failure
   - Nicht jeder Software-Fault führt zu einem Software-Failure

   Beispiel[1]:

   ```
   1 int double (int param) {
   2   int result;
   3   result = param * param;
   4   return result;
   5 }
   ```

   - Wird `double(3)` aufgerufen, ist das Ergebnis 9, aber wir erwarten 6.
   - Das Ergebnis 9 ist ein Software-Failure
   - Der Grund für den Failure ist der Software-Fault in Zeile 3 („`* param`" anstatt „`* 2`")

   c) Was verbirgt sich hinter dem Begriff *Regression Testing*?

   Lösung:

   - Nach jeder Änderung werden *alle* Tests ausgeführt
   - Stellt sicher, dass alles, was vor der Änderung funktioniert hat, auch nach der Änderung noch funktioniert
   - Voraussetzung: Tests müssen deterministisch und wiederholbar sein

   d) Nennen Sie die entscheidenden Vor- und Nachteile von *Testing* bzw. *Model Checking*!

   Lösung:

   Model Checking

   - System wird als Modell in einer formalen Sprache beschrieben
   - Vollständige Überprüfung gegen das Modell
   - Beweise möglich

[1] `https://stackoverflow.com/a/47963772`

- Modelle sind in der Erstellung sehr komplex und zeitaufwändig

Testing

- Einfacher umzusetzen als Model Checking
- Fehler können gefunden, aber korrekte Funktionsweise nicht bewiesen werden

e) Birgt es Gefahren, wenn eine Test-Suite ausschließlich Unit-Tests enthält? Wenn ja, warum?

Lösung:

Es fehlen:

- Integration Tests
  - Module werden kombiniert und als Gruppe getestet
  - Erfüllen Module im Zusammenspiel funktionale und nicht-funktionale Anforderungen?
- System Tests
  - Black-Box-Test des kompletten Systems
  - Orientierung oft an Use Cases
  - GUI, Usability, Performance, Barrierefreiheit, Stresstests, ...
- Acceptance Tests
  - Funktionstest, die der Kunde ausführt, um Qualität zu bewerten
  - Echte statt simulierte Daten

2. **Unit-Testing**

Gegeben sind folgende Funktionen. Definieren Sie Testcases um eine möglichst gute Testabdeckung zu erreichen.

- Eine Memberfunktion der `BigInteger`-Klasse:

```
/**
 * @param val value to be multiplied by this BigInteger.
 * @return a BigInteger whose value is (this * val).
 */
public BigInteger multiply(BigInteger val)
```

Lösung:

Sinnvolle Werte für `this` und `val`:

- 0
- 1
- -1
- sehr kleine positive Zahl
- sehr große positive Zahl (größer als `Long.MAX_VALUE`)
- sehr kleine negative Zahl
- sehr große negative Zahl

Damit kommt man auf $7 \cdot 7 = 49$ Testcases.

- Die Funktion `max()` aus `Math`:

```
/**
 * @param a  an argument
 * @param b  another argument
 * @return the larger of a and b.
 */
public static int max(int a, int b)
```

Lösung:

- Verhältnis zwischen a und b:
  - $a < b$
  - $a = b$
  - $a > b$
- Werte für a und b:
  - 0
  - < 0
  - > 0
  - maximum Integer
  - minimum Integer

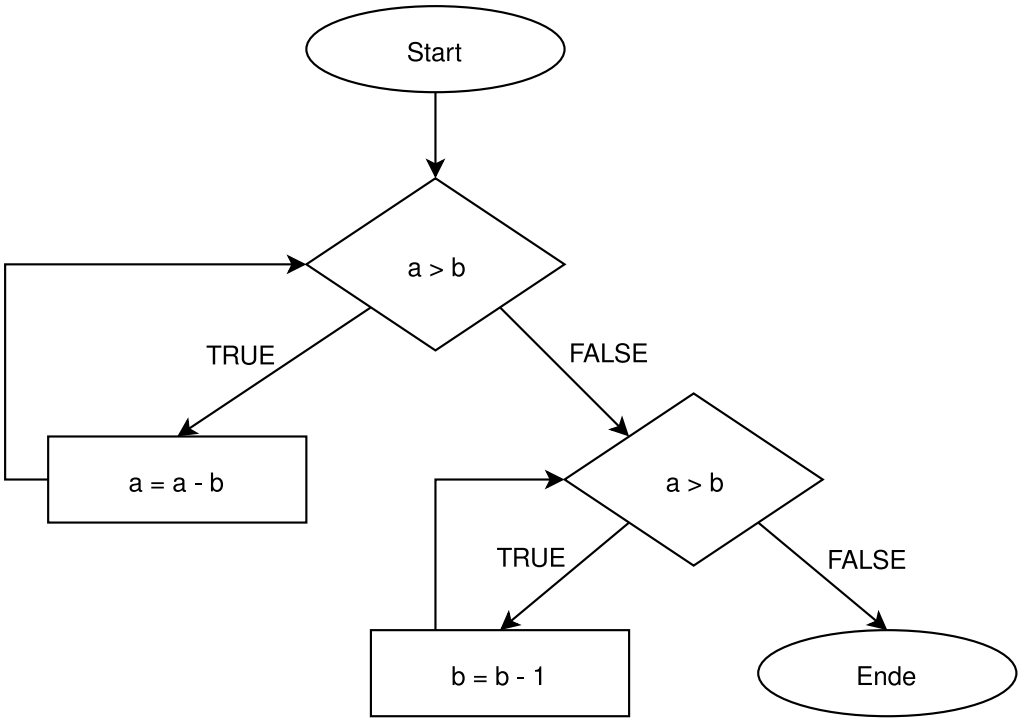
Übungsblatt 6

1. **Kontrollflussgraph**

   Zeichnen Sie den Kontrollflussgraphen für folgendes Python-Programm. Wie viele Tests werden für einen C_0-Test benötigt?

```python
#!/bin/env python3

def some_function(a, b):
    if a > b:
        while a > 1:
            a = a - b
        return a, b
    else:
        while (b * b) > a:
            b = b - 1
    return a, b


def main():
    a = int(input("a = "))
    b = int(input("b = "))

    a, b = some_function(a, b)
    print("Results:")
    print("  a =", a)
    print("  b =", b)


if __name__ == '__main__':
    main()
```

   Lösung:



   Für einen vollständigen C_0-Test werden mindestens drei verschiedene Inputs benötigt (zum Beispiel (3, 1), (1, 1) und (1, 3)).

2. **Softwarequalität**

   Nennen Sie mindestens drei Softwaremetriken, die Sie benutzen würden um Code-Qualität zu bewerten? Warum haben sie sich für diese Metriken entschieden?

   Lösung:

   Beispiele für Softwaremetriken:

   - Lines of code
   - Bugs per lines of code
   - Comment density
   - Zyklomatische Komplexität
   - Execution time
   - Test coverage
   - Anzahl an Klassen
   - Entwicklungszeit
   - Kundenzufriedenheit
   - ...

   Metriken müssen stets an das Einsatzgebiet angepasst werden (keine universellen Standards) und sind trotzdem oft nicht sinnvoll zur Messung von Softwarequalität.

3. **Softwareevolution**

   a) Was verbirgt sich hinter *Lehman's Laws* und warum sind sie noch heute aktuell?

      Lösung:

      Gesetze bzw. Hypothesen, die beschreiben, wie und warum ein großes Softwaresystem sich weiterentwickelt.

      - Continuing Change:
        Ein System muss kontinuierlich angepasst werden, sonst sinkt die Zufriedenheit mit der Zeit
      - Increasing Complexity:
        Die Komplexität des Systems steigt stetig, es sei denn das System wird in Stand gehalten (Maintenance) oder an der Reduzierung dessen gearbeitet
      - Self Regulation:
        Systementwicklungsprozesse sind selbstregulierend mit der Verwaltung von Produkt- und Prozessmaßen
      - Conservation of Organisational Stability:
        Die durchschnittliche „Effective Global Activity Rate" in einem entwickelnden System bleibt über die Lebensdauer des Produktes unveränderlich
      - Conservation of Familiarity:
        Um eine zufriedenstellende Evolution zu erreichen, müssen alle Beteiligten (z.B. Entwickler, Vertriebsmitarbeiter und Kunden) ihre Verhaltensweisen beibehalten
      - Continuing Growth:
        Die Funktionalität eines System muss konstant erweitert werden, um die Zufriedenheit zu gewährleisten
      - Declining Quality:
        Die Qualität eines Systems wird sinken, sofern keine Adaption an veränderte Umweltfaktoren geschieht
      - Feedback System:
        Evolutionsprozesse Stellen ein mehrstufiges Multi-Loop-, Multi-Agent-Feedback-System dar

b) Beschreiben Sie mit eigenen Worten, was sich hinter den Begriffen *Refactoring* und *Reengineering* verbirgt.

Lösung:

Refactoring:

- Veränderung interner Strukturen mit gleichbleibender Funktionalität
- Zum Beispiel Auflösen von Gott-Klassen in mehrere Module
- Verbesserung der Lesbarkeit, Verständlichkeit, Wartbarkeit und Erweiterbarkeit

Reengineering:

- Rekonstruktion eines Systems um Wartbarkeit und Verständlichkeit zu verbessern
- Portierung auf andere Plattformen
- Alte Schnittstellen könne aufgelöst und nur strukturiert werden

c) Warum werden Softwaresysteme über die Zeit hinweg immer komplexer und wie kann man diesem entgegenwirken?

Lösung:

- Ausbau des ursprünglichen Systems/Evolution der Software
- Bug-Fixes, Implementierung neuer Funktionalitäten u.ä. müssen in bestehendes System integriert werden
- Refactoring und Reengineering
- Löschen von nicht mehr verwendeten Code (Feature wurde anderweitig implementiert oder gar gelöscht)

4. **Softwarewartung**

a) Erläutern Sie, was unter dem Begriff *Wartung* im Umfeld der Softwareentwicklung zu verstehen ist!

Lösung:

Änderungen des Systems ab erster Installation (im Spiralmodell ab erstem Prototypen)

b) Welche Arten der Softwarewartung werden unterschieden? Zählen Sie diese Arten auf und geben Sie jeweils eine knappe Erklärung!

Lösung:

Korrektive Wartung

- Auffinden und Korrigieren von Fehlern der Software

Adaptive Wartung

- Anpassung der Software an neue Systemumgebung, Hardware, Änderungen von Standards, ...

Präventive Wartung

- Vermeidung von potentiellen Problemen

Perfektionierende Wartung

- Reengineering mit Redesign und Refactoring
- Optimierung der Performance
- Verbesserung der Wartbarkeit

c) Ist ein *gutes Design* ein adäquater Ersatz für späteres Refactoring? Erläutern Sie ihre Entscheidung.

Lösung:

- Sehr schwer/unmöglich auf Anhieb einen korrekten Systementwurf zu erstellen

    - unbekannte Problemdomäne
    - unverständliche Anforderungen
    - unvorhersehbar, wie sich das System weiterentwickeln wird
- Softwareevolution nur bis zu einem gewissen Grad berücksichtigen
    - keine zu frühen Optimierungen
    - Abstraktion nicht immer sinnvoll
- Gutes Design ist nicht gleichbedeutend mit gutem Code
- Testdriven-Development: Refactoring als fester Bestandteil des Entwicklungsprozesses