

1. Aufgabe: Versionierung / Versionsverwaltung

In der Veranstaltung wurden drei Versionen vorgestellt, die bei der (klassischen Form der) Weiterentwicklung eines Softwareprodukts, welches bereits produktiv ist, mindestens vorhanden sein müssen. Geben Sie die drei Versionen an und erläutern Sie diese bezüglich der Arbeitsschritte (1 – 2 Sätze je Version).

Lösung:

Entwicklungsversion:

In dieser Version wird in einzelnen Releases das Softwareprodukt erstellt und weiterentwickelt. Innerhalb eines Releases werden die Phasen des Wasserfallmodells durchlaufen.

Releaseversion:

In dieser Version werden vor einem Release diejenigen Teile der Entwicklungsversion zusammengestellt, die produktiv werden sollen. Diese Version wird kurz vor dem Release getestet.

Produktivversion:

Diese Version ist die ausgetestete Releaseversion, die produktiv geschaltet wird, dort nochmals getestet wird und dann zwecks Nutzung dem Kunden zur Verfügung steht.

2. Aufgabe: Design Patterns

a) MVC Pattern

Gegeben seien die folgenden Klassen (ohne get- und set-Methoden), die eine Variante des MVC Pattern realisieren. Geben Sie das zugehörige UML Klassendiagramm an!

Klasse View

```
public class View{  
  
    private Control control;  
    private ModelX modelX;  
    private ModelY modelY;  
}
```

Klasse Control

```
public class Control{  
  
    private View view;  
    private ModelX modelX;  
    private ModelY modelY;  
}
```

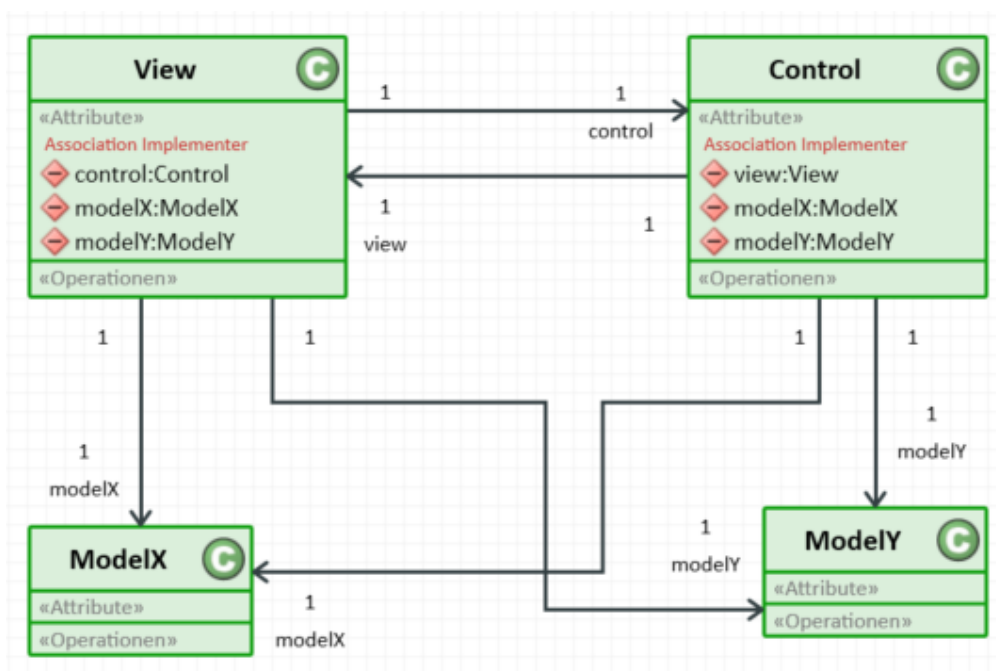
Klasse ModelX

```
public class ModelX{  
}
```

Klasse ModelY

```
public class ModelX{  
}
```

Lösung



b) Singleton Pattern

Ändern Sie die Klasse *ModelX* aus dem Aufgabenteil a) ab. Geben Sie denjenigen Quellcode an, mit welchem man erreicht, dass man maximal ein Objekt vom Typ *ModelX* erzeugen kann.

Geben Sie weiterhin die Anweisung zum Erhalt eines *ModelX*-Objekts an.

Lösung

Klasse ModelX

```
public class ModelX {  
  
    private static ModelX theInstance;  
  
    private ModelX() {  
    }  
  
    public static ModelX getInstance() {  
        if(theInstance == null) {  
            theInstance = new ModelX();  
        }  
        return theInstance;  
    }  
}
```

Anweisung

```
ModelX modelX = ModelX.getInstance();
```

c) Fabrik-Methode Pattern

Gegeben seien die Klassen *Dateienverwaltung* und *DateienService*, erstere siehe unten.

Klasse Dateienverwaltung

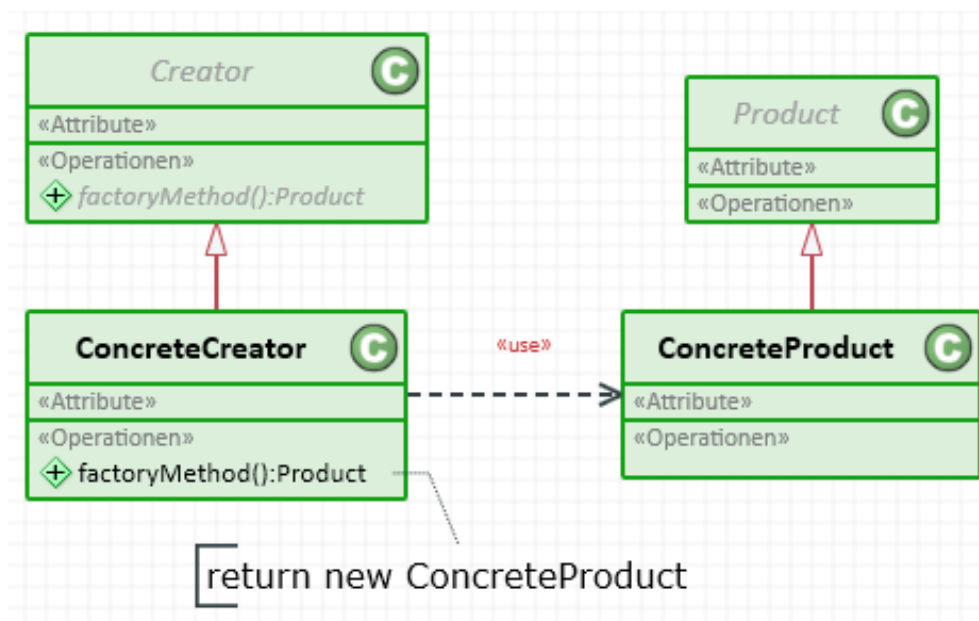
```
import java.io.IOException;  
  
public class Dateienverwaltung {  
  
    private DateienService dateienService = new DateienService();
```

```

public static void main(String[] args) {
    Dateienverwaltung dv = new Dateienverwaltung();
    try {
        dv.dateienService leseDatumAusCsvDatei();
        dv.dateienService.gibDatumInKonsoleAus();
    }
    catch (IOException e) {
        e.printStackTrace();
    }
}
}

```

Gegeben seien die Klassen *Creator*, *Product*, *ConcreteCsvReaderCreator* und *ConcreteCsvReaderProduct*, die das Fabrik-Methode Pattern implementieren.



Karl Eilebrecht, Gernot Starke, Patterns kompakt

Die Klasse *Product* enthält weiterhin eine abstrakte Methode `public abstract String leseDatumAusCsvDatei() throws IOException`. Deren Implementierung in der Klasse *ConcreteCsvReaderProduct* liest eine Zeile aus einer Datei *Datum.csv* und gibt diese als String zurück.

Die Datei *Datum.csv* enthält die Zeile `24;12;2000`. Das Programm liest diese Zeile und gibt sie in der Form `24.12.2000` in der Konsole aus, siehe *main*-Methode der Klasse *Dateienverwaltung*.

Geben Sie den Quellcode der Klasse *ConcreteCsvReaderCreator* an und ergänzen Sie den Quellcode der Klasse *DateienService*.

Lösung

Klasse ConcreteCsvReaderCreator

```
public class ConcreteCsvReaderCreator extends Creator{

    public Product factoryMethod(){
        return new ConcreteCsvReaderProduct();
    }
}
```

Klasse DateienService

```
import java.io.*;

public class DateienService {

    // enthaelt ein Datum in der Form TT.MM.JJJJ
    private String datum;

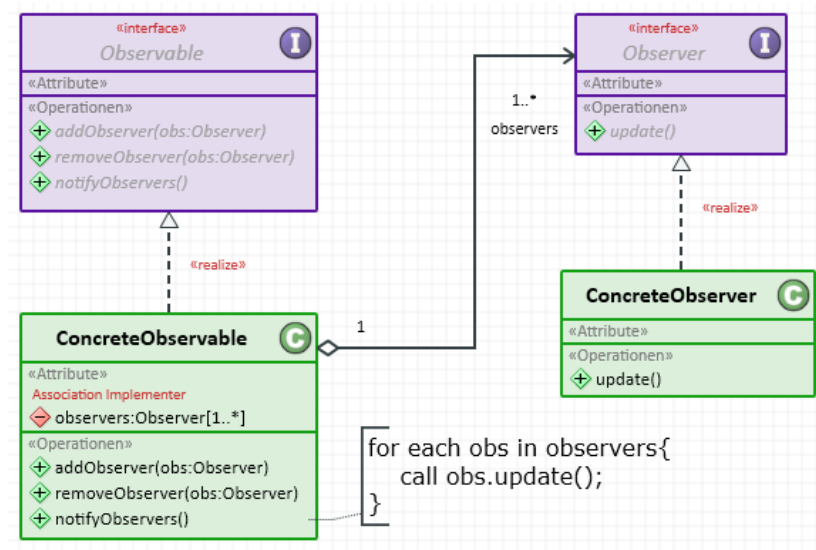
    public void leseDatumAusCsvDatei()
        throws IOException{
        // Hier ergaenzen! Mit Hilfe des Pattern Fabrik-Methode
        // soll das Datum aus Datum.csv gelesen werden (erste
        // Zeile) und dann formatiert und im Attribut datum
        // abgespeichert werden.
        Creator creator = new ConcreteCsvReaderCreator();
        Product product = creator.factoryMethod();
        String[] zeile = product.leseDatumAusCsvDatei()
            .split(";");
        this.datum = zeile[0] + "." + zeile[1] + "." + zeile[2];
    }

    public void gibDatumInKonsoleAus() {
        System.out.println(this.datum);
    }
}
```

d) Observer Pattern

Gegeben seien die Klassen *DateienService* und *Dateienverwaltung* aus dem Aufgabenteil c). Erweitern Sie diese. *DateienService* soll *Observable* werden entsprechend der folgenden Variante des Observer Patterns. *Dateienverwaltung* soll ein Observer von *DateienService* sein und immer, wenn das Attribut *datum* aktualisiert wird, dieses in die Konsole schreiben. Insbesondere wird der Aufruf *dv.dateienService.gibDatumInKonsoleAus();* in der main-Methode nicht mehr benötigt.

Sie können davon ausgehen, dass die Interfaces *Observable* und *Observer* vorhanden sind.



Karl Eilebrecht, Gernot Starke, Patterns kompakt

Lösung

Klasse DateienService

```

import java.io.*;
import java.util.Vector;

public class DateienService implements Observable
{
    private Vector<Observer> observers = new Vector<Observer>();

    // enthaelt ein Datum in der Form TT.MM.JJJJ
    private String datum;

    public void leseDatumAusCsvDatei()
        throws IOException{
        // siehe Aufgabenteil c)
        ...
        notifyObservers();
    }

    public void gibDatumInKonsoleAus() {
        System.out.println(this.datum);
    }
}

```

```

@Override
public void addObserver(Observer obs) {
    this.observers.add(obs);
}

@Override
public void removeObserver(Observer obs) {
    this.observers.remove(obs);
}

@Override
public void notifyObservers() {
    for(int i = 0; i < this.observers.size(); i++) {
        this.observers.get(i).update();
    }
}
}

```

Klasse Dateienverwaltung

```

import java.io.IOException;

public class Dateienverwaltung implements Observer {

    DateienService dateienService = new DateienService();

    public Dateienverwaltung() {
        this.dateienService.addObserver(this);
    }

    public void update() {
        dateienService.gibDatumInKonsoleAus();
    }

    public static void main(String[] args) {
        Dateienverwaltung dv = new Dateienverwaltung();
        try {
            dv.dateienService.leseDatumAusCsvDatei();
            // dv.dateienService.gibDatumInKonsoleAus();
        }
        catch (IOException e) {
            e.printStackTrace();
        }
    }
}

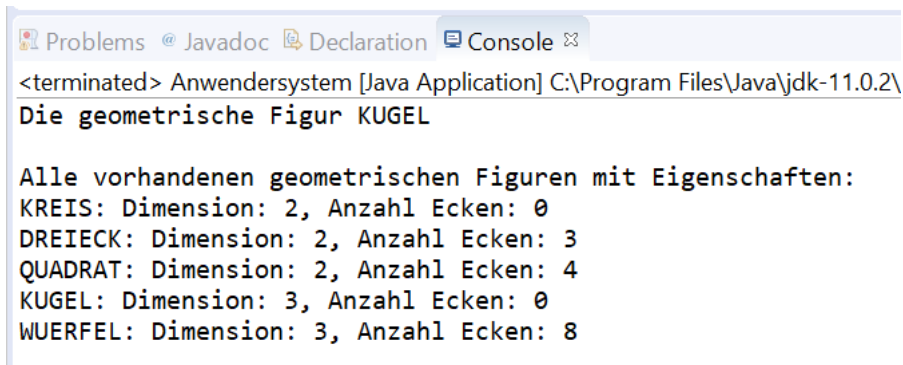
```

3. Aufgabe: Vertiefung Java

a) Enumeration

Erstellen Sie eine Enumeration *GeometrischeFigur* mit einem Kreis, Dreieck, Quadrat, einer Kugel und einem Würfel. Diese haben die Eigenschaften *dimension* und *anzahlEcken*, welche im Konstruktor belegt werden. Die Werte der Eigenschaften werden mittels get-Methoden herausgegeben.

Erstellen Sie eine Klasse *Anwendersystem* mit einer main-Methode, welche die folgende Konsolenausgabe erzeugt. Benutzen Sie eine for each – Schleife für die Ausgabe.



```
Problems @ Javadoc Declaration Console
<terminated> Anwendersystem [Java Application] C:\Program Files\Java\jdk-11.0.2\
Die geometrische Figur KUGEL

Alle vorhandenen geometrischen Figuren mit Eigenschaften:
KREIS: Dimension: 2, Anzahl Ecken: 0
DREIECK: Dimension: 2, Anzahl Ecken: 3
QUADRAT: Dimension: 2, Anzahl Ecken: 4
KUGEL: Dimension: 3, Anzahl Ecken: 0
WUERFEL: Dimension: 3, Anzahl Ecken: 8
```

Lösung

Enumeration GeometrischeFigur

```
public enum GeometrischeFigur {

    // Enumerationskonstanten
    KREIS(2, 0),
    DREIECK(2, 3),
    QUADRAT(2, 4),
    KUGEL(3, 0),
    WUERFEL(3, 8);

    private int dimension;
    private int anzahlEcken;

    GeometrischeFigur(int dimension, int anzahlEcken){
        this.dimension = dimension;
        this.anzahlEcken = anzahlEcken;
    }

    public int getDimension() {
        return dimension;
    }
}
```



```

    public int getAnzahlEcken() {
        return anzahlEcken;
    }
}

```

Klasse Anwendersystem

```

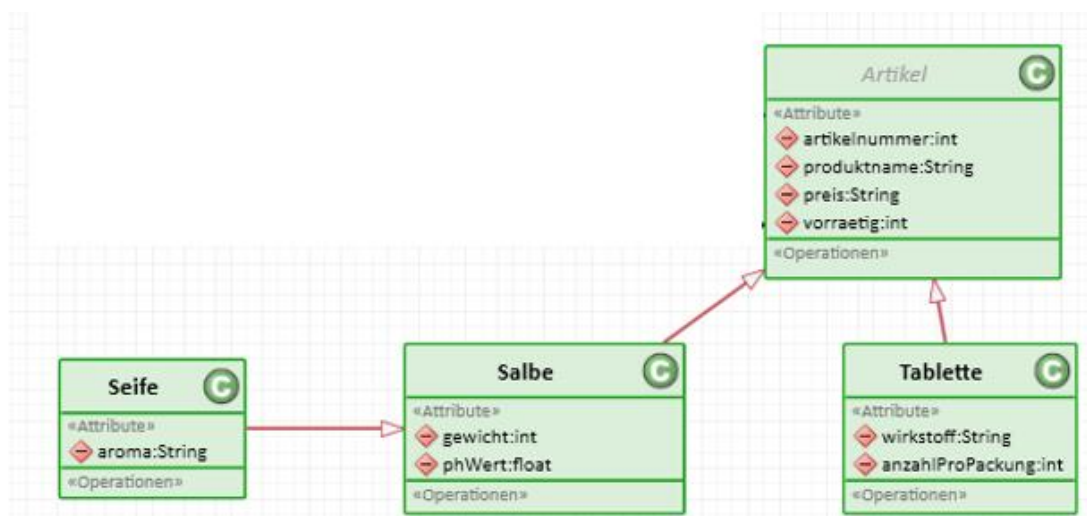
public class Anwendersystem {

    public static void main( String[] args){
        GeometrischeFigur[] geometrischeFiguren
            = GeometrischeFigur.values();
        GeometrischeFigur kugel = GeometrischeFigur.KUGEL;
        System.out.println(
            "Die geometrische Figur " + kugel + "\n");
        System.out.println("Alle vorhandenen geometrischen "
            + "Figuren mit Eigenschaften:");
        for(GeometrischeFigur geomFigur : geometrischeFiguren){
            System.out.println(geomFigur + ": "
                + "Dimension: " + geomFigur.getDimension()
                + ", Anzahl Ecken: "
                + geomFigur.getAnzahlEcken());
        }
    }
}

```

b) Generics

- i) Erstellen Sie die Klassen zu dem folgenden UML Klassendiagramm inklusive Konstruktoren, in welchen die Attribute mit Werten belegt werden, und get-und set-Methoden.



Weiterhin sollen die Klassen jeweils eine Methode *public String gibAttributeZurueck()* enthalten, die die Werte der Attribute, mit Komma getrennt, zurückgibt. Verwenden Sie Überschreibung.

Ergänzen Sie die vorliegende Klasse *Anwendersystem*, um die folgende Ausgabe in der Konsole zu erhalten. Benutzen Sie ein Attribut *artikelliste* vom Typ *ArrayList* und für die Ausgabe eine for each - Schleife.

```
1, Tablette gegen Entzündungen, 12,90 Euro, 50, Eisenkraut, 50
2, Zinksalbe, 8,90 Euro, 30, 75, 5.5
3, Kernseife, 3,90 Euro, 20, 100, 6.5, neutral
```

Lösung

Klasse Anwendersystem

```
import java.util.ArrayList;

public class Anwendersystem {

    // Hier ergaenzen
    private ArrayList<Artikel> artikelliste
        = new ArrayList<>();

    public void fuelleArtikelliste() {
        this.artikelliste.add(new Tablette(
            1, "Tablette gegen Entzündungen",
            "12,90 Euro", 50, "Eisenkraut", 50));
        this.artikelliste.add(new Salbe(
            2, "Zinksalbe", "8,90 Euro", 30, 75, 5.5f));
        this.artikelliste.add(new Seife(
            3, "Kernseife", "3,90 Euro", 20, 100, 6.5f,
            "neutral"));
    }

    public void gibArtikellisteAus() {
        // Hier ergaenzen
        for(Artikel artikel : this.artikelliste) {
            System.out.println(
                artikel.gibAttributeZurueck());
        }
    }

    public static void main(String[] args) {
        Anwendersystem anw = new Anwendersystem();
        anw.fuelleArtikelliste();
        anw.gibArtikellisteAus();
    }
}
```

Klasse Artikel

```
public abstract class Artikel {

    private int artikelnummer;
    private String produktname;
    private String preis;
    private int vorraetig;

    public Artikel(int artikelnummer, String produktname,
        String preis, int vorraetig) {
        super();
        this.artikelnummer = artikelnummer;
        this.produktname = produktname;
        this.preis = preis;
        this.vorraetig = vorraetig;
    }

    public int getArtikelnummer() {
        return artikelnummer;
    }

    public void setArtikelnummer(int artikelnummer) {
        this.artikelnummer = artikelnummer;
    }

    public String getProduktname() {
        return produktname;
    }

    public void setProduktname(String produktname) {
        this.produktname = produktname;
    }

    public String getPreis() {
        return preis;
    }

    public void setPreis(String preis) {
        this.preis = preis;
    }

    public int getVorraetig() {
        return vorraetig;
    }

    public void setVorraetig(int vorraetig) {
        this.vorraetig = vorraetig;
    }
}
```

```

        public String gibAttributeZurueck() {
            return getArtikelnummer() + ", "
                + getProduktname() + ", "
                + getPreis() + ", " + getVorraetig();
        }
    }
}

```

Klasse Tablette

```

public class Tablette extends Artikel{

    private String wirkstoff;
    private int anzahlProPackung;

    public Tablette(int artikelnummer, String produktname,
        String preis, int vorraetig,
        String wirkstoff, int anzahlProPackung) {
        super(artikelnummer, produktname, preis, vorraetig);
        this.wirkstoff = wirkstoff;
        this.anzahlProPackung = anzahlProPackung;
    }

    public String getWirkstoff() {
        return wirkstoff;
    }

    public void setWirkstoff(String wirkstoff) {
        this.wirkstoff = wirkstoff;
    }

    public int getAnzahlProPackung() {
        return anzahlProPackung;
    }

    public void setAnzahlProPackung(int anzahlProPackung) {
        this.anzahlProPackung = anzahlProPackung;
    }

    public String gibAttributeZurueck() {
        return super.gibAttributeZurueck() + ", "
            + getWirkstoff() + ", " + getAnzahlProPackung();
    }

}

```

Klasse Salbe

```
public class Salbe extends Artikel{
    private int gewicht;
    private float phWert;

    public Salbe(int artikelnummer, String produktname,
        String preis, int vorraetig, int gewicht,
        float phWert) {
        super(artikelnummer, produktname, preis, vorraetig);
        this.gewicht = gewicht;
        this.phWert = phWert;
    }

    public int getGewicht() {
        return gewicht;
    }

    public void setGewicht(int gewicht) {
        this.gewicht = gewicht;
    }

    public float getPhWert() {
        return phWert;
    }

    public void setPhWert(float phWert) {
        this.phWert = phWert;
    }

    public String gibAttributeZurueck() {
        return super.gibAttributeZurueck() + ", "
            + getGewicht() + ", " + getPhWert();
    }

}
```

Klasse Seife

```
public class Seife extends Salbe{

    private String aroma;

    public Seife(int artikelnummer, String produktname,
        String preis, int vorraetig, int gewicht,
        float phWert, String aroma) {
        super(artikelnummer, produktname, preis, vorraetig,
            gewicht, phWert);
        this.aroma = aroma;
    }

}
```

```

    public String getAroma() {
        return aroma;
    }

    public void setAroma(String aroma) {
        this.aroma = aroma;
    }

    public String gibAttributeZurueck() {
        return super.gibAttributeZurueck() + ", "
            + getAroma();
    }
}

```

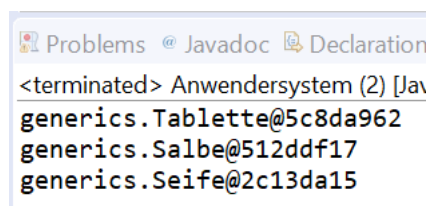
- ii) Erstellen Sie eine Klasse *ArrayListMitAusgabe*, welche von *ArrayList* ableitet und die folgende Methode *gibListeAlsZeilenZurueck()* enthält. Sie gibt ein Array von Strings zurück, welches die einzelnen Listenelemente als Strings enthält.

```

    public String[] gibListeAlsZeilenZurueck() {
        String[] ergebnis = new String[this.size()];
        for(int i = 0; i < this.size(); i++) {
            ergebnis[i] = this.get(i).toString();
        }
        return ergebnis;
    }
}

```

Ändern Sie das Attribut *artikelliste* und die Methode *gibArtikellisteAus* der Klasse *Anwendersystem* ab. Sie soll *ArrayListMitAusgabe* benutzen. Allerdings erhalten Sie noch nicht die gewünschte Konsolenausgabe sondern die folgende (, *generics* ist hier der Name des packages, in welchem die Klassen liegen).



```

<terminated> Anwendersystem (2) [Ja
generics.Tablette@5c8da962
generics.Salbe@512ddf17
generics.Seife@2c13da15

```

Nehmen Sie Änderungen in der Klasse *ArrayListMitAusgabe* vor, so dass Sie die gewünschte Ausgabe erhalten. Benutzen Sie eine Typeinschränkung mittels *extends*.

Lösung

Klasse Anwendersystem

```
import java.util.ArrayList;

public class Anwendersystem {

    // Hier ergaenzen
    private ArrayList<Artikel> artikelliste
    = new ArrayList<>();
    private ArrayListMitAusgabe<Artikel> artikelliste
        = new ArrayListMitAusgabe<>();

    public void fuelleArtikelliste() {
        ...
    }

    public void gibArtikellisteAus() {
        // Hier ergaenzen
        for(Artikel artikel : this.artikelliste) {
        System.out.println(
        artikel.gibAttributeZurueck());
        }

        String[] zeilen = this.artikelliste
            .gibListeAlsZeilenZurueck();
        for(String zeile : zeilen) {
            System.out.println(zeile);
        }
    }

    public static void main(String[] args) {
        Anwendersystem anw = new Anwendersystem();
        anw.fuelleArtikelliste();
        anw.gibArtikellisteAus();
    }
}
```

Klasse ArrayListMitAusgabe

```
import java.util.ArrayList;

public class ArrayListMitAusgabe<T extends Artikel>
    extends ArrayList<T>{
```

```

public String[] gibListeAlsZeilenZurueck() {
    String[] ergebnis = new String[this.size()];
    for(int i = 0; i < this.size(); i++) {
        ergebnis[i] = this.get(i).toString();

        ergebnis[i]
            = this.get(i).gibAttributeZurueck();
    }
    return ergebnis;
}
}

```

iii) Kommentieren Sie die Änderungen für den Aufgabenteil ii) in der Klasse *Anwendersystem* aus. Basis ist das Ergebnis des Aufgabenteils i).

Überladen Sie die Methode *gibArtikellisteAus* mit einer Methode, welche die Artikel einer als Parameter vorgegebenen Artikelliste in der Konsole ausgibt. Die Elemente der vorgegebenen Artikelliste sollen vom Typ *Artikel* oder einer Superklasse von *Artikel* sein.

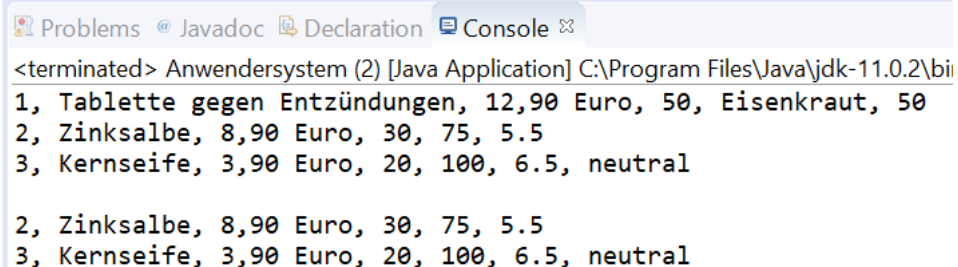
Erstellen Sie eine Methode *selektiereSeifenSalben*, welche aus einer vorgegebenen Liste *quelle* von Artikeln die Seifen und Salben in eine vorgegebene Liste *ziel* hineinkopiert. Verwenden Sie Upper- und Lower bounded wildcards.

Erweitern Sie die main-Methode folgendermaßen, um die folgende Ausgabe in der Konsole zu erhalten.

```

// Aufgabenteil iii)
System.out.println("");
ArrayList<Salbe> listeSeifenSalben
    = new ArrayList<Salbe>();
anw.selektiereSeifenSalben(
    listeSeifenSalben, anw.artikelliste);
anw.gibArtikellisteAus(listeSeifenSalben);

```



```

<terminated> Anwendersystem (2) [Java Application] C:\Program Files\Java\jdk-11.0.2\bin
1, Tablette gegen Entzündungen, 12,90 Euro, 50, Eisenkraut, 50
2, Zinksalbe, 8,90 Euro, 30, 75, 5.5
3, Kernseife, 3,90 Euro, 20, 100, 6.5, neutral

2, Zinksalbe, 8,90 Euro, 30, 75, 5.5
3, Kernseife, 3,90 Euro, 20, 100, 6.5, neutral

```


Lösung

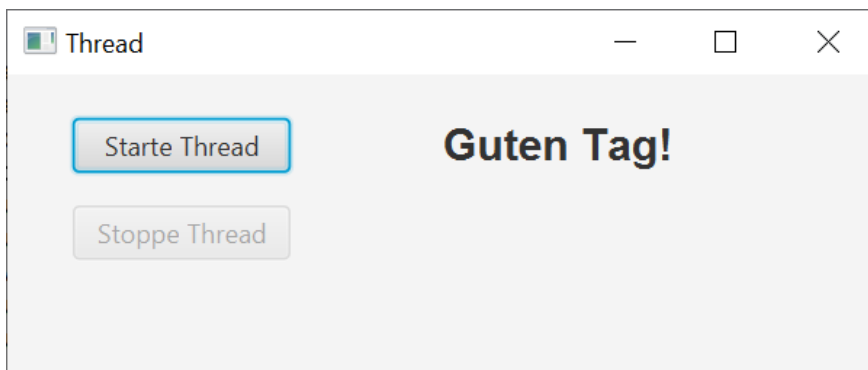
In der Klasse Anwendersystem

```
// Aufgabenteil iii)
public void selektiereSeifenSalben(
    ArrayList<? super Salbe> ziel,
    ArrayList<? extends Artikel> quelle) {
    for(Artikel artikel : quelle) {
        if(artikel instanceof Salbe) {
            ziel.add((Salbe)artikel);
        }
    }
}

// Aufgabenteil iii)
public <T extends Artikel> void gibArtikellisteAus(
    ArrayList<T> artikelliste) {
    for(Artikel artikel : artikelliste) {
        System.out.println(artikel.gibAttributeZurueck());
    }
}
```

c) Innere anonyme Klasse und Lambda-Ausdrücke

Gegeben ist die Vorgabe zu einer Anwendung mit der folgenden Oberfläche.



Beim Klick auf *Starte Thread* wandert das Label *Guten Tag!* nach unten. Mit dem Klick auf *Stoppe Thread* hört diese Aktion wieder auf. Es werden für die Aktionen beider Buttons innere anonyme Klassen verwendet.

Erstellen Sie eine innere, aber nicht anonyme Klasse für die Aktion des Buttons *Starte Thread*. Erstellen Sie einen Lambda-Ausdruck für die Aktion des Buttons *Stoppe Thread*.

Lösung

In Klasse Fenster

```
private void initListener(){
    btnStart.setAction(new EventHandlerBtnStart());
    btnStopp.setAction(actionEvent -> {
        timer.setStopp(true);
        btnStart.setDisable(false);
        btnStopp.setDisable(true);
    });
}

private class EventHandlerBtnStart
    implements EventHandler<ActionEvent>{

    @Override
    public void handle(ActionEvent e) {
        starteTimer();
        btnStart.setDisable(true);
        btnStopp.setDisable(false);
    }
}
```

d) Supplier und Lambda-Ausdruck

- i) Gegeben ist die Vorgabe zu einer Anwendung zu einem Benutzer mit Passwort. Ändern Sie diese ab.
Die Klasse *PasswortSupplier* soll vom Typ *Supplier<String>* sein.
Zum Erstellen der Länge des Passworts deklarieren Sie innerhalb der main-Methode eine Variable vom Typ *IntSupplier*. Diese initialisieren Sie mit Hilfe eines Lambda-Ausdrucks.

Lösung

In Klasse Anwendung

```
public static void main(String[] args) {
    IntSupplier intSupplier = () -> {return 8;};
    Benutzer benutzer = new Benutzer("musterperson",
        new PasswortSupplier(intSupplier.getAsInt()).get());
    System.out.println(benutzer.toString());
}
```

Klasse PasswortSupplier

```
package benutzerMitPasswort;

import java.security.SecureRandom;
import java.util.function.Supplier;

public class PasswortSupplier implements Supplier<String>{

    private final int laenge;
    private static String ERLAUBTE_CHARS
        = "0123456789abcdefghijklmnopqrstuvwxyz%&?!";

    public PasswortSupplier(int laenge){
        this.laenge = laenge;
    }

    @Override
    public String get() {
        SecureRandom random = new SecureRandom();
        StringBuffer password = new StringBuffer();
        for (int i = 0; i < this.laenge; i++) {
            password.append(ERLAUBTE_CHARS.charAt(
                random.nextInt(ERLAUBTE_CHARS.length())));
        }
        return password.toString();
    }
}
```

- i) Ergänzen Sie die Klasse *Benutzer* um eine statische Methode *zufaellicheLaengePasswort*, welche einen pseudozufällig gewählten int-Wert zwischen 8 und 16 erzeugt und herausgibt. Erstellen Sie mit Hilfe dieser Methode innerhalb der main-Methode eine Variable vom Typ *IntSupplier*, die Sie für die Erzeugung eines Benutzers verwenden. Ist *zufaellicheLaengePasswort* eine *Pure Function*?

Lösung

In Klasse Benutzer

```
public static int zufaellicheLaengePasswort() {
    SecureRandom random = new SecureRandom();
    return random.nextInt(8) + 8;
}
```

Klasse Anwendung

```
package benutzerMitPasswort;

import java.util.function.IntSupplier;

public class Anwendung {

    public static void main(String[] args) {
        // IntSupplier intSupplier = () -> {return 8;};
        IntSupplier intSupplier
            = Benutzer :: zufaelligeLaengePasswort;
        Benutzer benutzer = new Benutzer("musterperson",
            new PasswortSupplier(intSupplier.getAsInt()).get());
        System.out.println(benutzer.toString());
    }
}
```

Ist *zufaelligeLaengePasswort* eine *Pure Function*? **JA!**

4. Aufgabe: Softwaretests

Gegeben sei die Methode

```
public double berechne(int a, int b)
    throws Exception {...}
```

Sie berechnet: $1 / (\sqrt{a} * (b - 15)^2)$

Falls *a* kleiner 0 ist, oder der Nenner gleich 0 ist, wird eine *IllegalArgumentException* geworfen.

- a) Ergänzen Sie den vorliegenden Quellcode eines JUnit Tests. Die Methode *berechne* liegt in der Klasse *Berechnungen* und soll durch diese JUnit Testklasse getestet werden. Implementieren Sie drei Testfälle. Die ersten beiden Testfälle sollen eine gewünschte Berechnung testen, der dritte Testfall soll eine gewünschte Exception testen. Sie können davon ausgehen, dass die Klasse *Berechnungen* einen Default-Konstruktor besitzt. Benutzen Sie aus der Klasse *Assertions* *assertTrue(boolean condition, String message)*, *assertEquals(double expected, double actual, double delta, String message)* und *fail(String message)*.

Lösung

```
import static org.junit.jupiter.api.Assertions.*;

import org.junit.jupiter.api.*;
```

```

public class BerechnungenTest{

    // Ergaenzen!!!
    private Berechnungen berechnungen;

    @BeforeEach
    public void setUp() throws Exception{

        // Ergaenzen!!!
        this.berechnungen = new Berechnungen();

    }

    @AfterEach

    public void tearDown() throws Exception{

        // Ergaenzen!!!
        this.berechnungen = null;

    }

    @Test

    public void testBerechne() throws Exception{

        // Ergaenzen!!!
        double diff = this.berechnungen.berechne(4, 14) - 0.5;
        assertTrue(diff < 0.01 && -diff < 0.01, "Fall (4, 14)");
        assertEquals(1, this.berechnungen.berechne(1, 16),
            0.01, "Fall (1, 16)");
        try{
            this.berechnungen.berechne(4,15);
            fail("Exception fehlt.");
        }
        catch(Exception exc){}

    }

}

```

- b) Ergänzen Sie vier weitere Testfälle innerhalb einer neuen Testmethode. Die ersten drei Testfälle sollen eine gewünschte Berechnung testen, der vierte Testfall soll eine gewünschte Exception testen. Sie können davon ausgehen, dass die Klasse Berechnungen einen Default-Konstruktor besitzt. Benutzen Sie aus der Klasse Assertions
- assertTrue(boolean condition, Supplier<String> messageSupplier),*
assertTrue(BooleanSupplier conditionSupplier, String message),
assertTrue(BooleanSupplier conditionSupplier,
Supplier<String> messageSupplier) und
T assertTrueThrows(Class<T> clazz, Executable executable. Testen Sie bei der Exception sowohl den Typ als auch die Nachricht.

Lösung

In Klasse BerechnungenTest

```
@Test
public void testBerechneMitLambdas()
    throws Exception{
    final double diff1
        = this.berechnungen.berechne(16, 17) - 0.0625;
    assertTrue(diff1 < 0.01 && -diff1 < 0.01, () -> {
        return "Fall (16, 17)";
    });
    final double diff2
        = this.berechnungen.berechne(16, 13) - 0.0625;
    assertTrue(() -> {
        return diff2 < 0.01 && -diff2 < 0.01;
    }, "Fall (16, 13)");
    final double diff3
        = this.berechnungen.berechne(16, 16) - 0.25;
    Assertions.assertTrue(() -> {
        return diff3 < 0.01 && -diff3 < 0.01;
    }, () -> {
        return "Fall (16, 16)";
    });
    Throwable exc = assertThrows(
        IllegalArgumentException.class,
        () -> {
            this.berechnungen.berechne(-1, 16);
        });
    assertEquals("Wurzel negativ!", exc.getMessage());
}
```

5. Aufgabe: SOLID

a) Single Responsibility Principle

Gegeben sei die folgende Klasse *Mitarbeiter*. Gegeben sei eine Klasse *Anwendersystem*, welche die folgende Ausgabe erzeugt.

- i) Inwiefern wird das Single Responsibility Principle nicht erfüllt?
- ii) Geben Sie eine mögliche Änderung an, so dass das Single Responsibility Principle erfüllt wird.

Klasse Mitarbeiter

```
public class Mitarbeiter {

    private int personalnummer;
    private String vorname;
    private String nachname;
    private String abteilung;
    private String gebaeude;
    private int bueronummer;
    private String strasseHnr;
    private String plz;
    private String ort;

    // Konstruktor, der Attribute mit Werten belegt
    ...
    // get- und set-Methoden
    ...

    public String gibAttributeZurueck() {
        return this.getPersonalnummer() + "\n"
            + this.getVorname() + " " + this.getNachname() + "\n"
            + this.getAbteilung() + "\n"
            + this.getGebaeude() + ": Raum "
            + this.getBueronummer() + "\n"
            + this.getStrasseHnr() + "\n"
            + this.getPlz() + " " + this.getOrt();
    }
}
```

```
Problems @ Javadoc Declaration Console
<terminated> Anwendersystem (4) [Java Application] C:\
Mitarbeiter:

1
Elke Musterfrau
Entwicklung
Turm-Gebäude: Raum 32
Kurze Straße 1
44795 Bochum
2
Max Mustermann
Vertrieb
L-Gebäude: Raum 13
Lange Straße 103
44795 Bochum
```

Lösung

zu i) Es gibt keine Single Responsibility für die Daten eines Mitarbeiters und eines Büros.

zu ii)

Klasse Mitarbeiter

```
public class Mitarbeiter {

    private int personalnummer;
    private String vorname;
    private String nachname;
    private String abteilung;
    private Buero buero;

    public Mitarbeiter(int personalnummer, String vorname,
        String nachname,
        String abteilung, String gebaeude,
        int bueronummer, String strasseHnr, String plz,
        String ort) {
        ...
        this.buero = new Buero(
            gebaeude, bueronummer, strasseHnr, plz, ort);
    }

    // get- und set-Methoden
    ...

    public String gibAttributeZurueck() {
        return this.getPersonalnummer() + "\n"
            + this.getVorname() + " " + this.getNachname() + "\n"
            + this.getAbteilung() + "\n"
            + this.getBuero().gibAttributeZurueck();
    }
}
```


Klasse Buero

```
public class Buero {

    private String gebaeude;
    private int bueronummer;
    private String strasseHnr;
    private String plz;
    private String ort;

    // Konstruktor, der Attribute mit Werten belegt
    ...
    // get- und set-Methoden
    ...
    public String gibAttributeZurueck() {
        return this.getGebaeude() + ": Raum "
            + this.getBueronummer() + "\n"
            + this.getStrasseHnr() + "\n"
            + this.getPlz() + " " + this.getOrt();
    }

}
```

b) Open Closed Principle

Gegeben sei die folgende Klasse *Holzbrett*. Gegeben sei eine Klasse *Anwendersystem*, welche die folgende Ausgabe erzeugt.

- i) Nennen Sie einen Grund dafür, dass das Open Closed Principle eingehalten werden sollte (1-2 Sätze).
- ii) Inwiefern wird das Open Closed Principle nicht erfüllt?
- iii) Geben Sie eine mögliche Änderung an, so dass das Open Closed Principle erfüllt wird.

Klasse Holzbrett

```
public class Holzbrett {

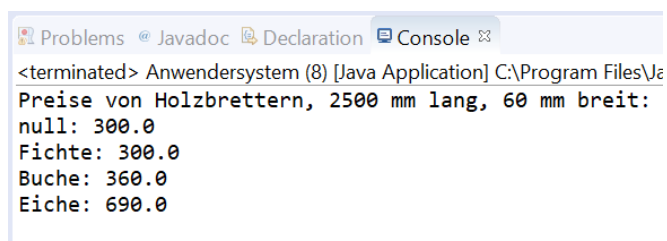
    private int nrHolzbrett;
    private String holzart;

    // Konstruktor, der Attribute mit Werten belegt
    ...
    // get- und set-Methoden
    ...
}
```

```

    public double berechnePreis(int laenge, int breite) {
        double ergebnis = laenge * breite * 0.002;
        if(getHolzart() != null) {
            if(getHolzart().equals("Douglasie")
                || getHolzart().equals("Buche")) {
                ergebnis = ergebnis * 1.2;
            }
            if(getHolzart().equals("Eiche")
                || getHolzart().equals("Lärche")) {
                ergebnis = ergebnis * 2.3;
            }
        }
        return ergebnis;
    }
}

```



```

<terminated> Anwendersystem (8) [Java Application] C:\Program Files\Ja
Preise von Holzbrettern, 2500 mm lang, 60 mm breit:
null: 300.0
Fichte: 300.0
Buche: 360.0
Eiche: 690.0

```

Lösung

zu i) Die Wahrscheinlichkeit, dass eine Klasse zu einem späteren Zeitpunkt geändert werden muss, sinkt. Damit sinkt auch die Fehleranfälligkeit bei der Wartung.

zu ii) Bei neuen Holzarten muss die Methode *berechnePreis* geändert werden.

zu iii)

Klasse Holzbrett

```

public class Holzbrett {
    ...
    public double berechnePreis(int laenge, int breite) {
        double ergebnis = laenge * breite * 0.002;
        if(getHolzart() != null) {
            if(getHolzart().equals("Douglasie")
                || getHolzart().equals("Buche")) {
                ergebnis = ergebnis * 1.2;
            }
            if(getHolzart().equals("Eiche")
                || getHolzart().equals("Lärche")) {
                ergebnis = ergebnis * 2.3;
            }
        }
        return ergebnis;
    }
}

```

Klasse HolzbrettSpezielleHolzart

```
public class HolzbrettSpezielleHolzart extends Holzbrett{

    public HolzbrettSpezielleHolzart(int nrHolzbrett,
        String holzart) {
        super(nrHolzbrett, holzart);
        // Achtung: Liskov Substitution Principle wird verletzt
        // (wegen Fachlichkeit)!
        if(holzart == null){
            throw new IllegalArgumentException(
                "Die spezielle Holzart darf nicht null sein.");
        }
    }

    @Override
    // Noch besser waere die Verwendung einer Liste mit
    // einer Aufstellung: Holzart-Faktor!
    public double berechnePreis(int laenge, int breite) {
        double ergebnis = super.berechnePreis(laenge, breite);
        if(getHolzart().equals("Douglasie")
            || getHolzart().equals("Buche")) {
            ergebnis = ergebnis * 1.2;
        }
        if(getHolzart().equals("Eiche")
            || getHolzart().equals("Lärche")) {
            ergebnis = ergebnis * 2.3;
        }
        return ergebnis;
    }
}
```

c) Liskov Substitution Principle

Gegeben sei die folgende Klasse *Artikel*, *Salbe* und *Anwendersystem*.

- i) Inwiefern wird das Liskov Substitution Principle nicht erfüllt?
- ii) Welche negative Auswirkung hat es, dass das Liskov Substitution Principle nicht erfüllt wird?
- iii) Geben Sie eine mögliche Änderung in den Klassen an, so dass das Liskov Substitution Principle erfüllt wird. Eine fachliche Änderung ist erlaubt. (Weitere Änderungen / Optimierungen brauchen Sie nicht vornehmen.)

Klasse Artikel

```
public class Artikel {

    private int artikelnummer;
    private String produktname;
```

```

private String preis;
private int vorraetig;

public Artikel(int artikelnummer, String produktname,
    String preis, int vorraetig) {
    super();
    this.artikelnummer = artikelnummer;
    this.produktname = produktname;
    this.preis = preis;
    this.vorraetig = vorraetig;
}

// get- und set-Methoden
...

public void gibAttributeAus() {
    System.out.print(getArtikelnummer() + ", "
        + getProduktname() + ", "
        + getPreis() + ", " + getVorraetig());
}
}

```

Klasse Salbe

```

public class Salbe extends Artikel{

    private int gewicht;
    private float phWert;

    public Salbe(int artikelnummer, String produktname,
        String preis, int vorraetig, int gewicht, float phWert) {
        super(artikelnummer, produktname, preis, vorraetig);
        this.gewicht = gewicht;
        this.phWert = phWert;
    }
    // get- und set-Methoden
    ...

    @Override
    public void gibAttributeAus() {
        if(getArtikelnummer() <= 0) {
            System.out.println(
                "Die Artikelnummer muss größer als 0 sein.");
        }
        else {
            super.gibAttributeAus();
            System.out.print(", "
                + getGewicht() + ", " + getPhWert());
        }
    }
}
}

```

Klasse Anwendersystem

```
public class Anwendersystem {

    public static void main(String[] args) {
        Artikel[] artikelArray = new Artikel[2];
        artikelArray[0] = new Artikel(1,
            "Tablette gegen Entzündungen",
            "12,90 Euro", 50);
        artikelArray[1] = new Salbe(0, "Zinksalbe",
            "8,90 Euro", 30, 75, 5.5f);
        for(int i = 0; i < artikelArray.length; i++) {
            artikelArray[i].gibAttributeAus();
            System.out.println("");
        }
    }
}
```

Lösung

zu i) Salben und Artikel verhalten sich unterschiedlich für den Fall, dass die Artikelnummer kleiner oder gleich 0 ist.

zu ii) Falls immer eine Ausgabe der Werte der Attribute erwartet wird, gibt es bei Salben mit einer Artikelnummer kleiner oder gleich 0 eine Überraschung.

zu iii)

Erste Möglichkeit: Man ergänzt die Abfrage der Artikelnummer auf größer 0 in der Klasse *Artikel*.

Zweite Möglichkeit: Man entfernt die Abfrage der Artikelnummer auf größer oder gleich 0 in der Klasse *Salbe*.

zu der ersten Möglichkeit: in Klasse Artikel

```
public void gibAttributeAus() {
    if(getArtikelnummer() <= 0) {
        System.out.println(
            "Die Artikelnummer muss größer als 0 sein.");
    }

    else {
        System.out.print(getArtikelnummer() + ", "
            + getProduktname() + ", "
            + getPreis() + ", " + getVorraetig());
    }
}
```

zu der zweiten Möglichkeit: in Klasse Salbe

```
@Override
public void gibAttributeAus() {
    super.gibAttributeAus();
    System.out.print(", " + getGewicht() + " "
        + getPhWert());
}
```

d) Interface Segregation Principle

Gegeben seien das folgende Interface *Berechnungen* und die Klassen *Kreis* und *Kugel*. Gegeben sei die folgende Klasse *Anwendersystem*, welche die folgende Ausgabe erzeugt.

- i) Inwiefern wird das Interface Segregation Principle nicht erfüllt?
- ii) Geben Sie eine mögliche Änderung an, so dass das Interface Segregation Principle erfüllt wird.

Interface Berechnungen

```
public interface Berechnungen {

    public double berechneRadius();
    public double berechneUmfang();
    public double berechneFlaeche();
    public double berechneVolumen();
}
```

Klasse Kreis

```
public class Kreis implements Berechnungen{

    private double xWert;
    private double yWert;
    private double radius;

    // Konstruktor, der Attribute mit Werten belegt
    ...
    // get- und set-Methoden
    ...

    @Override
    public double berechneRadius() {
        return this.getRadius();
    }
}
```

```

@Override
public double berechneUmfang() {
    return 2 * Math.PI * this.getRadius();
}

@Override
public double berechneFlaeche() {
    return Math.PI * this.getRadius() * this.getRadius();
}

@Override
public double berechneVolumen() {
    return 0;
}
}

```

Klasse Kugel

```

public class Kugel extends Kreis implements Berechnungen{

    private double zWert;

    public Kugel(double xWert, double yWert, double zWert,
        double radius) {
        super(xWert, yWert, radius);
        this.zWert = zWert;
    }

    public double getZWert() {
        return zWert;
    }

    public void setZWert(double zWert) {
        this.zWert = zWert;
    }

    @Override
    // Berechnung der Oberflaeche
    public double berechneFlaeche() {
        return 4 * Math.PI * this.getRadius() * this.getRadius();
    }

    @Override
    public double berechneVolumen() {
        return (4 * Math.PI * this.getRadius() * this.getRadius()
            * this.getRadius()) / 3;
    }
}

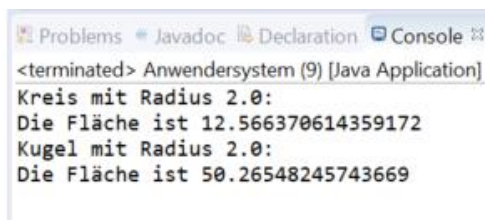
```

Klasse Anwendersystem

```
public class Anwendersystem {

    public static void gibFlaecheAus(
        Berechnungen[] geometrischeFiguren) {
        for(int i = 0; i < geometrischeFiguren.length; i++) {
            System.out.println(geometrischeFiguren[i]
                .getClass().getSimpleName()
                + " mit Radius "
                + geometrischeFiguren[i].berechneRadius()
                + ": ");
            System.out.println("Die Fläche ist "
                + geometrischeFiguren[i].berechneFlaeche());
        }
    }

    public static void main(String[] args) {
        Berechnungen[] geometrischeFiguren = new Berechnungen[2];
        geometrischeFiguren[0] = new Kreis(0, 0, 2);
        geometrischeFiguren[1] = new Kugel(0, 0, 0, 2);
        gibFlaecheAus(geometrischeFiguren);
    }
}
```



```
Problems Javadoc Declaration Console
<terminated> Anwendersystem (9) [Java Application]
Kreis mit Radius 2.0:
Die Fläche ist 12.566370614359172
Kugel mit Radius 2.0:
Die Fläche ist 50.26548245743669
```

Lösung

zu i) Die Klasse *Kreis* darf nicht das Interface *Berechnungen* implementieren, da eine Berechnung des Volumens keinen Sinn macht. Die Methode *gibFlaecheAus* der Klasse *Anwendersystem* hat einen Parameter vom Typ *Berechnungen*. Die Methoden aus *Berechnungen* braucht man nicht alle.

zu ii)

—Interface Berechnungen Interface Kreisberechnungen

```
public interface Kreisberechnungen {

    public double berechneRadius();
    public double berechneUmfang();
    public double berechneFlaeche();
}
```


Interface Volumenberechnung

```
public interface Volumenberechnung {  
  
    public double berechneVolumen();  
}
```

Klasse Kreis

```
public class Kreis implements Kreisberechnungen{  
    ...  
  
    @Override  
public double berechneVolumen() {  
    return 0;  
}  
}
```

Klasse Kugel

```
public class Kugel extends Kreis implements Kreisberechnungen,  
    Volumenberechnung{  
  
    ...  
}
```

Klasse Anwendersystem

```
public class Anwendersystem {  
  
    // ganz genau passt Kreisberechnungen nicht wegen der  
    // Berechnung des Umfangs  
    public static void gibFlaecheAus(  
        Kreisberechnungen[] geometrischeFiguren) {  
        for(int i = 0; i < geometrischeFiguren.length; i++) {  
            System.out.println(geometrischeFiguren[i]  
                .getClass().getSimpleName()  
                + " mit Radius "  
                + geometrischeFiguren[i].berechneRadius()  
                + ": ");  
            System.out.println("Die Fläche ist "  
                + geometrischeFiguren[i].berechneFlaeche());  
        }  
    }  
}
```

```

    public static void main(String[] args) {
        Kreisberechnungen [] geometrischeFiguren
            = new Kreisberechnungen[2];
        geometrischeFiguren[0] = new Kreis(0, 0, 2);
        geometrischeFiguren[1] = new Kugel(0, 0, 0, 2);
        gibFlaecheAus(geometrischeFiguren);
    }
}

```

e) Dependency Inversion Principle

Gegeben seien das Interface *Berechnungen* und die Klassen *Kreis*, *Kugel* und *Anwendersystem* zur Aufgabe zum Interface Segregation Principle.

Welche Klassen sind high level, welche low level? Was sind in dem Beispiel Abstraktionen? In wie weit wird das Dependency Inversion Principle erfüllt? Geben Sie für Erfüllung und auch Nichterfüllung Begründungen an.

Lösung

high level: Anwendersystem

low level: Kreis, Kugel

Abstraktion: Berechnungen

Das Dependency Inversion Principle wird in der Methode *gibFlaecheAus* der Klasse *Anwendersystem* erfüllt, da diese auf die Abstraktion *Berechnungen* zugreift, nicht direkt auf *Kreis* oder *Kugel*.

Das Dependency Inversion Principle wird in der main-Methode der Klasse *Anwendersystem* nicht erfüllt, da konkrete Objekte vom Typ *Kreis* oder *Kugel* kreiert werden.

6. Aufgabe: Clean Code

Gegeben ist eine Klasse *Produkt*.

```
public class Produkt {

    private int identnummer;
    private String produktname;
    private String preis;

    public Produkt(int identnummer, String produktname) {
        super();
        this.identnummer = identnummer;
        this.produktname = produktname;
    }

    public Produkt(int identnummer, String produktname,
        String preis) {
        super();
        this.identnummer = identnummer;
        this.produktname = produktname;
        this.preis = preis;
    }

    // get- und set-Methoden
    ...

    public float gibAttributeUndPreisBeiMengenrabattAus(
        int anzahl) {
        float ergebnis = this.preis;
        if(anzahl >= 10) {
            ergebnis = ergebnis * 0.9f;
        }
        gibAttributeAus();
        System.out.println(
            "Mengenrabatt bei " + anzahl + " Stück: "
            + ergebnis + " Euro");
        return ergebnis;
    }

    public void gibAttributeAus() {
        System.out.println(
            "Werte der Eigenschaften des Produkts");
        System.out.println("Identnummer: " + this.identnummer);
        System.out.println("Produktname: " + this.produktname);
        System.out.println("Preis:          " + this.preis
            + " Euro");
    }
}
```

- i) Geben Sie zu den Konstruktoren der Klasse *Produkt* den Quellcode zur Anwendung des Refaktorisierungsmusters *Chain Constructor* an.
- ii) Das *Integration Operation Segregation Principle (IOSP)*, welches zum Erreichen des roten Grades des Clean Code angewendet werden muss, wird in der Klasse *Produkt* nicht eingehalten. Geben Sie den Quellcode einer Möglichkeit zur Einhaltung des IOSP an.
- iii) Welche Tugend und welches Prinzip des roten Grades des Clean Code würden Sie verletzen, wenn Sie in dem vorherigen Aufgabenteil ii) dieser Aufgabe mehr als eine mögliche Änderung angeben? Geben Sie jeweils Begründungen an.
- iv) Wofür steht DRY und wieso sollte man DRY einhalten?

Lösung

zu i)

```
public Produkt(int identnummer, String produktname,
               String preis) {
    this(identnummer, produktname);
    this.preis = preis;
}
```

zu ii)

anstatt `public float gibAttributeUndPreisBeiMengenrabattAus(int anzahl)`

```
public void gibAttributeUndPreisBeiMengenrabattAus(
    int anzahl) {
    float reduzierterPreis
        = berechnePreisBeiMengenrabatt(anzahl);
    gibAttributeAus();
    gibPreisBeiMengenRabattAus(anzahl, reduzierterPreis);
}
```

```
private float berechnePreisBeiMengenrabatt(int anzahl) {
    float ergebnis = this.preis;
    if(anzahl >= 10) {
        ergebnis = ergebnis * 0.9f;
    }
    return ergebnis;
}
```

```
public void gibPreisBeiMengenRabattAus(int anzahl,
    float preisBeiMengenrabatt) {
    System.out.println("Mengenrabatt bei " + anzahl
        + " Stück: " + preisBeiMengenrabatt + " Euro");
}
```

zu iii)

Tugend: Tue nur das Nötigste.

Es ist die Angabe von nur einer Möglichkeit gewünscht, daher ist die Angabe einer weiteren Möglichkeit Zeitverschwendung.

Prinzip des roten Grades: Vorsicht vor Optimierungen!

Die Angabe von mehr als einer Möglichkeit ist zusätzlicher Aufwand.

zu iv)

DRY steht für Dont repeat yourself. Jede Doppelung von Code oder auch nur Handgriffen leistet Inkonsistenzen und Fehlern Vorschub.