Übung 11: Abstrakte Basisklassen

In der letzten Übung hatten wir den Unterschied zwischen Basisklassen (`extends`) und Interfaces (`implements`) erarbeitet. Da die unterschiedlichen Bremsen (Scheiben-, Trommel- und Cantileverbremsen) zwar mechanisch sehr unterschiedlich aufgebaut sind, aber alle im Prinzip bremsen können, wählten wir hier ein Interface. Das hatte den Nachteil, dass nun die einzelnen Bremsvarianten nun eben die Bremse implementieren, aber nicht erweitern. Möchte man nun aber z.B. Attribute wie `hersteller` für alle Bremsen vorschreiben, so gibt es zwei Möglichkeiten:

a) Umwandlung des Interfaces in eine Klasse mit den entsprechenden Attributen

```
public class Bremse {
    private String hersteller;
    public Bremse(String h) {
        this.hersteller = h;
    }
    public void bremsen() {
    }
}

public class Scheibenbremse extends Bremse{
    public Scheibenbremse(String h) {
        super(h);
    }
}
```

b) Erweiterung des Interfaces um entsprechende Getter/Settermethoden

```
public interface Bremse {
    void bremsen();
    String getHersteller();
    void setHersteller(String h);
}

public class Scheibenbremse implements Bremse{
    @Override
    public void bremsen() {}

    @Override
    public String getHersteller() {
        return null;
    }

    @Override
    public void setHersteller(String h) {}
}
```

Die erste Version a hat den Nachteil, dass man nun in `Bremse` die Methode `bremsen` implementieren muss, ohne aber zu wissen, welche Bremsenart es ist -- sofern das überhaupt semantisch sinnvoll ist! Die zweite Version (B) hat den Nachteil, dass man nun in jeder Realisierung von `Bremse` eine separate Variable für den `hersteller` anlegen muss.

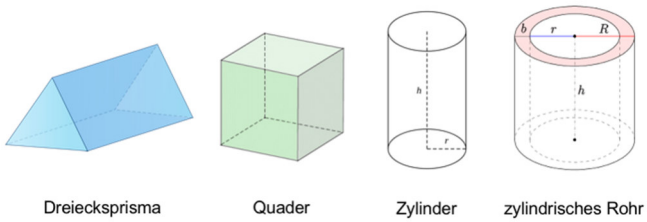
Die Lösung ist eine abstrakte Basisklasse (`abstract class`), in der die Methode `bremsen` als `abstract` markiert ist, und nicht implementiert wird (kein `{ ... }`, sondern `;`)

Aufgabe 1: Die abstrakte Basisklasse Bremse

Gegeben ist ein Teil der Musterlösung der letzten Aufgabe, in der `Bremse` ein Interface ist.

- Zeichnen Sie ein UML Diagramm. Modellieren Sie dazu die Bremse als abstrakte Basisklasse, welche `Hersteller` (`String`) und `Seriennummer` (`String`) als private Attribute modelliert und die Methoden `void bremsen()` und `void brauchtService()` vorschreibt.
- Passen Sie die vorgegebene Implementierung so an, dass sie dem UML Diagramm entspricht. Verwenden Sie Fantasiewerte für Hersteller und Seriennummer.
- Stellen Sie sicher, dass alle Tests in `BremseTests` erfolgreich laufen.

Aufgabe 2: dreidimensionale Formen



Die oben abgebildeten Formen Dreiecksprisma, Quader, Zylinder und zylindrisches Rohr sind einfache Volumina, welche sich aus der Grundfläche mal der Höhe berechnen. Modellieren Sie die vier Formen unter der Verwendung einer gemeinsamen Basisklasse.

Im der Angabe sind bereits die Klassendateien `Volumen.java`, `DreiecksPrisma.java`, `Quader.java`, `Zylinder.java` und `ZylindrischesRohr.java` angelegt.

- Zeichnen Sie ein UML Diagramm, das die Klassen in geeignete Vererbungsbeziehungen setzt.
- Ein Volumen soll mit der Methode `double volumen()` das eigentliche Volumen (also Grundfläche mal Höhe) berechnen, Codeduplikate aber vermieden werden.
- Verdeutlichen Sie sich, welche Werte zur Berechnung des Volumens nötig sind, und in welcher Klasse diese bestimmt werden können.
- Vervollständigen Sie die vorgegebenen Klassenrümpfe entsprechend Ihrem Diagramm, und implementieren Sie die Methoden.
- Stellen Sie sicher Tests in `VolumenTests` erfolgreich sind.

Hinweise

- Rechnen Sie durchwegs mit Gleitkommazahlen.
- Die Dreiecksfläche kann mit dem Satz von Heron aus den drei Seitenlängen berechnet werden.
- Der Querschnitt eines Rohres kann mit dem Innen- und Aussenradius berechnet werden.
- Es ist eine abstrakte Basisklasse erforderlich, welche sowohl konkrete als auch abstrakte Methoden hat.
- Mit genug Vorüberlegungen kann Top-Down gearbeitet werden; unter Umständen fällt Ihnen aber Bottom-Up einfacher. Implementieren Sie hierzu zunächst die Logik für zwei Klassen, um dann Gemeinsamkeiten zu isolieren.