Technische Hochschule Rosenheim
Technical University of Applied Sciences

Objektorientierte Programmierung
Kapitel 1 – Professionelle Softwareentwicklung

Prof. Dr. Kai Höfig

Was bisher geschah…

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- In der Vorlesung Grundlagen der Programmierung wurden folgende Inhalte behandelt:
  - Grundlegende Sprachkonzepte
  - Kontrollstrukturen
  - Objektorientierung
  - Klassen
  - Characters und Strings
  - Arrays und Container
  - Packages
  - Vererbung
  - Ausnahmen

Professionelle Softwareentwicklung

Technische Hochschule Rosenheim
Technical University of Applied Sciences

Die wesentlichen Elemente professioneller Softwareentwicklung sind

1. Modellierung des Problems, im Idealfall vor dem Beginn der Implementierung. Die Modellierung basiert auf der Problemstellung und der daraus erfolgten Spezifikation.

2. Implementierung der Funktionalität.

3. Implementierung von Tests, um die Funktionalität zu testen.

4. Versionierung

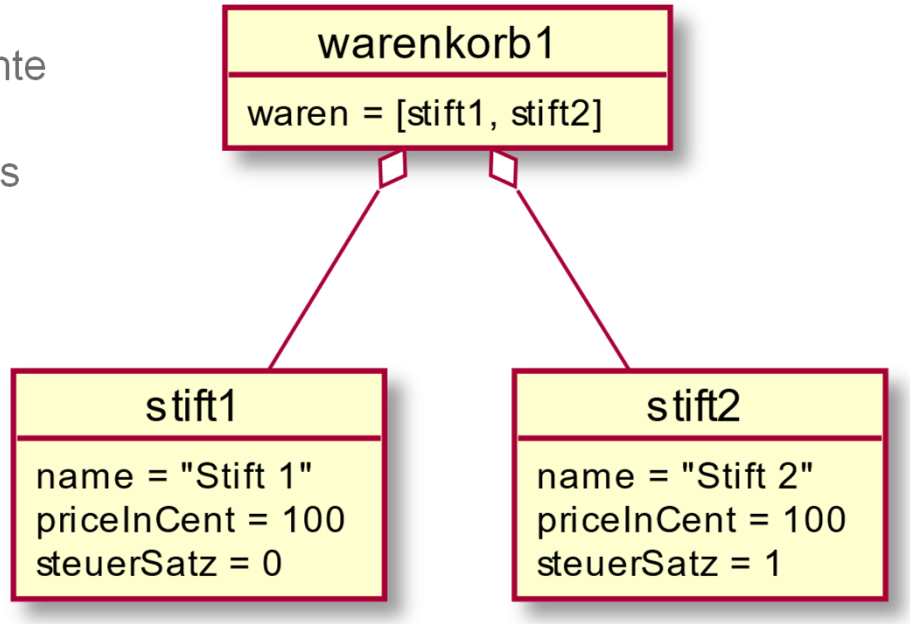
Modellierung eines Beispiels mit UML

Wiederholung Objektdiagramm

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Wir beginnen mit einem kleinen Beispiel. Für ein Webshopsystem brauchen wir zunächst *Waren*, welche einen Namen, Preis und Steuersatz (z.B. 19% oder ermäßigt 7%) haben. Waren werden dann einem *Warenkorb* hinzugefügt.

- Ein entsprechendes ***Objektdiagramm*** könnte wie folgt aussehen, Wobei der Pfeil mit der leeren Raute eine Aggregation darstellt: Das Objekt `warenkorb1` enthält die Objekte `stift1` und `stift2`.



- Hinweis: Eine **Komposition** wird durch einen Pfeil mit ausgefüllter Raute dargestellt; im Unterschied zur Aggregation zeigt diese an, dass ein Objekt nicht ohne seine Teile bestehen kann. Der Warenkorb würde in diesem Fall aus den Stiften bestehen, statt sie nur zu enthalten. Diese Semantik ist *common sense*

Modellierung eines Beispiels mit UML

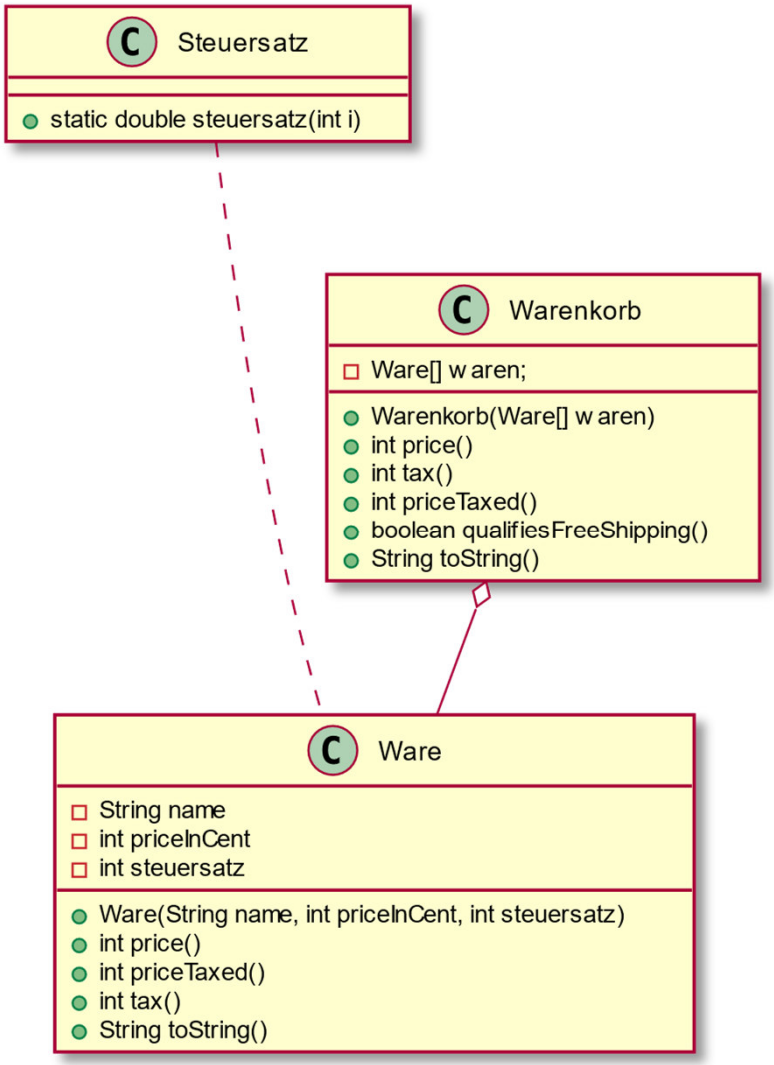
Wiederholung Klassendiagramm

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Möchte man diese kleine Modellwelt in einer objektorientierten Sprache wie Java implementieren, so abstrahiert man das Diagramm in ein *Klassendiagramm*.

- Neben den Assoziationen sind hier auch die Sichtbarkeiten modelliert: das rote Viereck steht für **private**, also von aussen nicht zugreifbar, der grüne Punkt für **public**. Man spezifiziert Attribute idR. privat, Methoden aber öffentlich. Dieses Prinzip nennt man auch Kapselung.

- Zusätzlich zu den beiden Klassen Warenkorb und Ware ist hier nun auch der Steuersatz modelliert. Dieser ist kein Objekt im strengen Sinne ist, da hier nur der tatsächliche Steuersatz konfiguriert wird. Entsprechend ist dieser mit der Ware **assoziiert**, dargestellt durch eine gestrichelte Linie. Die Ware verwendet den Steuersatz, um den Preis mit Steuer über `priceTaxed()` zu berechnen.



Implementierung in Java
Steuersatz

Technische Hochschule Rosenheim
Technical University of Applied Sciences

Der Steuersatz könnte nun wie folgt implementiert werden:

```
public class Steuersatz {
    public static double steuersatz(int i) {
        switch (i) {
            case 0: return 0.19;
            case 1: return 0.07;
            default: throw new IllegalArgumentException(i + " ist kein gültiger Steuersatz");
        }
    }
}
```

Es wird also der normale (0) und ermäßigte (1) Steuersatz codiert, und bei anderen Anfragen eine ungeprüfte Ausnahme vom Typ `IllegalArgumentException` geworfen (realisiert durch die default Anweisung).

Implementierung in Java

Warenkorb

Technische Hochschule Rosenheim
Technical University of Applied Sciences

Die Ware könnte nun wie folgt implementiert werden:

Die Sichtbarkeiten werden durch die Schlüsselwörter `private` und `public` gesetzt. Gibt es eine Verschattung von Variablen, wie z.B. das Konstruktorargument `name` und das Attribut `name`, so kann die Selbstreferenz `this` verwendet werden, um an das verschattete Attribut zu gelangen.

```
public class Ware {
    private String name;
    private int priceInCent;
    private int steuersatz;

    public Ware(String name, int priceInCent, int steuersatz) {
        this.name = name;
        this.priceInCent = priceInCent;
        this.steuersatz = steuersatz;
    }

    public int price() {
        return priceInCent;
    }

    public int priceTaxed() {
        return priceInCent + tax();
    }

    public int tax() {
        return (int) Math.round(Steuersatz.steuersatz(steuersatz) * priceInCent);
    }
}
```

Implementierung in Java
Warenkorb

Technische Hochschule Rosenheim
Technical University of Applied Sciences

Der Warenkorb könnte nun wie folgt implementiert werden:

```
public class Warenkorb {
    private Ware[] waren;

    public Warenkorb(Ware[] waren) {
        this.waren = waren;
    }

    public int price() {
        int p = 0;
        for (Ware w : waren)
            if (w != null)
                p += w.price();
        return p;
    }

    public boolean qualifiesFreeShipping() {
        return price() >= 300;
    }

    /* ... */
}
```

Hierbei sehen wir, dass die Waren als Array realisiert wurden, über dessen Inhalt mit einer for Schleife iteriert werden kann.

Bei Bedarf zu Wiederholen

- Klassen, Objekte und Arrays anlegen
- Bedingungen mit if und else
- Iteration mit for und while
- Geprüfte und ungeprüfte Ausnahmen, try-catch-throw

Testen mit JUnit (5)

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Zusätzlich zur Implementierung der Klassen und Methoden erfordert professionelle Softwareentwicklung das Testen auf Korrektheit, wobei testen i.A. keinen Korrektheitsbeweis liefern kann. Es gibt verschiedene Möglichkeiten und Toolkits um dieses zu vereinfachen bzw. zu automatisieren, in dieser Veranstaltung verwenden wir JUnit 5.

- Dabei werden die Testdateien *oft* in gesonderten Verzeichnissen geführt, da diese nicht zum Kunden ausgeliefert werden. So wird der Anwendungscode i.d.R. unter src/main/java abgelegt, Testcode aber unter src/test/java. Das machen wir in dem Projekt zur Vorlesung zur besseren Übersicht allerdings nicht.

- Der Testtreiber (z.B. IntelliJ oder Gradle) führt nun alle mit `@Test` annotierten Methoden als einzelne Testcases durch.

- Man spricht bei einfachen Tests, welche eine Klasse oder Methode isolieren (oder einfaches Zusammenspiel betrachten) von **Unittests**. Sie stellen "im Kleinen" sicher, dass die Komponenten das tun, was sie sollen.

- Wird die Software (oder wesentliche Teile davon) im Gesamten gestestet, so spricht man von *Integrationstests* (**Integration Tests**). Diese stellen nun sicher, dass die einzelnen bereits getesteten Komponenten auch korrekt ineinandergreifen, sodass die Software das gewünschte Ergebnis liefert.

Beispiel Junit5 Testfall

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Ein Test in JUnit ist eine Klasse mit speziell annotierten Methoden, hier ein Beispiel:

- JUnit liefert Hilfsmethoden, welche das Testen erleichtern. Da ein Großteil von Tests darauf beruht, dass Software bei bestimmter Eingabe eine bestimmte Ausgabe produziert, gibt es in der Klasse `Assertions` - eine Liste an Hilfsmethoden, um erwartetes Verhalten zu prüfen.

```
class SteuersatzTest {
    @Test
    void testSteuersatz() {
        Assertions.assertEquals(0.19, Steuersatz.steuersatz(0));
        Assertions.assertEquals(0.07, Steuersatz.steuersatz(1));

        Assertions.assertThrows(IllegalArgumentException.class, new Executable() {
            @Override
            public void execute() throws Throwable {
                Steuersatz.steuersatz(-1);
            }
        });

        Assertions.assertThrows(IllegalArgumentException.class, new Executable() {
            @Override
            public void execute() throws Throwable {
                Steuersatz.steuersatz(2);
            }
        });
    }
}
```

- `Assertions.assertEquals(0.19, Steuersatz.steuersatz(0))` prüft also, dass der Rückgabewert von `Steuersatz.steuersatz(0)` gleich 0.19 ist.

Assertions

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Der folgende, etwas unhandliche Codeausschnitt prüft, ob bei Argumenten anders als 0 oder 1 eine Exception geworfen wird:

```
Assertions.assertThrows(IllegalArgumentException.class, new Executable() {
    @Override
    public void execute() throws Throwable {
        Steuersatz.steuersatz(2);
    }
});
```

- Dies kann übrigens seit Java 8 verkürzt als *Lambdaausdruck* geschrieben werden:

```
Assertions.assertThrows(IllegalArgumentException.class,
        () -> Steuersatz.steuersatz(-2));
```

Übersicht über die Klasse Assertions
https://junit.org/junit5/docs/5.0.1/api/org/junit/jupiter/api/Assertions.html

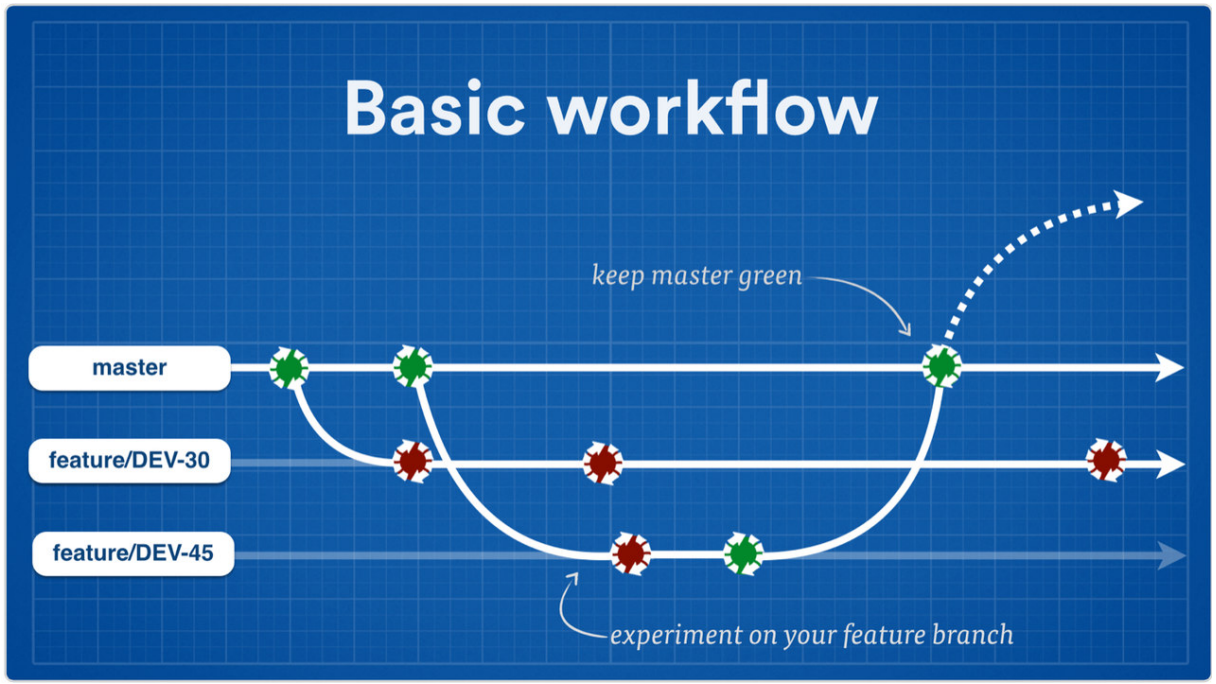
Kurze Einführung zu Java Lambda Expressions
https://www.youtube.com/watch?v=GxZWMgpMuLs

Versionierung mit Git

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Oft arbeiten mehrere Entwickler an einem Projekt und man sollte zu Sicherungs- und Dokumentationszwecken in regelmäßigen Abständen Sicherungspunkte (snapshots) anlegen.

- Die Versionierungssoftware Git hilft hierbei. Vereinfacht gesehen soll es einen Hauptbestand des Quellcodes geben (*master*), und neue Features sollen dann jeweils in separaten *Branches* implementiert werden. Ist ein Feature in einem Branch fertig gestellt, so wird es in master durch einen *Merge* eingebracht:



Eine Anleitung für Git finden Sie unter
http://rogerdudler.github.io/git-guide/

Eine Anleitung für die Integration von Git (nicht GitHub!) in IntelliJ gibt es zum Beispiel unter
https://www.jetbrains.com/help/idea/using-git-integration.html