title: 01 Professionelle Softwareentwicklung

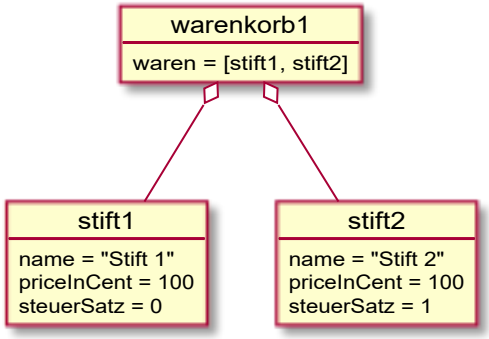
permalink: /01-professionelle-softwareentwicklung/

Professionelle Softwareentwicklung

Modellierung mit UML

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.

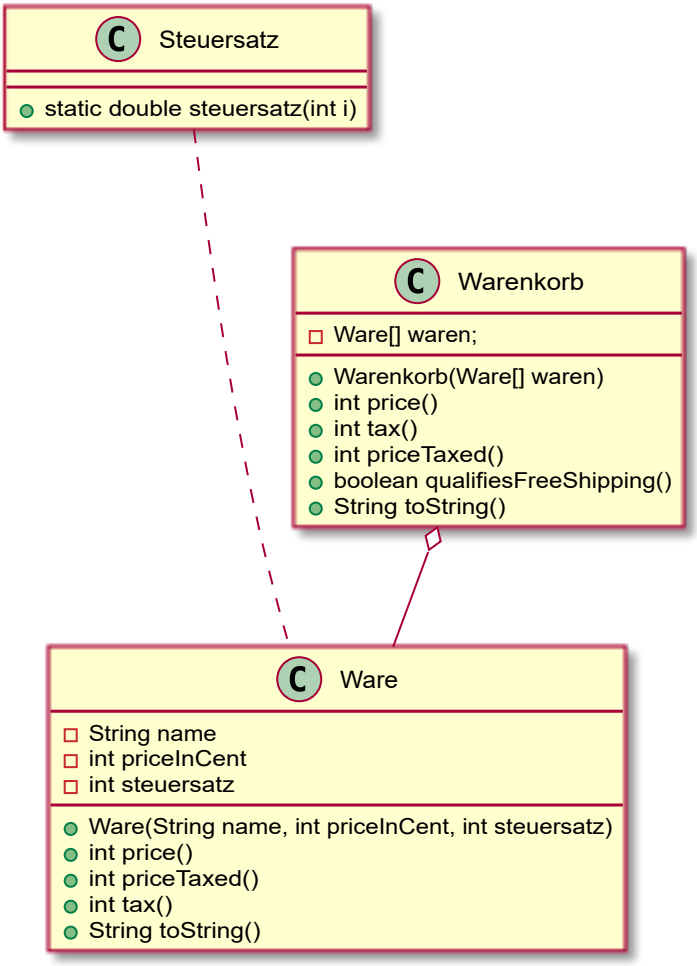
Ein entsprechendes *Objektdiagramm* könnte wie folgt aussehen:



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.

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`. Wie Sie sehen können spezifiziert man also Attribute in der Regel 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 (`priceTaxed()`) zu berechnen.

Bei Bedarf zu Wiederholen

- Objekt- und Klassendiagramme
- UML

Implementierung in Java

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).

Die Ware könnte nun wie folgt implementiert werden:

```
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);
        }
}
```

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.

Der Warenkorb könnte 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)

Zusätzlich zur Implementierung der Klassen und Methoden erfordert professionelle Softwareentwicklung das Testen auf Korrektheit. 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`.

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

```
public class SteuersatzTests {
        @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);
                        }
                });
        }
}
```

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

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.

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

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(-1);
        }
});
```

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

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

Begrifflichkeit

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.

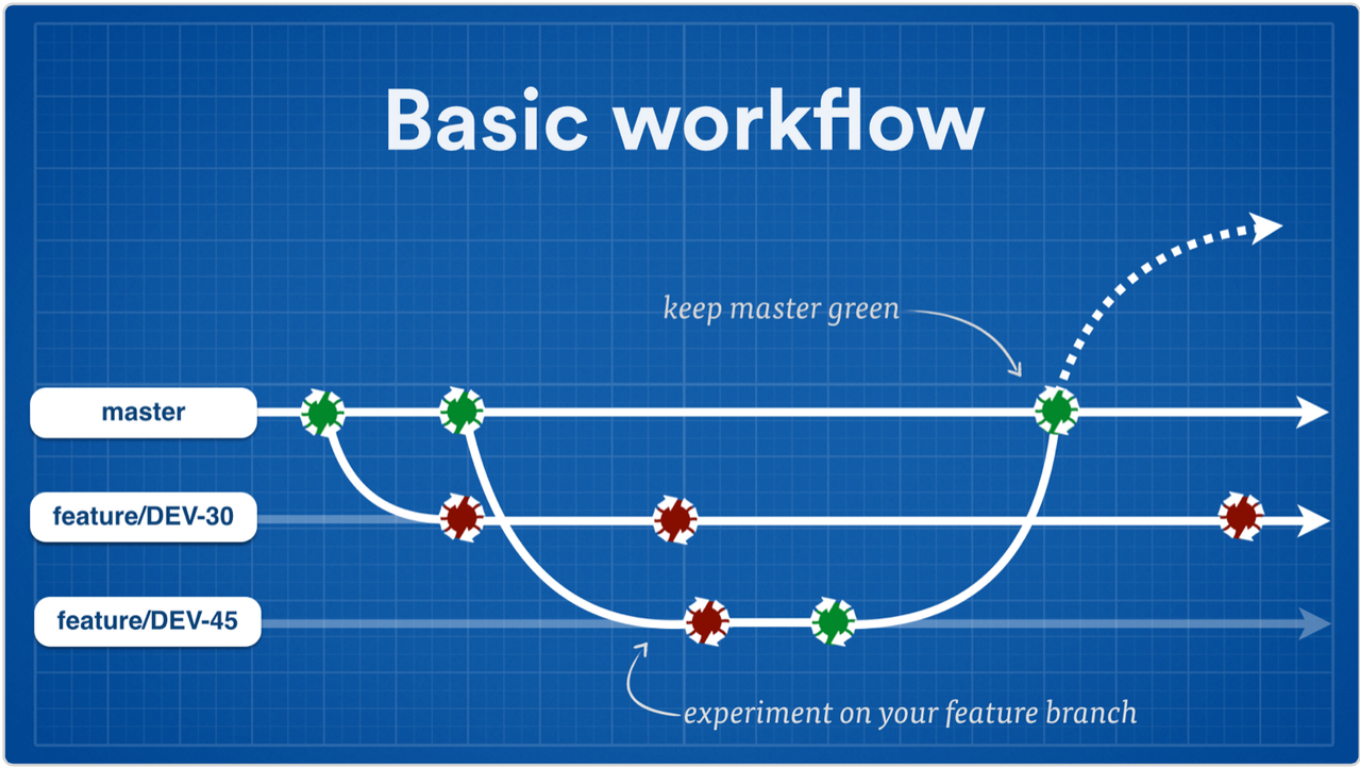
Versionierung mit Git

Die wesentlichen Elemente professioneller Softwareentwicklung sind also

- Modellierung des Problems, im Idealfall *vor* dem Beginn der Implementierung. Die Modellierung basiert auf der Problemstellung und der daraus erfolgten Spezifikation.
- Implementierung der Funktionalität.
- Implementierung von Tests, um die Funktionalität zu testen.

Nun ist es aber so, dass oft mehrere Entwickler an einem Projekt arbeiten und weiterhin man zu Sicherungs- und Dokumentationszwecken in regelmäßigen Abständen Sicherungspunkte (snapshots) anlegen sollte.

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:



Die erste Übung (https://inf-git.fh-rosenheim.de/wif-oop-ss19) beinhaltet ein kleines Tutorial zu Git, bitte machen Sie sich damit vertraut.

■