Testen

Zentrale Konzepte:

- Modultests
- Regressionstests
- Testen mit JUnit

Fehler in einem Programm

- Frühe Fehler sind in der Regel Syntaxfehler.
  - Diese kann der Compiler erkennen.
- Spätere Fehler sind in der Regel logische Fehler.
  - Hier kann der Compiler nicht helfen.
  - Auch als Bugs bekannt.
- Einige logische Fehler zeigen sich nicht direkt.
  - Kommerzielle Software ist selten fehlerfrei.

Strategien

- **Fehlervermeidung:** Am besten so programmieren, dass die Fehlerwahrscheinlichkeit minimiert wird.
- Durch den Einsatz von Softwaretechniken wie Kapselung kann man die Fehlerwahrscheinlichkeit verringern.
- Durch die Verwendung von Softwarepraktiken wie Modularisierung und Dokumentation kann man die Chancen auf Fehlererkennung erhöhen.
- **Testen**: dient der Überprüfung, ob ein Stück Software (oder Methode oder Klasse) das gewünschte Verhalten zeigt.
- **Fehlerbeseitigung** (Debugging): bezeichnet die Suche nach der Ursache eines Fehlers und deren Beseitigung.

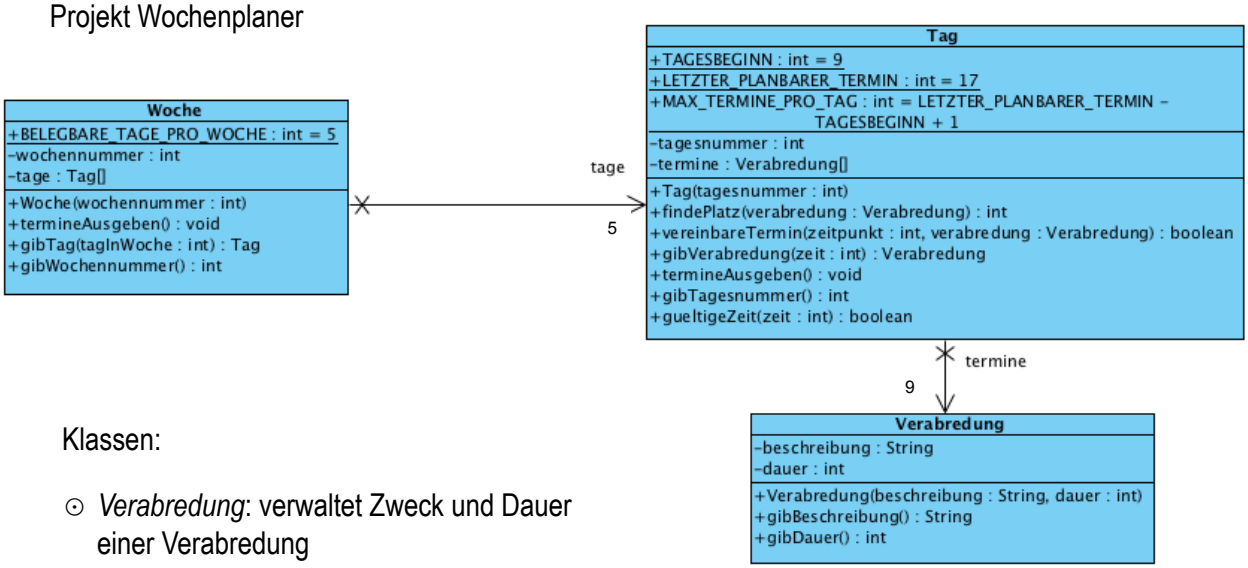
Techniken zum Testen

- Modultests
- Tests automatisieren
- JUnit

Modultests (Unit Tests)

- Modultests testen einzelne Bestandteile eines Programms:
  - Gruppe von Klassen, die gemeinsam eine Aufgabe lösen
  - Eine einzelne Klasse
  - Eine einzelne Methode
- Modultests können durchgeführt werden, sobald der zu testende Bestandteil programmiert ist. D.h. Modultests können während der Entwicklung eines Programms eingestreut werden um das Verhalten einzelner Einheiten zu testen.
- Vorteil: Je früher Fehler gefunden und behoben werden, desto kürzer wird die Entwicklungszeit
- Vorteil: Man erstellt so eine Menge von **Testfällen**, die man auch nach Änderungen/ Erweiterungen am Programm einsetzen kann, um Folgefehler aufzuspüren.

Projekt Wochenplaner



Klassen:

- *Verabredung*: verwaltet Zweck und Dauer einer Verabredung
- *Tag*: bezieht sich auf einen bestimmten Tag im Jahr und verwaltet die Verabredungen (Termine) des Tages.
- *Woche*: bezieht sich auf eine bestimmte Woche im Jahr und verwaltet die Tage von Montag bis Freitag.

Planen von Tests: Herangehensweise

- Bei Klassen mit Sammlungen muss geprüft werden, ob sie sich in folgenden Konstellationen korrekt verhält:
  - Leere Sammlung
  - Volle Sammlung (bei Sammlungen mit begrenzter Kapazität)
- Bei Sammlungen müssen die Grenzbereiche getestet werden, d.h.
  - erstes Element
  - letztes Element
- Positives Testen: Testen von Fällen, die funktionieren sollten
- Negatives Testen: Testen von Fällen, die nicht funktionieren sollten

Testfälle: Testen der Klasse *Tag*

- Hat das Attribut *termine* genug Platz, um die benötigte Menge an Eintragungen aufzunehmen?
  - Prüfen durch Ausgabe der Array-Länge
- Aktualisiert die Methode *vereinbareTermin* das Attribut *termine* korrekt, wenn ein Termin vereinbart wurde?
  - Prüfen mit einstündiger Verabredung um 9, 13 und 17 Uhr (Grenzbereiche testen)
  - Prüfen mit zusätzlicher einstündiger Verabredung um 14 Uhr (positiver Test)
  - Prüfen mit zusätzlicher zweistündiger Verabredung um 16 Uhr (negativer Test)
- Gibt die Methode *termineAusgeben* korrekt die Liste der vereinbarten Termine aus?
  - Prüfen der Ausgabe (leere Sammlung, volle Sammlung)
- Liefert die Methode *findePlatz* das korrekte Ergebnis, wenn Platz für eine neue Verabredung gebraucht wird?

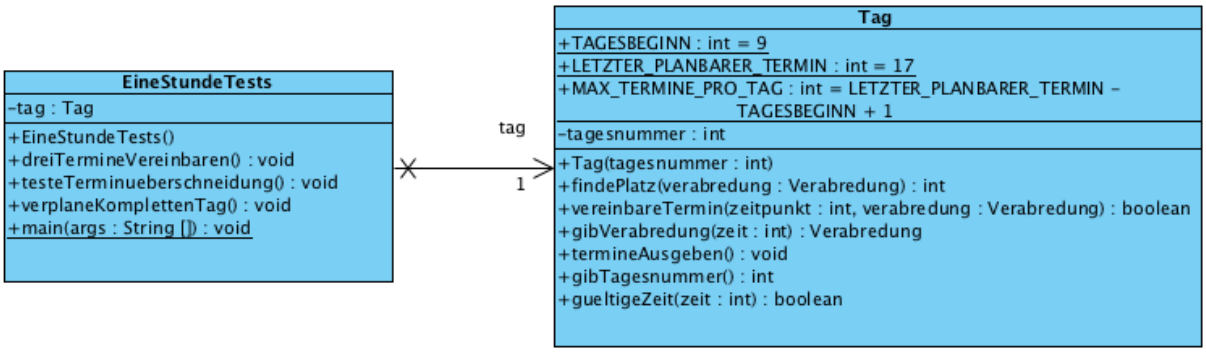
- Die Korrektur eines Fehlers kann an einer anderen Stelle einen neuen Fehler einschleusen.

  => Nach jeder Korrektur muss das Programm erneut getestet werden

- **Regressionstests**
  - ... sind Tests, die bereits erfolgreich durchgeführt wurden
  - ... müssen nach jeder Änderung wiederholt werden um auf Nachfolgefehler zu prüfen
  - ... sollten automatisiert werden

    => Programm übernimmt diese Arbeit, d.h. ein Programm, das die Tests ausführt.

Klasse *EineStundeTests :* enthält Regressionstest für die Klasse Tag



Klasse *EineStundeTests*: enthält Methoden, die einzelne Tests auf einem Tag-Objekt ausführen

- *dreiTermineVereinbaren()*
- *testeTerminüberschneidung()*
- *verplaneKomplettenTag()*

JUnit

- ***JUnit*** ist ein Java-Test-Rahmenwerk (Test-*Framework*) für automatisierte Regressionstests.
- **Testklassen** definieren Testfälle für bestimmte Zielklassen einer Anwendung.
- **Testfälle** sind Methoden, die Tests auf den Zielklassen ausführen.
- **Zusicherungen** werden verwendet, um anzugeben, welche Ergebnisse die Methodenaufrufe haben sollen. Damit kann automatisiert geprüft werden, ob der Test erfolgreich war.
- **Testgerüste** werden verwendet, um das Wiederholen von Tests zu unterstützen, indem ein Satz von Testobjekten erstellt wird.

JUnit Testklasse *TagTest*

```
public class TagTest
{
    public TagTest(){}

    ...

    /**
     * Ueberpruefe, dass Doppelbelegungen nicht zulaessig sind.
     */
    @Test
    public void testDoppelbelegung()
    {
        Tag tag1 = new Tag(1);
        Verabredung termin1 = new Verabredung("Java Vorlesung", 1);
        Verabredung termin2 = new Verabredung("Fehler", 1);
        assertTrue(tag1.vereinbareTermin(9, termin1));
        assertFalse(tag1.vereinbareTermin(9, termin2));
    }

}
```

Standardkonstruktor wird benötigt.

Annotation @Test definiert Testmethode.

Testfälle

- Eine *JUnit* Testklasse enthält Methoden, die sog. Testfälle durchführen. Eine solche Methode muss mit der Annotation **@Test** markiert sein, damit *JUnit* die Methode als Testfall erkennen kann und diese automatisch ausführt.

```
@Test
public void doppelbelegung()
{
        Tag tag1 = new Tag(1);
        Verabredung termin1 = new Verabredung("Java Vorlesung", 1);
        Verabredung termin2 = new Verabredung("Fehler", 1);

        assertTrue(tag1.vereinbareTermin(9, termin1));
        assertFalse(tag1.vereinbareTermin(9, termin2));
}
```


Zusicherungen

- Eine **Zusicherung** ist ein boolescher Ausdruck für eine Bedingung, von der wir erwarten, dass sie wahr ist. Wenn sie nicht wahr ist, dann sagen wir, dass die Zusicherung fehlgeschlagen ist. Dies weist auf einen Fehler im Programm hin, der vom *JUnit*-Framework ausgewertet und angezeigt wird.

- In *JUnit* können solche Zusicherungen mit Hilfe einer assert-Anweisung formuliert werden. Z.B. mit

Zusicherung	Ist wahr, wenn ...
assertTrue(String message, boolean condition)	die Bedingung wahr ist
assertFalse(boolean condition)	die Bedingung falsch ist
assertNotNull(Object object)	der Wert von *object* nicht null ist
assertNull(Object object)	der Wert von *object* null ist
assertEquals(Object expected, Object actual)	die Objekte gleich sind (prüft mit equals)
assertEquals(long expected, long actual)	die Werte gleich sind (int oder long)
assertEquals(double expected, double actual)	die Werte gleich sind (float oder double)

Statischer Import

- Alle assert-Methoden sind statische Methoden der JUnit Klasse *Assert*.
- Normalerweise muss man bei der Verwendung von statischen Methoden den Namen der Klasse mit angeben

```
Assert.assertTrue(tag1.vereinbareTermin(9, termin1));
```

- Um diese Methoden in eigenen Testklassen zu verwenden zu können ohne den Namen der Klasse davor zu schreiben, müssen die Methoden per statischem Import importiert werden.

```
import static org.junit.Assert.assertTrue;
```

Import von Annotationen

- Die verwendeten Annotationen müssen im Quellcode ebenfalls importiert werden.

```
import org.junit.Test;
```


Testgerüst

- Ein **Testgerüst** besteht aus einer Menge von Objekten in einem definierten Zustand, die das Testszenario für einen Modultest liefern.
- Die Objekte des Testgerüstes müssen in Attributen der Testklasse gespeichert werden.
- In *JUnit* können solche Testgerüste in einer Methode aufgebaut werden, die mit der Annotation ***@Before*** markiert ist. Z.B.

```
@Before
public void setUp() {
        tag1 = new Tag(1);
        termin1 = new Verabredung("Java Vorlesung", 1);
        termin2 = new Verabredung("Java Uebung", 1);
        termin3 = new Verabredung("John treffen", 1);
}
```

Attribute der Testklasse

- Die Methode *mit @Before markierte Methode* wird von *JUnit* vor jeder Testmethode aufgerufen.

Testen auf geworfene Exceptions

- Zu dem Test einer Methode kann auch gehören, dass geprüft wird, ob in einer bestimmten Situation eine dafür vorgesehene Exception geworfen wird.
- Die erwartete Exception kann man als Parameter der Annotation **@Test** hinzufügen, z.B.

Erwartete Exception

```
@Test(expected=IndexOutOfBoundsException.class)

public void outOfBounds() {
        ArrayList<String> list = new ArrayList<String>();
        String text = list.get(1);
}
```

- Die zugehörige Testmethode schlägt fehlt, wenn keine Exception bzw. eine andere Exception geworfen wird.

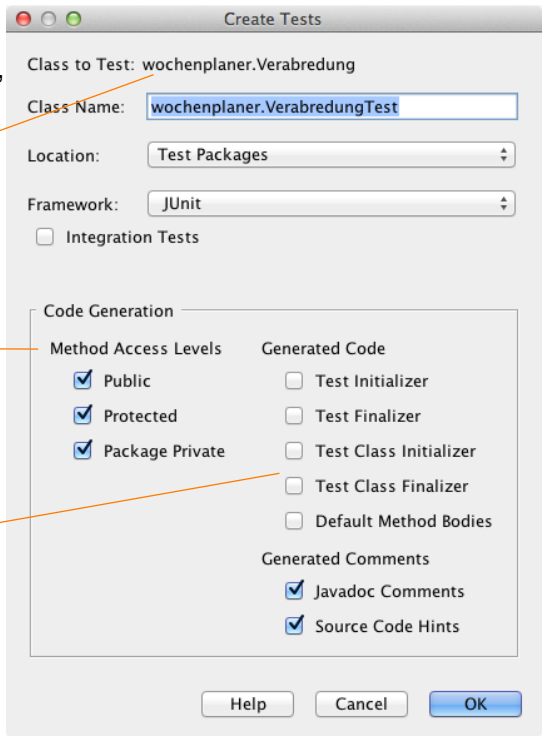
Nutzen von *JUnit* in *Netbeans*

- Testklasse erstellen: Zu testende Klasse wählen, *Tools->Create/Update Tests*

Klasse, die getestet werden soll

Autom. Erstellen von leeren Testmethoden

Autom. Erstellen von zusätzlichen Methoden



Nutzen von *JUnit* in *Netbeans*

- ⊙ Testklasse ausführen: Zu testende Klasse wählen, im Kontextmenü: *Test File*