universität innsbruck

Unit Tests

Programmiermethodik

Lukas Kaltenbrunner, Simon Priller
Universität Innsbruck

Testen

Testen (1)

- Testen ist der Vorgang, ein Programm oder einen Teil davon mit der Absicht auszuführen, möglicherweise enthaltene Fehler zu finden.
- Für die Bestimmung des Testergebnisses werden die spezifizierten Anforderungen (Soll) mit den gelieferten Ergebnissen (Ist) verglichen.
  - Pass: Gelieferte Ergebnisse stimmen mit den spezifizierten Anforderungen überein.
  - Fail: Gelieferte Ergebnisse stimmen nicht mit den spezifizierten Anforderungen überein.
  - Error: Während der Ausführung des Tests trat ein unerwarteter Fehler auf.
- Für das Testen ist es erforderlich, dass die Anforderungen bekannt sind.
- Testen dient der Sicherstellung von Softwarequalität.
- Debugging != Testen

Testen (2)

- Auswahl Testfälle immer abhängig von der zu testenden Anwendung.
- Um die Anzahl der Tests möglichst klein zu halten, werden jene Tests gewählt, die das Programm in kritische Situationen bringen.
  - In diesen Situationen sind Fehler am ehesten zu erwarten.
- Zwei Vorgehensweisen bei der Konstruktion eines Testfalls
  - Black-Box-Test
  - White-Box-Test

Black-Box-Test

- Nur die Anforderungen des Programms werden berücksichtigt.
  - Die Schnittstelle gibt die Testfälle vor.
- Der Source-Code spielt keine Rolle, das Programm wird als Black-Box betrachtet.
- Änderungen am Quelltext (nicht Schnittstelle) erfordern keine neue Implementierung der Tests.
- Black-Box-Testtechniken sind beispielsweise Äquivalenzklassentests, Grenzwertanalyse, kombinatorisches Testen.

White-Box-Test

- Test orientiert sich am Quelltext des Programms.
- Änderungen am Quelltext erfordern teilweise neue Tests bzw. alte Tests werden überflüssig.
- White-Box-Testtechniken können in kontrollflussorientierte und datenflussorientierte Überdeckungskriterien unterteilt werden.
  - Kontrollflussorientierte Tests: Mit den Testfällen sollen – je nach Überdeckungskriterium – die einzelnen Anweisungen, Zweige, Bedingungen oder Pfade explizit ausgetestet werden.
  - Datenflussorientierte Tests: Zugriffe auf Variablen sind maßgeblich für die Testerstellung

Anweisungsüberdeckung

- Anweisungsüberdeckung ist ein kontrollflussorientiertes White-Box-Überdeckungskriterium.
- Dabei sollen alle Anweisungen mindestens einmal ausgeführt werden.

```
public static int hammingDistance(final String first, final String second) {
    if (first == null || second == null) {
        throw new IllegalArgumentException("error description...");
    }
    if (first.length() != second.length()) {
        throw new IllegalArgumentException("error description...");
    }
    int distance = 0;
    for (int i = 0; i < first.length(); ++i) {
        if (first.charAt(i) != second.charAt(i)) {
            ++distance;
        }
    }
    return distance;
}
```

- Testfälle:
  - `hammingDistance(null, null);`
  - `hammingDistance("len", "length");`
  - `hammingDistance("test", "task");`

Testarten (Überblick)

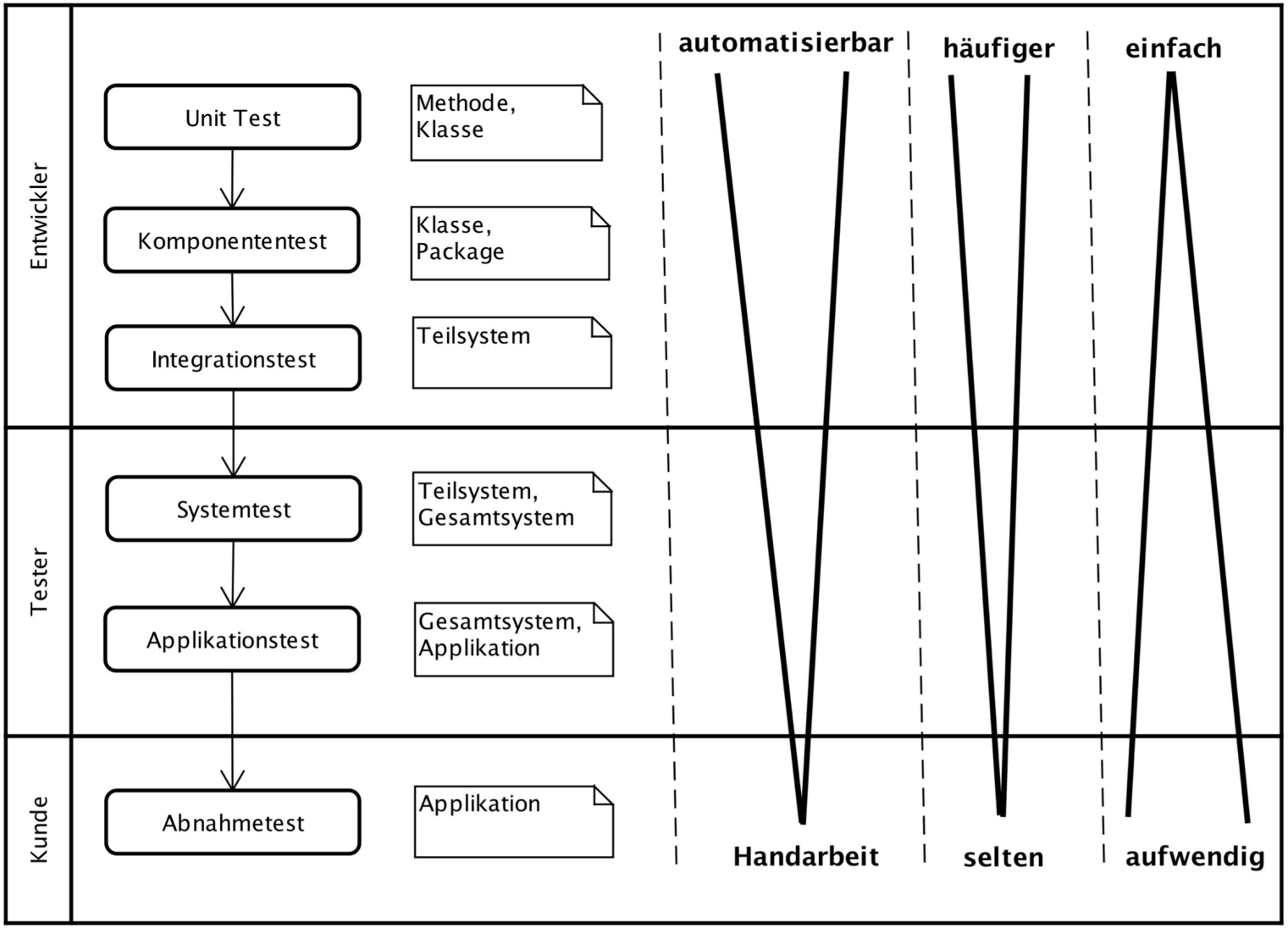


Abbildung entnommen aus „Der Weg zum Java-Profi: Konzepte und Techniken für die professionelle Java-Entwicklung.“, Michael Inden

Abgrenzung zur Verifikation

- Verifikation = formaler Korrektheitsbeweis
  - Es wird versucht, mit formalen Methoden den Nachweis zu führen, dass ein Programm nur richtige Ergebnisse produzieren kann.
  - Liefert eine endgültige Aussage zur Korrektheit.
  - Schon für sehr kleine Programme kann dieser Ansatz sehr aufwendig sein!
- Testen = systematisches Ausprobieren
  - Es wird eine bestimmte Anzahl von Tests konstruiert, mit denen das Programm „probeweise“ ausgeführt wird.
  - Mit Testen kann aber nur die Anwesenheit von Fehlern nachgewiesen werden, nicht aber deren Abwesenheit!
  - In der Regel wird eine sehr große Anzahl von Testfällen benötigt um ein Programm ausführlich zu testen.
    - Das verursacht aber meist sehr hohe Kosten.
    - Endgültiger Korrektheitsbeweis ist damit auch nicht möglich.

Unit-Tests mit JUnit

Unit-Tests (1)

- Beim Unit-Test wird eine kleine Einheit eines Programms betrachtet und auf mögliche Fehler untersucht.
- Die Einheit wird isoliert vom Rest des Gesamtsystems betrachtet.
  - Testen eines ausgewählten Softwarebausteins (meist Methoden)
- Unit-Tests können automatisiert werden.
  - Testfälle werden direkt als Quellcode entwickelt.
- Testen sollte immer ein wichtiger Bestandteil des Entwicklungsprozesses sein.
- Es gibt Programmiertechniken, bei denen dieses Testen ein elementarer Teil des Entwicklungsprozesses ist.
  - Extreme Programming (XP)
  - Test-Driven Development (TDD)

Unit-Tests (2)

- FIRST-Prinzipien
  - Fast: schnell ablaufen (sonst werden sie seltener ausgeführt).
  - Independent: keine Abhängigkeiten zwischen Tests.
  - Repeatable: wiederholt in jeder Umgebung ausführbar.
  - Self-Validating: durch Assertions (keine manuellen Überprüfungen).
  - Timely: zeitnah geschrieben (kurz vor oder nach Produktionscode).

Test-Driven Development

- Schritte:
  1. Zuerst werden die Ergebnisse in Form von Testfällen implementiert.
     - Damit die Testfälle übersetzt werden können, werden zunächst nur leere Methoden oder Methoden mit einer einzigen return-Anweisung bereitgestellt.
       - Übersetzung funktioniert, Tests schlagen noch alle fehl.
  2. Danach wird der Code entwickelt.
     - Die Methodenrümpfe werden schrittweise vervollständigt, bis am Ende alle Tests fehlerfrei durchlaufen werden.
  3. Räume den Anwendungscode auf (Refactoring)
     - Beispielsweise Code-Duplikate entfernen, Code-Richtlinien einhalten usw.
     - Nach dem Refactoring laufen alle Tests immer noch fehlerfrei durch. Beginne wieder bei Schritt 1.
- Vorteile bei diesem Vorgehen
  - Tests werden zuerst erstellt.
  - Tests können nicht vernachlässigt werden.
  - Tests geben notwendige Funktionalität vor.

JUnit

- JUnit bietet ein einheitliches Framework zur Organisation und systematischen Durchführung von Unit-Tests in Java.
- Aktuelle Version 5.8.2 (November 2021)
- Weit verbreitetes Unit-Testing-Framework
- Die Bibliothek ist nicht Teil des JDK sondern ist unter https://junit.org/junit5/ erhältlich und dokumentiert.
  - Installation am einfachsten direkt über die IDE.
- Testfälle werden direkt als Java-Programme erstellt.

Exkurs: Annotationen

- Anführen von Metadaten im Quelltext (seit Java 1.5), nicht Teil des eigentlichen Quellcodes.
- Annotationen starten immer mit einem @-Zeichen.
- Annotationen beziehen sich auf das folgende Code-Element (z.B. Klasse, Methode, Feld).
- Java API Beispiele:
  - `@Override`
    - Annotierte Methode überschreibt eine Methode einer Superklasse bzw. eines Interfaces.
  - `@Deprecated`
    - Markiert veraltete Code-Elemente.
- JUnit-Beispiele:
  - `@Test`
    - Markiert eine Methode als Testmethode.
  - `@DisplayName`
    - Deklaration einer benutzerdefinierten Testbeschreibung.
  - `@ParameterizedTest`
    - Markiert eine Methode als parametrisierte Testmethode.
  - `@Disabled`
    - Markiert eine Testklasse oder Testmethode als deaktiviert.

Testklasse

- Es gibt eine Top-Level-Testklasse für jede zu testende Klasse.
- Die Testklasse darf nicht abstrakt sein.
- Die Testklasse muss genau einen Konstruktor haben.
- Die Testklasse importiert benötigte JUnit-Klassen und Methoden.
  - Für das Importieren der JUnit-Methoden werden meist statische Imports verwendet.
    - Beispiel:
      ```
      import static org.junit.jupiter.api.Assertions.*;
      ```
- Die Testklasse definiert Testmethoden und Verwaltungsmethoden.
  - Diese Methoden dürfen weder abstrakt noch privat sein.
  - Der Rückgabewert der Methoden muss `void` sein.

Testmethoden

- Testmethoden müssen mit `@Test`, `@ParameterizedTest`, `@RepeatedTest`, `@TestFactory` oder `@TestTemplate` gekennzeichnet werden.
- Jede Testmethode sollte jeweils nur eine Funktionalität überprüfen (meist ein Assert pro Testmethode).
  - Der Methodenname ist frei wählbar (sollte den Testfall beschreiben)
- Üblicherweise besteht ein Test aus den drei Teilen Given, When, Then (GWT-Stil).
  - Given – Voraussetzungen für den Testfall werden aufgestellt.
  - When – Aktionen die im Testfall überprüft werden sollen.
  - Then – Abgleich der erwarteten Ergebnisse mit den berechneten Werten.

Verwaltungsmethoden

- Zweck der Verwaltungsmethode wird festgelegt mit einer Annotation.
  - `@BeforeEach`
    - Wird vor jedem einzelnen Test aufgerufen.
  - `@AfterEach`
    - Wird nach jedem einzelnen Test aufgerufen.
  - `@BeforeAll`
    - Wird einmal vor allen Tests aufgerufen (muss `static` sein).
  - `@AfterAll`
    - Wird einmal nach allen Tests aufgerufen (muss `static` sein).
- Diese Verwaltungsmethoden können für die Initialisierung und das Zurücksetzen von Daten hilfreich sein.
- Allerdings können diese Verwaltungsmethoden die Lesbarkeit einzelner Testfälle stark einschränken.

Assertions-Klasse (1)

- In der Assertions-Klasse bietet JUnit verschiedene Methoden an, um Annahmen im Test zu überprüfen.
- Methoden für den Vergleich von Wahrheitswerten:
  - `assertFalse(boolean condition)`
  - `assertTrue(boolean condition)`
  - …
- Methoden für den Vergleich von Werten `expected` und `actual`:
  - `assertEquals(long expected, long actual)`
  - `assertEquals(double expected, double actual)`
  - `assertEquals(double expected, double actual, double delta)`
  - `assertEquals(Object expected, Object actual)`
  - …

Assertions-Klasse (2)

- Methoden für den inhaltlichen Vergleich von Arrays:
  - `assertArrayEquals (int[] expected, int[] actual)`
  - …
- Methoden zur Überprüfung, ob eine erwartete Exception geworfen wird:
  - `assertThrows(Class<T> expectedType, Executable executable)`
  - …
- Methoden zur Überprüfung, ob eine variable Anzahl von Assertions stimmen:
  - `assertAll(Executable… executables)`
  - …
- Viele weitere, siehe API-Dokumentation!
- Alle Vergleichsmethoden sind überladen mit zusätzlichem Parameter `message`.
  - Text wird beim Fehlschlagen des Tests ausgegeben.
    - `assertFalse(boolean condition, String message)`

Beispiel Assertions

```java
@Test
public void sizeEmpty() {
    final ArrayStack stack = new ArrayStack();

    final int size = stack.size();

    assertTrue(0 == size);
}
```

```java
public class ArrayStack implements Stack {

    private final String[] data;
    private int position = 0;
    …
    public int size() {
        return position;
    }
    …
}
```

src/at/ac/uibk/pm/testing/stack/ArrayStack.java

Use Meaningful Assertions

```java
@Test
public void sizeEmpty() {
    final ArrayStack stack = new ArrayStack();

    final int size = stack.size();

    assertEquals(0, size);
}
```

- Vorher:
  - Vergleich ist über `assertTrue` bzw. `assertFalse` durchaus möglich.
  - Im Fehlerfall geht allerdings Information verloren.
  - Feedback wenig aussagekräftig (z.B. `AssertionFailedError: expected: <true> but was: <false>`)
- Nachher:
  - assert-Methode ist auf Vergleich abgestimmt
  - Feedback ist detaillierter (z.B. `AssertionFailedError: expected: <0> but was: <3>`)

Describe Your Tests

```java
@Test
@DisplayName("an empty stack should have size 0")
public void sizeEmpty() {
    final ArrayStack stack = new ArrayStack();

    final int size = stack.size();

    assertEquals(0, size);
}
```

- Vorher:
  - Bei den Testergebnissen werden die Methodennamen angezeigt.
- Nachher:
  - Bei den Testergebnissen wird die Beschreibung aus `DisplayName` angezeigt.
  - Der Testfall ist dokumentiert.

Exceptions

- Testen des Normalfalls:
  - Werden Methoden, welche Checked Exceptions werfen, getestet, können die aufgetretenen Exceptions in der Testmethode weitergereicht werden.
- Testen des Ausnahmefalls:
  - Überprüfung, ob erwartete Exception wirklich geworfen wird.
  - Test ist erfolgreich, wenn die erwartete Exception von der getesteten Methode geworfen wird.
  - Test scheitert, wenn die Exception nicht geworfen wird.

Testen des Ausnahmefalls

```java
@Test
@DisplayName("pop on empty stack")
public void popEmpty() {
    final ArrayStack stack = new ArrayStack();

    try {
        stack.pop();
        fail("Test should fail since stack is empty.");
    } catch (EmptyStackException ignored) {

    }
}
```

< > Let Framework Handle Exceptions

```java
@Test
@DisplayName("pop on empty stack")
public void popEmpty() {
    final ArrayStack stack = new ArrayStack();

    Executable when = () -> stack.pop();

    assertThrows(EmptyStackException.class, when);
}
```

`Executable` aus org.junit.jupiter.api.function

Lambda-Ausdruck (mehr dazu in der VO zu funktionaler Programmierung)

- Vorher:
  - Beim Lesen muss die Bedeutung (Semantik) der Variablen selbst herausgefunden werden.
  - Variablen geben keinerlei Auskunft über Inhalt.
- Nachher:
  - Lesbarer Code

Parametrisierte Tests

- Ausführen desselben Tests mit unterschiedlichen Daten, welche als Argument der Testmethode übergeben werden.
- Diese Tests werden mithilfe der Annotation `@ParameterizedTest` realisiert.
- Zusätzlich muss eine Quelle für die Argumente festgelegt werden.
- Beispiele möglicher Quellen:
  - `@ValueSource`
    - Array aus Literalen
  - `@CsvSource`
    - Liste aus Comma-separated-values

Beispiel parametrisierte Tests (1)

```java
public static boolean isPrime(final int primeCandidate) {
    if (primeCandidate <= 1) {
        return false;
    }
    for (int divisor = 2; divisor * divisor <= primeCandidate; ++divisor) {
        if (primeCandidate % divisor == 0) {
            return false;
        }
    }
    return true;
}
```

```java
@ParameterizedTest(name = "isPrime({0}) => true")
@ValueSource(ints = {2, 3, 5, 7, 11, 13, 17, 19, 15601})
void isPrime(int value) {
    assertTrue(MathUtils.isPrime(value));
}
```

src/at/ac/uibk/pm/testing/math/MathUtilsTest.java

Beispiel parametrisierte Tests (2)

```java
public static long fibonacciNumber(final int n) {
    if (n < 0) {
        throw new IllegalArgumentException(
                String.format("Expected non-negative integer but got %d", n));
    }
    long previous = 0;
    long current = 1;
    if (n <= 1) {
        return n;
    }
    for (int i = 2; i <= n; ++i) {
        final long currentTmp = current;
        current = Math.addExact(current, previous);
        previous = currentTmp;
    }
    return current;
}
```

```java
@ParameterizedTest(name = "fibonacciNumber({0}) => {1}")
@CsvSource({"0, 0", "1, 1", "2, 1", "3, 2", "4, 3", "5, 5", "6, 8"})
void fibonacciNumberInput(final int input, final int expectedOutput) {
    assertEquals(expectedOutput, MathUtils.fibonacciNumber(input));
}
```

src/at/ac/uibk/pm/testing/math/MathUtilsTest.java

Objekte mit Abhängigkeiten (1)

- Bei einem Element, welches getestet werden soll, wird von object-under-test oder system-under-test (SUT) gesprochen.
- Ein Kollaborateur ist ein Element, welches durch das SUT aufgerufen wird.
- SUT und Kollaborateure können sich beeinflussen.
  - Indirekte Eingabe
    - Kollaborateure beeinflussen das SUT beispielsweise über Rückgabewerte von Methoden, Verändern des Zustands eines Parameters oder Werfen eine Ausnahme.
  - Indirekte Ausgabe
    - SUT beeinflusst den Zustand eines Kollaborateurs.
- Eine Einheit soll bei Unit-Tests isoliert vom Rest des Gesamtsystems betrachtet werden.
- Was wird als eine Einheit betrachtet?
  - Klassischer bzw. Detroit-Style: Eine Klasse inklusive der Abhängigkeiten
  - London-Style: Eine Klasse

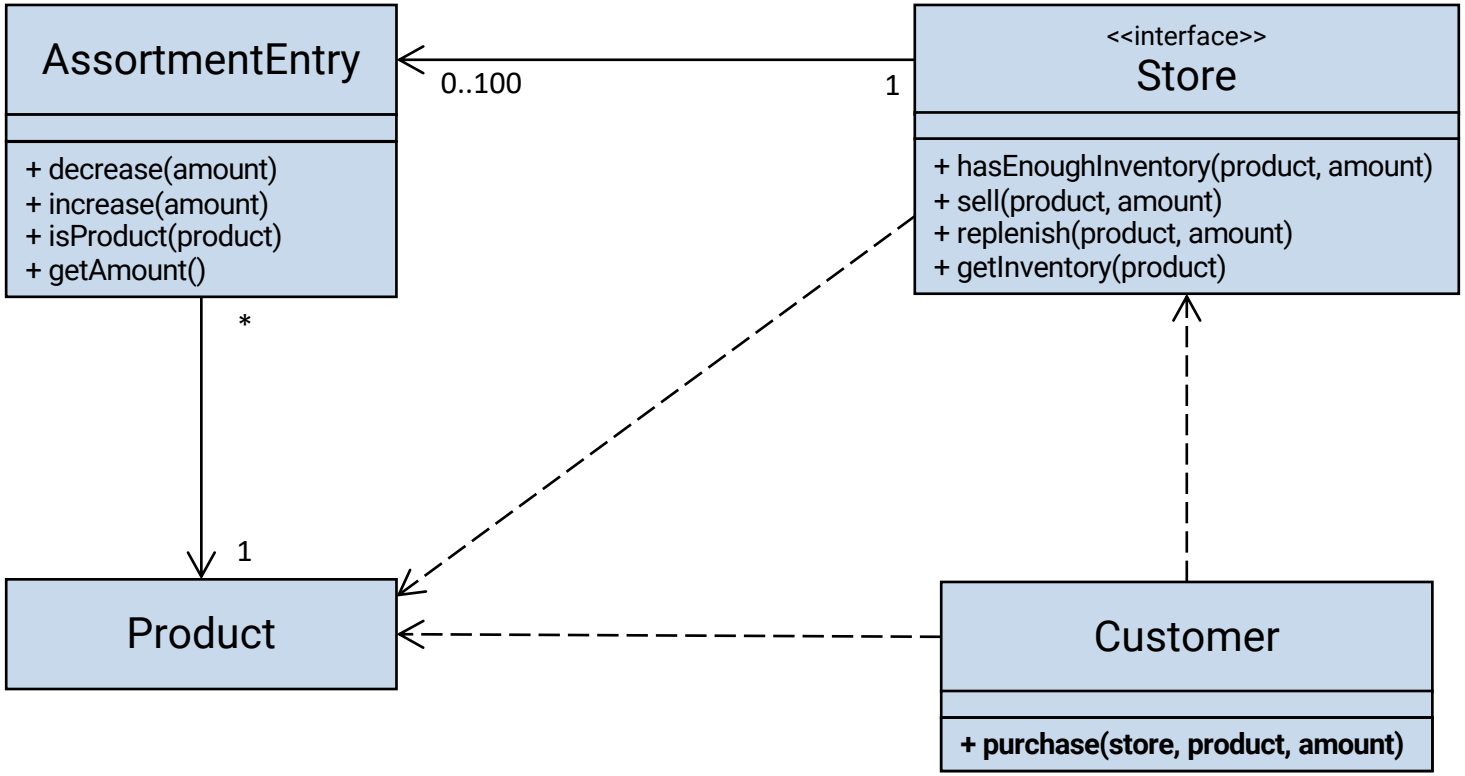
Objekte mit Abhängigkeiten (2)

- Klassischer bzw. Detroit-Style:
  - Verwendung von realen Objekten, wenn möglich.
  - Tests dürfen keine Abhängigkeiten teilen, welche das Ergebnis anderer Tests oder die Wiederholbarkeit beeinflussen.
    - Dateisystem
    - Datenbank
    - Datum
    - Zeit
    - …
  - Test-Doubles werden eingeführt, um solche Abhängigkeiten zu isolieren.
- London-Style:
  - Verwendung eines Mocks für jedes Objekt mit relevantem Verhalten.

Test-Doubles

- Dummy
  - Objekt, das als Parameter übergeben aber nicht verwendet wird.
- Fake
  - Alternative und vereinfachte Implementierung mit abgewandelter Funktionsweise.
- Stub
  - Minimalistische Implementierung mit vordefinierten Rückgabewerten für Methoden, welche während des Tests aufgerufen werden.
- Mock
  - Objekt, welches das vordefinierte Verhalten mit erwarteten Methodenaufrufen überprüft.
  - Dynamische Generierung durch Frameworks wie beispielsweise Mockito, EasyMock oder jMock.

Beispiel Objekte mit Abhängigkeiten (1)



Beispiel Objekte mit Abhängigkeiten (2)

```
public class Customer {

    public boolean purchase(Store store, Product product, int amount) {
        if (!store.hasEnoughInventory(product, amount)) {
            return false;
        }
        store.sell(product, amount);
        return true;
    }
}
```

Beispiel Objekte mit Abhängigkeiten (3)

- Klassischer Ansatz: Es werden keine Test-Doubles verwendet, da es keine geteilten Abhängigkeiten gibt.

```
@Test
@DisplayName("purchase some of the available products")
public void purchaseSomeProducts() {
    Customer customer = new Customer();
    Store store = new GroceryStore();
    Product product = new Product("Milk", 129);
    store.replenish(product, 10);

    boolean success = customer.purchase(store, product, 7);

    assertAll(
            () -> assertTrue(success),
            () -> assertEquals(3, store.getInventory(product))
    );
}
```

Exkurs: Mockito

- Mockito ist ein Beispiel eines Mocking-Frameworks.
- Hilfreiche Methoden
  - `mock()`
    - Wird verwendet um einen Mock einer Klasse zu erstellen.
  - `when()` und `thenReturn()`
    - Verhalten bei einem Methodenaufruf beschreiben.
  - `verify()`
    - Überprüfen von einem Methodenaufruf.

Beispiel Objekte mit Abhängigkeiten (4)

- London Ansatz: Es wird für das Objekt des Interfaces `Store` ein Mock verwendet.

```
@Test
@DisplayName("purchase some of the available products")
public void purchaseSomeProducts() {
    Customer customer = new Customer();
    Store store = mock(Store.class);
    when(store.hasEnoughInventory(any(), anyInt())).thenReturn(true);
    Product product = new Product("Milk", 129);

    boolean success = customer.purchase(store, product, 7);

    assertTrue(success);
    verify(store, times(1)).sell(product, 7);
}
```

Quellen

- Bernhard Lahres, Gregor Rayman, Stefan Strich: **Objektorientierte Programmierung: Das umfassende Handbuch**, Rheinwerk Verlag, 5. Auflage, 2021
- Joachim Goll, Cornelia Heinisch: **Java als erste Programmiersprache**, Springer Vieweg, 8. Auflage, 2016
- Michael Inden: **Der Weg zum Java-Profi: Konzepte und Techniken für die professionelle Java-Entwicklung**, dpunkt.verlag, 5. Auflage, 2021
- Gerard Meszaros: **xUnit Test Patterns: Refactoring Test Code**, Addison-Wesley, 2007
- Vladimir Khorikov: **Unit Testing: Principles, Practices, and Patterns**, Manning Publications, 2020
- Martin Fowler: **Mocks Aren't Stubs**, besucht am 30.03.2022, http://www.martinfowler.com/articles/mocksArentStubs.html