

Verteilte Systeme und Komponenten

# **Automatisiertes Testing**

**Wie man effektiv Software testet.**

Roland Gisler



# Inhalt

- Einführung und Motivation
- Test First Ansatz
- Abgrenzung zwischen Unit- und Integrationstests.
- Messen der Codeabdeckung von Tests
- Design: Dependency Injection
- Test Doubles (Mocking / Stellvertreter)
- Testen mit Hilfe von Containern (Integration)
- Zusammenfassung und Quellen

# Lernziele

- Sie kennen die verschiedenen Testarten und sind in der Lage gute Unit- und Integrationstests zu schreiben.
- Sie beherrschen die Entwicklung nach dem Test-First Prinzip.
- Sie nutzen Werkzeuge zur Messung der Codeabdeckung aktiv zu Verbesserung Ihres Codes und der Testfälle.
- Sie kennen das Prinzip der Dependency Injection (DI).
- Wie wissen was Test Doubles sind und können Mocking-Frameworks einsetzen.
- Sie sind in der Lage Container-Technologien für automatisierte Tests zu nutzen.

# Einführung


# Software Testing - Einführung

- Bei vielen Entwickler\*innen verpönt und unbeliebt. Warum?
    - «Ich kann programmieren, ich mache keine Fehler.»
    - «Ich will programmieren, nicht testen.»
    - «Ich muss den Termin einhalten, es muss schon fertig sein!»
    - «Wenn ich keinen Fehler finde, war das Testen verlorene Zeit!»
  - Testen hat teilweise ein schlechtes Image und gilt als 'uncool'...
    - Einerseits in einzelnen Köpfen (Entwickler\*innen).
    - Andererseits aber auch in der Firmenkultur (Management).
  - Aber: Tests gewährleisten, dass (speziell auch wichtige) Software möglichst fehlerfrei arbeitet!
    - z.B. medizinische Geräte, Verkehrsmittel, Atomkraftwerke etc.
- ➔ Sie erinnern sich: Ariane 5 und andere Geschichten...**

## Motivation für Testen – Erinnern Sie sich?

- Wir testen um **X** Fehler zu finden? **Nein!**
- Besser:

**Wir testen kontinuierlich während der Implementation, um die Gewissheit zu haben dass es funktioniert!**



- Fehler finden bevor man Sie gemacht hat!
  - Fehler korrigieren bevor man Sie implementiert hat!
  - Oder mindestens Fehler schon im Ansatz (wenn es noch niemand anders gemerkt hat) finden!
- Wie macht man das?

# Test First!

# Test First Methodik

- Entwickelt aus XP (extrem programming, u.a. von E. Gamma)
- Ganz einfacher Ansatz:  
**Vor** der Implementation immer **zuerst** die Testfälle schreiben!

Viele positive Effekte:

- Beim Schreiben der Testfälle denkt man auch an die konkrete Implementation des zu testenden Codes. Dabei **reift** diese buchstäblich **besser** heran!
- Dabei fallen einem viele **Ausnahmen** und **Sonderfälle** ein, welche man bei der Implementation wie **selbstverständlich** auch **berücksichtigt**.
- Ist die **Implementation** eines SW-Elementes **fertig**, kann dieses ohne aufwändige Integration **sofort getestet** werden!

# Unit Tests



# Unit Tests

- Werden häufig mit Komponenten-, Modul- und Entwicklertests gleich gesetzt.
- Sind funktionale Test von einzelnen, in sich abgeschlossenen Units (typisch Klasse, aber auch Komponente oder Modul).
- Ziele von guten Unit Tests:
  - **Schnell** und **einfach ausführbar, selbstvalidierend** (mit **assert\*-Methoden**) und **automatisiert**.
  - Möglichst **ohne** Abhängigkeiten zu anderen Klassen, Komponenten oder Modulen (lose Kopplung).
  - Werden während der Entwicklung geschrieben und ausgeführt.
    - In der Entwicklungsumgebung (IDE).
    - Im automatisierten Buildprozesses (CI).
- Gute Unterstützung durch Frameworks (z.B. JUnit, TestNG etc.)

# Unit Tests: Nutzen

## ■ 👍 Positiv:

- Neue oder veränderte Komponenten können sehr **schnell** getestet werden (regressiv).
- Testen ist vollständig in die Implementationsphase integriert.
- Test First Ansatz ist möglich.
- Automatisiertes, übersichtliches Feedback / Reporting.
- Messung von ➔ Codeabdeckung kann integriert werden.

## ■ 👎 Negativ:

- Für GUI(-Komponenten) etwas aufwändiger.
- Qualität und Nachvollziehbarkeit der Testfälle muss im Auge behalten werden: Qualität vor Quantität!
- In manchen Architekturen / Umgebungen schwierig umsetzbar.

# **Integrationstests mit JUnit**

# Integrationstest mit JUnit - Namenskonvention

- **Wichtig:** JUnit (und andere Frameworks) können zur Automatisierung (fast) **aller** Testarten verwendet werden!
- Für Integrationstests existiert für JUnit (und Apache Maven) eine eigene Namenskonvention:
  - Klassenname **XyzIT** für Integrationstests.
  - Werden auch unter **src/test/java** abgelegt.
- Die Unterscheidung ergibt sich «**nur**» durch den Zeitpunkt der Ausführung, und deren (Laufzeit-)Abhängigkeiten.
- Integrationstest können mit Apache Maven mit dem eigenen Stage **integration-test** bzw. **verify** ausgeführt werden.
  - Getrennte Plugins **surefire** und **failsafe** weisen die Testresultate auch getrennt (Unit und Integration) aus.

# Abgrenzung: Unit vs. Integrations Tests

- Wird häufig individuell festgelegt und kontrovers diskutiert.
- Unit Tests sind wirklich Unit Tests, wenn sie (unter anderem)...
  - auf einem beliebigen System und jederzeit lauffähig sind.
  - (bei Java) auch auf unterschiedlichen Betriebssystemen laufen.
- Konsequenz:
  - Testfälle welche z.B. mit dem **Dateisystem** interagieren, sind in strenger Sichtweise bereits **Integrationstests**!
  - Testfälle die z.B. Sockets verwenden (auch wenn nur auf «localhost») sind bereits Integrationstests!
- Unit Tests sollten somit **nie** aufgrund von «Fremdeinflüssen» fehlschlagen!
  - Beispiel: falscher Pfad ('\' vs. `/' ), Platz, Zugriffsrechte etc.

# Empfehlungen zu Unit und Integrations Tests

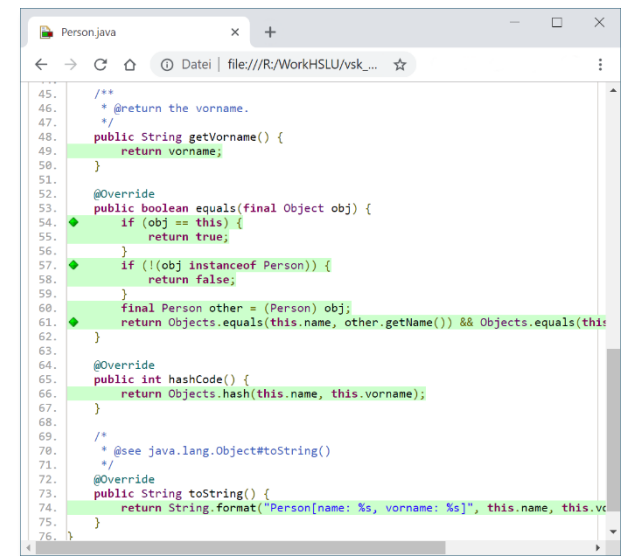


- Machen Sie eine bewusste Trennung zwischen den beiden Kategorien.
- Einigen Sie sich im Team / Organisation auf eine gemeinsame Philosophie wie sie die Kategorien exakt aufteilen.
  - Akademische Sturheit hilft nicht – pragmatische Einheitlichkeit hingegen schon!
  - Kann auch mal projektspezifisch sein!
- Je mehr als Unit Test realisiert werden kann, je besser!
- **Aber jeder automatisierte** Test ist ein grosser Gewinn!
- Nutzen Sie zusätzliche Hilfsmittel wie → Code Coverage oder → Test Doubles (Mocking), oder → Testcontainer um die Motivation zu steigern.

# **Messung der Code Coverage**

# Was ist Code Coverage?

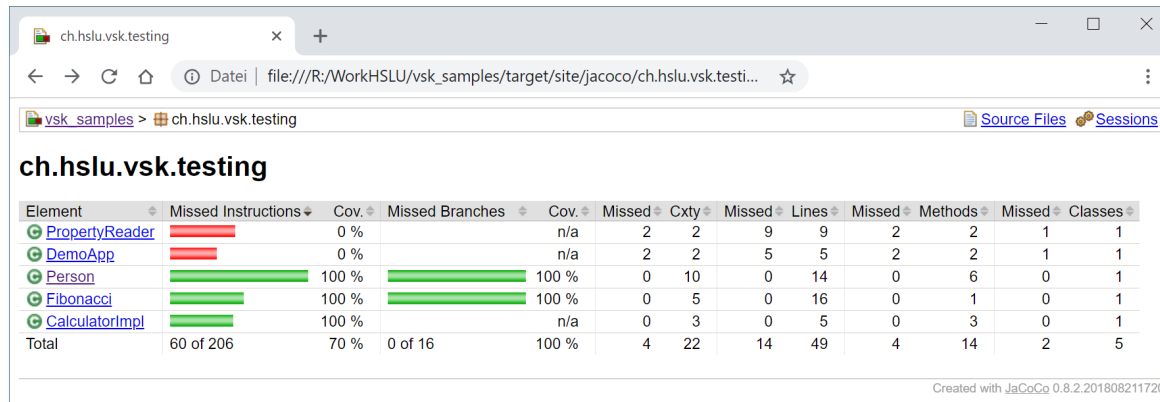
- Code Coverage ist eine Metrik welche zur Laufzeit misst, welche Quellcodezeilen ausgeführt wurden.
- Diese Messung erfolgt typisch während der Ausführung der Testfälle.
  - Kann auch zur 'normalen' Laufzeit erfolgen, dann z.B. zur Messung welche Funktionen tatsächlich genutzt werden!
- Somit kann eine Aussage gemacht werden, wie umfassend der Code tatsächlich genutzt bzw. getestet wurde!
- Umfangreiche, statistische Aufbereitung der Daten möglich:
  - Nach Testfall, Komponente, Package, Teilsystem etc.
  - Auf Buildserver z.B. auch historisiert, d.h. zeitliche Veränderung der Werte wird sichtbar gemacht.





# Coverage - Motivation für Einsatz

- Herausforderung: Wie implementiert man mit möglichst geringem Aufwand trotzdem möglichst umfassende Testfälle (Effizienz!)?
  - Wie kann man die Qualität von Testfällen beurteilen?
- ➔ Messen der durch die Testfälle erreichten Codeabdeckung!
  - Während der Ausführung der Tests wird gemessen, welche Statements/Codezeilen tatsächlich vom Test erfasst wurden.





- **Vorsicht:** Eine hohe Coverage ist **kein** Beweis für gute Testfälle oder gar die Fehlerfreiheit des Codes!

# Coverage - Was kann man messen?

- Man unterscheidet verschiedene Messtechniken und -werte:
  - Statement Coverage (Line Coverage)
  - Branch Coverage
  - Decision Coverage
  - Path Coverage
  - Function Coverage
  - Race Coverage
- Messwerte sind unterschiedlich Aufwändig und Aussagekräftig.

# Coverage - Was wird gemessen? (1)

- Statement Coverage 
  - Misst ob (und wie häufig) eine Codezeile durchlaufen wurde.
  - Problem: Handelt es sich bei der Zeile z.B. um einen logischen Vergleich/Ausdruck, ist ein einmaliger Durchlauf nicht repräsentativ.
- Branch Coverage
  - Prüft, dass alle Zweige einer bedingten Anweisung ausgeführt wurden.
- Decision Coverage 
  - Bei Fallunterscheidungen (**if**, **while** etc.) wird geprüft, dass alle Teilausdrücke in der Bedingung auf **true** und **false** aufgelöst wurden (strenger als Branch Coverage).

## Coverage - Was wird gemessen? (2)

- Path Coverage

- Bei der Path Coverage wird gemessen, ob alle möglichen Kombinationen von Programmablaufpfäden durchlaufen wurden.
- Problem: Die Anzahl der Möglichkeiten steigt exponentiell mit der Anzahl Entscheidungen → in der Praxis nicht durchführbar.

- Function Coverage

- Misst auf der Basis der Funktionen ob sie aufgerufen wurden.

- Race Coverage

- Konzentriert sich auf Codestellen die parallel ablaufen.

# Coverage - Technische Umsetzung

- Instrumentierung des Quellcodes 🖱️
  - Der Quellcode wird durch einen Preprocessor vor dem Compilieren mit Statements zur Coverage-Messung ergänzt.
  - Nachteil: Modifizierter Quellcode (man denke an Debugging).
- Instrumentierung des Bytecodes 🙌
  - Der Bytecode wird bei/nach der Kompilierung mit Bytecode zur Coverage-Messung ergänzt.
  - Nachteil: `class`-Dateien müssen separiert werden (Deployment).
- Just-in-time Instrumentierung zur Laufzeit 👍
  - Instrumentiert den Bytecode direkt während des Classloadings.
  - Vorteile: Nur ein Binary, unabhängig von Compiler, jederzeit und überall ad-hoc aktivierbar!
  - Nachteil: Teilweise «Konkurrenz» bei Bytecode-Manipulation.

# Coverage - Wer misst und wann?

- Gemessen wird die Abdeckung bei der Ausführung des Codes durch den (modifizierten) Code selber.
  - Das Coverage-Werkzeug **instrumentiert** den Code.
  - Messung sehr häufig bei der Ausführung der Testfälle.
  - Daten werden typisch in eine spezifische Datei persistiert.
- Ein populäres Coverage Werkzeug für Java ist JaCoCo.
  - Bestandteil von EclEmma, welches zu Eclipse gehört.
  - Details siehe <https://www.eclemma.org/jacoco/index.html>
  - Ist in VSK-Projekten (Basis: oop\_maven\_template) integriert.
- Es ist ein Buildtool, die IDE oder der Buildserver werten die Resultate «nur» aus und stellen diese dann ansprechend dar.
  - Mit Apache Maven wird mit `mvn jacoco:report` ein HTML-Report erzeugt, siehe `./target/site/jacoco/index.html`

# Live Demo's / Screencast's

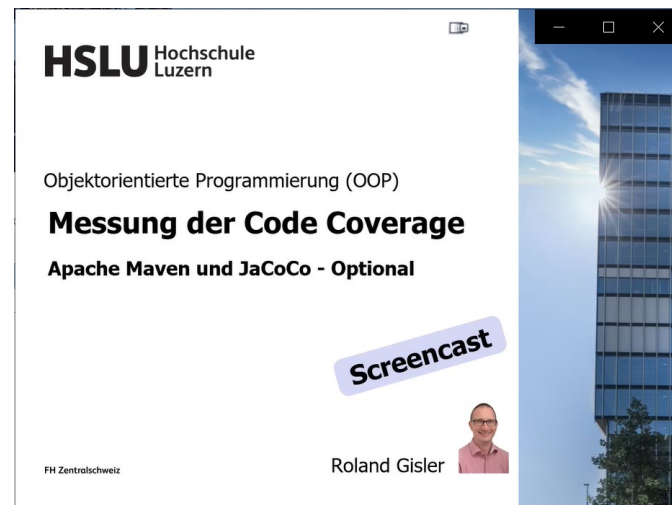
- Messung der Code Coverage mit JaCoCo / Console:

[E02\\_SC04\\_TestingCoverageConsole.mp4](#)

- Messung der Code Coverage in Eclipse:

[E02\\_SC05\\_TestingCoverageEclipse.mp4](#)

(Screencasts aus Modul OOP)



# Dependency Injection

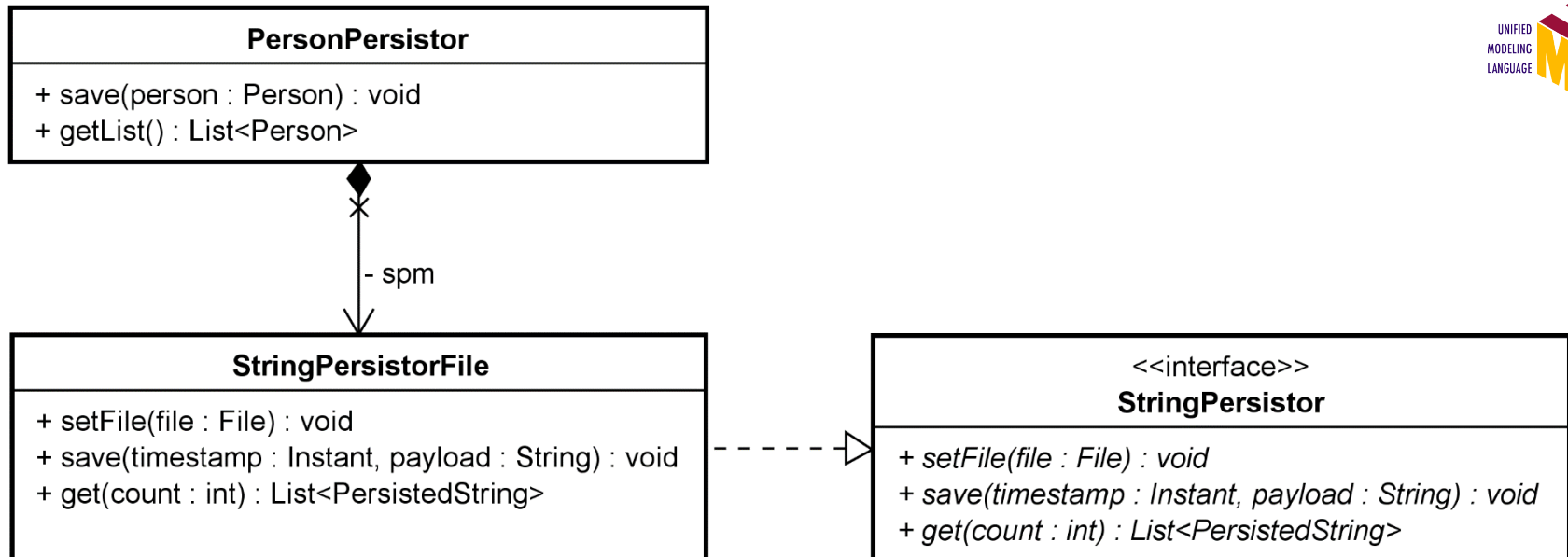


# Schlechte Testbarkeit

- Oft stellen Entwickler\*innen fest, dass sich selbst einfache Klassen oder Komponenten «schlecht» testen lassen.
  - Das führt sehr schnell zum Verzicht auf (Unit-)Tests, *oder:*
  - Es werden wieder vermehrt Integrationstests implementiert.
- Bei näherer Betrachtung sind oft zu viele bzw. **zu stark gekoppelte** Abhängigkeiten die Ursache.
  - Deren negative Auswirkungen fallen bei der ersten Anwendung (und das sind die Testfälle!) sehr schnell auf.
- Die eigentliche Ursache ist somit schlicht **schlechtes Design!**
- Darum: Lässt sich eine Softwareeinheit nur «schlecht» (kompliziert und/oder aufwändig) testen, sollte man das **Design immer kritisch hinterfragen!**

## Beispiel: PersonPersistor

- Wir wollen eine Klasse **PersonPersistor** implementieren, welche **Person**-Objekte als **Strings** serialisiert in eine Datei speichert.
- Wir haben bereits eine erprobte und getestete Implementation eines universellen **StringPersistorFile** → Wiederverwenden!
- Ein erster, primitiver Lösungsansatz (dabei bleibt es leider zu oft!):



## Beispiel: PersonPersistor mit hoher Kopplung – **Schlecht!**

- PersonPersistor erzeugt sich seinen StringPersistor selber.

```
public final class PersonPersistor {  
  
    private final StringPersistorFile spm =  
        new StringPersistorFile();  
  
    ...  
}
```



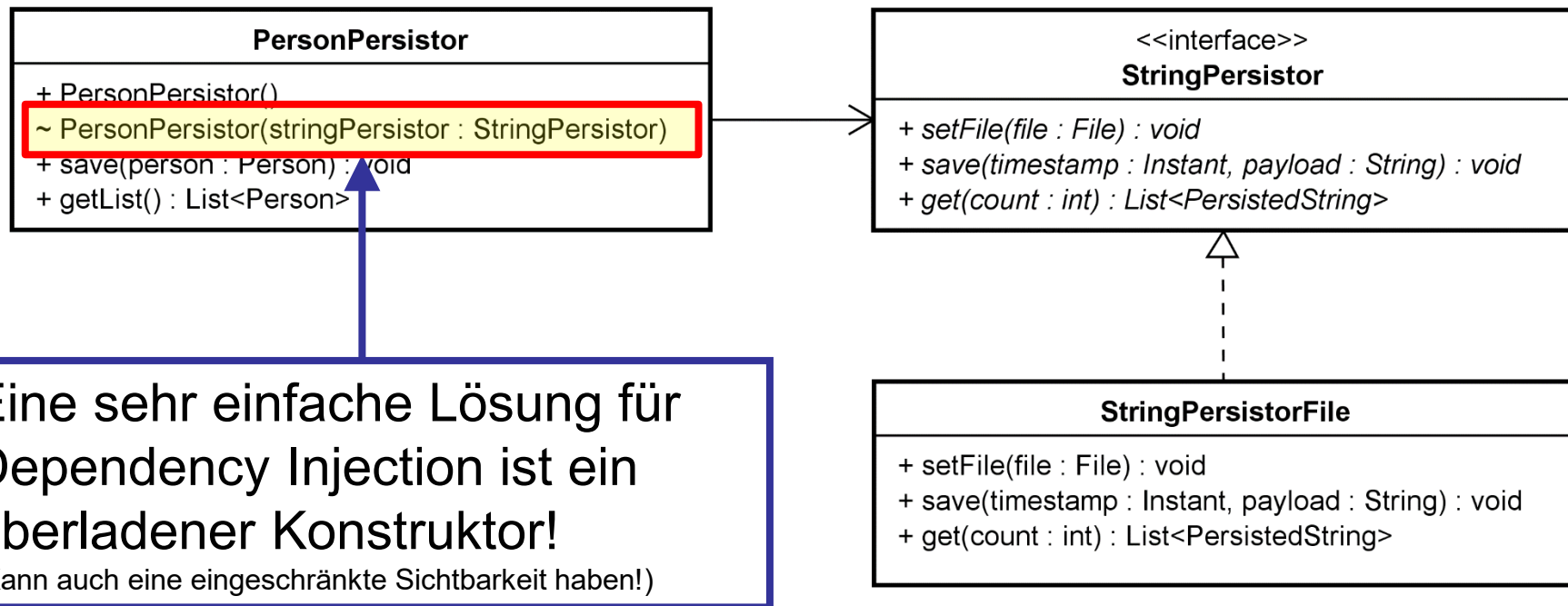
- In der Realisierung sehr einfach, aber das ist auch der einzige Vorteil dieser Lösung!
- Dagegen stehen einige negative Aspekte:
  - Typen und Implementation fest verdrahtet (starke Kopplung).
  - Abhängigkeit zu einer Implementationsklasse (obschon ein Interface existieren würde!).
  - Unflexible Implementation, wie testet man diese?

# Probleme beim Testen des PersonPersistor

- Der **PersonPersistor** enthält die Logik um **Personen** in **Strings** zu serialisieren und wieder zu deserialisieren. Diese Funktionalität **wollen** wir testen.
- Da intern ein **StringPersistorFile** verwendet wird, ist das aber schwierig, weil dieser auf das Dateisystem zugreift, und wir damit sofort in der Kategorie **→Integrationstest** sind!
- Zudem testen wir beim Test des schlanken **PersonPersistors** den (hoffentlich schon getesteten) **StringPersistorFile** ungewollt nochmal mit. **→** Selektivität des Testfalles sinkt!
- Es wäre besser, wenn wir (mindestens für das Testen) die Dependency auf den intern verwendeten **StringPersistorFile** durch etwas anderes ersetzen könnten!

# Lösung: Dependency Injection (DI) – gut!

- Die Lösung: Wir verwenden Dependency Injection!
- Eine Klasse/Komponenten erzeugt ihre Abhängigkeiten nicht selber, sondern lässt sich diese (wahlweise) auch **von Aussen** übergeben.



# Vorteile beim Einsatz von Dependency Injection

- Man ersetzt den konkreten Typ durch ein **Interface**, womit die Kopplung stark abnimmt.
  - **DIP** – **D**ependency **I**nversion **P**rinciple (aus → S.O.L.I.D.)
  - Dadurch können auch verschiedene, alternative Implementationen genutzt werden.
  - Es resultiert eine bessere «**Seperation of Concerns**» (SoC).
- Das Beste: Die Testbarkeit wird dadurch massiv vereinfacht!  
Man kann während der Tests eine alternative Implementation als Platzhalter → **Test Double** einfügen.
  - Integrationstests werden somit wieder zu Unit Tests!
  - Es resultieren schnellere und selektivere Tests!
- Was sind Test Doubles?

## Beispiel: Test mit einer Fake-Implementation

- Für die Testausführung wird die Fake-Implementation **StringPersistorMemory** (statt **File**) verwendet, welche die Strings nur im Memory «speichert»:

```
@Test
public void testGetEmptyList() {
    final PersonPersistor instance =
        new PersonPersistor(new StringPersistorMemory());
    assertThat(instance.getList()).isEmpty();
}
```

- Somit resultiert für diesen Testfall:
  - Er hat **keine** Abhängigkeit zum Dateisystem mehr.
  - Er ist wieder ein **Unit Test** (statt Integrationstest).
  - Er testet viel weniger Dritt-Code und wird dadurch **selektiver!**

# **Effektives Testen mit Test Doubles**



# Nachteile von Fake-Implementationen beim Testen

- So elegant die Idee ist, so hat sie auch grosse Nachteile: Je besser und umfangreicher man testet, umso grösser wird die Anzahl von unterschiedlichen Fake-Implementationen.
  - Unterhalt der Fake-Implementationen erhöht aufwand.
  - Übersichtlichkeit leidet (was ist Echt und was ist Fake?).
- Die Lösung: Man kann während der Tests eine zur Laufzeit, dynamisch erstellte (!) Implementation als Platzhalter → **Test Double** einfügen.
  - Integrationstests werden somit wieder zu Unit Tests!
  - Es resultieren schnellere und selektivere Tests!
  - Wir haben nicht unzählige «Fake»-Implementation.
- Was sind Test Doubles? → siehe Input [EP 23 TestDoubles.pdf](#).

## Beispiel 2: Test mit einem Mock

- Für die Testausführung wird ein Mock (oder Spy) verwendet, welcher direkt im Testfall erzeugt und konfiguriert wird:

```
@Test
public void testGetEmptyList() {
    final StringPersistor mock =
        mock(StringPersistor.class);
    when(mock.get(0)).thenReturn(Collections.emptyList());
    final PersonPersistor instance =
        new PersonPersistor(mock);
    assertThat(instance.getList()).isEmpty();
}
```

- Der Testfall
  - Hat **keine** Abhängigkeit mehr zu einer Implementation.
  - Seine **Selektivität** ist somit **maximal!**

Hinweis: Das `mock`-Objekt ist ein Proxy (Proxy-Pattern nach GoF) und wird hier mit Mockito erzeugt.

# Live Demo's / Screencast's

- Unit Test mit (schlechter) Integration:

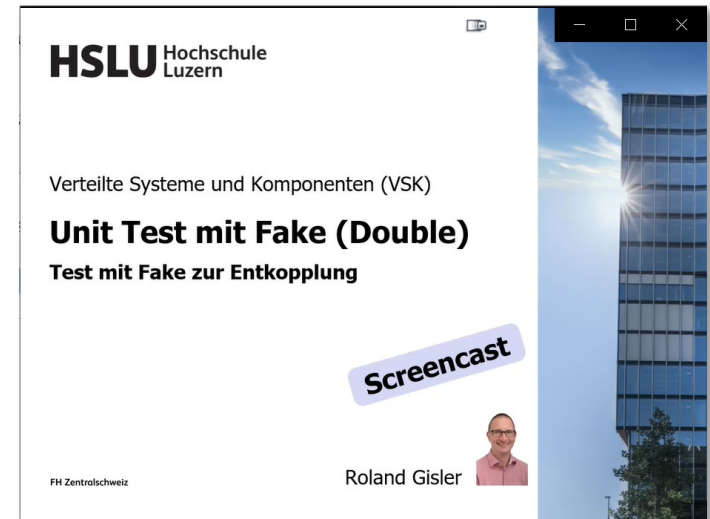
[EP 22 SC03 TestingBadIntegration.mp4](#)

- Unit Test mit einer Fake-Implementation (Test-Double):

[EP 22 SC04 TestingDoublesFake.mp4](#)

- Unit Test mit Mocking-Framework (Mockito):

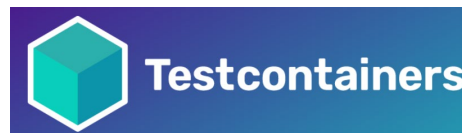
[EP 22 SC05 TestingDoublesMock.mp4](#)



# **(Integrations-)Testen mit Containern**

# Das «Leiden» bei Integrations-Test

- Selbst wenn Integrations-Tests eigentlich automatisiert sind, stellen sie uns vor grosse Probleme: Die nötigen «Umsysteme» müssen meist in mühsamer Arbeit manuell installiert und konfiguriert werden.
  - z.B. Datenbankserver, Applikationsserver, Webserver etc.
  - viel Handarbeit, fragil, grosse Fehleranfälligkeit.
  - Testdatenmanagement: Bei manchen Projekten ein sehr wichtiges (und häufig unterschätztes) Thema!
  - Dokumentationsaufwand, Versionierung!
- Eine sehr potente Lösung für einige dieser Herausforderungen: Docker und Testcontainers - <https://www.testcontainers.org/>



# Integrations-Tests mit Docker-Containern

- Die Grundidee: Wir bauen für die Tests individuelle Container, welche schnell und einfach konfiguriert, hochgefahren und danach ohne Spuren wieder abgeräumt werden können.
  - Noch dazu: Das geht alles schnell! (vgl. virtuelle Maschine)
- **Testcontainers** ist ein JUnit ergänzendes Framework, welches:
  - Eine Java-API zu vielen Docker-Aufgaben anbietet.
  - Vorbereitete Images (als Klassen!) zur Verfügung stellt.
  - Das Hoch- und Runterfahren weitgehend automatisiert.
  - Sich automatisch um das Portmapping kümmert!
- Für maximale Flexibilität können eigene Images sogar innerhalb eines Testfalles (ad-hoc) mit einer sehr eleganten Fluent-API erstellt und konfiguriert werden.
  - Vergleiche: Dynamische Mock-Erstellung und Konfiguration.

# Beispiel 1: Echo Server – Testen des Startes – 1/2

- Testcontainers erlaubt uns Container zu definieren, welche dann vollautomatisch vor jedem (!) einzelnen Test hochgefahren werden.

```
@Testcontainers
class EchoServerDefaultPortIT {

    @Container
    GenericContainer<?> echoServer
        = new GenericContainer<>(DockedImageName.parse("repo/echoserver:latest"))
            .withStartupTimeout(Duration.ofSeconds(1))
            .withExposedPorts(EchoServer.DEFAULT_PORT);

    ...
}
```

- **@Testcontainer** markiert die Testklasse für das Framework.
- **@Container** definiert einen konkreten Container (als Objekt).
  - in max. 1s muss er oben sein (auf Port reagieren), sonst failure!
- Container erhält dynamische IP und Port, stellt sicher, dass es keine Kollisionen mit bereits laufenden Containern gibt!

## Beispiel 1: Echo Server – Testen des Startes – 2/2

- Der erste Testfall prüft, ob der Server korrekt hochfährt:

```
@Test
void testServerStart() {
    assertThat(server.getLogs())
        .contains("started").contains("5555");
}
```

- Mit `getLogs()` kann man auf die Ausgaben des Containers (Container-Log) zugreifen.
- Beispiel-Ausgabe des Servers beim Start:

```
2022-10-27 18:14:29,313 INFO - EchoServer on 'Oracle Corp.' ←
started - listening on tcp://*:5555
```

➔ Somit wurde der Server offenbar korrekt gestartet.



## Beispiel 2: Test ob ein Echo kommt – 2/3

- Der zweite Testfall prüft, ob der Server ein Echo gibt (und loggt):

```
@Test
void testServerGetEchoAndLog() {
    final String url = String.format("tcp://%s:%s",
                                     echoServer.getHost(), echoServer.getFirstMappedPort());
    try ( ZMQ.Socket socket = new ZContext().createSocket(SocketType.REQ)) {
        socket.connect(url);
        socket.send("Hallo!".getBytes(ZMQ.CHARSET));
        assertEquals(new String(socket.recv(), ZMQ.CHARSET).isEqualTo("Hallo!");
    }
    assertEquals(echoServer.getLogs()).contains("Hallo!");
}
```

- `getHost()` und `getFirstMappedPort()` liefern die dem Container dynamisch zugewiesene IP bzw. Port.
- Beispiel mit ZeroMQ, prüft auf die korrekte (Echo-)Antwort, und ob das im Server auch geloggt wird.

## Beispiel 3: Individuelles Test-Image bauen(ad-hoc)

- Wir können mit Testcontainers sogar ad-hoc Images bauen, dazu steht uns eine elegante Fluent-API zur Verfügung:

```
@Container
GenericContainer<?> echoServer
    = new GenericContainer<>(new ImageFromDockerfile("ad-hoc/echoserver-temurin", false)
        .withFileFromFile("./target/echoserver.jar", new File("./target/echoserver.jar"))
        .withDockerfileFromBuilder(builder -> builder
            .from("eclipse-temurin:17.0.4.1_1-jre-alpine")
            .expose(5555)
            .workDir("/app")
            .copy("./target/echoserver.jar", "/app/")
            .cmd("java", "-jar", "echoserver.jar")
            .build()))
        .withStartupTimeout(Duration.ofSeconds(1))
        .withExposedPorts(5555);
```

- **withFileFromFile()** – Beispiel für Zugriff auf Build-Kontext
- **withDockerfileFromBuilder()** – FluentAPI mit Builder-Pattern, welches quasi die Syntax des **Dockerfile** nachbildet.
  - Ist das Elegant? 😊 Ginge natürlich auch mit einem **Dockerfile**.

# Live Demo's / Screencast's

- Integrationstests mit Testcontainers (Docker Container):

[EP 22 SC06 Testcontainers.mp4](#)



# Herausforderungen von Testcontainern

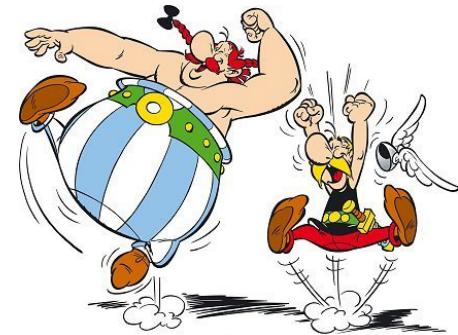
- Das Konzept ist faszinierend und hat eine grosse Mächtigkeit!
  - Es stellt aber auch grosse Ansprüche an die Infrastruktur:
    - Images müssen gespeichert werden, bzw. sollten gezielt verwaltet (und auch gelöscht!) werden.
    - Eigene Registries und Repositories werden unverzichtbar.
    - Was vorher «individuell» auf einem Rechner installiert wurde, zentralisiert sich jetzt z.B. auf einer Buildinfrastruktur.
  - Docker-in-Docker («dind») ist möglich, muss aber explizit berücksichtigt und vorgesehen werden.
- ➔ Aus diesen Gründen beschränken wird uns in VSK vorerst auf eine rein «lokale» Anwendung!
- Persönlicher Docker-Account, Ausführung «nur» auf eigenem (Entwickler\*innen-)Rechner.

# Auftrag für das Projekt

- Testen Sie!
  - Primärer Fokus: **Automatisierte** Tests!
- Versuchen Sie möglichst nach Test-First Methodik zu entwickeln.
  - Beginnen Sie bei einfachen, kleinen Klassen mit **Unit Tests**!
- Erfahren und erkennen Sie die Abgrenzung zu **Integrationstests**
  - Sie können auch Integrationstest automatisieren!
  - Dazu verwenden Sie JUnit und ggf. Testcontainer.
  - Klassen der Integrationstest enden auf **\*IT**.
  - Ausführen (nur lokal) mit dem Target: **mvn integration-test**
  - Optional: Coverage-Report erstellen mit **mvn jacoco:report**
- Optional: Lokale Versuche mit Testcontainern.

# Zusammenfassung

- Nicht nachträgliches Testen zur Fehlersuche, sondern kontinuierliches Testen zur Bestätigung, dass es funktioniert!
- Test First - Ansatz als sehr attraktive Methode aus dem XP-Ansatz.
- Messung der Code Coverage als Motivationsfaktor.
- Designregeln für gute Testbarkeit: SRP, SoC, Dependency Injection.
- Test Doubles (Mocking) um Testfälle noch stärker zu entkoppeln und mehr Unit Tests (statt Integration) machen zu können.
- Integrations-Tests mit Container-Technologien machen das (Test-)Leben noch viel spannender.



Testen macht **noch mehr** Spass!

**Fragen?**

# Quellen 1

- JUnit Testframework, <https://junit.org/junit5/>
- AssertJ, <https://assertj.github.io/doc/>
- Code Coverage Messung:
  - EclEmma, JaCoCo, <http://www.eclemma.org/jacoco/>
  - Clover, <http://www.atlassian.com/software/clover/> (komerz.)
- Mocking Frameworks, Open Source (Beispiele):
  - Mockito, <https://site.mockito.org/> - Empfehlung
  - EasyMock, <http://easymock.org/>
  - MockRunner, <http://mockrunner.github.io/>
- Docker Container:
  - Testcontainers, <https://www.testcontainers.org/>