

Enterprise Application Development with Java EE

Developer Testing



 Berner Fachhochschule
CAS Mobile Application
Development

Jonas Bandi
IvoryCode GmbH
jonas.bandi@ivorycode.com

ABOUT ME

Jonas Bandi
jonas.bandi@gmail.com
Twitter: @jbandi

- Freelancer: www.ivorycode.com
- Dozent an der Berner Fachhochschule:
Applikationsentwicklung mit JavaScript und HTML5
- In-House Kurse: Web-Technologien im Enterprise
UBS, Postfinance, Mobiliar, BIT ...



JavaScript Schulungen?
Visit: <http://ivorycode.com/#schulung>

ADMINISTRATIVES

Fragen, Diskussionen und Anregungen erwünscht!

Unterlagen

Git Repo:

<https://bitbucket.org/jonasbandi/cas-eadj-2016>

Initial Checkout:

```
git clone https://bitbucket.org/jonasbandi/cas-eadj-2016.git
```

Update:

```
git reset --hard  
git pull
```

(setzt alle lokalen
Veränderungen zurück)

Slides & Exercises: 00-CourseMaterial

Ziele des Kurses

- Betrachtung des Software Testings aus der Perspektive des Entwicklers
- Theoretischer Hintergrund zum Software Testing
- Überblick über aktuelle Werkzeuge
- Spezialitäten beim Testen von Java EE
- Konkrete Demonstration von Lösungsansätzen für das Testen einer Java EE Anwendung

Program

Testing Basics

Big Picture

Begriffserklärungen

**Automatisiertes
Testing,
Unit-Testing, Mocking,
DB-Integration**

Java EE Testing

Java Testing **JUnit**

JUnit, Hamcrest

Mockito

DB-Unit, Liquibase

Selenium

Java EE Testing 

EJB-Remote Tests

Arquillian

Was ist Testen?

Der Softwaretest ist eine analytische, dynamische Maßnahme zur Qualitätssicherung von Software.

[Wikipedia]

- Analytisch: Erst nach der Erstellung durchführbar
- Dynamisch: Die Ausführung des Codes ist notwendig

Warum Testen wir?

- Um die Qualität unserer Software zu überprüfen.

*Testing shows the presence,
not the absence of bugs.*

- Edsger W. Dijkstra (1970)

- Wir testen um Fehler zu finden.
 - Tests können nur die Anwesenheit und nicht die generelle Abwesenheit von Fehlern nachweisen!
 - Ziel: Fehlerarmut nachweisen
 - Das Ziel einer effizienten **Teststrategie** ist es, dies mit möglichst wenig Aufwand zu erreichen.

Warum schreiben wir Tests?

- Sicherstellen, dass das System korrekt funktioniert
- Abnahme durchführen
- Regression verhindern
- Refactoring ermöglichen
- Das Design verbessern
- Die Entwicklung vorantreiben
- Definition of Done
- Interaktion mit dem Kunden verbessern
- Das System dokumentieren

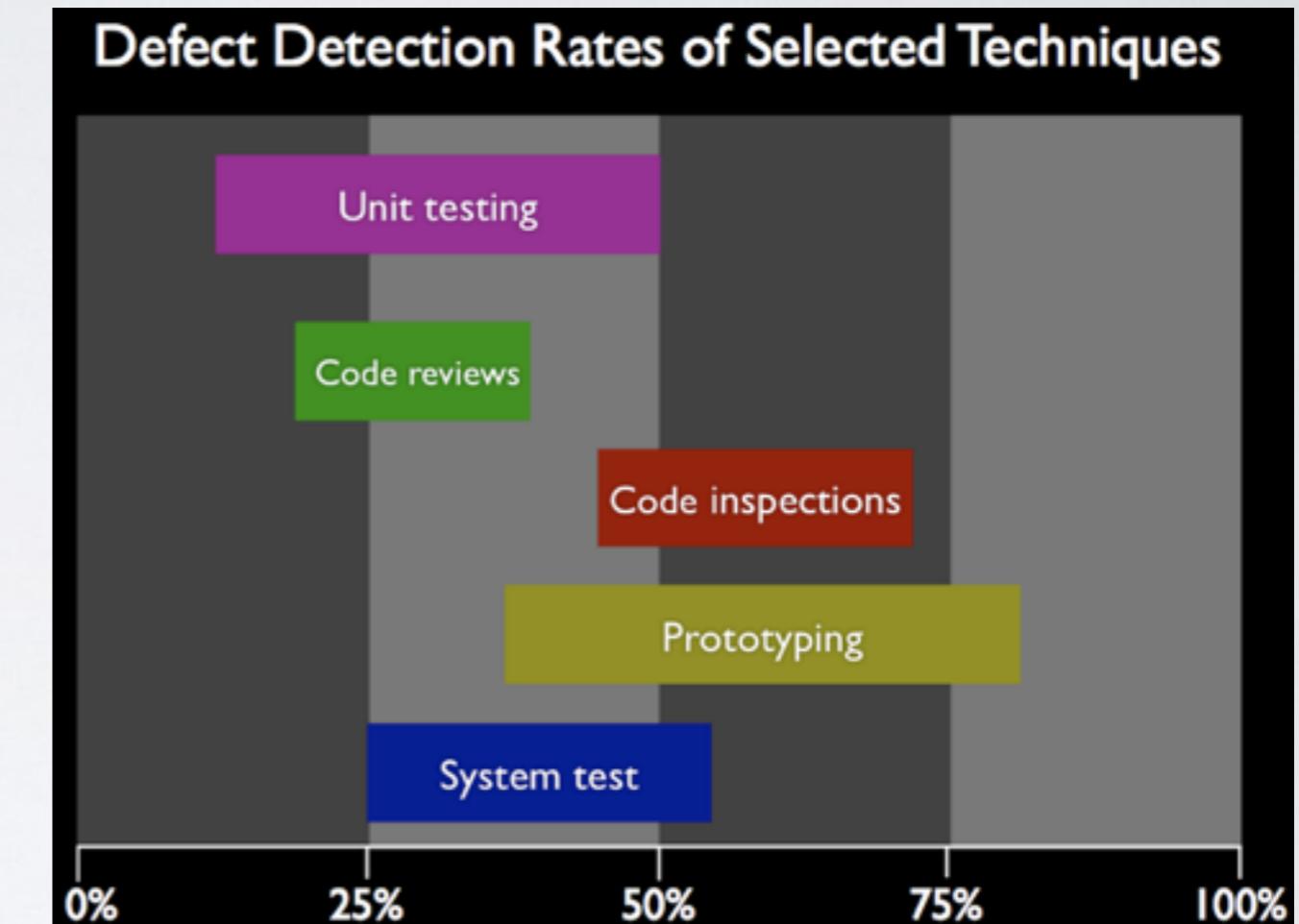
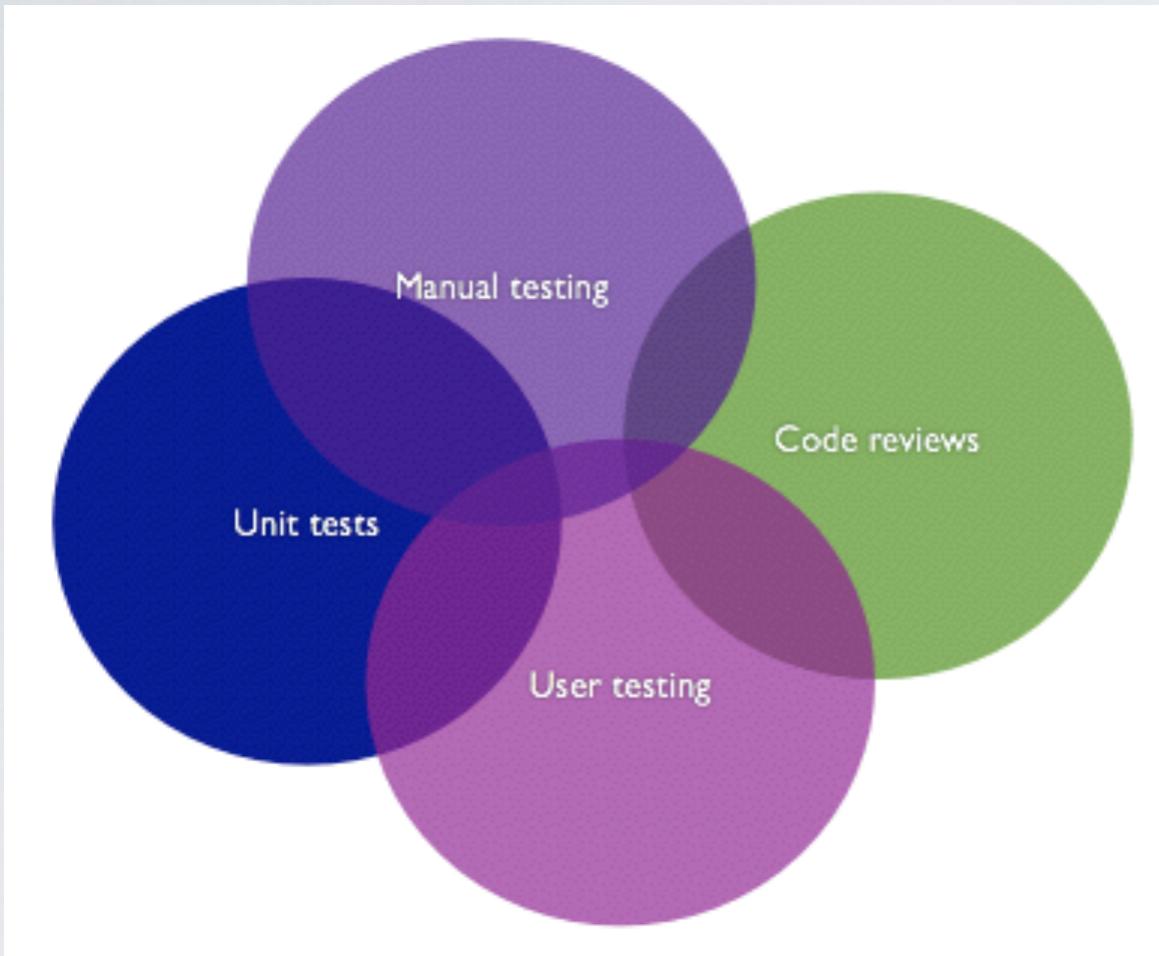
“Nothing makes a system more flexible than a comprehensive suite of tests! Far above good architecture and good design!”

- Robert C. Martin

“We have as many testers as we have developers. And testers spend all their time testing, and developers spend half their time testing. We're more of a testing, a quality software organization than we're a software organization.”

- Bill Gates

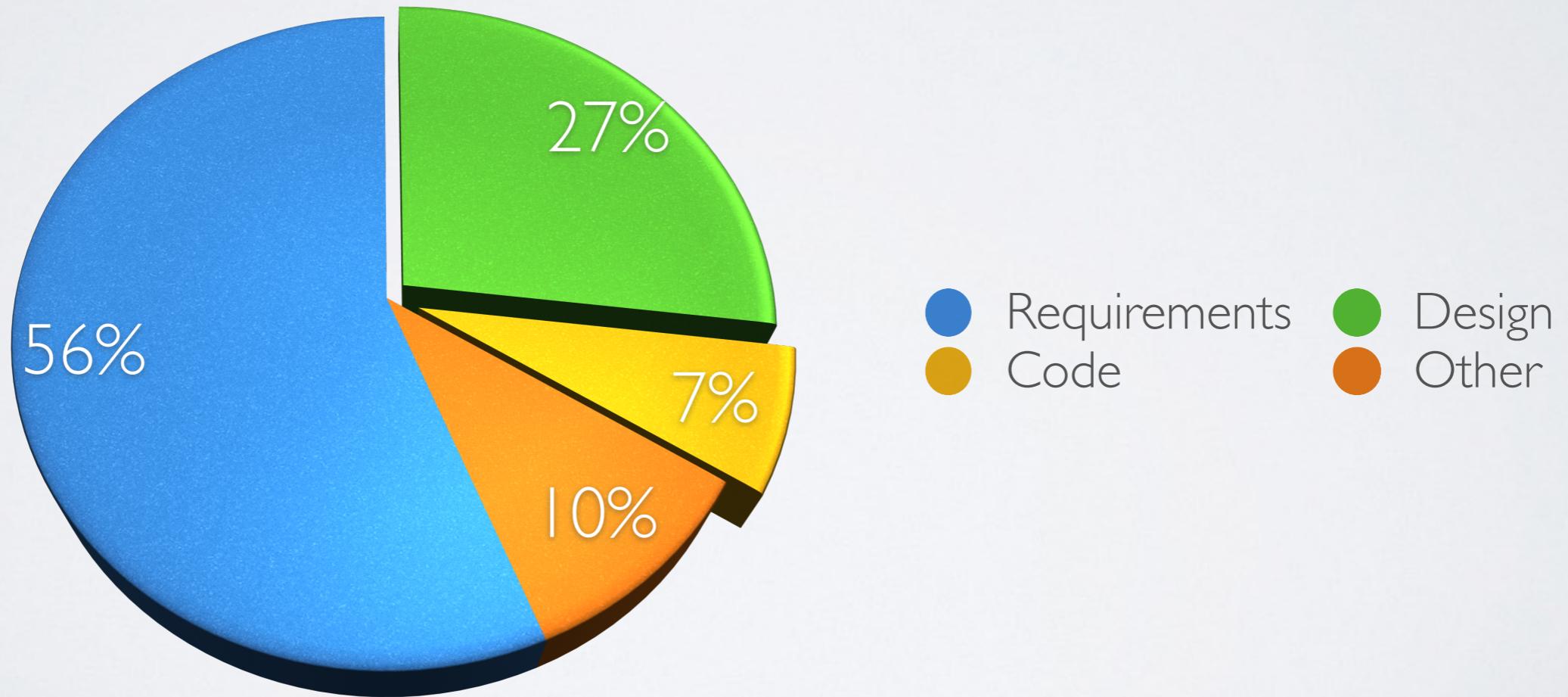
Developer Testing - the Big Picture



- Developer Testing alleine garantiert nicht hohe Qualität
- Unit Testing hat eine relativ niedrige Defect Detection Rate
- Testing is overrated - <http://railspikes.com/2008/7/11/testing-is-overrated>
- Code Complete - Steve McConnell

Developer Testing - the Big Picture

- Where errors are introduced
(CHAOS Report, Standish Group)



Much of the essence of building a program is in fact the debugging of the specification.

Fred Brooks (No Silver Bullet, 1986)

You cannot inspect quality into a product. The quality is there or it isn't by the time it's inspected.

W. Edwards Deming (Out Of The Crisis, 1982)

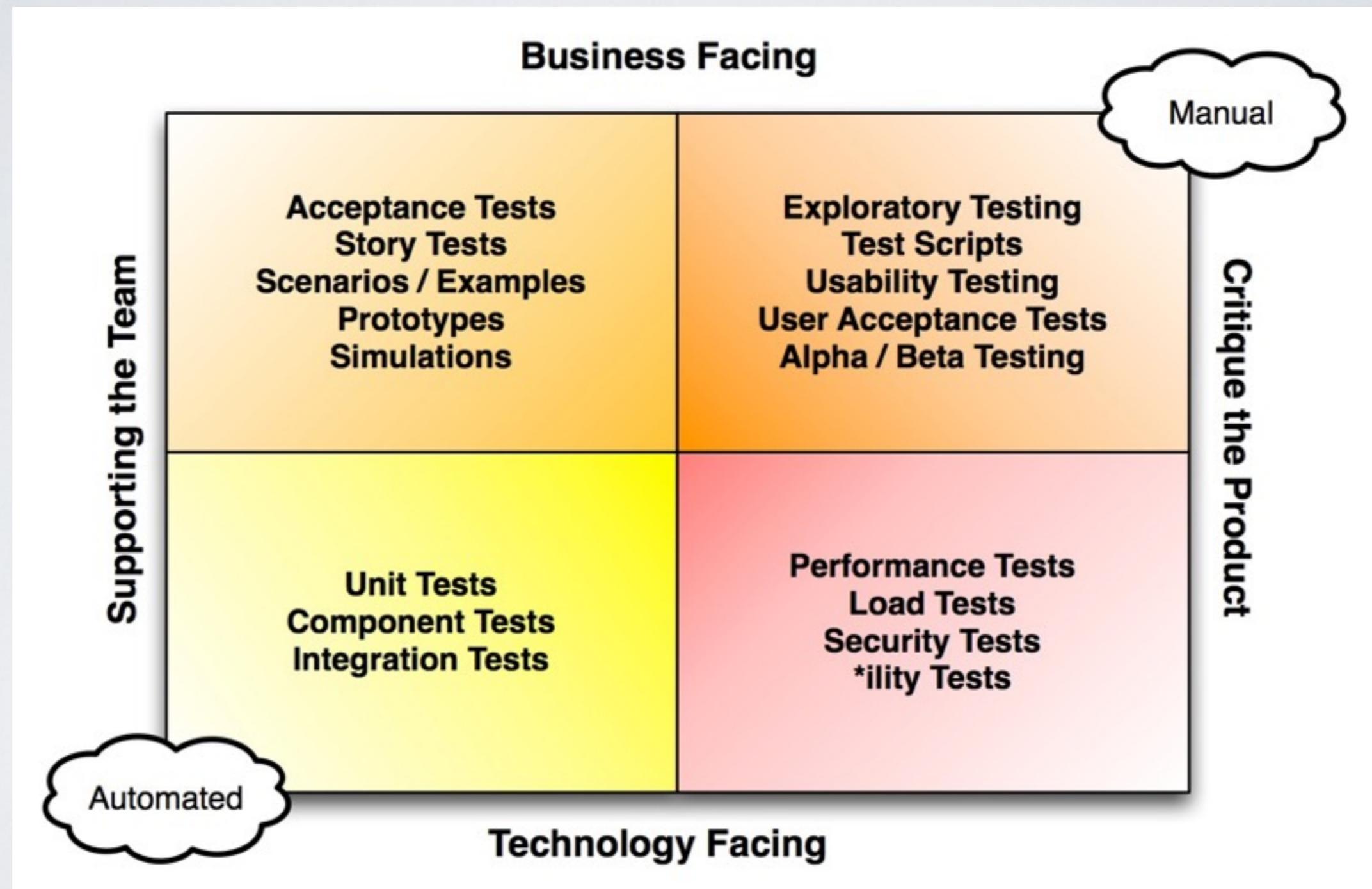
Thinking you can improve quality by doing more testing is like thinking you can loose weight by weighing you more. [...] If you want to improve your software, don't test more; develop better.

Steve McConnell (Code Complete, 1993)

Brainstorming

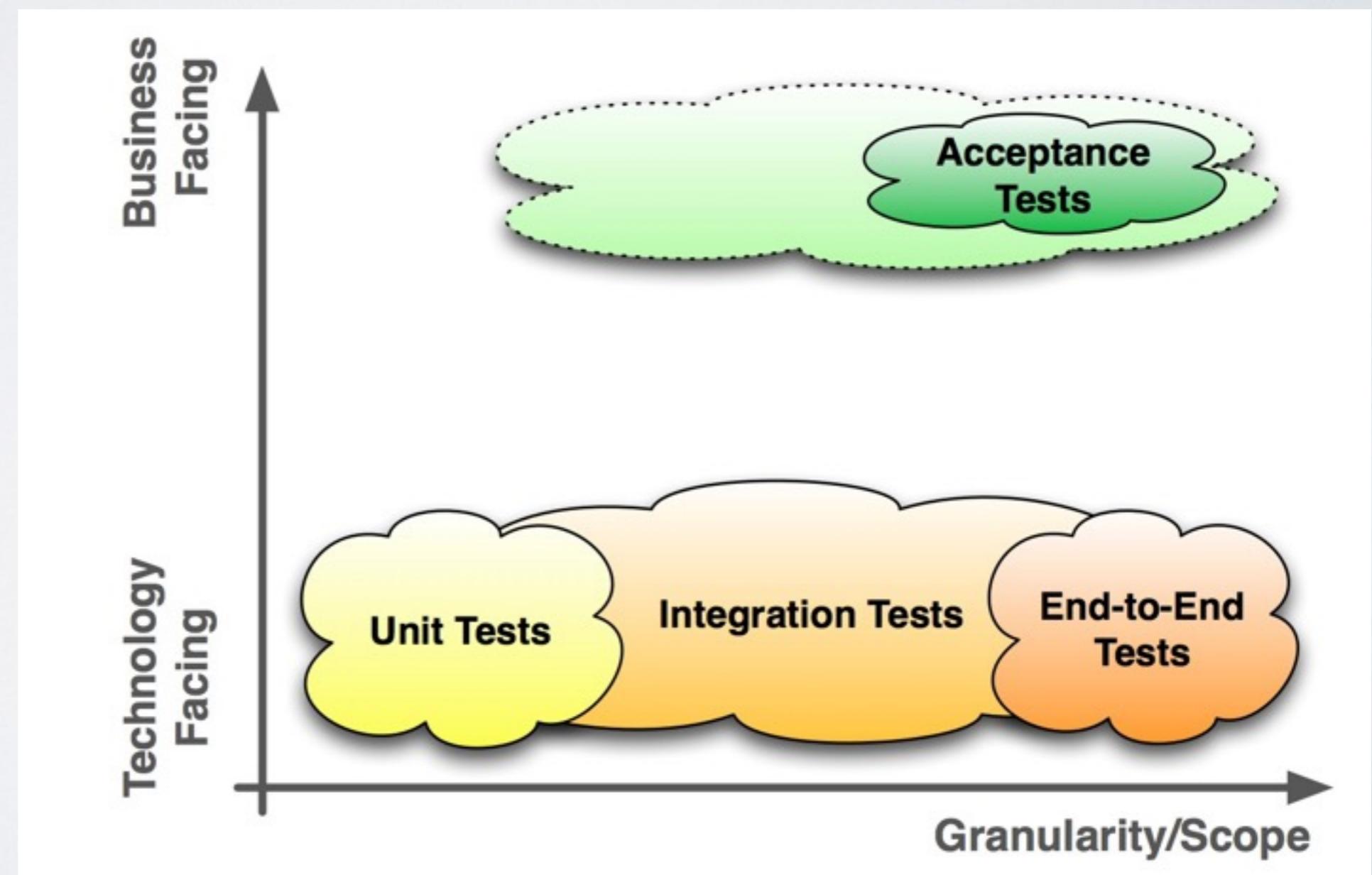
- Welche Arten von Tests kennen Sie?
Schreiben Sie diese auf.
- Wie würden Sie diese Arten von
Tests gruppieren oder klassifizieren?
- Zeit: 5 min

Testing Quadrants

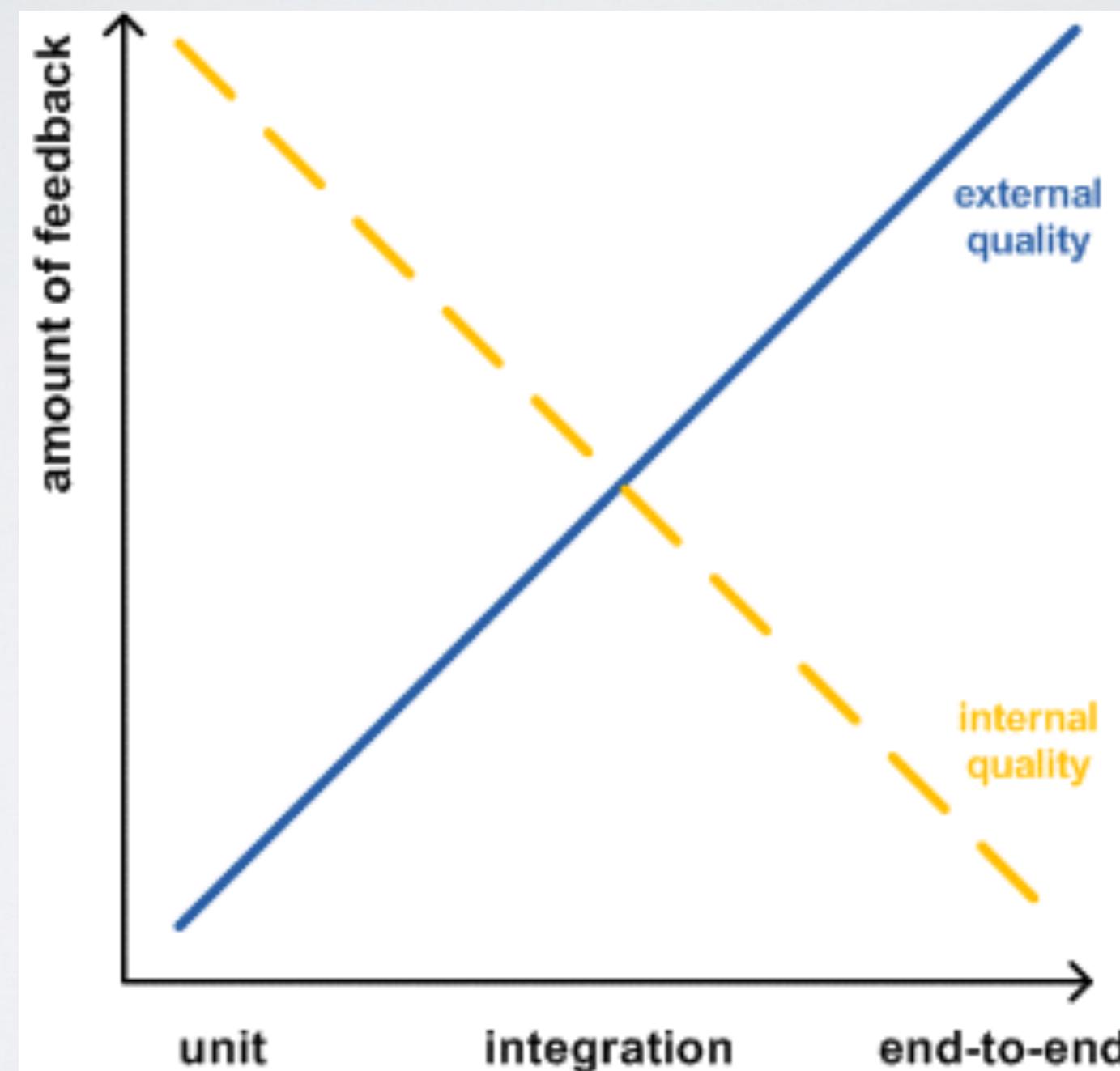


Granularity vs. Goal

- Are we building the right thing?
- Are we building it right?



Internal vs. External Quality

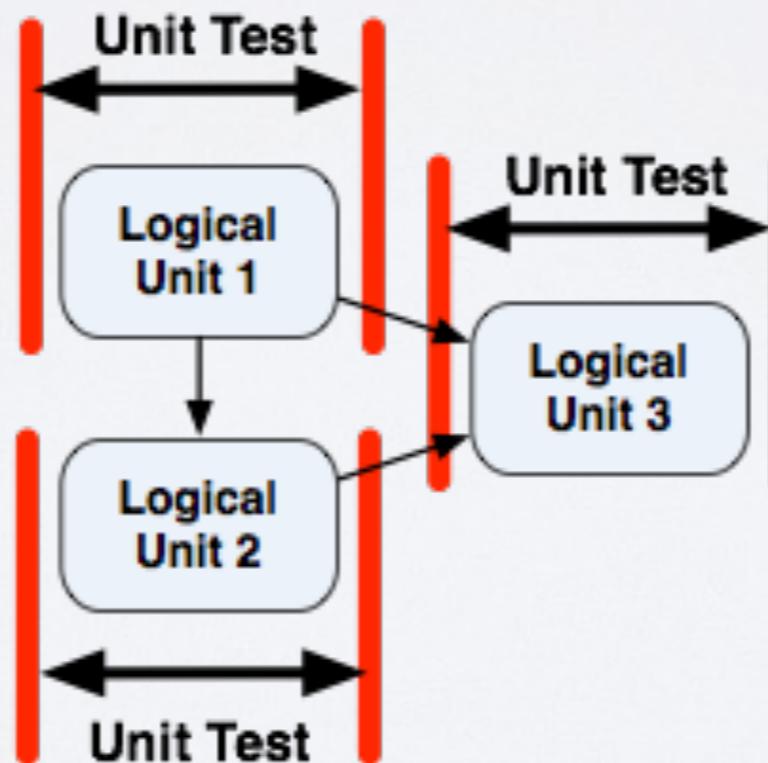


Software-Testing: Begriffserklärungen

- Unit Test
- Integration Test / Component Test
- End-to-End Test / Functional Test/ System Test
- Acceptance Test
- Regression Test
- Performance Test
- White-Box-, Black-Box-, Grey-Box- Testing

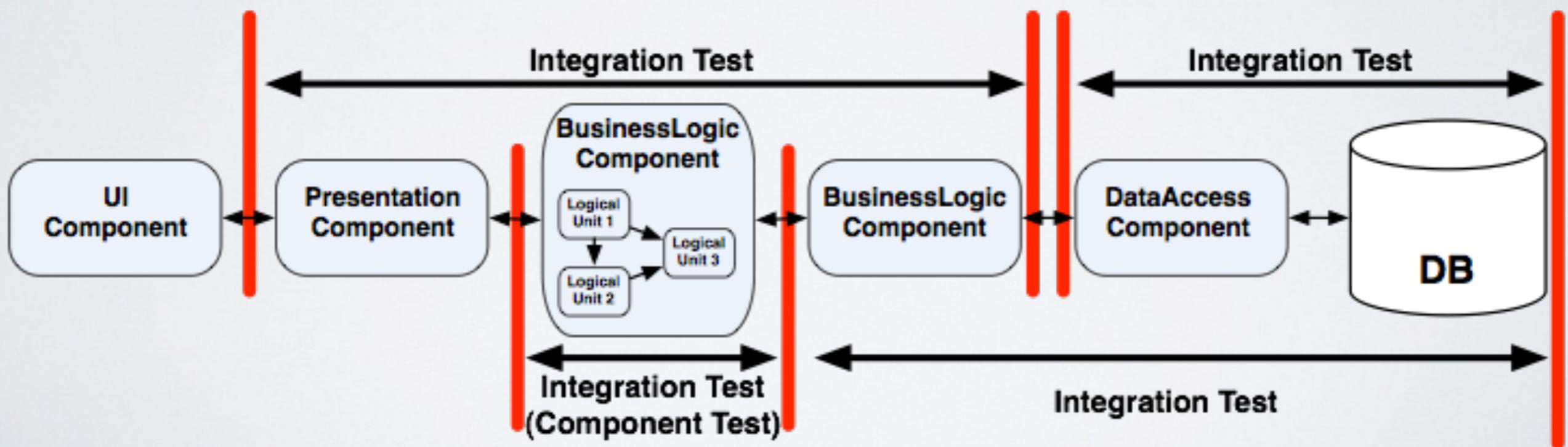
Begriffserklärungen

- Unit Test
 - Eine *einzige* logische Einheit wird *isoliert* getestet.
 - Eine *logische Einheit* ist in der Regel eine *Klasse*



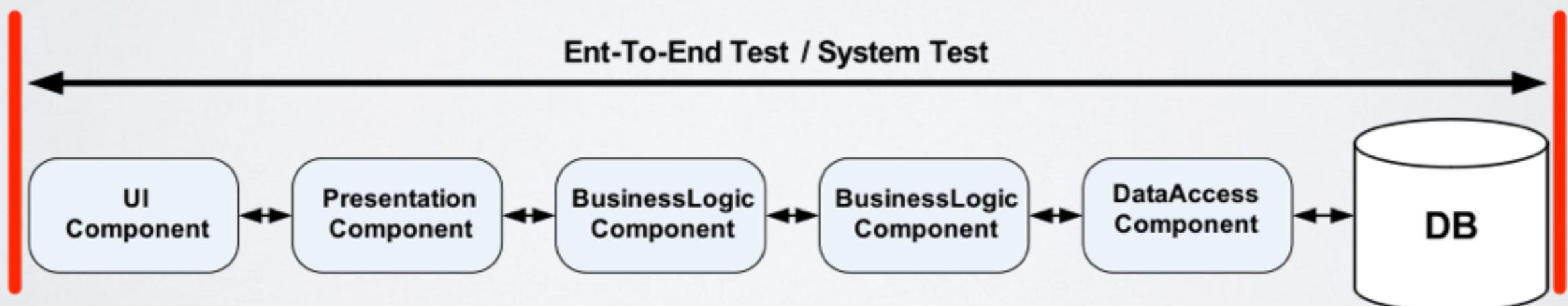
Begriffserklärungen

- Integration Test
 - Mehrere logische Einheiten werden *zusammen* getestet.
 - Somit wird auch die *Interaktion* dieser logischen Einheiten getestet



Begriffserklärungen

- End-to-End Test / Functional Test / System Test
 - Alle logischen Einheiten und ihre Interaktionen werden als Ganzes getestet



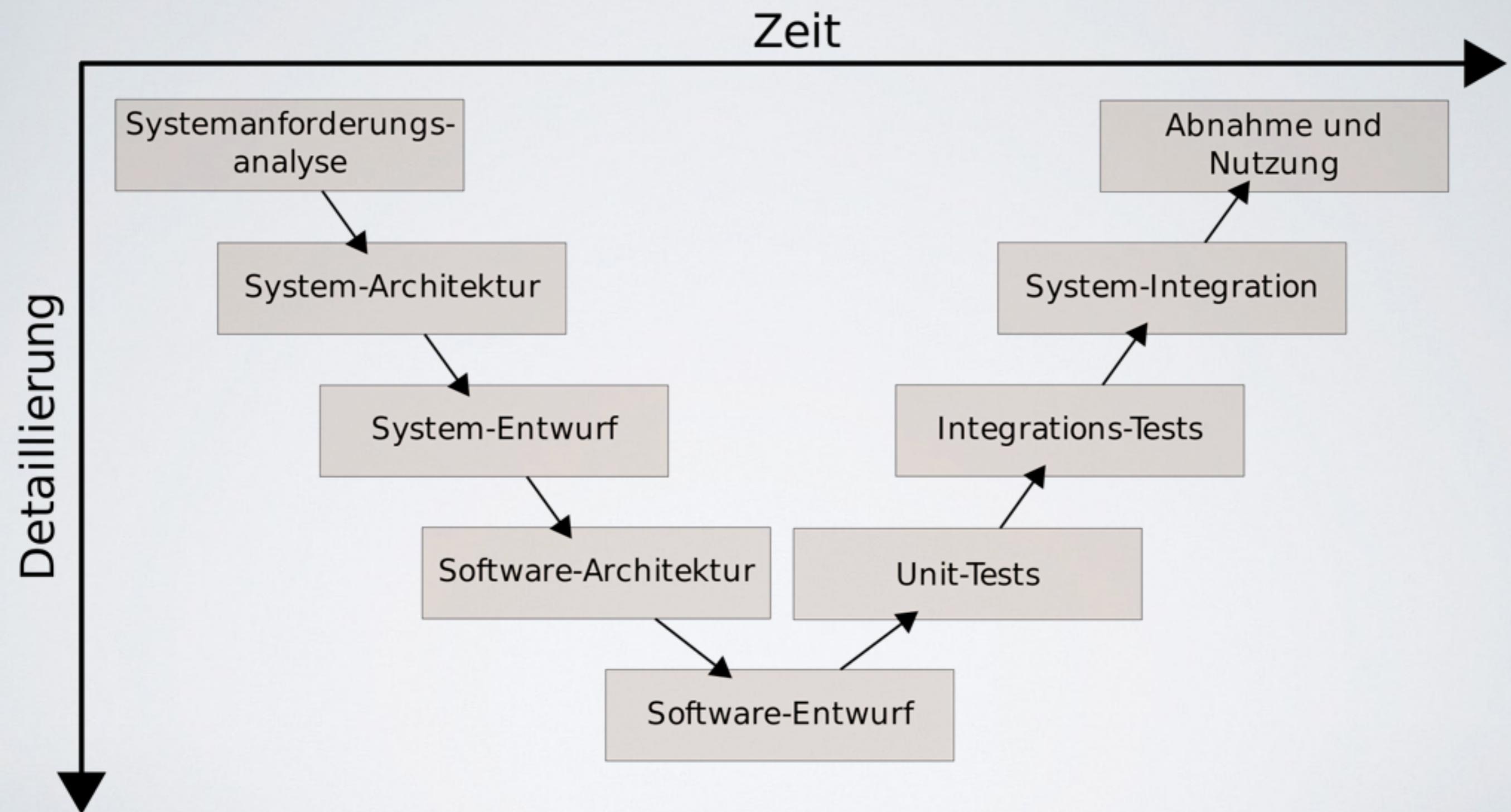
Begriffserklärungen

- Abnahme Test / Acceptance Test
 - Test der vom Benutzer geschrieben und ausgeführt wird
 - Oft die Basis für die Abnahme eines Gesamt- oder Teilsystems
 - Idealerweise werden sie auch schon früher im Entwicklungsprozess eingesetzt.

Begriffserklärungen

- Regression Test
 - Test der dazu dient, nach einer Änderung der Software die Unversehrtheit bereits getesteter Bereiche zu überprüfen.
 - Grundsätzlich ist jeder Unit-Test und Integration-Test, der regelmäßig ausgeführt wird ein Regression-Test.
 - „Das hat doch mal funktioniert...“, „Das haben wir doch schon mal gefixt“

Big-Picture: V-Model



Begriffserklärungen

- Funktionaler Test
 - Überprüft die Korrektheit einer logische Einheit
- Nicht funktionaler Test
 - Überprüft nicht funktionale Anforderungen (Performance, Verfügbarkeit...)

Black-Box-Test

- Der Test hat keinerlei Kenntnis über den inneren Aufbau der zu testenden Einheit.
- Die zu testende Einheit wird ausschliesslich über ihre spezifizierten Schnittstellen getestet.
- Werden meist verwendet um eine Spezifikation zu überprüfen.
- Werden idealerweise nicht vom Programmierer der zu testenden Einheit realisiert.

White-Box-Test

- Der Test hat Kenntnis über den inneren Aufbau der zu testenden Einheit.
- Der Test hat Zugriff auf den inneren Aufbau (Zustand) der zu testenden Einheit.
- Werden meist verwendet um eine Implementation zu überprüfen.
- Werden normalerweise vom Programmierer der zu testenden Einheit realisiert.

Grey-Box-Testing

- Der Test hat Kenntnis über den inneren Aufbau der zu testenden Einheit.
- Die zu testende Einheit wird ausschliesslich über ihre spezifizierten Schnittstellen getestet, wobei aber die Kenntnis über den inneren Aufbau ausgenutzt wird, um spezielle Situationen zu prüfen.
- Der Test kann den Kontext der zu testenden Einheit beeinflussen (z.B. Datenbank, config-files...)

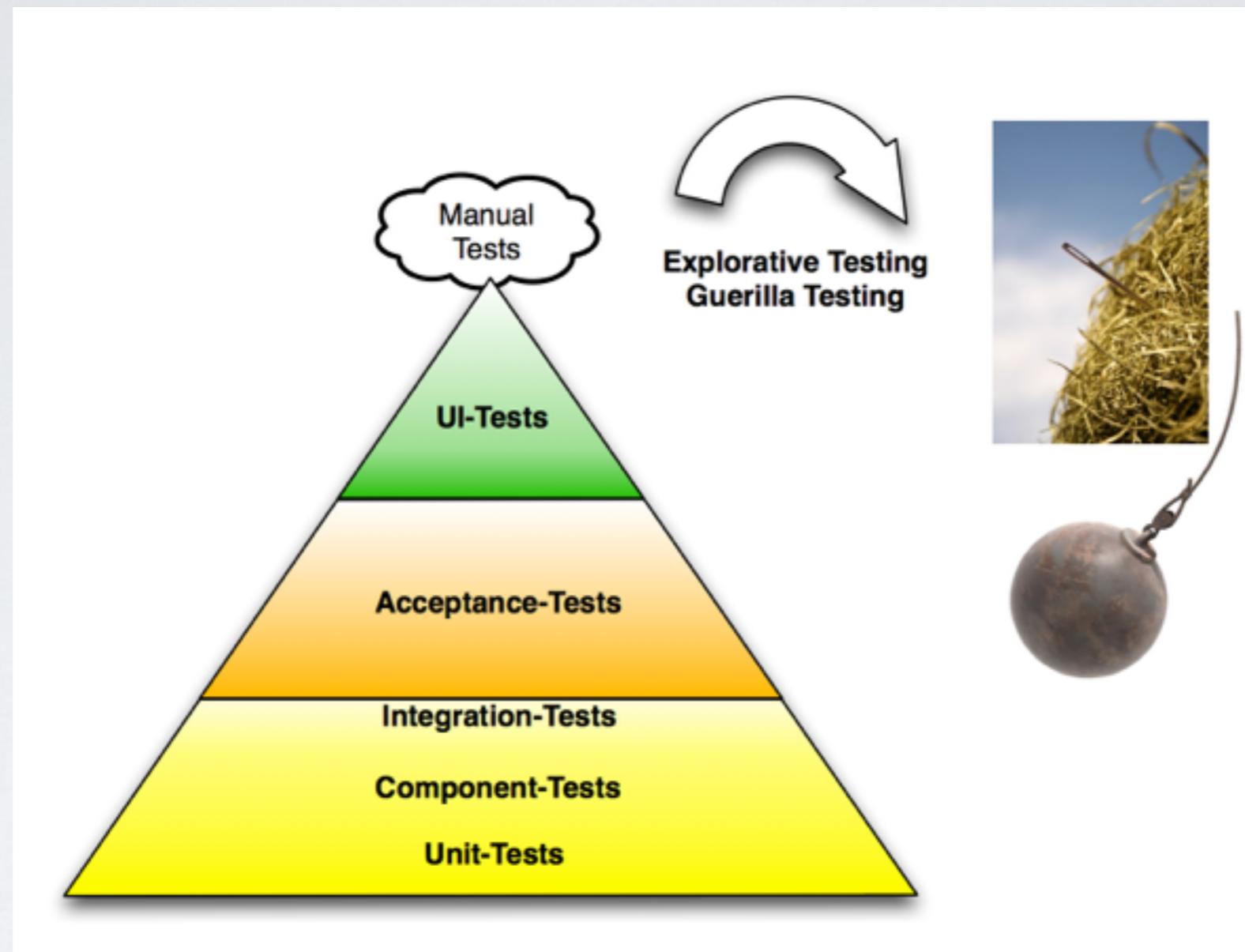
Ziele des automatisierten Testen

- Repetitive manuelle Arbeit wird vermindert
 - die Zuverlässigkeit und Reproduzierbarkeit der Tests wird erhöht
- Produktivität
 - Schneller Feedback erlaubt rasche, effiziente Iterationen
 - Bugs werden sofort erkannt und können rasch auf eine bestimmte Ursache zurückgeführt werden
- Sicherheit und Agilität
 - Unerwünschte Effekte bei Veränderungen werden sofort ersichtlich
 - Ein „Sicherheitsnetz“ erlaubt und ermutigt auch radikalere Veränderungen
- Qualität und Stabilität
 - Unerwünschte Effekte können sofort gefixt werden
- Kontrolle
 - Zustand des Systems ist jederzeit bekannt

Preis des automatisierten Testen

- Automatisiertes Testen ist aufwändig!
- Aufwand:
 - $\frac{1}{2}$ bis 2/3 der Codezeilen eines Projektes!
 - Kontinuierliche Wartung notwendig
 - Unterscheidung zwischen Programmcode und Testcode ist nicht sinnvoll
- Martin Fowler: [<http://martinfowler.com/bliki/TestCancer.html>, 6.12.07]
“Typically there are as many lines of code of tests as there are of functional code”
- Feedback aus den vergangenen Kursen:
 - 30-40% des Projektbudgets
 - 5 Zeilen Test-Code pro Zeile Applikations-Code (Medizinal-Software)

Testing Pyramide

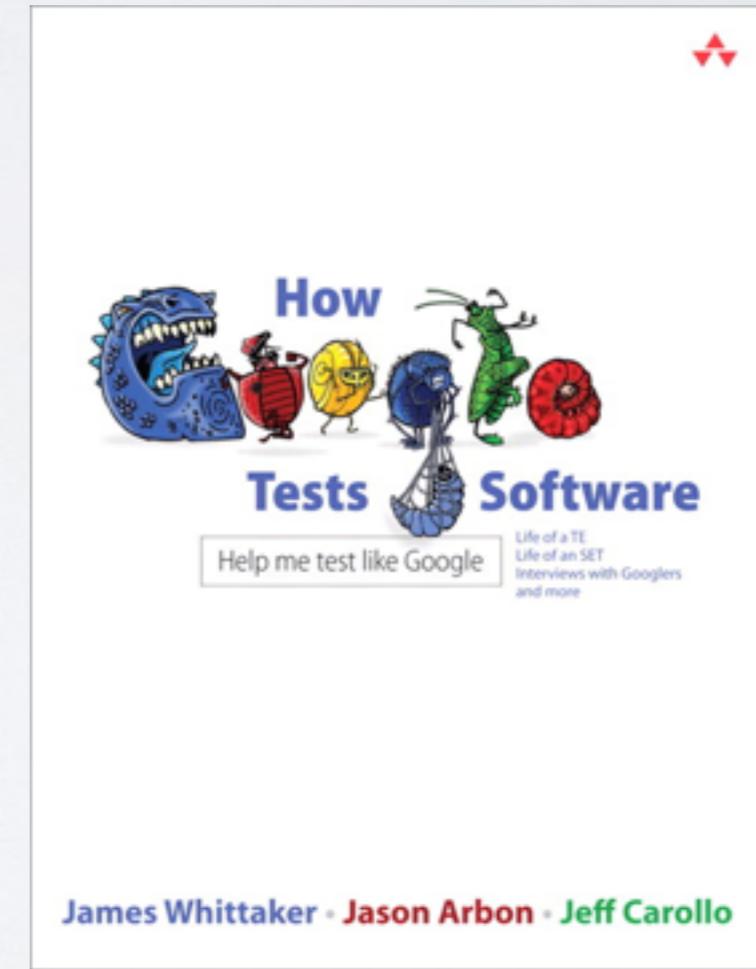


- Automatisierung ermöglicht mehr manuelles Testen (Explorative Testing, Guerilla Testing)
- Checking vs. Testing

Kritik am automatisierten Testing

- Zu aufwändig
 - Automatisiert Tests sind eine längerfristige Investition, welche bewusst geplant und gemanagt werden muss (Design, Patterns, Wartung...). Das Ziel sind nicht möglichst viele, sondern möglichst effiziente Tests d.h. eine sinnvolle Teststrategie.
- Tests kann man nicht verkaufen („Kunde kauft keine Tests“)
 - Der Kunde kauft keine Code-Zeilen
 - Softwareentwicklung ist ein kompliziertes Handwerk und Tests sind ein notwendiges Werkzeug
 - Eine nicht getestete Zeile Code ist nicht würdig verkauft zu werden...

- Tests should be made a first-class citizen and treated like any other feature.
- How Google Tests Software
<http://www.amazon.com/Google-Tests-Software-James-Whittaker/dp/0321803027>



Automated Tests are Code

Tests are important long-term artifacts of your delivery process, as important as other long-term artifacts including code. You will need to maintain them in the long term. You should write tests in a way that makes them easy to maintain, or you'll end up crying in the shower.

- Gojko Adzic

- Tests are a long-term investment!
- Expressiveness over convenience
 - Test Code is usually more concrete than production code
 - Tests are about examples
- Refactor and abstract
- Focus on what matters
- If it's hard to test that is a clue

Debugging vs. Testing

“Debugging is a skill that should not be desired!”

- Robert C. Martin

- Naiver Ansatz beim Programmieren:
 - Es kompiliert -> es läuft!
 - Sobald ein Fehler auftritt wird der Debugger gestartet...
 - Programming through debugging
 - Designing through debugging
- Tests haben den Effekt, dass Funktionalität getrennt von der Implementation betrachtet werden kann
 - Kann als Design-Werkzeug genutzt werden
 - Kann als Kontroll-Werkzeug genutzt werden

Defect Driven Testing

- Good Practice für Bug-Fixing:
 - Den Bug lokalisieren (idealerweise mit der Hilfe von Tests)
 - Schreiben eines Tests, welcher den Bug reproduziert
 - Fixing des Bugs
 - Test beweist, dass der Bug gefixt ist
 - Investition für die Zukunft („Das hatten wir doch schon mal gefixt...“)

Unit Testing

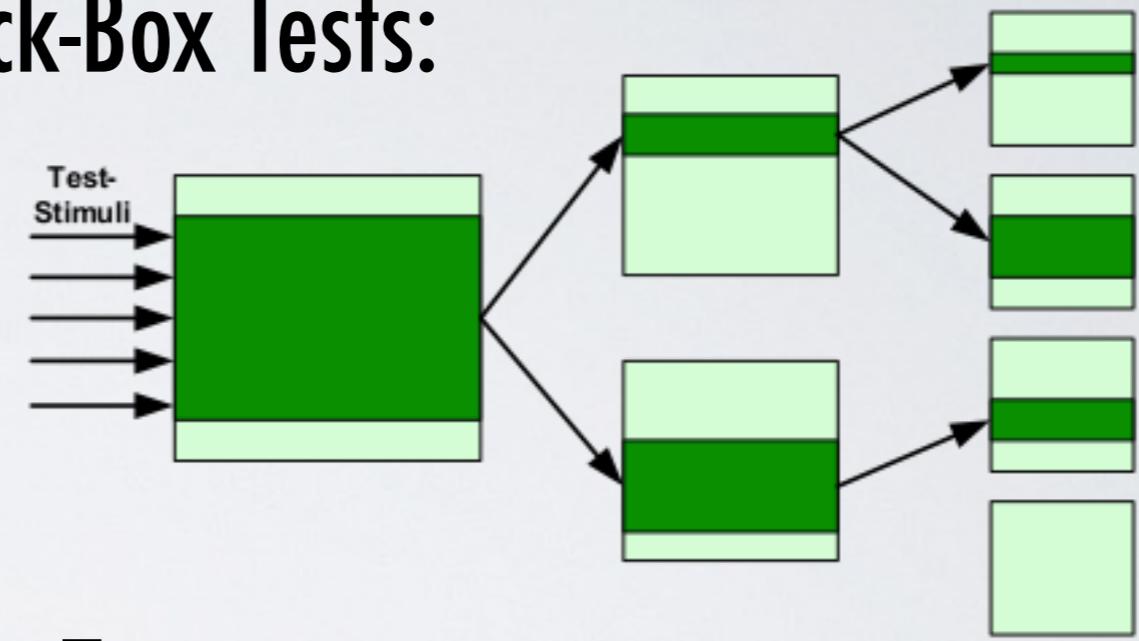
„Fail early to succeed sooner!“

- Quick Feedback
- Safety-net when doing changes
- Divide & Conquer

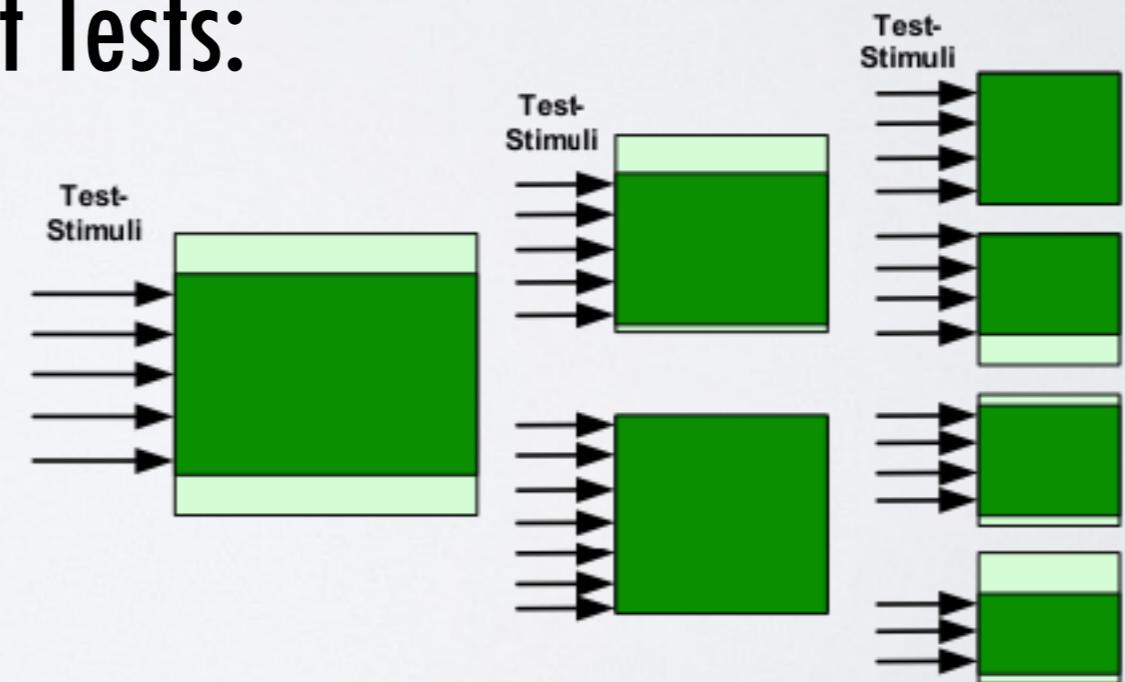
Unit-Tests im Detail

- Ziele:
 - Komponenten in Isolation testen
 - Divide & Conquer
 - Kleine überblickbare Einheiten testen
 - Gute Test-Abdeckung soll mehr Bugs aufdecken
 - Reproduzierbarkeit garantieren

Black-Box Tests:



Unit Tests:



Unit-Tests im Detail

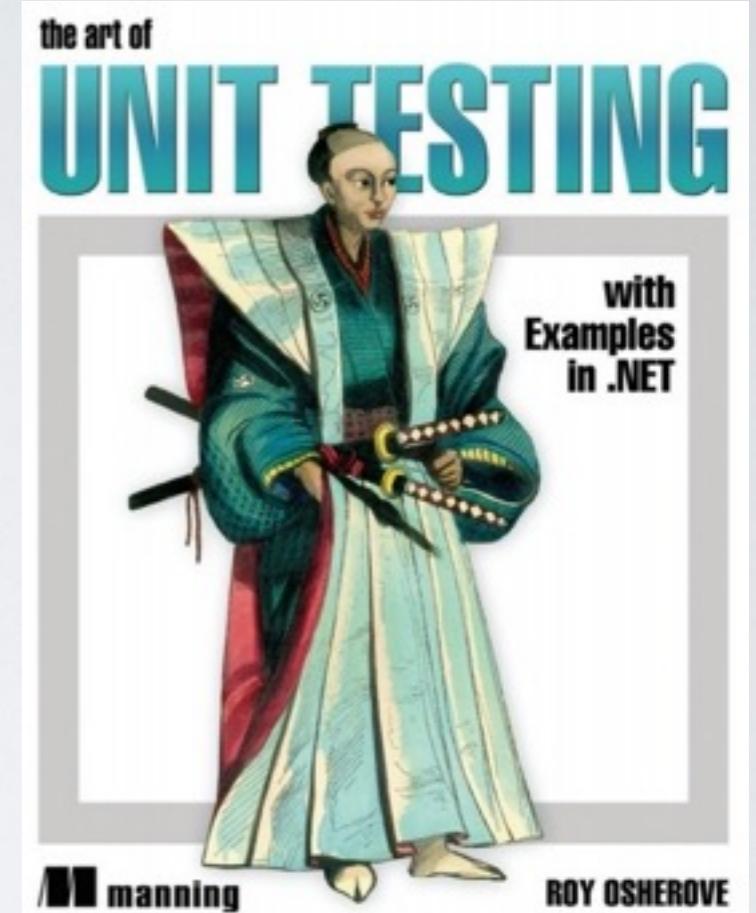
- Designwerkzeug
 - Ein Test ist eine präzise Mikro-Spezifikation
 - Fördert bewährte Design-Prinzipien (Entkopplung, Single Responsibility, Dependency Injection ...)
 - Erzwingt die Perspektive des Clients
- Entwicklungswerkzeug
 - Neuer oder veränderter Code kann rasch ausgeführt werden.
- Dokumentation
 - Anhand von Unit-Tests kann die vorgesehene Verwendung einfach nachvollzogen werden

Eigenschaften guter Unit-Tests

- Schnell
 - “A unit test that takes 1/10 of a second to run is a slow unit test” (“Working Effectively with Legacy Code”, Michael C. Feathers)
- Unterstützen die Lokalisierung eines spezifischen Problems (Feedback)
 - Was bedeutet ein Fehlschlagen des Tests?
- Lesbar
- Zuverlässig
 - deterministisch, wiederholbar

Unit-Test Definition

- A unit test is a fast, in-memory, consistent, automated and repeatable test of a functional unit-of-work in the system.
- A test is not a unit test if:
 - It talks to the database
 - It communicates across the network
 - It touches the file system
 - It can't run at the same time as any of your other unit tests
 - You have to do special things to your environment (such as editing config files) to run it.



Probleme mit Unit-Tests

- Not reflecting quality of the whole system
- Maintenance
- Keeping them small and decoupled
- Keeping them fast
- Tests have bugs
- You can't test what is missing

Unit-Test Frameworks

- Technisches Hilfsmittel, welches das isolierte Testen einer logischen Einheit erleichtert.
- Ermöglicht es die Tests einfach wiederholbar und automatisierbar zu machen.
- Es können sowohl Unit-Tests wie auch Integration-Tests implementiert werden.
- Unit-Test-Frameworks gibt es für alle erdenklichen Sprachen
 - http://en.wikipedia.org/wiki/List_of_unit_testing_frameworks

JUnit

```
public class SimpleTest
{
    @Test
    public void testMultiplication()
    {
        assertEquals ("Multiplication", 6, 3*2);
    }
}
```

Rund um JUnit hat sich ein sehr reiches Ökosystem entwickelt.

@RunWith: JUnit 4 ermöglicht anderen Frameworks die Kontrolle über die Testausführung zu übernehmen.



JUnit Lambda

- Nächste Generation von JUnit
- Basierend auf Java 8

<http://junit-team.github.io/junit5/>

Hamcrest

A set of matchers to make test expectations/verifications more expressive and declarative.

```
assertThat(app.greet(), is(anyOf(  
    notNullValue(),  
    instanceOf(Integer.class))))  
);
```

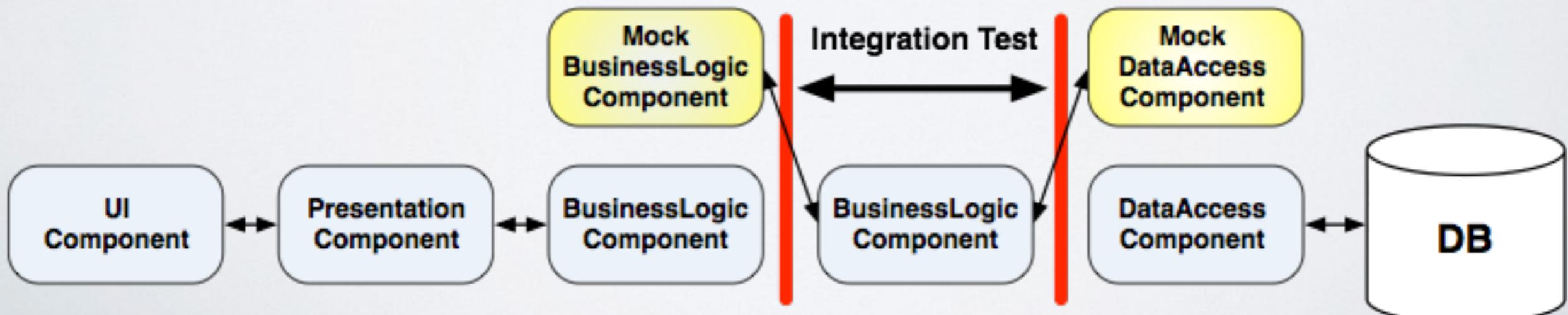
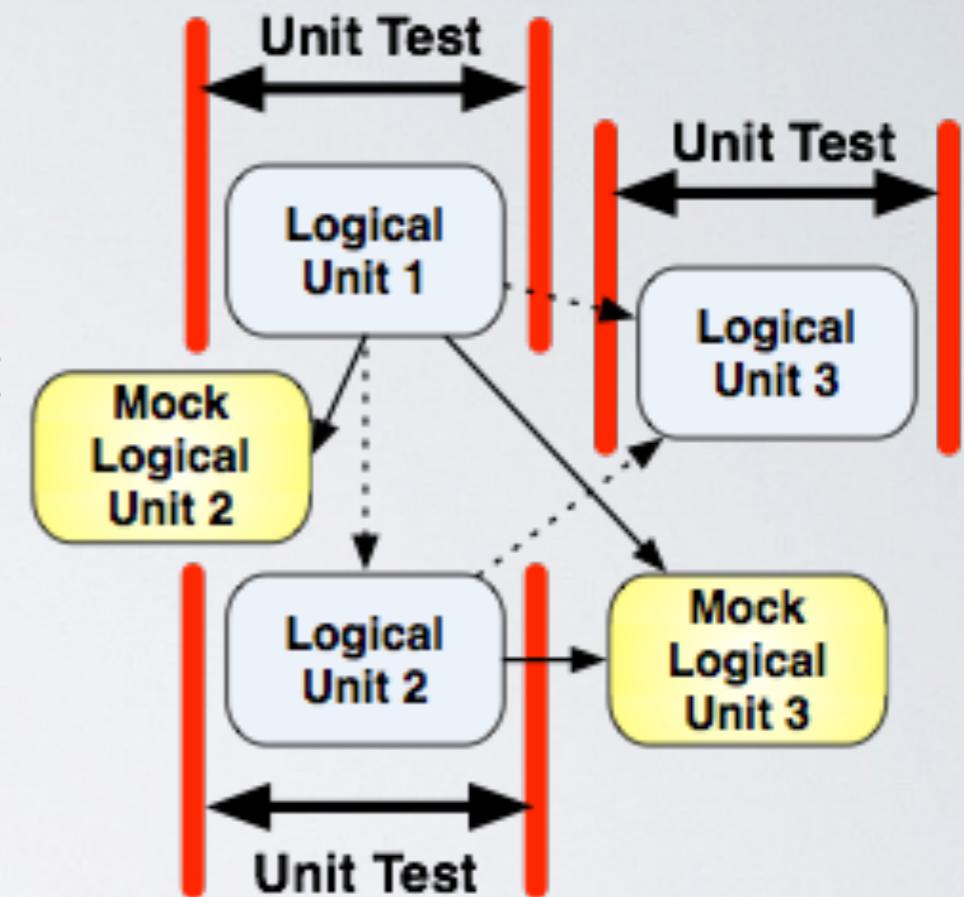
Unit Test Gliederung

AAA - Arrange Act Assert

- Arrange: der Kontext des Tests wird erstellt
- Act: der eigentliche Test wird ausgeführt
- Assert: das erwartete Verhalten wird verifiziert

Mocking

- Eine Schwierigkeit von Unit-Testing ist es, eine Unit isoliert zu testen.
 - Eine Unit ist meist in einem Umfeld eingebettet, mit welchem sie interagiert
- Das Ziel von Mocking ist es, das Umfeld einer Unit zu simulieren, so dass die Unit isoliert getestet werden kann.



Mocking: Test Doubles

- Test Doubles sind Objekte, die in einem Test eingesetzt werden, um die Umgebung des zu testenden Objekts zu simulieren
- Dummy
 - Ist nur ein Platzhalter und wird im Test nie wirklich verwendet.
- Fake
 - Hat eine funktionstüchtige Implementierung, die aber stark vereinfacht ist (z.B. In-Memory-DB)
- Stub
 - Interagiert mit dem zu testende Objekt, die Interaktion wird aber nicht direkt überprüft um den Testausgang zu bestimmen
- Mock
 - Erwartet und überprüft eine klar definierte Interaktion mit dem zu testenden Objekt
- Test Spy
 - Interaktionen mit dem zu testenden Objekt werden vom Spy festgehalten und können nachträglich inspiziert werden.

Mocking Frameworks

- Erlauben das einfache Erstellen von Mock- resp. Stub-Objekten
- Die Mock- und Stub-Objekte werden aufgrund bestehender Interfaces oder Klassen definiert.
- Die erwartete Interaktion mit den Mock- und Stub-Objekten und deren Verhalten wird programmatisch definiert, bevor der eigentliche Test startet.
- Bekannte Java Frameworks:
JMock, EasyMock, Mockito, Jmockit, PowerMock



```
@Test
public void verifyMock() {

    // mock creation
    List mockedList = mock(List.class);

    // using mock object
    mockedList.add("one");
    mockedList.clear();

    // verification
    verify(mockedList).add("one");
    verify(mockedList).clear();
}
```

Mocking Gliederung

- Klassische Mocks: Record - Replay - Verify
 - Record: erwartetes Verhalten wird spezifiziert
 - Replay: der eigentliche Test wird ausgeführt
 - Verify: das erwartete Verhalten wird überprüft
- Mit Test-Spys
 - Arrange: Spys erstellen
 - Act: Spys benutzen
 - Assert: Erwartete Interaktionen spezifizieren und überprüfen

Diskussion: Stubbing vs. Mocking

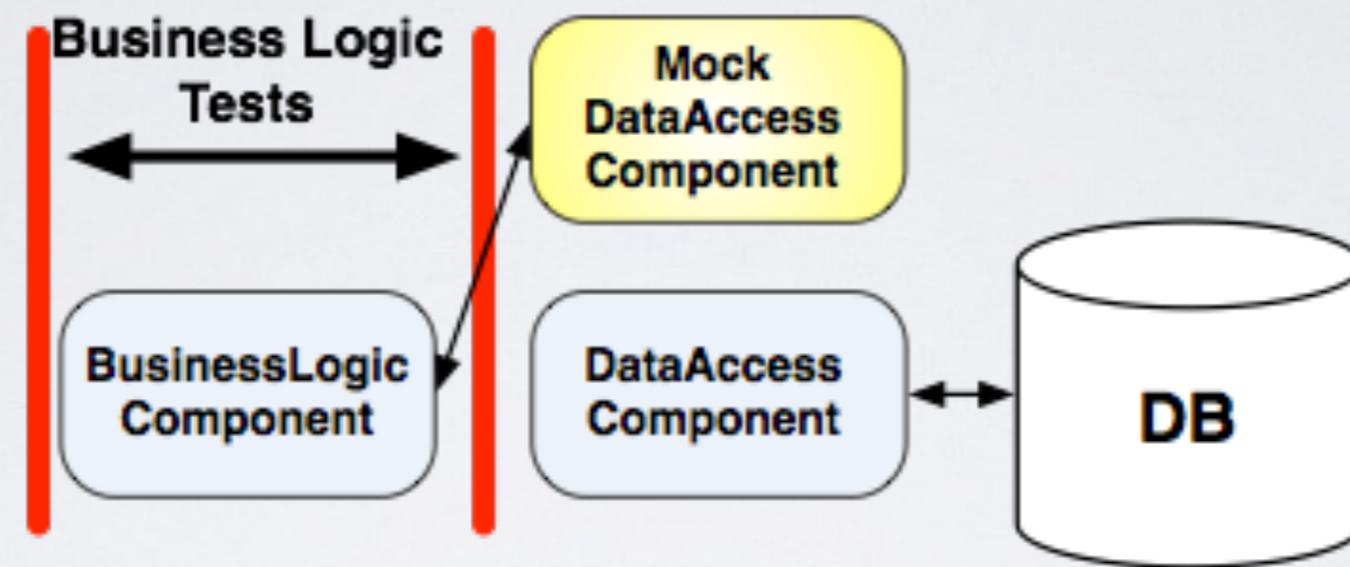
- Wann sollte Stubbing eingesetzt werden, wann sollte Mocking eingesetzt werden?
- Was sind die Konsequenzen?

Was ist mit Infrastruktur?

- Vermeide das Testen von Infrastruktur in Unit Tests!
 - Infrastruktur sollte getestet sein
 - Tests werden komplex und unleserlich
 - Tests werden meist langsam
- Vermeide das Mocken von Infrastruktur!
 - Komplexität: Man programmiert die Infrastruktur erneut!
- Wichtig: Trennung von Businesslogik und Infrastruktur!
- Design for Testability

DB Integration Testing

- Mocking der DB soweit möglich



- Einbindung der DB
 - DB in einen definierten Zustand bringen
 - Tests durchführen
 - Veränderungen verifizieren
 - (Aufräumen)

DB Integration Testing: Patterns

- **Sandboxes:** Verwenden von verschiedenen DBs
 - Jeder Entwickler hat eine eigene (lokale) DB mit Testbeständen
 - Entwicklungsdatenbank mit realistischen Datenbeständen
 - Integrationsdatenbank / Testdatenbank: Letzter Schritt vor der produktiven Datenbank (z.B. für Abnahmetest)
 - Aber: Das Managen verschiedener Datenbanken und Datenbestände ist aufwändig! Gute Versionisierungsstrategie nötig!
- **Table Truncation Setup/Teardown:** Alle beteiligten Tabellen werden vor/nach jedem Test geleert. Das Test-Setup ist verantwortlich, dass gültige Daten vorhanden sind.
- **Transaction Rollback Teardown:** Der gesamte Test läuft in einer Transaktion. Am Schluss wird ein Rollback durchgeführt.
 - Trennung von Transaktionssteuerung und Business-Logik (-> Design for Testability)

DB Integration Testing: Probleme

- Probleme:
 - Schema-Management
 - Daten-Management
 - Geschwindigkeit: DB-Zugriffe sind um Faktoren langsamer als In-Memory-Calls.

DBUnit

- <http://www.dbunit.org/>
- Ermöglicht einfaches exportieren und importieren von Daten in die DB und aus der DB.
- Daten werden in XML-Files gespeichert.
- Die DB kann einfach vor jedem Test in einen definierten Ausgangszustand gebracht werden.
- Datenbestände können (einfach) verglichen werden.

Alternative: Liquibase <http://liquibase.org/>

In-Memory Datenbanken

- Derby (>10.5.1.1)
- HSQLDB
- H2
- SQLite

`jdbc:derby:memory:musicstoredb;create=true`

- + Schnell
- + Keine Installation
- + Keine Seiteneffekte
- Automatisiertes Einspielen von Schema und Datenbeständen notwendig
- +/- Schema immer aktuell

Übung 1: POJO Testing

Brainstorming

- Welche Schwierigkeiten gibt es beim Java EE Testing? Schreiben Sie diese auf.
- Zeit: 5 min

Strategien zum Testen von Java EE

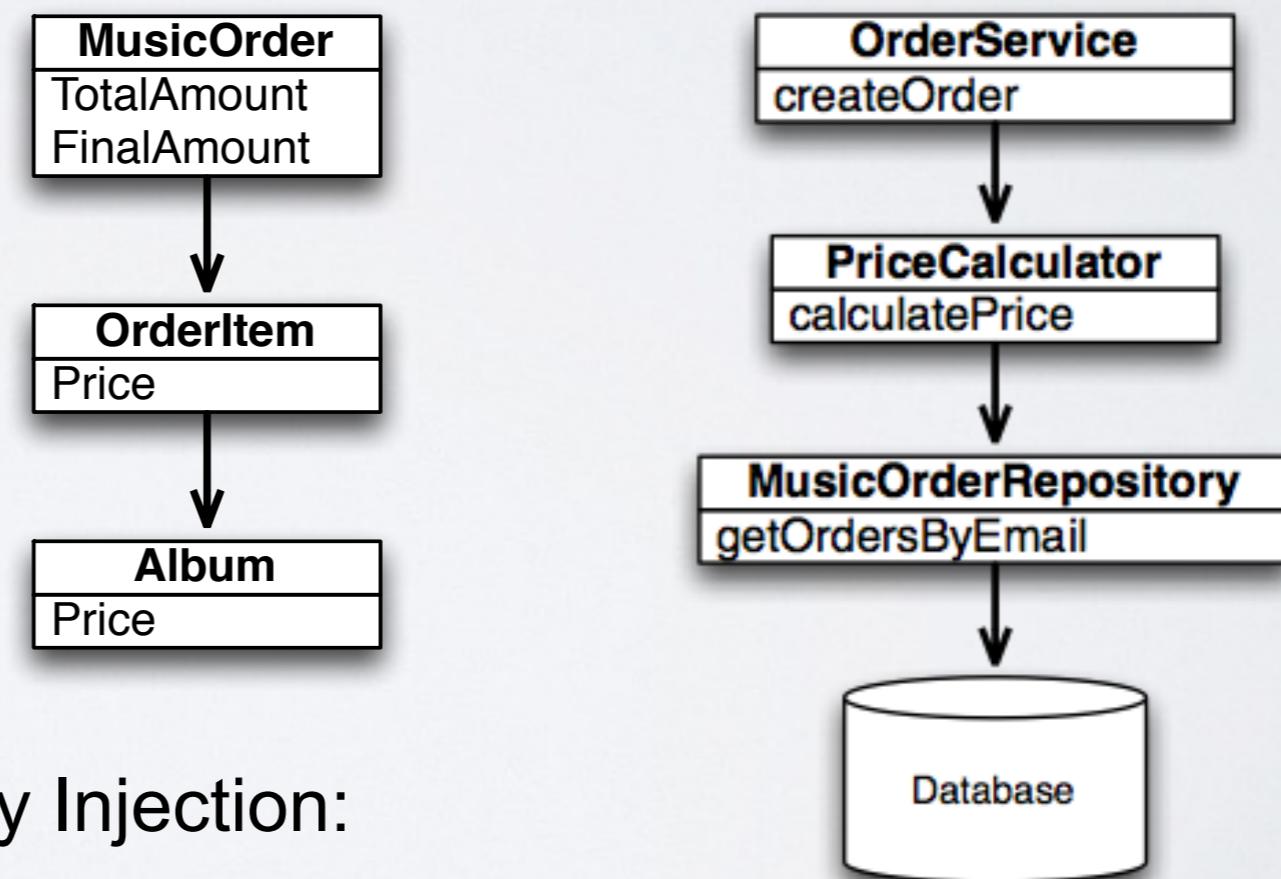
- **Teststrategie:** Welche Fehler sollen von welchen Tests gefunden werden.
- Typische Strategien:
 - Unit-Tests mit Mocking für Business-Logik
 - Kann sich über mehrere Komponenten erstrecken
 - Integrationstests Data-Access
 - In-Memory DB
 - “Reale” DB
 - Integrationstests Businesslogik In-Container
 - Test-Client ausserhalb des Containers
 - Test-Client innerhalb des Containers
 - Integrations/Acceptance-Tests der Applikation über den Presentation-Layer

Unit-Tests für Business-Logik

- Ziel: Korrektheit der Businesslogik sicherstellen
- EJBs sind grundätzlich POJOs
- Trennung von Businesslogik und Infrastruktur
- Delegation von Businesslogik an reine POJOs
- Design for Testability
- Mocking einsetzen um Abhängigkeiten zu durchbrechen.

Unit-Tests für Business-Logik

- Beispiel Musicstore:
 - Entities: MusicOrder-OrderItem-Album
 - Services: OrderManager, PriceCalculator



Libraries helfen bei Dependency Injection:

<http://jglue.org/cdi-unit/>

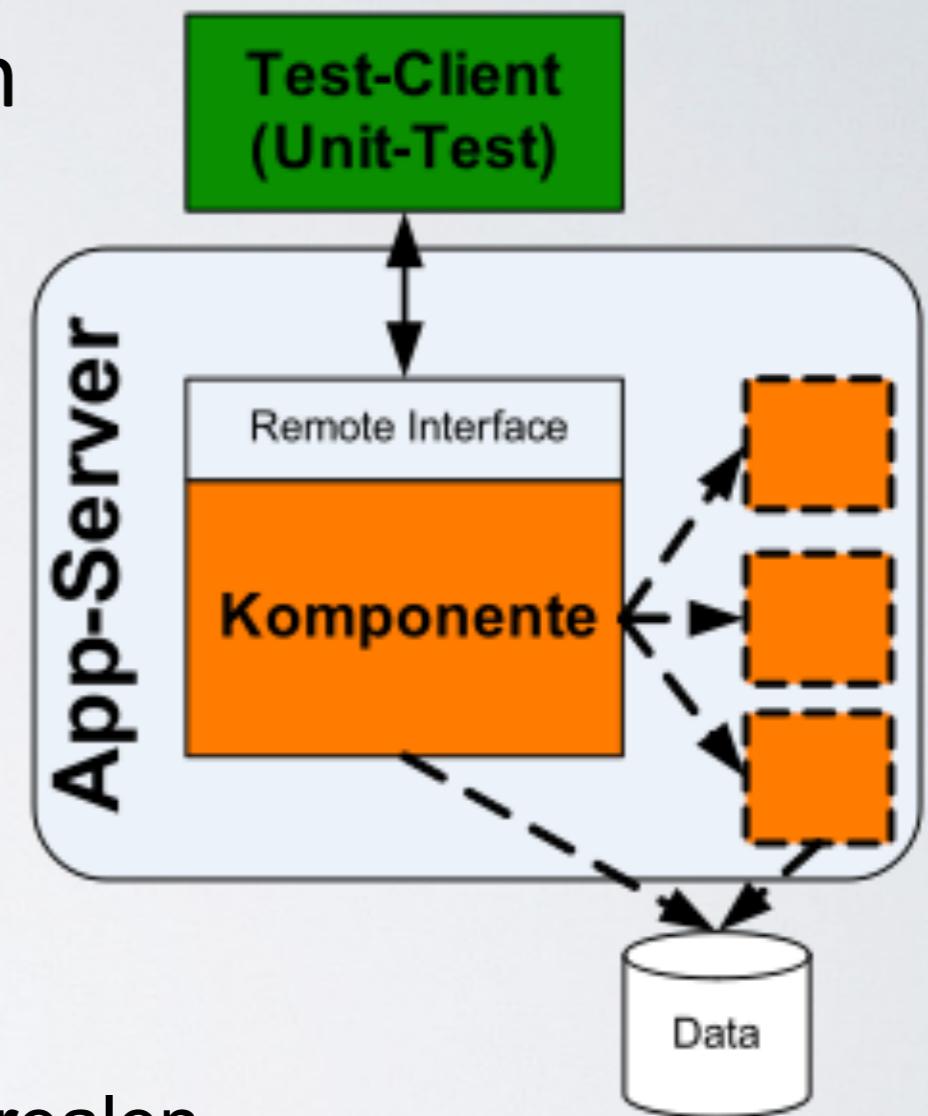
<https://easygloss.dev.java.net/>

Integrationstests Data-Access

- Ziel:
 - Fachliche Korrektheit der Queries sicherstellen
 - Technisch korrekte Anbindung an die DB sicherstellen
 - Fehler im Mapping aufdecken
- Benötigen Data-Access-Infrastruktur und Datenbank
- Können ausserhalb des Containers laufen

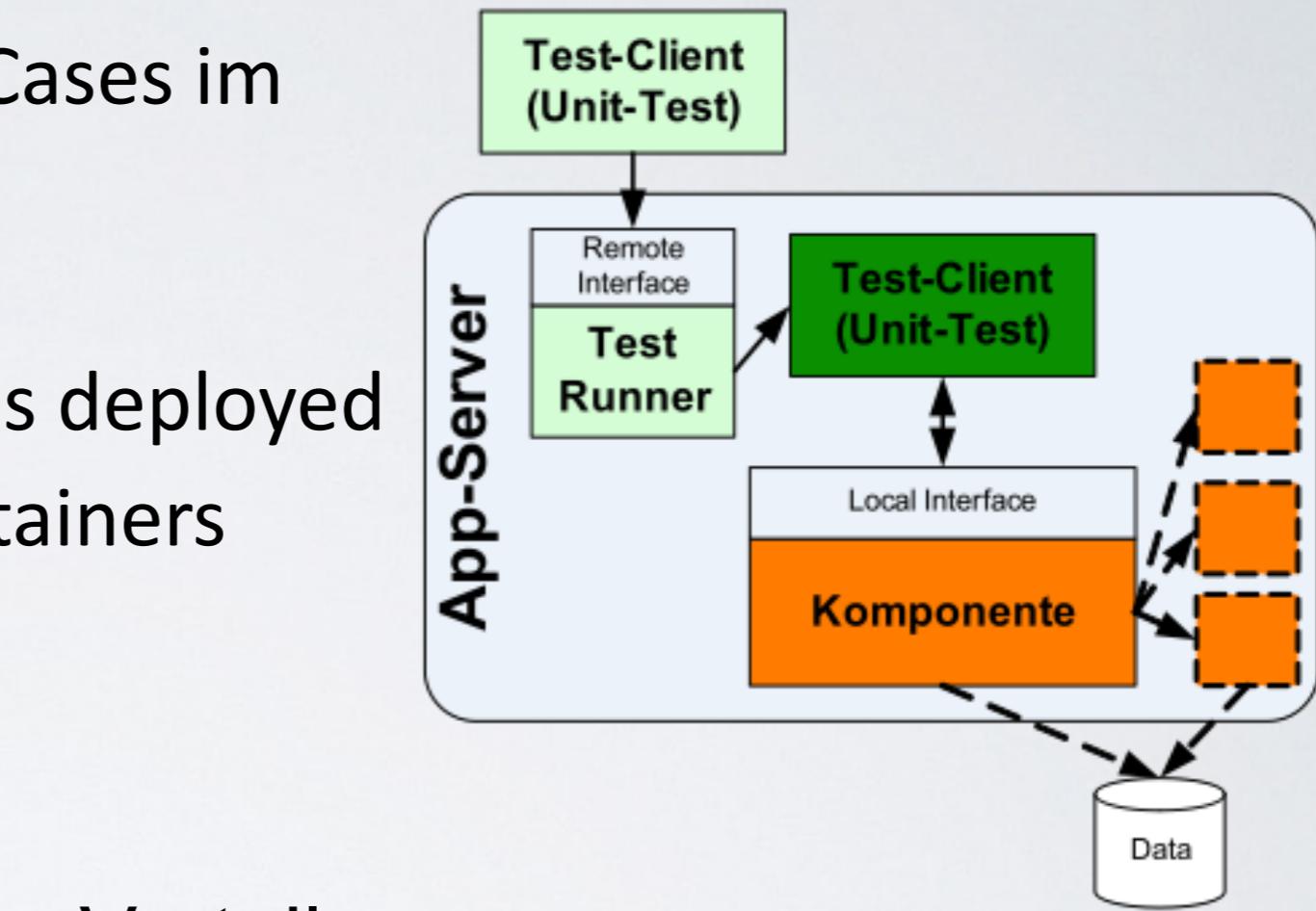
Integrationstests Businesslogik In-Container

- Ziel: Korrektheit einzelner UseCases im Container verifizieren.
- EJBs werden deployed
- Test ist ein Remote-Client
- Klassischer Integration-Test
- Nachteile:
 - Lokale Interfaces können nicht getestet werden
 - Tests werden oft schwerfällig und kompliziert
- Vorteile:
 - Test sehr nahe an ‚realen Verhältnissen‘
 - Keine zusätzliche Infrastruktur notwendig



Integrationstests Businesslogik In-Container (Client im Container)

- Ziel: Korrektheit einzelner UseCases im Container verifizieren.
- EJBs werden deployed
- Test-Infrastruktur wird ebenfalls deployed
- Tests laufen innerhalb des Containers (spezielles EJB oder Servlet)



- Nachteile:
 - Zusätzliche Infrastruktur
 - Testinfrastruktur muss innerhalb des Containers laufen
 - Deployment nötig

- Vorteile:
 - Feingranulares Testing (lokale Interfaces)
 - Zugriff auf Container-Resourcen
 - Nahe an ‚realen Verhältnissen‘

Arquillian

<http://arquillian.org/>



```
14 @RunWith(Arquillian.class)
15 public class HelloEJBTest {
16     @EJB
17     private HelloEJB helloEJB;
18
19     @Deployment
20     public static JavaArchive createTestArchive() {
21         return ShrinkWrap.create(JavaArchive.class, "helloEJB.jar")
22             .addClasses(HelloEJB.class, HelloEJBBean.class);
23     }
24
25     @Test
26     public void testHelloEJB() {
27         String result = helloEJB.sayHelloEJB("Michael");
28         assertEquals("Hello Michael", result);
29     }
30 }
```

Arquillian Ecosystem

Arquillian Persistence Extension

- Integration with DBUnit

Arquillian Drone & Graphene

- Integration with WebDriver

Arquillian Warp

- Allows Grey-Box Testing by triggering a test as a client and verify expectations on the server

Integrations/Acceptance-Tests der Applikation über den UI-Layer

- Ziel:
 - Basisfunktionalität der Applikation sicherstellen
 - Korrektes Verhalten des UI in dedizierten UseCases sicherstellen
- Goldene Regel:
 - Eindeutige Identifikatoren für UI-Elemente verwenden
 - JSF: Component ID:

```
<h:inputText value="#{foo.text}" id="input_foo_text"/>
```

Integrations/Acceptance-Tests der Applikation über den UI-Layer

- Beispiel Bookstore
 - Verwendung von Canoo WebTest
 - Smoke Tests: Laden aller Pages
 - Durchspielen verschiedener UseCases

<http://webtest.canoo.com>

<http://seleniumhq.org>, WebDriver

<http://watir.com> , <http://watij.com>

<http://htmlunit.sourceforge.net>

Automatisiertes UI-Testing

- In der Regel die höchste Stufe des Integration-Tests
- Sollte nicht als einzige Test-Strategie eingesetzt werden!
 - Langsam
 - Fehleranfällig (z.B. Layout ...)
 - Wenig aufschlussreich auf Fehlerursache
- Als Ergänzung zu logischen Unit-Tests und Integration-Tests sehr sinnvoll
- Aber: Aufwändig!

UITest Automation Tools are Snake Oil

<http://blog.objectmentor.com/articles/2010/01/04/ui-test-automation-tools-are-snake-oil>

<http://seleniumhq.org>, WebDriver
<http://webtest.canoo.com>

Diskussion

- Welche Strategien würden Sie einsetzen um den Bookstore zu testen. Welche nicht?
Warum?
- Welche Strategien würden Sie einsetzen um in Ihrem Umfeld Applikationen zu testen?
Welche haben sich bewährt? Welche nicht?
Warum?

Übung 2: Arquillian