



OPITZ CONSULTING

■■■ überraschend mehr Möglichkeiten!



Test Driven Development

OC|Expert Camp /
OPITZ CONSULTING München

Thomas Papendieck,
Senior-Consultant

- 1 Mind Changes
- 2 Anforderungen an UnitTests
- 3 UnitTest in Java



Mind Changes



Mind Change 1

Test Driven Development erzeugt keine Tests sondern ***ausführbare Spezifikationen***. Diese ausführbaren Spezifikationen sind Teil der Dokumentation des eigenen Codes für andere Entwickler.

Mind Change 2

UnitTests verifizieren **öffentlich beobachtbares Verhalten**, nicht Code.
Öffentlich beobachtbares Verhalten sind Rückgabewerte und Kommunikation mit Abhängigkeiten (anderen *Units*).



Mind Change 3:

Code Coverage hat als konkrete Zahl keine Bedeutung. Die einzige "*Coverage*" Metrik von Bedeutung ist **100% Requirements Coverage**. Die kann man aber nicht messen, sondern nur per konsequentlichem TDD sicherstellen.



Mind Change 4:

Code Coverage hat als konkrete Zahl keine Bedeutung. Es gibt Code, der nur Infrastruktur also keine Geschäftslogik implementiert und *too simple to fail* ist, bzw. dessen Korrektheit durch Integrationstest verifiziert wird. Für solchen Code sollten keine Unitests geschrieben werden.

Das setzt voraus, dass das die Konzepte *Single Layer of Abstraction*(SLA) und *Single Responsibility Pattern*(SRP) konsequent auch auf Objekt/Klassenebene umgesetzt sind.

Anforderungen an UnitTests



Was ist ein guter UnitTest?

- Fast
 - Independent
 - Repeatable
 - Self Checking
 - Timely
-
- Readable
 - Trustworthy
 - Fast
 - Maintainable

Unit Test in Java

- 3.1 Werkzeuge
- 3.2 Code Vorlagen



Starterkit für Java Entwickler

- IDE including testplugin (eclipse + infinitest / Netbeans + ? /...)
- build – tool (maven / gradle / ant / ...)
- SCM (git / subversion / ...)
- xUnit testing framework (JUnit / NUnit /...)
- mocking framwork (Mockito / ...)

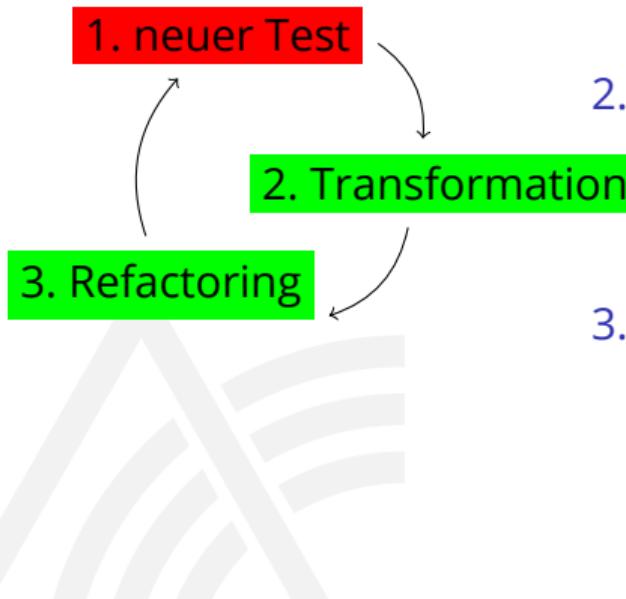
einen JUnit Test schreiben

```
1 class PlainOldJavaClass{  
2  
3     @Test // marks a method so that its been called by JUnit  
4     public // test methods must be public  
5     void // test methods must not return anything  
6     calculate_worstGame_returnsZero() // no parameters  
7     {  
8         // arrange: create and configure dependencies and variables  
9         String playerResultAllZero = "0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0";  
10        int expectedResult = 0;  
11  
12        // act: create tested unit (if possible) and call tested method.  
13        int calculatedResult = new BowlingCalculator().calculate(playerResultAllZero);  
14  
15        // assert: check the Result by calling one of JUnits assert methods.  
16        // the assert method throws an exception when the check fails.  
17        assertEquals("allzerosin", // give usefull short(!) additional description  
18                      // skip when in doubt  
19                      expectedResult,  
20                      calculatedResult);  
21    }  
22}
```

Mockito verwenden

```
1 class PlainOldJavaClass{  
2     @Test  
3     public void testedMethod__preconditions__expectedBehavior() {  
4         // create mock  
5         MyInterfaceOrClass mockObject = mock( MyInterfaceOrClass.class );  
6         //create a spy  
7         MyClass spyObject = spy(new MyClass());  
8         // configure behavior:  
9         doThrow(new RuntimeException("simulate_error"))  
10            .when( mockObject ).anyMethod();  
11         doReturn(mockObject, null, mockObject)  
12            .thenThrow(new RuntimeException("simulate_error_after_4th_call"))  
13            .when( spyObject ).methodWithReturnValue();  
14  
15         // alternative for non void methods:  
16         .when( spyObject.methodWithReturnValue() )  
17            .thenReturn(mockObject, null, mockObject)  
18            .thenThrow(new RuntimeException("simulate_error_after_4th_call"));  
19  
20         // check call of method  
21         verify(spyObject).expectedMethodCall(mockObject, any(OtherInterfaceOrClass.class));  
22     }  
23 }
```

Test Driven Development



1. Schreibe einen neuen Test, gerade so viel dass er fehl schlägt (nicht kompilieren ist fehlschlagen).
2. Schreibe gerade so viel Produktivcode, dass der Test erfüllt wird. Zukünftige Anforderungen nicht beachten! (so simpel wie möglich, alles ist erlaubt, Transformations-Prämissen beachten).
3. Verbessere den Code (Produktion und Test), ohne einen Test zu brechen und ohne neue Funktinalität (Geschäftslogik) hinzuzufügen.



Vielen Dank für die Aufmerksamkeit.



Thomas Papendieck

Senior-Consultant

Norsk-Data-Straße 4
61348 Bad Homburg

thomas.papendieck@opitz-consulting.com

+49 6172 66260 1523



WWW.OPITZ-CONSULTING.COM



@OCC_WIRE



OPITZCONSULTING



opitzconsulting



opitz-consulting-bcb8-1009116