UnitTests? Yes, but right!

Thomas Papendieck
Senior–Consultant

OPITZ CONSULTING GmbH

Location:
Codecentric
Frankfurt (Main)

Agenda

Requirements on UnitTests

Requirements on tested code

required tooling

What is a good UnitTest?

- R
- T
- F
- M

What is a good UnitTest?

- Readable
- Trustworthy
- Fast
- Maintainable

What what does Readable mean?

- what is tested
  test name expresses preconditions, desired result
- short
  put common preparations in separate methods
- what properties of input are important
  choose variable names carefully
  declare and initialize input variables explicitly
- what properties of output are important
  choose variable name carefully
  simple verification (methods with ”speaking names”)

What what does Trustworthy mean?

- business requirement

  Does our code implement a business requirement?
- technical

  does the test really execute the (right) part of the production code?
  when failing, does the test fail for the right reason?
  - replace dependencies
  - ”test first”
  - (almost) no custom logic in tests

What what does Fast mean?

- Tests can be executed frequently without delaying work (each safe at best).
- sophisticated and potentially slow logic in dependencies must be replaced.

What what does Maintainable mean?

- stable against changes in other units
  replace dependencies
- stable against changes in tested unit
  test single assumption

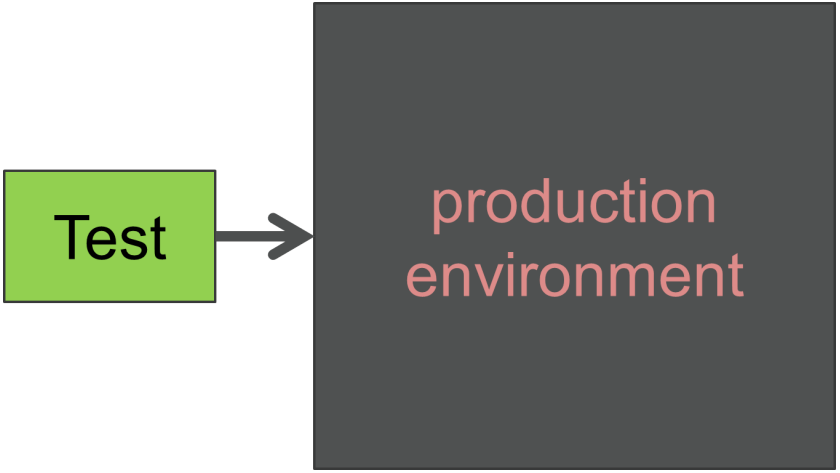
Testability of production code

Quality of Code (as by means of ”Clean Code”) influences quality of Tests (as by means of RTFM– requirements)

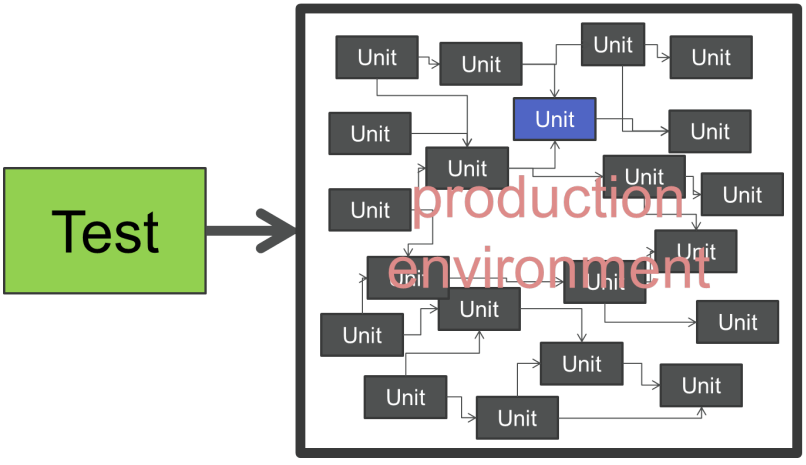
Most important ”Clean Code” rules:

- Separation of concerns
- single resposibility
  object creation is **not** concern of business logic
- Tell! Don’t ask.
- Law of Demeter (Don’t talk to strangers)
  avoid ”message chaining”

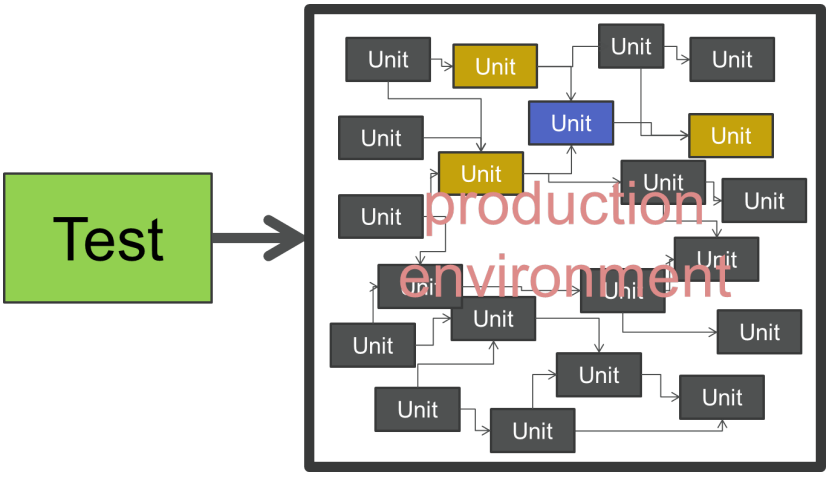
Testability of production code



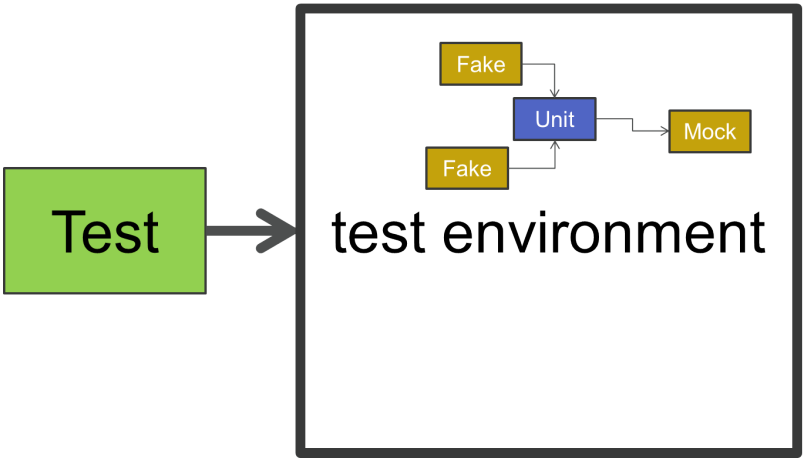
Testability of production code



Testability of production code



Testability of production code



Types of Replacements

- Stub
  - empty implementation of an interface
- Fake
  - alternative implementation of an interface or extension of an existing implementation.
  - extremely simplified behavior (no logic)
- Mock
  - ”enhanced” Fake
  - configurable behavior
  - verification of method calls and the parameters given

What enables dependency replacement?

- separate dependency instantiation from business logic
- no call to `new` operator
- introduce ”seams”
  - dependency injection
  - low visibility getter
- no `static` (public) methods
- no `final` (public) methods
- avoid Singelton pattern (not singelton as a concept)

write ”Clean Code”[1]

- programming against interfaces
- SoC/SRP (Feature envy)
- Law of Demeter
- DRY
- same level of abstraction

[1] ”Clean Code” by Robert C Martin, ISBN-13: 978-0132350884

starter kit for Java developers

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

Writing a JUnit Test

```
1  class PlainOldJavaClass{
2     @Test // marks a method so that its been called by JUnit
3     public // test methods must be public
4     void  // test methods must not return anything
5     testedMethod__preconditions__expectedBehavior() // no parameters
6     {
7       // arrange: create and configure dependencies and variables
8       int input = 4;
9       int expectedResult = 2;
10
11      // act: create tested unit (if possible)  and call tested method.
12      int calculatedResult = (int) Math.sqr(input);
13
14      // assert: check the Result by calling one of JUnits assert methods.
15      // the assert method throws an exception when the check fails.
16      assertEquals("square root of 4", // always describe what you check.
17                   expectedResult,
18                   calculatedResult);
19    }
20 }
```

Using Mockito

```
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("should not be called")).when(mockObject).anyMethod();
10         doReturn(mockObject).thenReturn(null).when(spyObject).getMyInterfaceOrClass();
11
12         // alternative for non void methods:
13         .when ( spyObject.getMyInterfaceOrClass() ) .thenReturn(mockObject,null,mockObject);
14
15         // check call of method
16         verify(spyObject).expectedMethodCall(mockObject,any(OtherInterfaceOrClass.class));
17     }
18 }
```

Test Driven Development

When developing your software follow this cycle:

- Test - write enough of a single test so that it fails.
  not compiling means fail...
- Implement - change the production code to make the test compile and pass.
  Use the simplest possible solution.
- Refactor - change the production code to make it ”clean”.
  Resolve code duplications, introduce design patterns, but no test must break at the end.
  Also refactor your test code.

Hands On

Starting points for your practice:

- this presentation and an empty project:
  `https://github.com/tpd-opitz/JUGF-UnitTest-yes-but-right`
- Sample projects with ”good” and ”bad” samples of production and test code.
  `https://github.com/tpd-opitz/eclipse-magazin-02-16-Beispiele`