TDD

TDD

What is it all about?

Introduction to Unit testing

- What does TDD mean?
- The basics of TDD
- Important Questions and Answers
- Some numbers
- Getting started

What does TDD mean?

- Test Driven Development
- Test Oriented Development
- Test Driven Design
- Test Driven Design & Development

Putting TDD in Context

- Engineering methodologies (Waterfall i.e.)
- Agile methodologies
  - Extreme programming
    - Test Driven Development
- What ever the methodology
  - Test Driven Development is always possible

What is a Unit Test?

- A Unit Test is a test of a small functional piece of code
  - A Unit Test is an automated piece of code that checks an other piece of code
- Look at the following function:

```
public boolean isLoginOk(String username,String password){
                boolean isLoginOk=false;
                // code comes here
                // getting data out of database
                // verifying username password combination
                return isLoginOk;
}
```

About the isLoginOk() function

- What do you like to check of this function?
  - You probably try to check a username password combination that returns true
  - Idem for a combination that returns false
  - Perhaps some special values? Null?

- How many unit tests do you expect that is needed to test this piece of functionality?

```
public boolean isLoginOk(String username,String password){
                boolean isLoginOk=false;
                // code comes here
                // getting data out of database
                // verifying username password combination
                return isLoginOk;
}
```

A:2
B:4
C: More than 4
D: More than 10

How many unit tests do you expect

- Reason Question: I like to establish a sort of consensus on how many unit tests to expect
- Probably at a minimum:
  - 2 to check for a positive login, returning true and a negative one for returning false
  - Plus checks when strange values are passed in: null values and the like.
- So we expect at a minimum 4 unit tests.

How much functionality is there?

- How many pieces of functionality, comparable with this one, isLoginOk() do you expect in an application of moderate size?
- 100, 300, 1000, 3000 pieces of functionality?
- Well, let's stick to 300 pieces
  - How many unit tests do you expect to find?
- Well, we expect 1200 Unit Tests
  - Roughly 300*4 =1200
- How many Unit Tests do we see in the wild?

How many Unit Tests are actually written?

- 300, 400 hmmm
- When people are truly doing TDD you expect 1200 UT's
- Why this gab?
  - Was it because of lack of time; the UT's were actually written after the production code and time was running short?
  - Or writing test was too difficult?
- Is the practice of TDD followed?
- Writing UT's != TDD at least not necessarily

Imagine you had these 1200 test!

- Which of the following statements holds true?
- Easier to find bugs!?
- Easier to maintain!?
- Easier to understand!?
- Easier to develop!?

Is it Easier to find bugs!?

- Who thinks that because of these 1200 UT finding bugs is easier?(show of hands.)
  - perhaps you expect it, but it is not always true.
  - I.E: What happens if you have bugs in your tests?
  - Or you think your tests pass, but they actually have a bug? It can take a long time of finding such a bug! This is especially true when people write a lot of logic in there UT's
- There are ways to make it true however, make very simple UT's!

Is it easier to maintain with 1200 UT's?

- Show of hands
- Not necessarily ! But there are ways to cope with this problem too!
  - I.E. imagine a test in which you instantiates a class and you have 25 other tests for this class, now you change the constructor for this class
  - 25 test do not compile, now you have to change each one of them.
  - Maintaining the code may actually break a lot of tests for the wrong reasons. So more tests, make it harder for people to maintain there code

Is it easier to understand with1200 UT's?

- What do I mean with understand?
- Well imagine you are a new developer on the team: do these test help you to get a grip on the software?
  - I.E: you are asked to fix a specific bug in a class you have never seen before. It's a huge class, there is a specific method were you are told the bug is. You have no idea what the methode does.
- The first thing what you do is?
  - look at the test? – ah, there are no tests!
- What do you do?.

There are no tests, what to do?

- What would you do?
- Ask the audience!
- look at the code
  - Which code?
- look at the references where method is called
  - to decipher what the method is supposed to do
  - what are the usecases where this method is used
  - what are the contexts in which it needs to be checked.

But when you have UT's..

- We can think of each test as a small piece of usecase that is tested
- It is a small piece of an automated usecase that tells you what a method should do.
- But this is not necessarily true for a UT
- Because a lot of people who write test write them in an unreadable manner
  - they don't pay attention to the naming of the tests, the names of parameters.

Unreadable UT are a shame

- because an UTis not solely an automated test
- It is part of the documentation of your api
- It is supposed to be:
  - executable documentation
- If you realize that, then you understand that naming is very important.

What is wrong with bad names?

- Names like: testLoginPass1, testLoginPass2,etc
- If such a test fails and someone looks at the test, he realizes that he has no idea what the test is doing.
- When should it fail? Why should it fail?
  - they start looking at the code of the test they don't understand, they start to debug the test
- The test doesn't help to understand the code, there is little value in the test to help you understand the code.

The question we posed was

- Which statements are true with 1200 UT's?

- Easier to find bugs!?
- Easier to maintain!?
- Easier to understand!?
- Easier to develop!?

- With a strict practice (TDD) the first 3 can be made true then it is also easier to develop

Automatic properties of TDD

- Does not help
  - Test Maintainability
  - Test Readability
- Does help
  - Trust-worthy tests
  - Design
  - Understandig problem at hand
  - Code Coverage
  - Contract first development
  - Simple production code

Automatic properties of TDD

- Incremental development
- Deliver early and often
- Improved quality

Where do these properties come from?

- you can make unit test before or after writing code
- but even if you write unit test before writing code it doesn't qualify for TDD.
- TDD: demands extra care in the field of procedures to:
  - write maintainable tests and to
  - write readable tests

Where do these properties come from?

- Test first although a strategy with a lot of payoff doesn't result automatically in testcode that can be better maintained or is more readable.
- You can write crappy unreadable testcode
- Important to make a distinction between proper TDD and unit testing
- But if you start with TDD there are a rewards regardless of the other 2 points

Where test according to TDD help

- It can help you write trustworthy tests, test that you really trust
- Test should not fail by default or are expected to fail
- It helps you to drive the design, when doing TDD you implement small tasks, which must be designed

Where test according to TDD help

- On the presumption that you don't implement by layer but by feature
- The important rewards are:
  - You can deliver by incremental development
  - you will always build something within a week, perhaps not with a ui but you can show some test in which there is some functionality
- The team can deliver early and often
  - you write tests and if something fails you immediately try to solve and only continue when it works

Where test according to TDD help

- If your boss comes and asks how far are you with your task, you can say:
  - 50% is done, you can show him the test in which you really test 50% of you functionality
- With TDD you have these test to follow, these are executable specifications.
- Of course code coverage is much better

People already do Unit Testing but

- Tests fail on a few crucial points
  - Test are often not structured
  - Test are not repeatable(by others)
- Why is repeatability important
  - You want other developers, who write code, give the opportunity to check that they didn't break your code
  - To check that, they must be able to run your tests
  - On the same hand, you want to be able to run there tests to see If your code doesn't break there's
- Not on all your test
- Not easy to do it the right way

To help UT use a framework

- The original was for SmallTalk
  - Kent Beck and Erich Gamma
- Ported to Various languages and platforms
  - Junit, Nunit,VBUnit,.
- Standard test architecture
- Introduction Junit

How we use the framework

- Write Tests
  - Make it easy to create and organize tests
- Run Tests
  - Allow running all of your tests, a group or just one
  - From commandline or GUI
- Review Results
  - Immediate Pass/Fail feedback

Declare a Test Case

```
@Test
public void isLoginOk_WithCorrectCredentials_ReturnsTrue() {


}
```

Mehtod under Test

scenario

Expected behavior

No reason to put test in the name!

```
@Test
public void isLoginOk_WithCorrectCredentials_ReturnsTrue() {


}
```

Mehtod under Test

scenario

Expected behavior

Implementing a test case

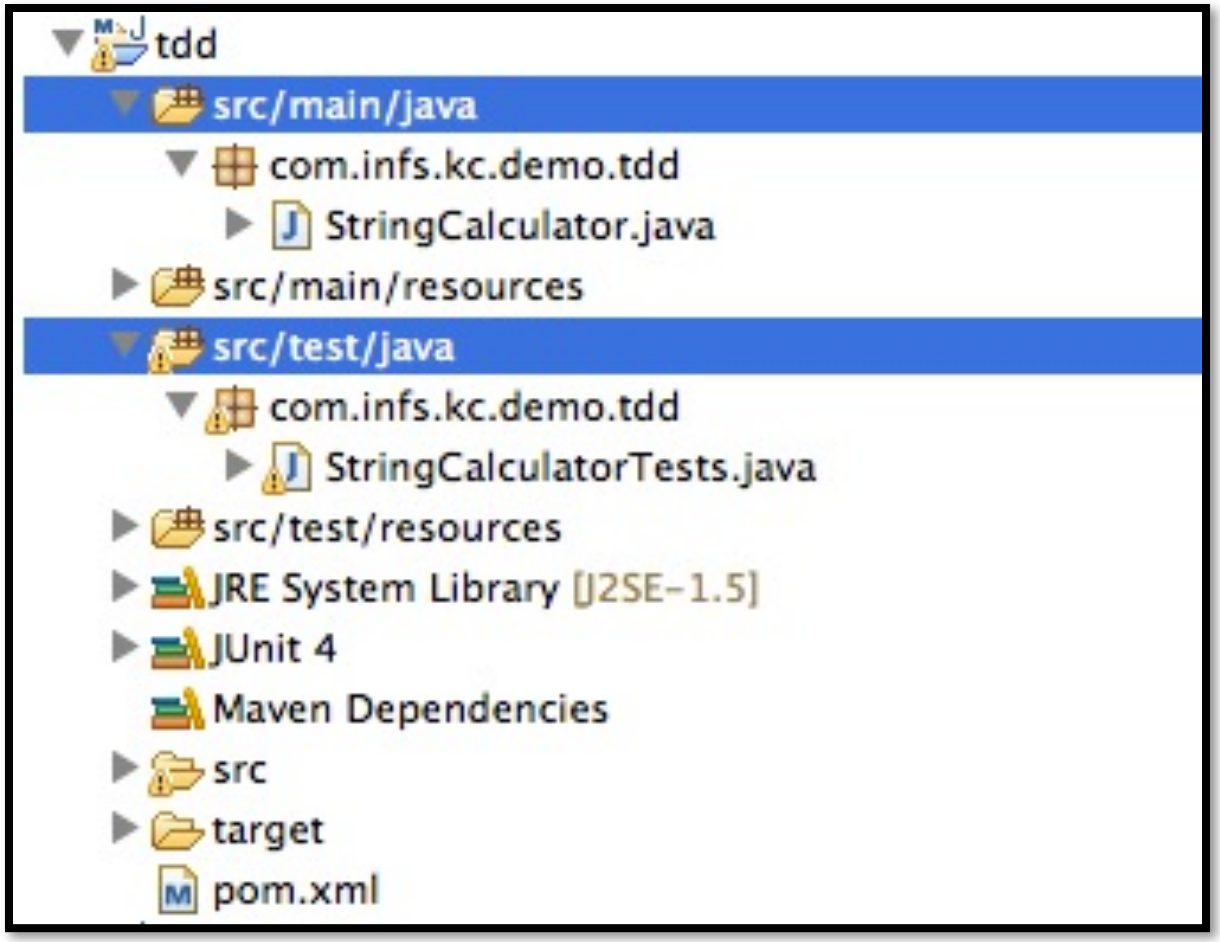
```
@Test
public void sum_2ints_sumThemUp() {
        StringCalculator calculator =
                        new StringCalculator();     // Arrange

        int actualSum=calculator.sum(1,2);       // Act
        int expectedSum=3;
        assertThat(actualSum, is(expectedSum)); // Assert
}
```

TDD is

- The act of writing tests before the actual production of code
- Verification that the test passes when it should
- Verification that it fails when it should

Separate production and test code



Test-Driven Development

- Make it Fail
  - No code without a failing test
- Make it Work
  - As simple as possible
- Make it Better
  - Refactor

Demo

- Calculator
  - which sums integers as strings
  - The integers are separated by a comma
  - Example
    - sum(“”)
    - sum(“1,2”)