BLG 475E: Software Quality and Testing
Fall 2017-18

Dr.Ayşe Tosun
tosunay@itu.edu.tr

İstanbul Teknik Üniversitesi

Outline

- What is unit testing for?
- How to do unit testing?
- Junit Basics
- Slicing and iterative test last development
- Test driven development

Slides are based on H. Erdogmus's and O. Dieste's training materials within the FidiPro ESEIL Project & the book «Pragmatic Unit Testing» by A. Hunt, D. Thomas, 2003.

What IS / IS NOT unit testing FOR?

- Unit testing isn't designed to achieve some corporate quality initiative.
- It's not a tool for the end-users, or managers, or team leads.
- Unit testing is done by programmers, for programmers. It's here for programmers’ benefit alone, to make their lifes easier.
- Unit tests are performed to prove that a piece of code does what the developer thinks it should do.
- It will make your designs better and drastically reduce the amount of time you spend debugging.
- It is also a programming philosophy

How to do unit testing

- Decide how to test the method in question – **before** writing the code itself

- Run the test itself, and probably all the other tests in that part of the system

- Make sure **all tests pass**

Planning tests (Example)

- A single method designed to find the largest number in a list of numbers
  - int Largest.largest(int[] list);
- e.g. [7,8,9] -> 9
- What other tests can you think of?
- The order should not matter
- The duplicate largest numbers should not matter
- Only one number in a list
- Negative numbers

Structuring unit tests

- When writing unit tests, there are some naming conventions you need to follow.
- We need to use some unit testing frameworks depending on the programming language to run our tests.
- We will learn unit test practices using Junit.

Some facts about JUnit

- JUnit is a unit testing framework for Java, created by Eric Gamma (patterns) and Kent Beck (TDD) in 1997
  - They say it was created in a flight from Zurich to the 1997 OOPSLA in Atlanta ☺
- Similar frameworks created for other languages
  - *De facto* standard
- Still evolving
  - Latest release: 5 (10th September) before: 4.12

Basic Concepts

- Assertions, to verify single expected results (typically, one parameter)
- Test methods, to verify one single case of a given feature = *test case*
- Test class; typically embodies all test methods for a given class

Assertions

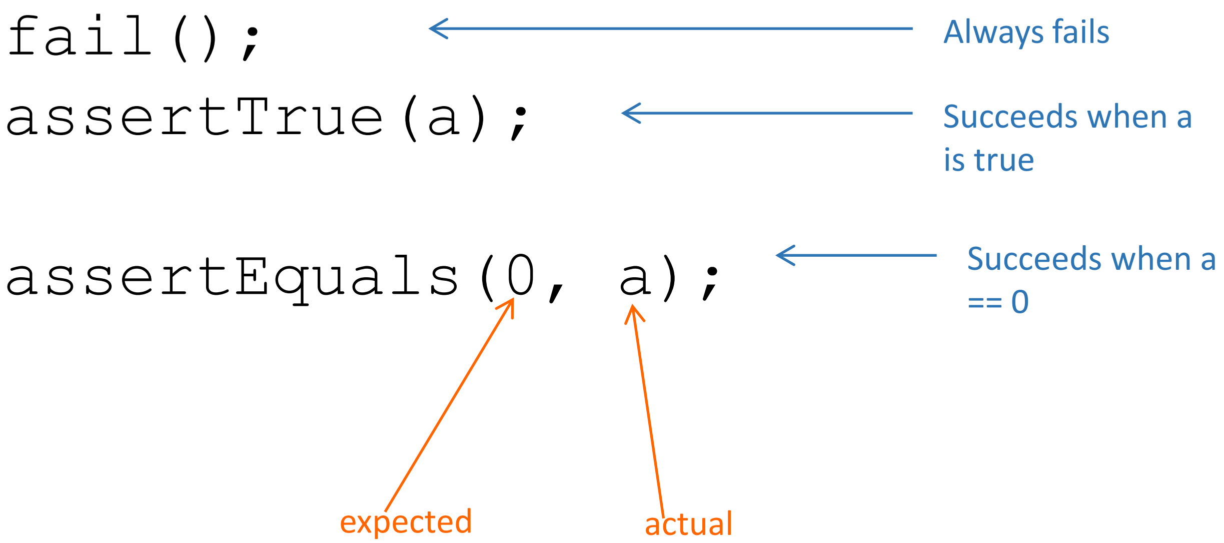
- Verify single expected results

```
fail
assertTrue          assertFalse
assertNull          assertNotNull
assertEquals
assertArrayEquals
assertSame          assertNotSame
```

- Those methods accept a `String` argument to describe the reason of a failure

Assertions

- e.g.:



```
fail();
assertTrue(a);

assertEquals(0, a);
```

Always fails

Succeeds when a is true

Succeeds when a == 0

expected

actual

Assertions

- All assertions accept a `String` argument (in the 1st position) to describe the reason of a failure

```
assertEquals(“reason here”, 0, a);
```

Test methods

- Defined using the `@Test` annotation:

```
@Test
public void <methodName>() { … }
```

Test methods

- Defined using the `@Test` annotation:

```
@Test
public void <methodName>() { … }
```

Mandatory

Test methods

- Defined using the `@Test` annotation:

```
@Test
public void <methodName>() { … }
```

Test method names must be meaningful; they should provide a clear idea what the test is for

In general, these good practices are independent of the UT framework (Junit, in this case)

Test methods

- Defined using the annotation:

```
@Test
public void <methodName>() { … }
```

- Test methods can contain any code:
  - Local variables, calculations
  - Control structures, call to helper methods
  - Etc.
- In particular, they contain one or several assertions

Test methods

- e.g.:

Focus on behavior, not implementation

```
       @Test
       public void pushThreeElements() {
              s.push(a);
              s.push(b);
              s.push(c);
              assertEquals(3,
s.getSize());
       }
```

Focus on behavior, not implementation

- Does it work?...

```
class Stack{
        int numElem;

        public void Push(…) {
                numElem++;
        }

        public int getSize() {
                return numElem;
        }
}
```

Focus on behavior, not implementation

- And this?

```
class Stack {
        Vector <Object> elems;

        public void push(…) {
                elems.add(…);
        }

        public int getSize() {
                return elems.size();
        }
}
```

Test class

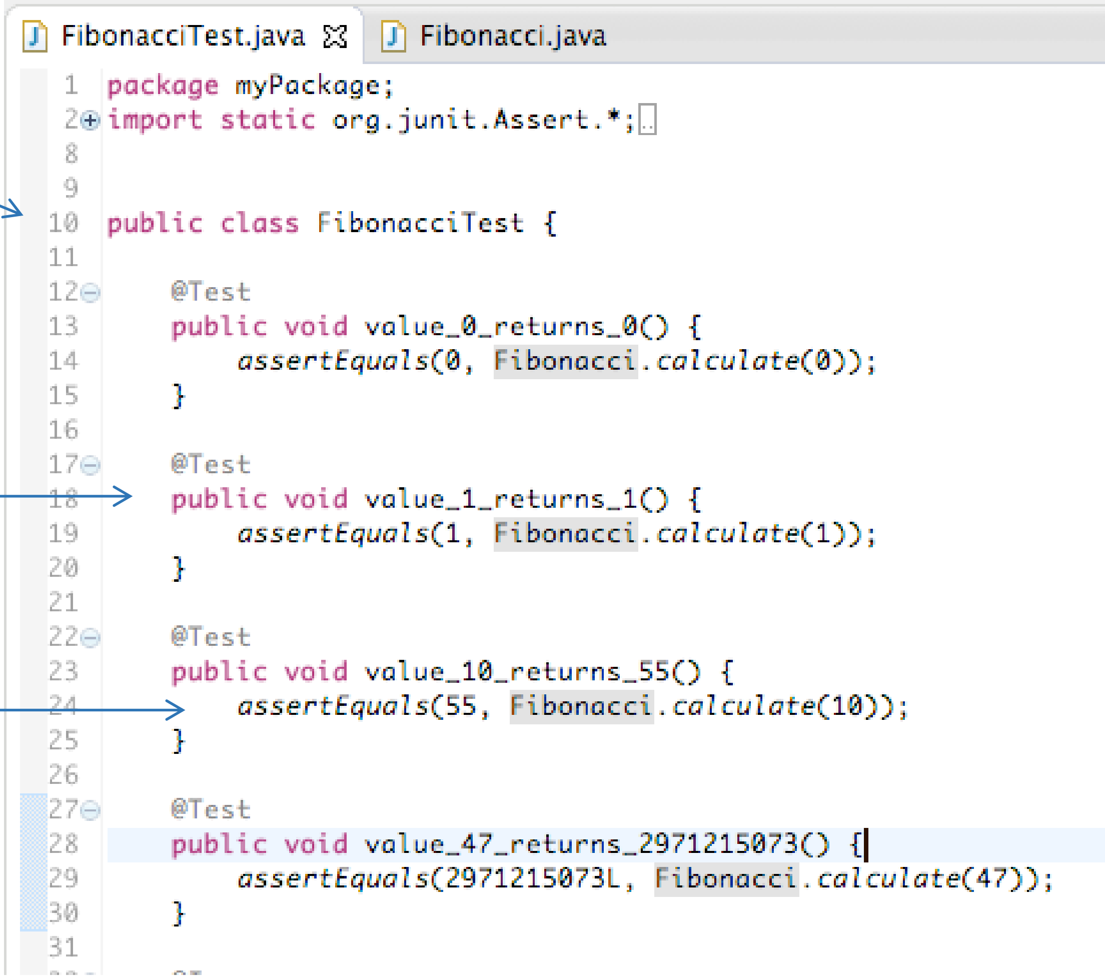
- A class that embodies a set of test methods and related code
- Test classes may become really complex, but we will focus on the basics so far

Typical test class structure

Typically, the test class for class <*Class*> is <*ClassTest*> *(but not always)*

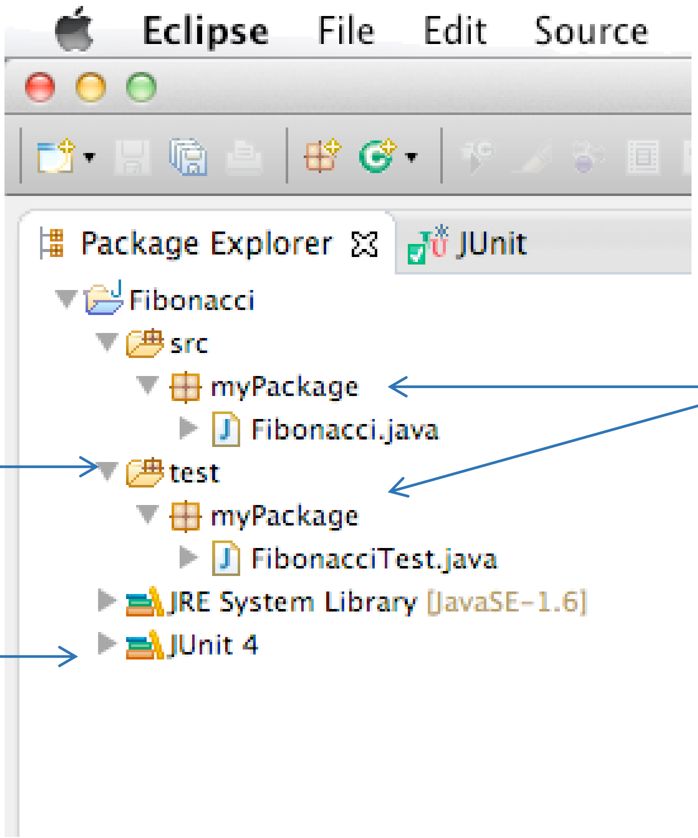
Test method, annotated with @Test

Assertion (one of several kinds of)



```java
package myPackage;
import static org.junit.Assert.*;


public class FibonacciTest {

    @Test
    public void value_0_returns_0() {
        assertEquals(0, Fibonacci.calculate(0));
    }

    @Test
    public void value_1_returns_1() {
        assertEquals(1, Fibonacci.calculate(1));
    }

    @Test
    public void value_10_returns_55() {
        assertEquals(55, Fibonacci.calculate(10));
    }

    @Test
    public void value_47_returns_2971215073() {
        assertEquals(2971215073L, Fibonacci.calculate(47));
    }

```

Typical code organization



Test classes are located under the *test* folder

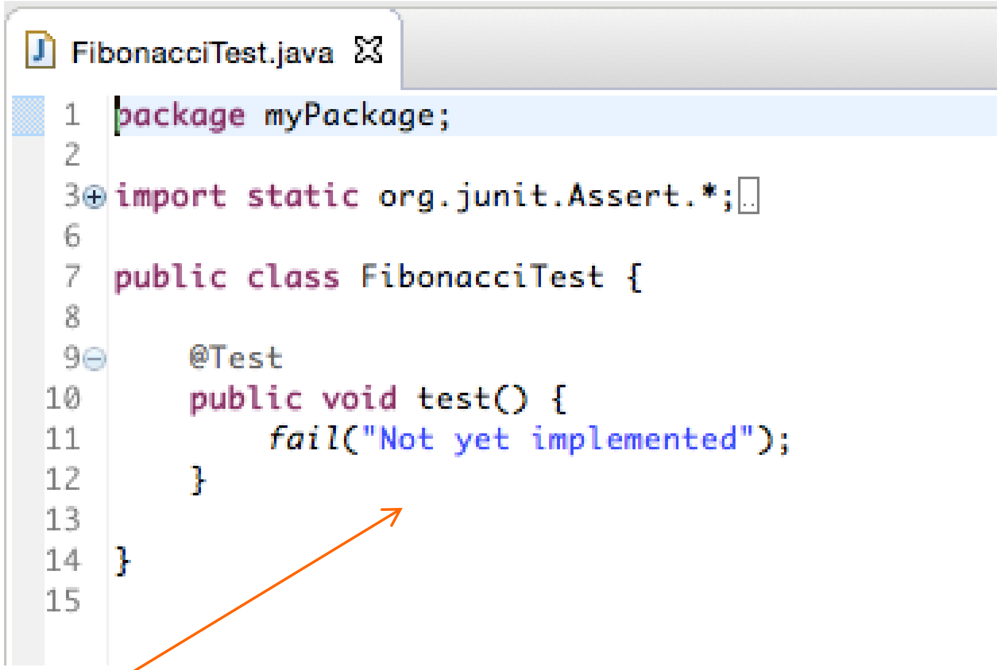
JUnit library must be present

Classes and test classes are located in the same package

Good practices

- Test method names must be meaningful
- Focus on behavior, not implementation

One point here…



```java
package myPackage;

import static org.junit.Assert.*;

public class FibonacciTest {

	@Test
	public void test() {
		fail("Not yet implemented");
	}

}

```

Unfinished tests should fail

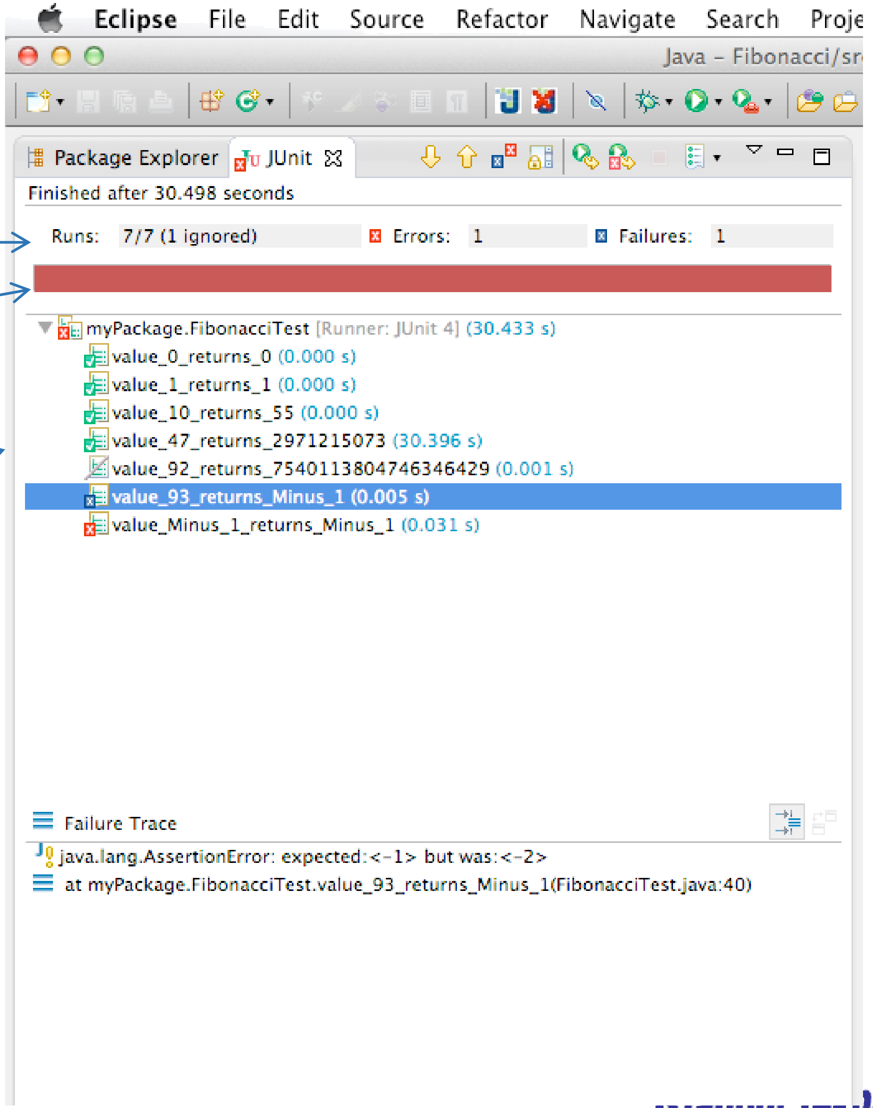
JUnit built-in runner

Status

Red/Green bar

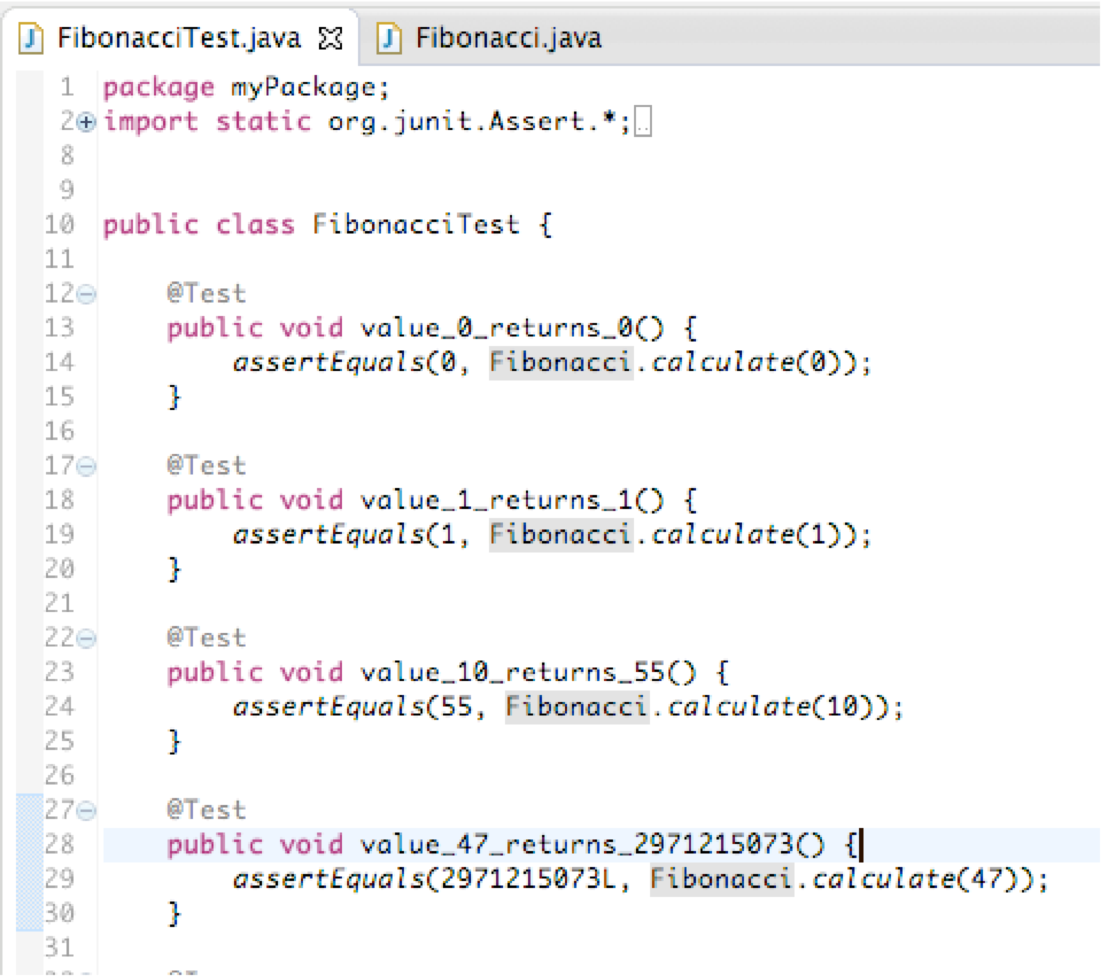
Run tests, results (success, failure, ignore, error) and execution time

Failure trace, including failing assertion information



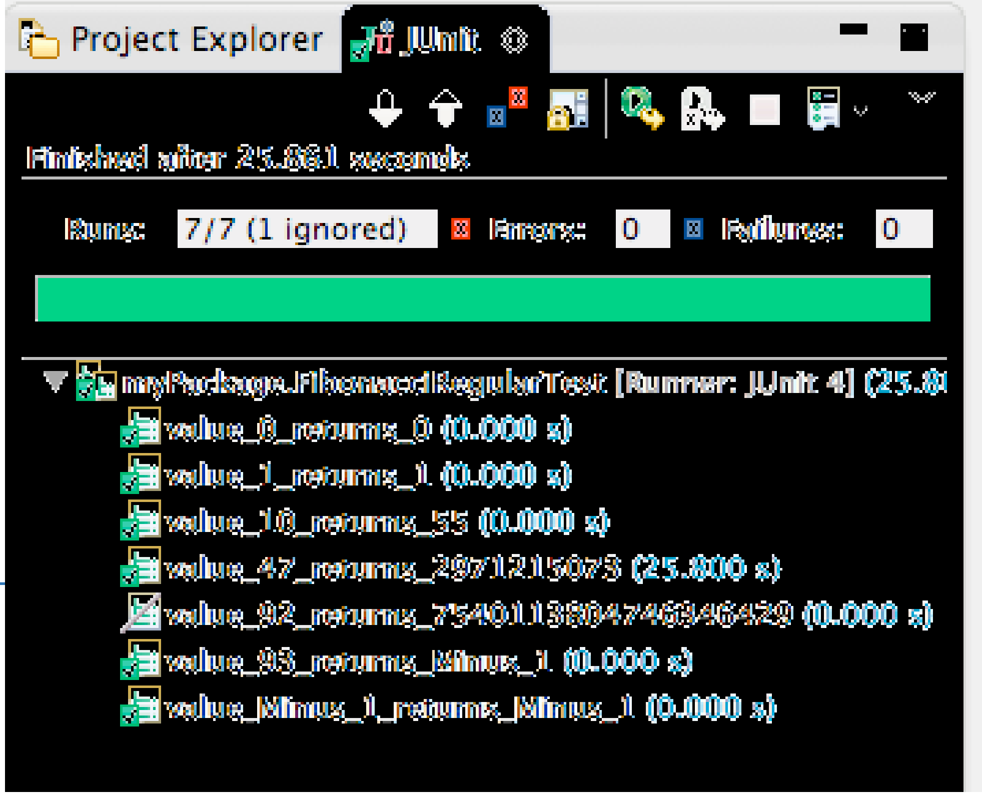
```
Finished after 30.498 seconds
Runs:  7/7 (1 ignored)    Errors:  1    Failures:  1

myPackage.FibonacciTest [Runner: JUnit 4] (30.433 s)
    value_0_returns_0 (0.000 s)
    value_1_returns_1 (0.000 s)
    value_10_returns_55 (0.000 s)
    value_47_returns_2971215073 (30.396 s)
    value_92_returns_7540113804746346429 (0.001 s)
    value_93_returns_Minus_1 (0.005 s)
    value_Minus_1_returns_Minus_1 (0.031 s)

Failure Trace
java.lang.AssertionError: expected:<-1> but was:<-2>
at myPackage.FibonacciTest.value_93_returns_Minus_1(FibonacciTest.java:40)
```

Target



```
package myPackage;
import static org.junit.Assert.*;


public class FibonacciTest {

    @Test
    public void value_0_returns_0() {
        assertEquals(0, Fibonacci.calculate(0));
    }

    @Test
    public void value_1_returns_1() {
        assertEquals(1, Fibonacci.calculate(1));
    }

    @Test
    public void value_10_returns_55() {
        assertEquals(55, Fibonacci.calculate(10));
    }

    @Test
    public void value_47_returns_2971215073() {
        assertEquals(2971215073L, Fibonacci.calculate(47));
    }

```

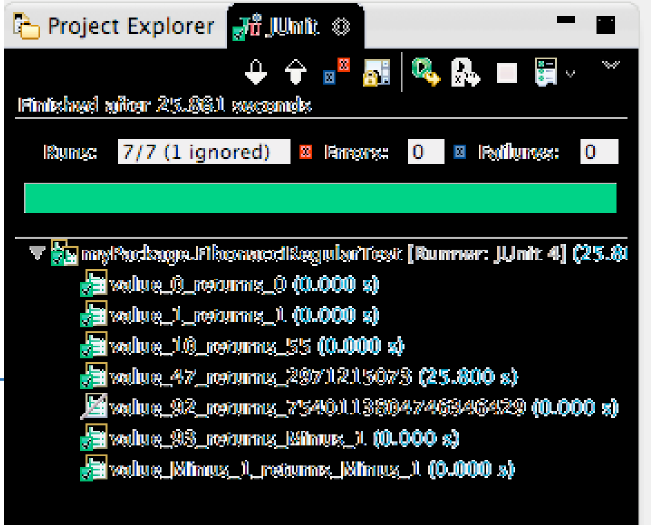
Results for `Fibonacci` class



```
Finished after 25.861 seconds

Runs: 7/7 (1 ignored)   Errors: 0   Failures: 0

myPackage.FibonacciRegularTest [Runner: JUnit 4] (25.8
    value_0_returns_0 (0.000 s)
    value_1_returns_1 (0.000 s)
    value_10_returns_55 (0.000 s)
    value_47_returns_2971215073 (25.800 s)
    value_92_returns_7540113804746346429 (0.000 s)
    value_93_returns_Minus_1 (0.000 s)
    value_Minus_1_returns_Minus_1 (0.000 s)
```

This tests takes almost 30 seconds!

Results for `Fibonacci` class

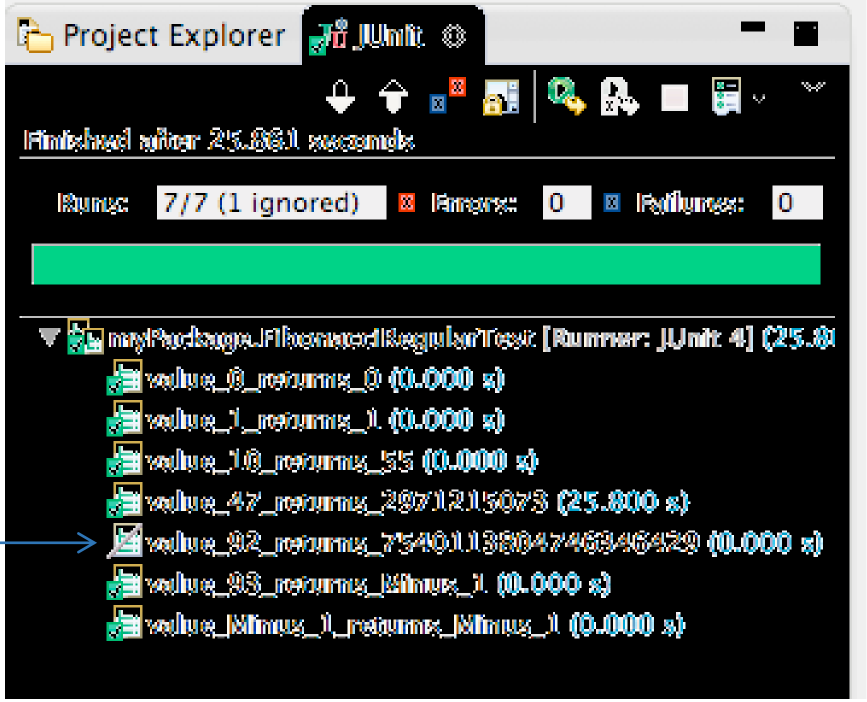


```
Finished after 25.861 seconds

Runs: 7/7 (1 ignored)   Errors: 0   Failures: 0

myPackage.FibonacciRegularTest [Runner: JUnit 4] (25.8
    value_0_returns_0 (0.000 s)
    value_1_returns_1 (0.000 s)
    value_10_returns_55 (0.000 s)
    value_47_returns_2971215073 (25.800 s)
    value_92_returns_7540113804746346429 (0.000 s)
    value_93_returns_Minus_1 (0.000 s)
    value_Minus_1_returns_Minus_1 (0.000 s)
```

This tests takes almost 30 seconds!

Test should provide fast feedback

Results for `Fibonacci` class

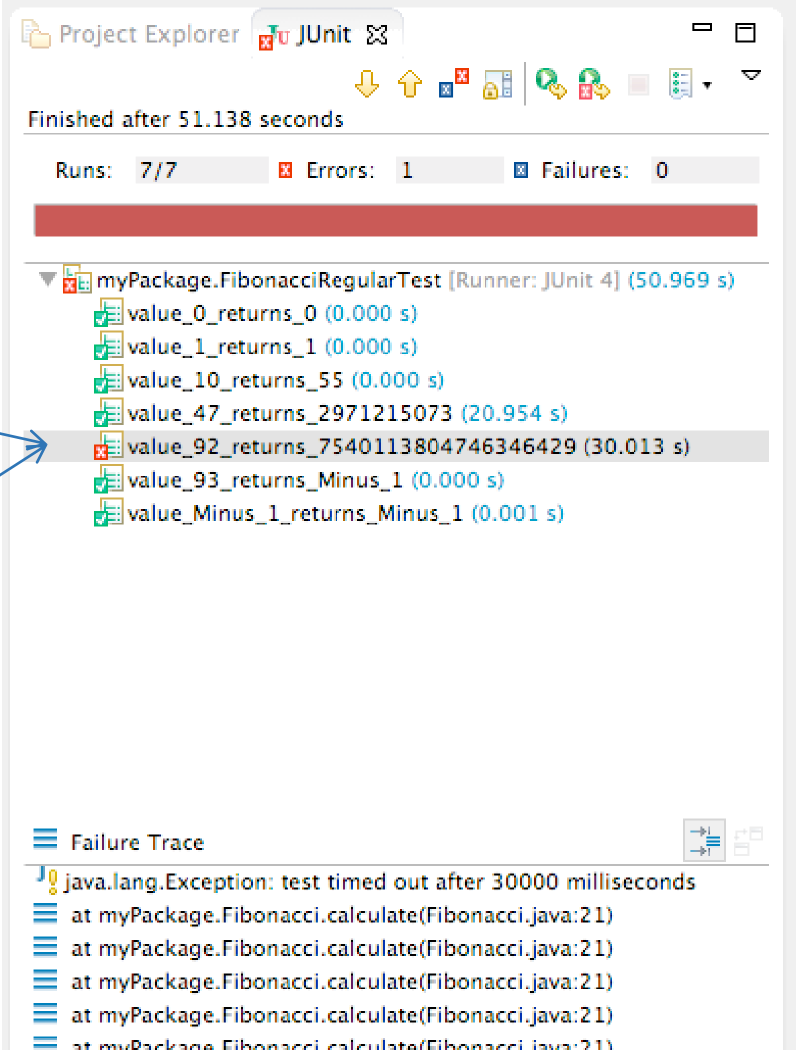


```
Finished after 25.861 seconds

Runs: 7/7 (1 ignored)   Errors: 0   Failures: 0

myPackage.FibonacciRegularTest [Runner: JUnit 4] (25.8
    value_0_returns_0 (0.000 s)
    value_1_returns_1 (0.000 s)
    value_10_returns_55 (0.000 s)
    value_47_returns_2971215073 (25.800 s)
    value_92_returns_7540113804746346429 (0.000 s)
    value_93_returns_Minus_1 (0.000 s)
    value_Minus_1_returns_Minus_1 (0.000 s)
```

Try this one!

(right now, this test is decorated with `@Ignore` besides `@Test`, to avoid test execution)

Results for `Fibonacci` class



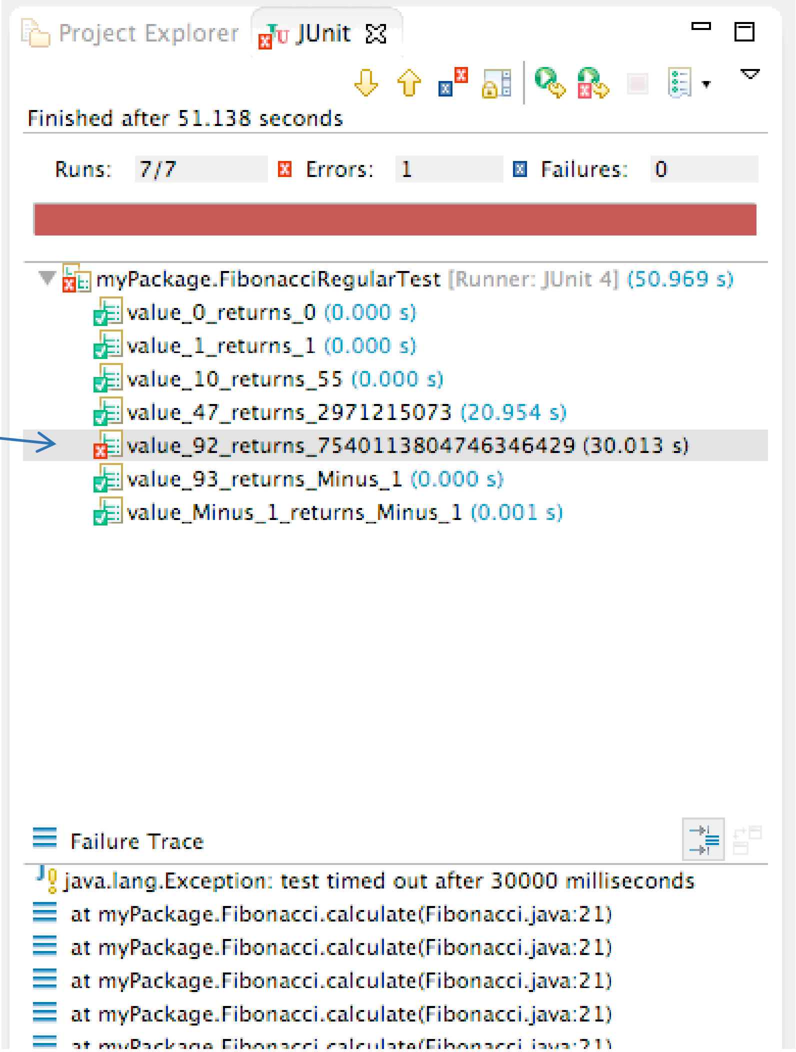
```
Finished after 51.138 seconds

Runs: 7/7     Errors: 1     Failures: 0

myPackage.FibonacciRegularTest [Runner: JUnit 4] (50.969 s)
    value_0_returns_0 (0.000 s)
    value_1_returns_1 (0.000 s)
    value_10_returns_55 (0.000 s)
    value_47_returns_2971215073 (20.954 s)
    value_92_returns_7540113804746346429 (30.013 s)
    value_93_returns_Minus_1 (0.000 s)
    value_Minus_1_returns_Minus_1 (0.001 s)

Failure Trace
java.lang.Exception: test timed out after 30000 milliseconds
at myPackage.Fibonacci.calculate(Fibonacci.java:21)
at myPackage.Fibonacci.calculate(Fibonacci.java:21)
at myPackage.Fibonacci.calculate(Fibonacci.java:21)
at myPackage.Fibonacci.calculate(Fibonacci.java:21)
at myPackage.Fibonacci.calculate(Fibonacci.java:21)
```

Don’t lose time. It will take quite a while to finish

We can use a trick provided by JUnit to avoid long lasting cases:

```
@Test (timeout =
<miliseconds>)
```

Results for `Fibonacci` class



```
Finished after 51.138 seconds

Runs: 7/7    Errors: 1    Failures: 0

myPackage.FibonacciRegularTest [Runner: JUnit 4] (50.969 s)
    value_0_returns_0 (0.000 s)
    value_1_returns_1 (0.000 s)
    value_10_returns_55 (0.000 s)
    value_47_returns_2971215073 (20.954 s)
    value_92_returns_7540113804746346429 (30.013 s)
    value_93_returns_Minus_1 (0.000 s)
    value_Minus_1_returns_Minus_1 (0.001 s)

Failure Trace
java.lang.Exception: test timed out after 30000 milliseconds
at myPackage.Fibonacci.calculate(Fibonacci.java:21)
at myPackage.Fibonacci.calculate(Fibonacci.java:21)
at myPackage.Fibonacci.calculate(Fibonacci.java:21)
at myPackage.Fibonacci.calculate(Fibonacci.java:21)
at myPackage.Fibonacci.calculate(Fibonacci.java:21)
```

However, long lasting test is a sign of poor (production code) design

Design should be testable (refactor if needed)

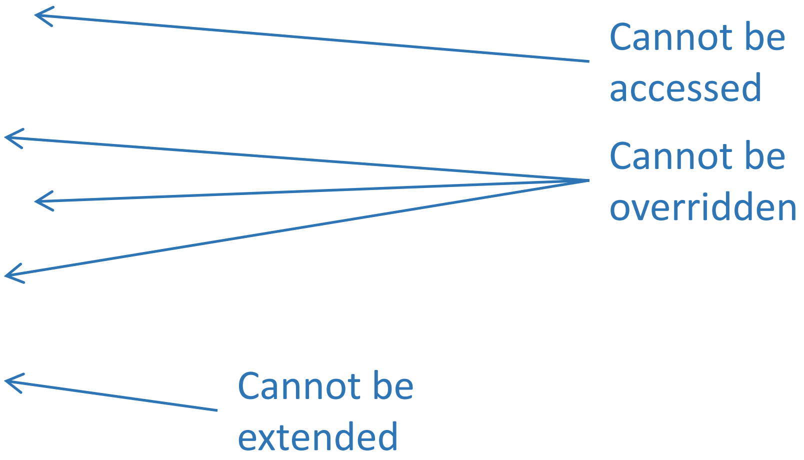
Wrong approach

A test should have just one reason to fail

```
         @Test
         public void calculateSeveralAverages() throws
NotEnoughElements {
                  Statistics s = new Statistics();
         s.add(1.0);
                  s.add(2.0);
                  assertEquals(1.5, s.average(), 0.01);
                  s.add(3.0);
                  assertEquals(2.0, s.average(), 0.01);
         }
```

Testable design

- Some programming decisions affect code testability
  - Not <u>necessarily bad decisions:</u> e.g.: private fields, methods
  - Good, actually
- Fields
  - `private`
- Methods
  - `final`
  - `private`
  - `static`
- Classes
  - `final`



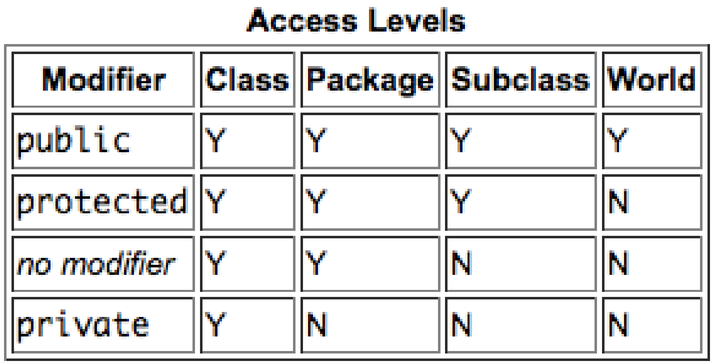
Cannot be accessed

Cannot be overridden

Cannot be extended

Java scoping rules

- See http://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html



Access Levels

Modifier	Class	Package	Subclass	World
`public`	Y	Y	Y	Y
`protected`	Y	Y	Y	N
no modifier	Y	Y	N	N
`private`	Y	N	N	N

Solutions?

- Fields
  - ~~`private`~~ → `package access`
- Methods
  - ~~`final`~~ → remove
  - ~~`private`~~ → `package (/protected)`
  - ~~`static`~~ → do not use unless they are predictable
- Classes
  - ~~`final`~~ → remove

Good practices

- Design should be testable (refactor if needed)
  - Use dependency injection to easily substitute collaborators with test doubles
    - Pay attention to external resources, system lookups, etc.
    - Use `new` with care
  - Avoid complex logic in constructors
    - Easier to double
  - Use `package access` instead of `private` to access attributes from test doubles in the same package
    - Protected is another alternative,
  - Same for methods
  - Do not use `final` methods or classes
  - Use only `static` methods if you are sure they do not needed to be substitute by doubles (e.g.: results are predictable)

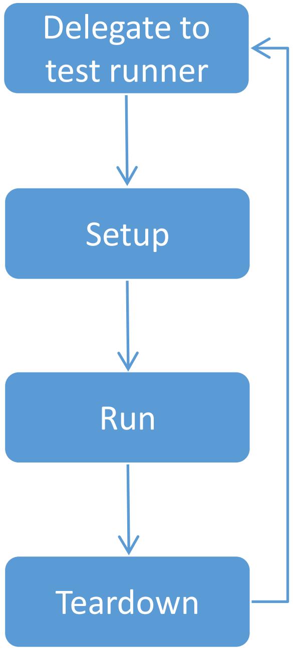
Summary of good practices

- Test method names must be meaningful
- Focus on behavior, not implementation
- Unfinished tests should fail
- Tests should provide fast feedback
- Design should be testable (refactor if needed)
- Tests should not depend on other tests
- A test should have just one reason to fail

Unit test execution cycle

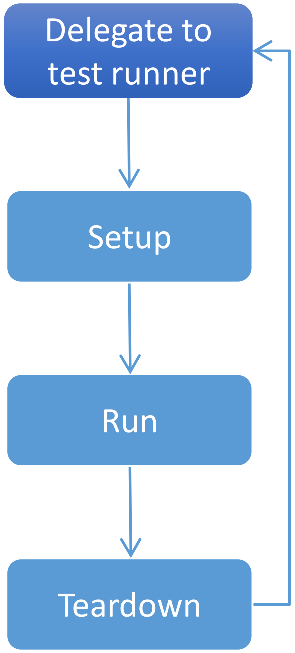
- Select a test runner & create a new instance of the test(s) class(es)
- Invoke any setup method(s) on the test class
- Run test method(s)
- Invoke any teardown method(s) on the test class

A.K.A. *setup-exercise-verify-teardown* cycle



Unit test execution cycle

- Select a test runner & create a new instance of the test(s) class(es)
- Invoke any setup method(s) on the test class
- Run test method(s)
- Invoke any teardown method(s) on the test class



Setup methods

- Help creating *test fixtures*
  - A set of objects needed to *consistently* run the test cases
- Defined using the annotations:

```
@Before
public void <methodName>() { … }
```

- or -

```
@BeforeClass
static public void <methodName>() { … }
```

Setup methods

- Help creating *test fixtures*
  - A set of objects needed to *consistently* run the test cases
- Defined using the annotations:

```
@Before
public void <methodName>() { … }
```

Runs before *each* test method is invoked = *runs n times*

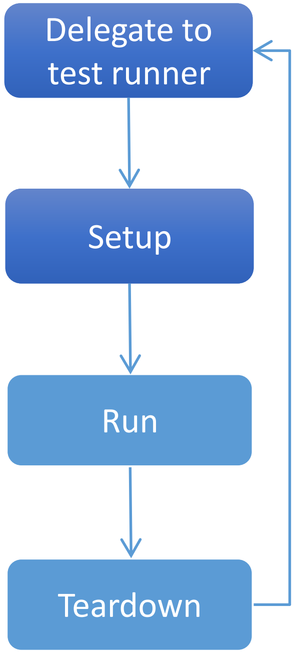
- or -

```
@BeforeClass
static public void <methodName>() { … }
```

Runs before *any* test method is invoked = *runs once*

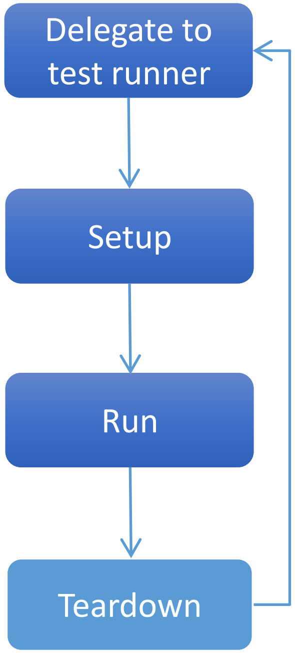
JUnit execution cycle

- Select a test runner & create a new instance of the test(s) class(es)
- Invoke any setup method(s) on the test class
- Run test method(s)
- Invoke any teardown method(s) on the test class



JUnit execution cycle

- Select a test runner & create a new instance of the test(s) class(es)
- Invoke any setup method(s) on the test class
- Run test method(s)
- Invoke any teardown method(s) on the test class



Teardown methods

- They are directed to restore the environment to the same condition it was before running the tests
- Defined using the annotations:

```
@After
public void <methodName>() { … }
```

- or -

```
@AfterClass
static public void <methodName>() { … }
```

Teardown methods

- They are directed to restore the environment to the same condition it was before running the tests
- Defined using the annotations:

```
@After
public void <methodName>() { … }
```

- or -

```
@AfterClass
static public void <methodName>() { … }
```

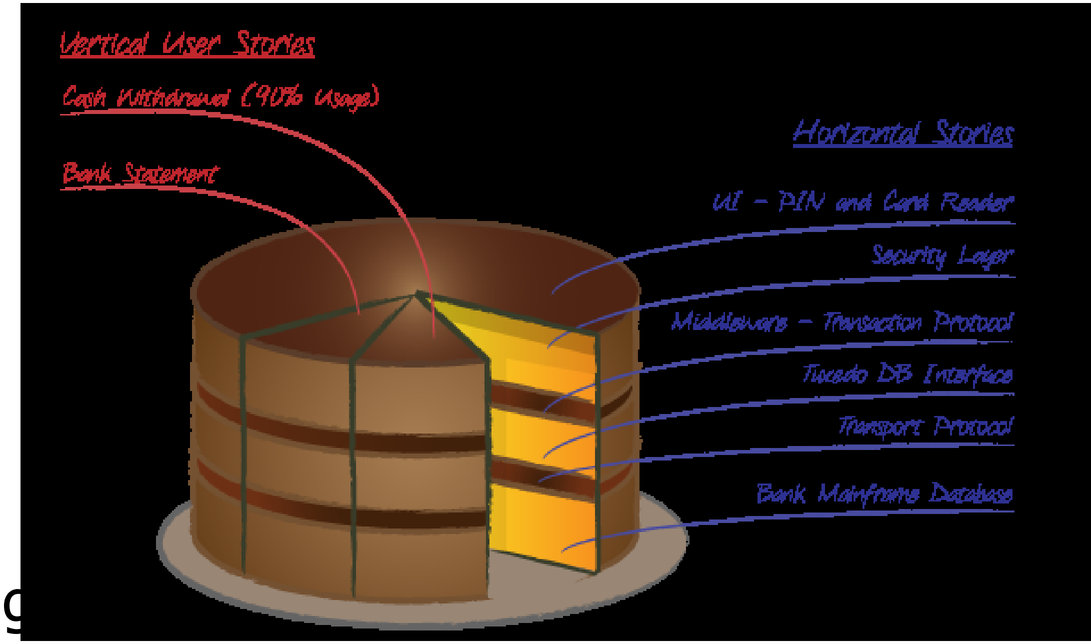
Runs after *each* test method is invoked = *runs n times*

Runs after *all* test methods have been invoked = *runs once*

Slicing

Splitting stories

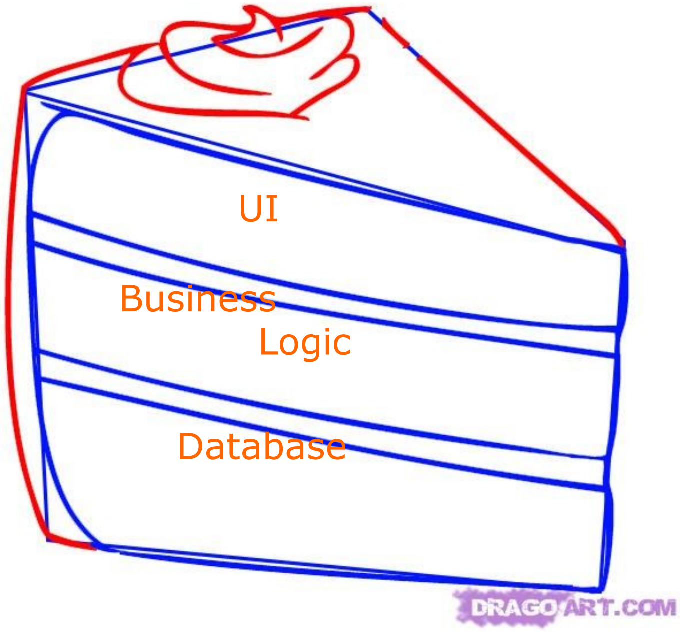
- In agile teams, story splitting is frequent
- Vertical slices are preferred
  - Comply INVEST guidelines
- Stories are split to:
  - Improve understanding, estimation, prioritization
  - Make progress visible, increased team satisfaction
  - Get faster feedback



I	Independent
N	Negotiable
V	Valuable
E	Estimable
S	Sized appropriately or Small
T	Testable

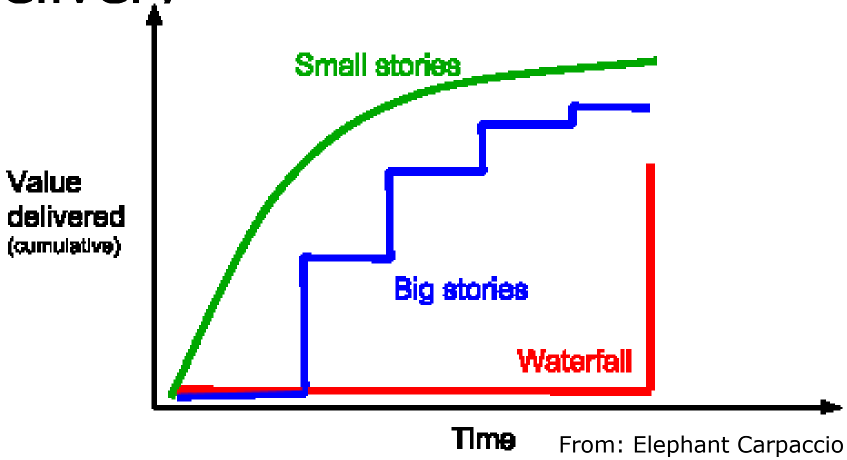
A word of warning

- Depending on the team's organization, tasks can be assigned in different ways
  - Team members have different fields of specialization (e.g.: database, UI, etc.)
    - Horizontal pieces
  - Team members are generalists
    - Vertical slices



Coding

- Stories can be split to trivial levels of complexity
  - Less up-front design, refactoring
  - Faster delivery



From: Elephant Carpaccio facilitation guide

Trivial (but illustrative) example

- Task

  Develop a procedure to calculate a Fibonacci number *n*

$$f_n = f_{n-1} + f_{n-2}$$

being:

$$f_0 = 0$$
$$f_1 = 1$$

Thanks: Timo Räty, Elektrobit, Oulu
Before: Kent Beck

Discussion

- What are the (thin) user stories here?

Discussion

- What are the (thin) user stories here?
- Do f(0), f(1), f(n) fit the concept of a (thin) user story?

Discussion

- What are the (thinner) user stories here?
- Do f(0), f(1), f(n) fit the concept of a (thin) user story?
  - Depends on what we understand by *business value*
  - Taken liberally
  - Let’s call them **slices**, to avoid confusion

Trivial (but illustrative) example

$$f_n = f_{n-1} + f_{n-2}$$
$$f_0 = 0$$
$$f_1 = 1$$

- Slices
  1. Calculate f(0)
  2. Calculate f(1)
  3. Calculate f(n)   *(recursive)*

Code

```
    /**
     * Function that calculates Fibonacci's number
(recursive)
     * @param n
     * @return     Fibonacci's number
     */
    public static long calculate(int n) {
        if (n == 0)
            return 0;

        if (n == 1)
            return 1;

        return(calculate(n-1) + calculate(n-2));
    }
```

No defensive programming!

Code

(with some defensive elements)

```
    /**
     * Function that calculates Fibonacci's number
(recursive)
     * @param n
     * @return      Fibonacci's number
     */
       if ((n > 92) || (n < 0))
           return -1;

       if (n == 0)
           return 0;

       if (n == 1)
           return 1;

       return(calculate(n-1) + calculate(n-2));
    }
```

Do not advocate defensive programming

(regular) Assertions or contract-based approaches are also possible

Test last approaches

Test last

- We all acknowledge that before releasing a US, it has to be tested
  - Our concern is UT, not integration or system testing
- The typical approach is running the tests after the code was complete
  - 1 US → 1 testing session
  - **Test-last approach**

Discussion

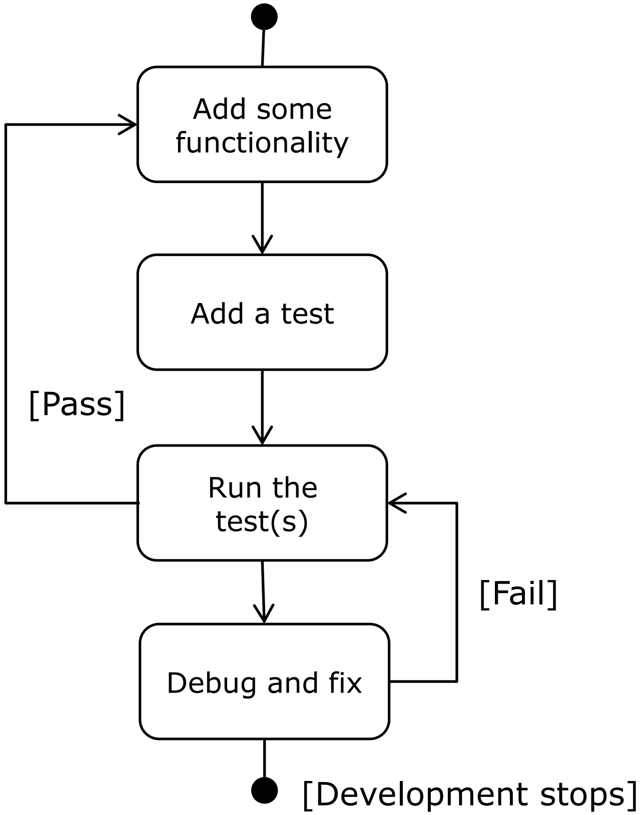
- When USs are further split,

Do we wait to test until the top level US is complete?

Incremental test last

- When USs are further split, each slice can be individually tested
  - Each slice produces a tiny piece of functionality
  - This functionality is retained throughout subsequent increments
- This strategy is called **incremental test last**

Incremental test-last



Incremental test last

- When USs are further split, each slice can be individually tested
  - Each slice produces a tiny piece of functionality
  - This functionality is retained throughout subsequent increments
- This strategy is called **incremental test last**
  - Only possible with automated tests

Good Practice: No production code without test code