

Introduction to Software Testing (*2nd edition*) Chapter 4

TDD Example

Paul Ammann & Jeff Offutt

<http://www.cs.gmu.edu/~offutt/softwaretest/>

September 2018

A Simple Example Using TDD

- TDD is different and feels awkward at first
 - Even if you don't decide to fully invest in TDD, trying it can help you think about testing and development in a different way
- TDD is most useful when faced with complex design decisions
- This example is based on Calc from chapter 3
 - No complex design decisions
 - The intent is to help you understand the process
- Calc is a class to perform simple calculator operations

Starting Point

We start TDD with no code, design, or requirements

We choose an initial piece of functional behavior ...

and write the first test ...

First Calc Test

```
import org.junit.Test;
import static org.junit.Assert.*;

public class CalcTest
{
    @Test public void testAdd()
    {
        assertTrue ("Calc sum incorrect",
            5 == Calc.add (2, 3));
    }
}
```

Class will
be called
"Calc"


Method signature:

- Name: add()
- 2 int parameters
- Returns int

This won't compile (red light!), so let's write
minimal code to make it compile ...

First Implementation of Calc

```
public class Calc
{
    static public int add (int a, int b)
    {
        return 5;
    }
}
```



Compiles
Test runs correctly
Green light!

Why “5”?

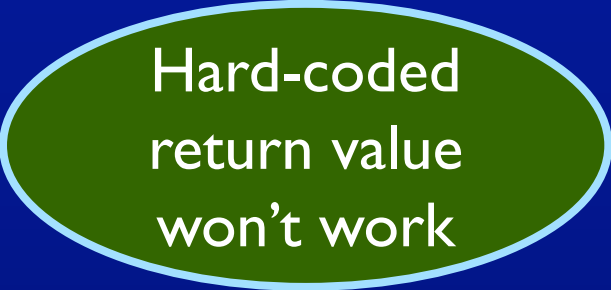
“Minimal” code to make test work

May seem silly at first, but that’s TDD

Second Calc Test

```
@Test public void testAddB()
{
    assertTrue ("Calc sum incorrect",
        7 == Calc.add (3, 4));
}
```

Hard-coded
return value
won't work



This will fail (red light!), so we have to change
the Calc class ...

Modification of Calc

```
public class Calc
{
    static public int add (int a, int b)
    {
        return a+b;
    }
}
```



Compiles
Test runs correctly
Green light!

Refactor?

Calc class is probably okay, but the first test could be changed to “testAddA()” for consistency

Third Calc Test

```
@Test public void testSubtract()  
{  
    assertTrue ("Calc difference incorrect",  
        5 == Calc.subtract (10, 5));  
}
```

Two decisions:

- name: subtract()
- First parameter is minuend

Now we implement the subtract() method ...

Continuing with Calc

You can continue this process on your own
(subtract, multiply, divide, memory, exponents, logs,
numeric bases, etc.)

TDD is not necessary for a class this
simple, but it's a good way to learn

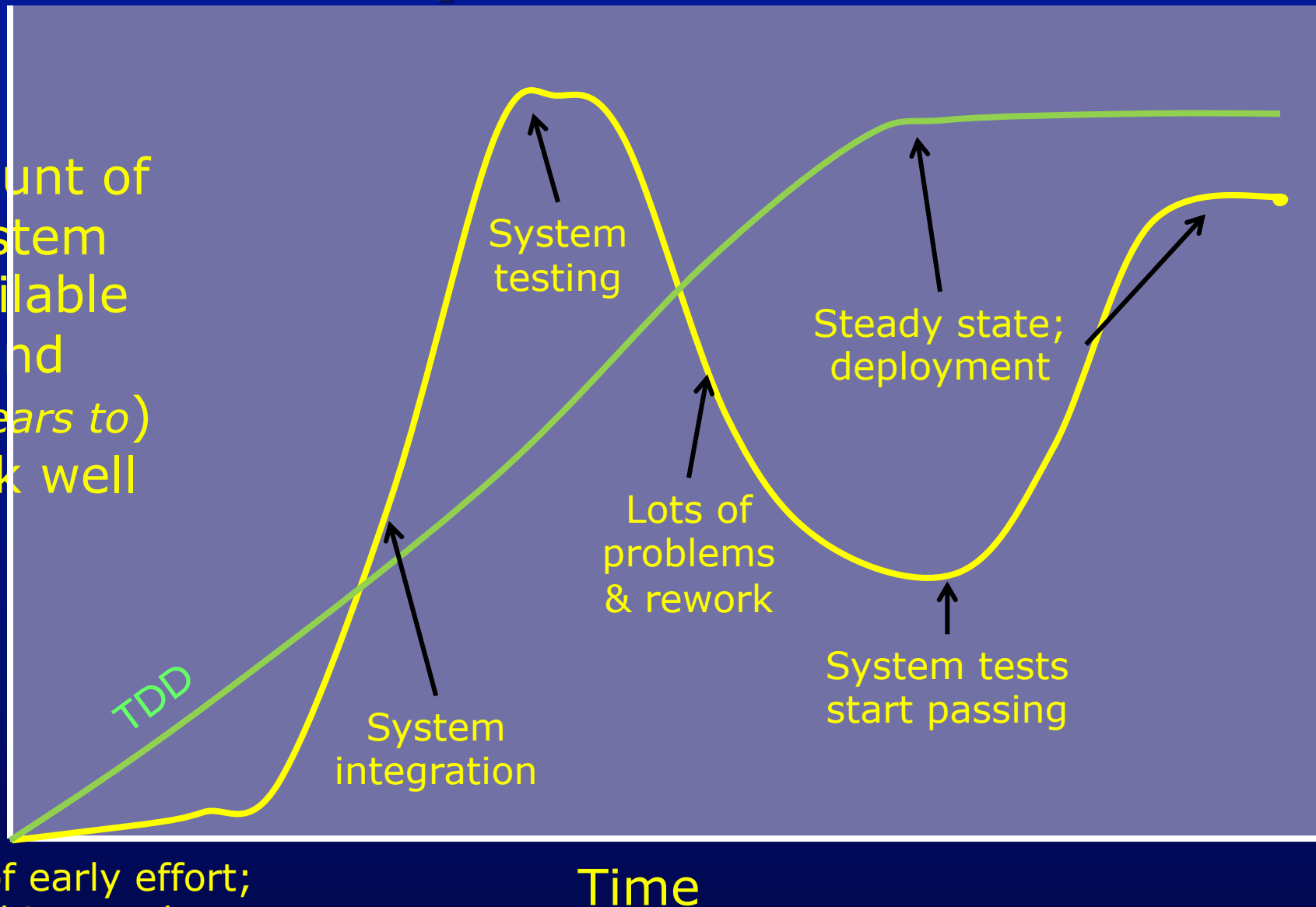
TDD can provide some benefits with
more complicated design decisions

Expected Benefits of TDD

- Program **always works** as expected
 - Correct relative to current tests
- Decisions are made when needed
 - Not weeks or months early
 - Decisions are more likely to be **correct**
- End result is more **reliable**
- Future **modifications** ...
 - Are cheaper
 - Are easier
 - Are less frequent

TDD & System Readiness

Amount of
system
available
and
(*appears to*)
work well



Lots of early effort;
nothing works

Time