Test Driven Development "TDD"

A process focusing on reliability, progress, and confidence

Clean code that works

- *To ensure that our software is reliable all the time*
  - *"Clean code **that works**"*
- *To ensure fast development progress*
- *To ensure that we dare restructure our software and its architecture*
  - *"**Clean code** that works"*

The Values

Keep focus

- Make one thing only, at a time!
- *Often*
  - *Fixing this, requires fixing that, hey this could be smarter, fixing it, ohh – I could move that to a library, hum hum, …*

Take small steps

- Taking small steps allow you to backtrack easily when the ground becomes slippery
- *Often*
  - *I can do it by introducing these two classes, hum hum, no no, I need a third, wait…*

The Values

Speed

- You are what you do! Deliver every 14 days!!!
- *Often*
  - *After two years, half the system is delivered, but works quite in another way than the user anticipate/wants…*
- Speed, not by being sloppy but by making less functionality of superior quality!

Simplicity

- Maximize the amount of work *not done!*
- *Often*
  - *I can make a wonderful recursive solution parameterized for situations X, Y and Z (that will never ever occur in practice)*

You are all employed today

Welcome to *PayStation Ltd.*

We will develop the main software to run pay stations.

[Demo]

Case: Pay Station

Welcome to *PayStation Ltd.*

Customer: AlphaTown

Requirements

- accept coins for payment
  - 5, 10, 25 cents
- show time bought on display
- print parking time receipts
- US: 2 minutes cost 5 cent
- handle buy and cancel

Stories

Story 1: Buy a parking ticket. A car driver walks to the pay station to buy parking time. He enters several valid coins (5, 10, and 25 cents) as payment. For each payment of 5 cents he receives 2 minutes parking time. On the pay station's display he can see how much parking time he has bought so far. Once he is satisfied with the amount of time, he presses the button marked "Buy". He receives a printed receipt, stating the number of minutes parking time he has bought. The display is cleared to prepare for another transaction.

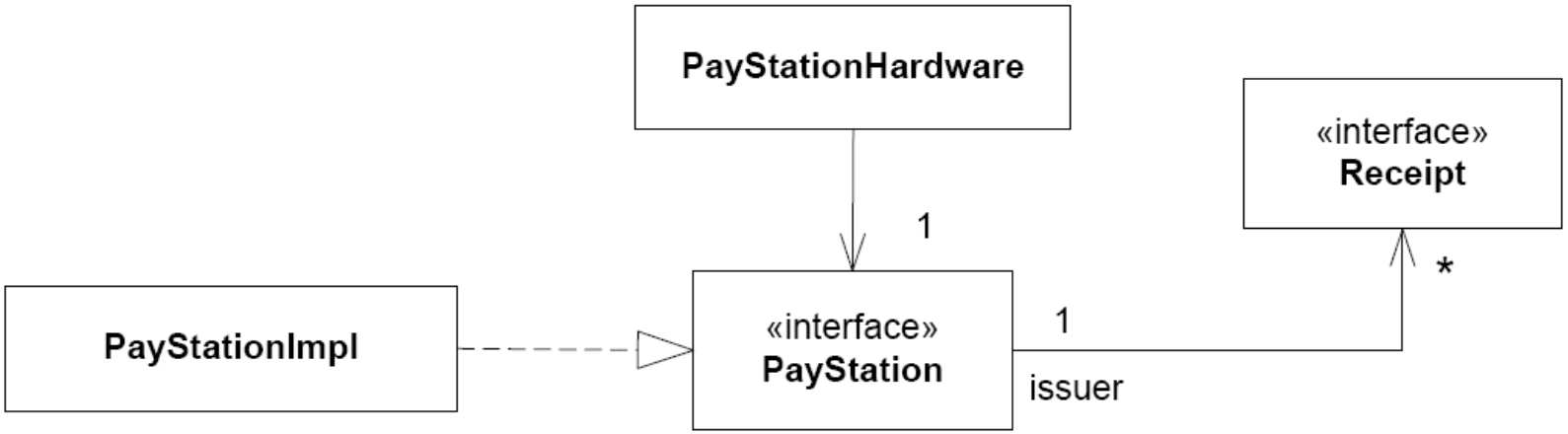
Story 2: Cancel a transaction. A driver has entered several coins but realize that the accumulated parking time shown in the display exceeds what she needs. She presses the button marked "Cancel" and her coins are returned. The display is cleared to prepare for another transaction.

Story 3: Reject illegal coin. A driver has entered 50 cents total and the display reads "20". By mistake, he enters a 1 euro coin which is not a recognized coin. The pay station rejects the coin and the display is not updated.

Design: Static View

For our purpose the design is given…

- Central *interface* **PayStation**



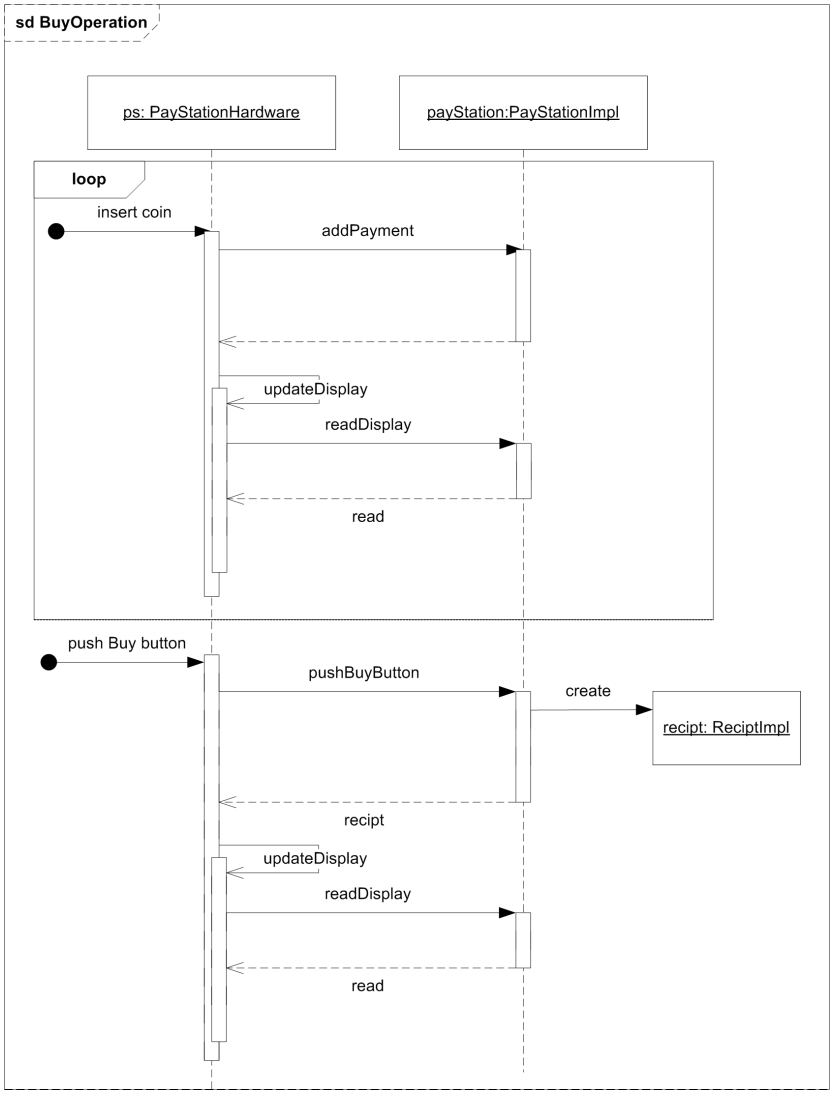
Design: Code View

```
public interface PayStation {

  /**
   * Insert coin into the pay station and adjust state accordingly.
   * @param coinValue is an integer value representing the coin in
   * cent. That is, a quarter is coinValue=25, etc.
   * @throws IllegalCoinException in case coinValue is not
   * a valid coin value
   */
  public void addPayment( int coinValue ) throws IllegalCoinException;

  /**
   * Read the machine's display. The display shows a numerical
   * description of the amount of parking time accumulated so far
   * based on inserted payment.
   * @return the number to display on the pay station display
   */
  public int readDisplay();

  /**
   * Buy parking time. Terminate the ongoing transaction and
   * return a parking receipt. A non-null object is always returned.
   * @return a valid parking receipt object.
   */
  public Receipt buy();

  /**
   * Cancel the present transaction. Resets the machine for a new
   * transaction.
   */
  public void cancel();
}
```

```
public interface Receipt {

  /**
   * Return the number of minutes this receipt is valid for.
   * @return number of minutes parking time
   */
  public int value();
}
```

Design: Dynamic View

The envisioned dynamics…



Minimum Terminology

[FRS chapter 2]

- **Production code:** source code that compiles to machine executable code that *will provide behavior fulfilling the requirements of the user/customer.*
- **Testing code**: source code that defines test cases for the production code.
- **Failure:** the damn thing does something wrong
- **Defect:** algorithmic cause of a failure
- **Test case:** definition of input values for a unit under test and the expected output
  - **Input: -37; UUT: Math.abs(x); expected output: +37**

AARHUS UNIVERSITET

TDD

Structuring the Programming Process

How do I program?

Before TDD I had programmed for ~25 years...

But – I *just did it*...

I could not really express what I did!

A teaching assistant once told our students that he himself used *divine inspiration* when programming. A great help to beginners!

Structuring the Process

TDD replace *tacit knowledge* with a formulated process

- *The rhythm*
  - The **five steps** in each highly focussed and fast-paced iteration.
- *Testing principles*
  - The **testing principles** that defines a term for a specific action to make in each step
  - Form: Name - Problem – Solution

The Fundamental TDD Principles

The Three Cornerstones in TDD

Principle: Test

TDD Principle: **Automated Test**

How do you test your software? Write an automated test.

Locke, Berkeley & Hume:

Anything that can't be measured

does not exist.

Kent Beck:

Software features that can't be demonstrated by automated tests simply don't exist.

Test

- Tests are often by developers considered something that *other stakeholders* are interested in...

- *Testing is boring...*
- But automated tests are something that we should write for our own sake...
  - to have confidence in our code
  - to dare change code radically
  - to know when we are done

- You will get a chance to judge for yourselves...

Principle: Test First

TDD Principle: Test First

When should you write your tests? Before you write the code that is to be tested.

Because you won't test after!

- Time has validated this observation ☺

My own experience

Developing module X

- write driver code “test program” for X
- edit-compile-debug...

Integrate X

- change X’s API here and there

Bang! Why does X not work???

- I did have a test program didn’t I?
- If I find it, it would take ages to make it run again
  - and is the bug in X or in the rewritten test program?

TDD principle

TDD Principle: **Test List**

What should you test? Before you begin, write a list of all the tests you know you will have to write. Add to it as you find new potential tests.

- *"Never take a step forward unless you know where your foot is going to land".* What is it we want to achieve in this iteration ???

- Another strategy: Keep it all in your head.
  - if you are a genius
  - if your problem is a "Mickey Mouse" problem

The Rhythm

The Iteration Skeleton

Each TDD iteration follows the Rhythm

The TDD Rhythm:

1. Quickly add a test
2. Run all tests and see the new one fail
3. Make a little change
4. Run all tests and see them all succeed
5. Refactor to remove duplication

(6. All tests pass again after refactoring!)

The Rhythm: Red-Green-Refactor

The Rhythm

Improve code quality

Clean part

Implement delta-feature that does not break any existing code

Works part

Introduce test of delta-feature

Size of an iteration

An iteration is *small* typically adding a very very small increment of behavior to the system

Iterations (=all 5 steps) typically last from 1 to 15 minutes. If it becomes bigger it is usually a sign that you *do not take small steps* and have *lost focus!*

Pay Station by TDD

Back to the Pay Station

Let's combine these first principles

- Test, Test First, Test List

Our starting point is the interfaces that define the domain model / business logic.

- User interface logic we can add later...

- that is: PayStation and Receipt

Exercise: Test List

Generate the Test List for these stories

Story 1: Buy a parking ticket. A car driver walks to the pay station to buy parking time. He enters several valid coins (5, 10, and 25 cents) as payment. For each payment of 5 cents he receives 2 minutes parking time. On the pay station's display he can see how much parking time he has bought so far. Once he is satisfied with the amount of time, he presses the button marked "Buy". He receives a printed receipt, stating the number of minutes parking time he has bought. The display is cleared to prepare for another transaction.

Story 2: Cancel a transaction. A driver has entered several coins but realize that the accumulated parking time shown in the display exceeds what she needs. She presses the button marked "Cancel" and her coins are returned. The display is cleared to prepare for another transaction.

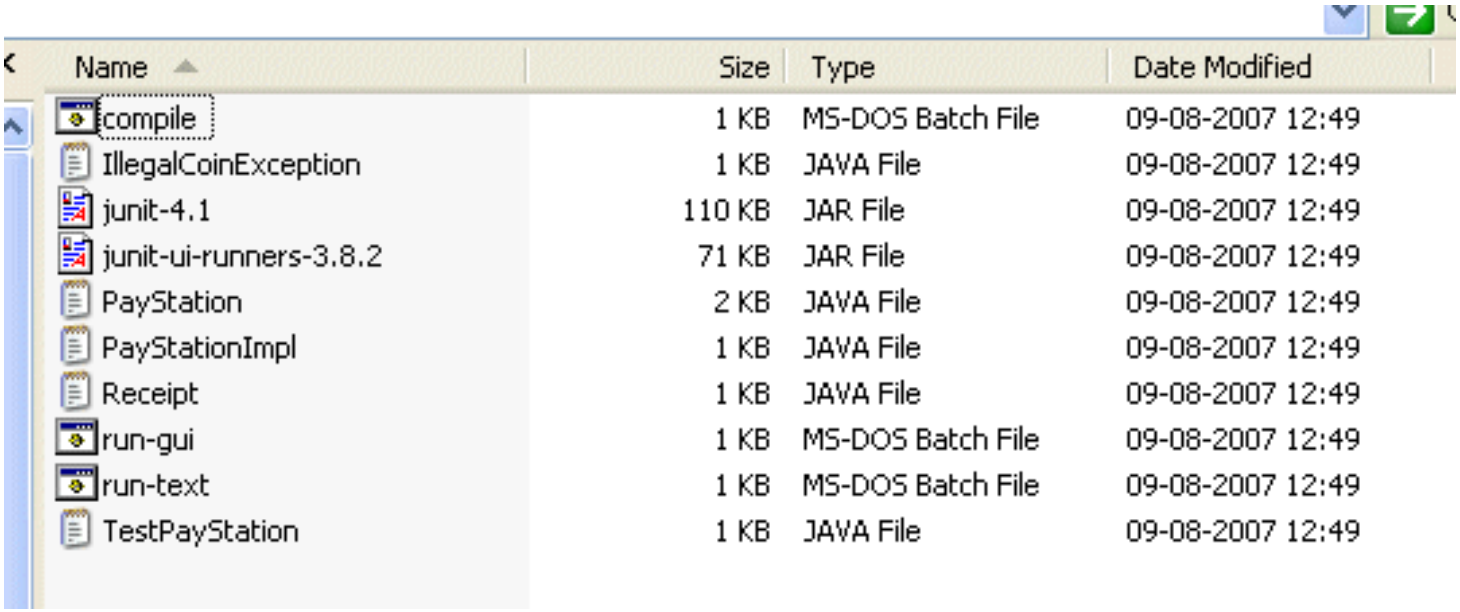
Story 3: Reject illegal coin. A driver has entered 50 cents total and the display reads "20". By mistake, he enters a 1 euro coin which is not a recognized coin. The pay station rejects the coin and the display is not updated.

My answer

* accept legal coin
* 5 cents should give 2 minutes parking time
* reject illegal coin
* readDisplay
* buy produces valid receipt
* cancel resets pay station

Iteration 0: Setting Up

Demo



Name	Size	Type	Date Modified
compile	1 KB	MS-DOS Batch File	09-08-2007 12:49
IllegalCoinException	1 KB	JAVA File	09-08-2007 12:49
junit-4.1	110 KB	JAR File	09-08-2007 12:49
junit-ui-runners-3.8.2	71 KB	JAR File	09-08-2007 12:49
PayStation	2 KB	JAVA File	09-08-2007 12:49
PayStationImpl	1 KB	JAVA File	09-08-2007 12:49
Receipt	1 KB	JAVA File	09-08-2007 12:49
run-gui	1 KB	MS-DOS Batch File	09-08-2007 12:49
run-text	1 KB	MS-DOS Batch File	09-08-2007 12:49
TestPayStation	1 KB	JAVA File	09-08-2007 12:49

Compile and Execution

Supplied code has a ‘compile.bat’ script

- compile.sh for linux

It will compile the java code (surprise!)

Next, run the script ‘run-test.bat’

- Verbose output ☹

Or you do the magic in your favorite IDE ☺

- *Eclipse has very nice support for JUnit*

JUnit 4.x Raw (no GUI ☹)

```
JUnit version 4.4
.E
Time: 0,047
There was 1 failure:
1) shouldDisplay2MinFor5Cents(TestPayStation)
java.lang.AssertionError: Should display 2 min for 5 cents
  expected:<2> but was:<0>

          at org.junit.Assert.fail(Assert.java:74)
          at org.junit.Assert.failNotEquals(Assert.java:448)
          at org.junit.Assert.assertEquals(Assert.java:102)
          at org.junit.Assert.assertEquals(Assert.java:323)
          at TestPayStation.shouldDisplay2MinFor5Cents(TestPayStation.java:20)
  [lines removed here]
          at org.junit.runner.JUnitCore.main(JUnitCore.java:44)

  FAILURES!!!
  Tests run: 1,  Failures: 1
```

Testing Code

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

/** Testcases for the Pay Station system.

   From the book "Reliable and Flexible Software Explained"
   Copyright: 2010 CRC Press
   Author: Henrik B Christensen
*/
public class TestPayStation {

  /**
   * Entering 5 cents should make the display report 2 minutes parking
   * time.
  */
  @Test
  public void shouldDisplay2MinFor5Cents() throws IllegalCoinException {
    PayStation ps = new PayStationImpl();
    ps.addPayment( 5 );
    assertEquals( "Should display 2 min for 5 cents",
                  2, ps.readDisplay() );
  }
}
```

JUnit version

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

/** Testcases for the Pay Station system.

   From the book "Reliable and Flexible Software Explained"
   Copyright: 2010 CRC Press
   Author: Henrik B Christensen
*/
public class TestPayStation {

  /**
   * Entering 5 cents should make the display report 2 minutes parking
   * time.
   */
  @Test
  public void shouldDisplay2MinFor5Cents() throws IllegalCoinException {
    PayStation ps = new PayStationImpl();
    ps.addPayment( 5 );
    assertEquals( "Should display 2 min for 5 cents",
                  2, ps.readDisplay() );
  }
}
```

expected value

computed value

(TestNG version)

```
package paystation.domain;

import org.testng.annotations.*;

/** Testcases for the Pay Station system.

   This source code is from the book
     "Flexible, Reliable Software:
       Using Patterns and Agile Development"
     published 2010 by CRC Press.
   Author:
     Henrik B Christensen
     Department of Computer Science
     University of Aarhus

   This source code is provided WITHOUT ANY WARRANTY either
   expressed or implied. You may study, use, modify, and
   distribute it for non-commercial purposes. For any
   commercial use, see http://www.baerbak.com/
*/
public class TestNGPayStation {
  /**
   * Entering 5 cents should make the display report 2 minutes
   * parking time.
   */
  @Test
  public void shouldDisplay2MinFor5Cents() throws IllegalCoinException {
    PayStation ps = new PayStationImpl();
    ps.addPayment( 5 );
    assert 2 == ps.readDisplay();
  }
}
```

expected value

computed value

(NUnit Version)

```
1  using NUnit.Framework;
2
3  /** Testcases for the Pay Station system.
4
5      Author: (c) Henrik Bærbak Christensen 2006
6  */
7
8  [TestFixture]
9  public class TestPayStation {
10
11   /** Testing that a nickel gives two minutes parking time */
12   [Test]
13   public void testEnterNickel() {
14
15     PayStation ps = new PayStationImpl();
16     ps.addPayment( 5 );
17     Assert.AreEqual( 2, ps.readDisplay() );
18   }
19
20 }
```

expected value

computed value

Iteration 1: 5 ¢ = 2 min Parking

Which one to pick

OK, I have the initial test list and I have the rhythm…

Where do I start???

> TDD Principle: **One Step Test**
>
> Which test should you pick next from the test list? Pick a test that will teach you something and that you are confident you can implement.

Step 1: Quickly add a test

- The test case
  - ps.addPayment(5);
  - ps.readDisplay() == 2;

In JUnit

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

/** Testcases for the Pay Station system.

   From the book "Reliable and Flexible Software Explained"
   Copyright: 2010 CRC Press
   Author: Henrik B Christensen
*/
public class TestPayStation {

  /**
   * Entering 5 cents should make the display report 2 minutes parking
   * time.
  */
  @Test
  public void shouldDisplay2MinFor5Cents() throws IllegalCoinException {
    PayStation ps = new PayStationImpl();
    ps.addPayment( 5 );
    assertEquals( "Should display 2 min for 5 cents",
                  2, ps.readDisplay() );
  }
}
```

Step 2

Step 2: Run all tests and see the new one fail

- Require that I implement a PayStationImpl *Temporary Test Stub*
  - All methods are *empty* or *return null*

```
public class PayStationImpl implements PayStation {
  private int insertedSoFar;
  private int timeBought;

  public void addPayment( int coinValue )
    throws IllegalCoinException {
  }
  public int readDisplay() {
    return 0;
  }
  public Receipt buy() {
    return null;
  }
  public void cancel() {
  }
}
```

Step 2

```
JUnit version 4.4
.E
Time: 0,047
There was 1 failure:
1) shouldDisplay2MinFor5Cents(TestPayStation)
java.lang.AssertionError: Should display 2 min for 5 cents
  expected:<2> but was:<0>

        at org.junit.Assert.fail(Assert.java:74)
        at org.junit.Assert.failNotEquals(Assert.java:448)
        at org.junit.Assert.assertEquals(Assert.java:102)
        at org.junit.Assert.assertEquals(Assert.java:323)
        at TestPayStation.shouldDisplay2MinFor5Cents(TestPayStation.java:20)
 [lines removed here]
        at org.junit.runner.JUnitCore.main(JUnitCore.java:44)

 FAILURES!!!
 Tests run: 1,  Failures: 1
```

Step 3

3. Make a little change

Exercise: What should I do?

Remember:

- *Keep focus!!!*
- *Take small steps!!!*

```
public class PayStationImpl implements PayStation {

  public void addPayment( int coinValue )
          throws IllegalCoinException {
  }

  public int readDisplay() {
    return 0;
  }

  public Receipt buy() {
    return null;
  }

  public void cancel() {
  }
}
```

Step 3

TDD Principle: **Fake It ('Til You Make It)**

What is your first implementation once you have a broken test? Return a constant. Once you have your tests running, gradually transform it.

This principle was *very* controversial to me!

Implement a solution that is known to be wrong and that must be deleted in two seconds???

Why???

Test-*Driven*

It is called test-*driven* because it is driven by tests...

Key Point: Production code is driven into existence by tests

In the extreme, you do not enter a single character into your production code unless there is a test case that demands it.

We only have one test case, 5c = 2 min, and the *simplest possible implementation* is 'return 2;'. No other test case force us to anything more!

Fake it ???

Fake it because:

- **focus!** You keep focus on the task at hand! Otherwise you often are lead into implementing all sorts of other code...

- **small steps!** You move faster by making many small steps rapidly than leaping,falling, and crawling back up all the time...

Warstory

I just have to change A a bit. However I soon find out that I need a new class B. Introducing B I can remove a lot of code from C; however this requires D to be massaged a bit. While doing this I discover a real bad bug in E that I have to fix first...

After two days – nothing at *all* is working – and I have no clue at all why I began all these changes...

Focus control

Fake It allows me to focus on the immediate problem:

I change A a bit. Done!

Step 4.

4. Run all tests and see them all succeed.

Nice feeling of success ☺

Remember to note the success on the test list

~~insert legal coin~~
~~5 cents should give 2 minutes parking time.~~
insert illegal coin
~~readDisplay~~
buy
cancel

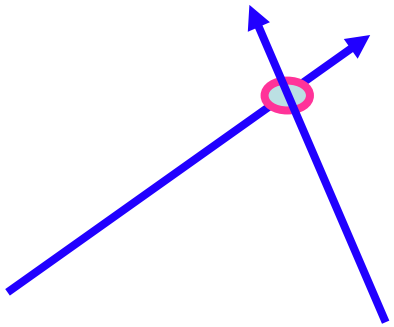
But – of course I am concerned! The implementation is wrong!

Step 4.

The point is that one test case is not enough to ensure a reliable implementation of the rate calculation! I need more test cases to drive the implementation.

TDD Principle: **Triangulation**

How do you most conservatively drive abstraction with tests? Abstract only when you have two or more examples.



Triangulation

The key point here is that *return 2* is actually the **correct** implementation of the *readDisplay* if the only occuring use case is a person buying for 5 cents!

The *conservative* way to *drive* a more correct implementation is to add more examples/stories/scenarios => more test cases!

The above implementation is *not* correct for entering e.g. 25 cents!

Triangulating

So I simply **remind myself** that *Fake It* is playing around in the production code by adding it to the test list:

~~insert legal coin~~
insert illegal coin, exception
~~5 cents should give 2 minutes parking time.~~
~~readDisplay~~
buy
~~cancel~~
25 cents = 10 minutes

Step 5

Refactor to remove duplication

I will come back to this....

Iteration 2: Rate Calculation

25 cent = 10 minutes

Step 1

1: Quickly add a test.

– but where?

TDD Principle: Isolated Test

How should the running of tests affect one another? Not at all.

Exercise: Why?

Step 1

Isolated Test guards you against the *ripple effect*

- Test 1 fails,
  - leaving objects in another state than if it had passed
- Test 2 assumes the object state left by Test 1
  - but is it different from that assumed – and Test 2 fails
- ... and all tests fail due to one single problem.

Morale: Isolate in order to overview failure consequences.

[Live programming]

Step 2: Fail…

Step 3: Make a little change…

```
public class PayStationImpl implements PayStation {
  private int insertedSoFar;
  public void addPayment( int coinValue )
    throws IllegalCoinException {
    insertedSoFar = coinValue;
  }
  public int readDisplay() {
    return insertedSoFar / 5 * 2;
  }
  public Receipt buy() {
    return null;
  }
  public void cancel() {
  }
}
```

Hi – this is wrong isn't it???

- What should I do ??

Tests drive implementation

* ~~accept legal coin~~
* reject illegal coin, exception
* ~~5 cents should give 2 minutes parking time.~~
* ~~readDisplay~~
* buy produces valid receipt
* cancel resets pay station
* ~~25 cents = 10 minutes~~
* enter two or more legal coins

Step 5.

Definition: **Refactoring**

Refactoring is the process of modifying and restructuring the source code to improve its maintainability and flexibility without affecting the system's external behavior when executing.

Testing code is also best maintained. I have duplicated the code that sets up the pay station object. I can move this to a *Fixture* which you define using the @Before annotation (JUnit) og @BeforeMethod annotation (TestNG).

@Before

```
1  import org.junit.*;
2  import static org.junit.Assert.*;
3
4  /** Testcases for the Pay Station system.
5      Author: (c) Henrik Bærbak Christensen 2006 */
6
7  public class TestPayStation {
8    PayStation ps;
9    /** Fixture for pay station testing. */
10   @Before
11   public void setUp() {
12     ps = new PayStationImpl();
13   }
14
15   /** Testing that a nickel gives two minutes parking time */
16   @Test
17   public void testEnterNickel() throws IllegalCoinException {
18     ps.addPayment( 5 );
19     assertEquals( 2, ps.readDisplay() );
20   }
```

The @Before method *always* run before *each* test method. This way each test case starts in a known and stable object configuration.

Magic constants

```
1  import org.junit.*;
2  import static org.junit.Assert.*;
3
4  /** Testcases for the Pay Station system.
5      Author: (c) Henrik Bærbak Christensen 2006 */
6
7  public class TestPayStation {
8    PayStation ps;
9    /** Fixture for pay station testing. */
10   @Before
11   public void setUp() {
12     ps = new PayStationImpl();
13   }
14
15   /** Testing that a nickel gives two minutes parking time */
16   @Test
17   public void testEnterNickel() throws IllegalCoinException {
18     ps.addPayment( 5 );
19     assertEquals( 2, ps.readDisplay() );
20   }
```

TDD Principle: **Evident Data**

How do you represent the intent of the data? Include expected and actual results in the test itself, and make their relationship apparent. You are writing tests for the reader, not just for the computer.

Evident Data

```
/**
 * Entering 5 cents should make the display report 2 minutes
 * parking time.
*/
@Test
public void shouldDisplay2MinFor5Cents() throws IllegalCoinException {
  ps.addPayment( 5 );
  assertEquals( "Should display 2 min for 5 cents",
                5 / 5 * 2, ps.readDisplay() );
}

/**
 * Entering 25 cents should make the display report 10 minutes
 * parking time.
*/
@Test
public void shouldDisplay10MinFor25Cents() throws IllegalCoinException {
  ps.addPayment( 25 );
  assertEquals( "Should display 10 min for 25 cents",
                25 / 5 * 2, ps.readDisplay() );
  // 25 cent in 5 cent coins each giving 2 minutes parking
}
```

Iteration 3: Illegal Coin

The standard 17 cent coin ☺

JUnit Syntax

JUnit allows you to state the exception a given test method *must* throw to pass:

```
/** Testing for illegal coin entry. */
@Test(expected=IllegalCoinException.class)
public void testEnterIllegalCoin() throws IllegalCoinException {
  ps.addPayment(17);
}
```

Iteration 4: Two Valid Coins

One Step Test: let us close the holes…

Step 1

What coins to use???

- I have already used a 5 and 25 cent, but have not tried 10 cent yet.

TDD Principle: **Representative Data**

What data do you use for your tests? Select a small set of data where each element represents a conceptual aspect or a special computational processing.

So – I decide to use a dime also!

- (but choosing the proper input values is a big topic on its own right!)

Rhythm

Step 1:

```
@Test
public void shouldDisplay14MinFor10and25Cents() throws IllegalCoinException {
  ps.addPayment(25);
  ps.addPayment(10);
  assert (10+25) / 5 * 2 == ps.readDisplay();
}
```

Step 2: RED

Step 3:

- insertedSoFar += coinValue;

Step 4: Huhh??? Fail???

- What happened?

Iteration 5: Buy

The Buy Scenario

Step 1+2

```
/**
 * Buy should return a valid receipt of the
 * proper amount of parking time
*/
@Test
public void shouldReturnCorrectReceiptWhenBuy()
        throws IllegalCoinException {
  ps.addPayment(5);
  ps.addPayment(10);
  ps.addPayment(25);
  Receipt receipt;
  receipt = ps.buy();
  assertNotNull( "Receipt reference cannot be null",
                 receipt );
  assertEquals( "Receipt value must be 16 min.",
                (5+10+25) / 5 * 2 , receipt.value() );
}
```

Now – step 3 – *make a little change...*

3. Make a little change

Ups? *Little* change??? We need *two* changes

- An implementation of Receipt
- Implementing the buy method

Small steps? What are my options?

- The old way: Do both in one go!
  - (I trust I could do that after 20 years of coding experience. However, the first time I did it last year, I actually got it mixed up. It was quite late though... ☺)
- Fix receipt first, buy next...
- Fix buy first, implement receipt later...

What would you do?

A) Do both in one go!

B) Fix receipt first, buy next...

C) Fix buy first, implement receipt later...

Let us vote ☺

Analysis

Take small steps tells us either B or C option:

- Fix receipt first, buy next...
  - This is the natural order, because buy *depends upon* receipt, and not the other way around
  - but I break the buy iteration!!!
    - ***I have lost focus!***
    - Implementing Receipt means fixing a bug in B, that require a new class C, that would be better of if D had another method, that...
- Complete buy first – do receipt next
  - But how on earth can I do that given the dependency structure?

What is your answer?

... and the winner is ...

[Drum roll...]

Fake it

Return a constant object

Fake it

Of course!

This is the whole point of Fake It!

(I has taken me a while to see that – but Fake It keeps me **focused**.)

I can complete buy by making a *fake receipt*. I keep focus! I am not lead astray.

Fake Receipt

Java supports anonymous inner classes that do the job beautifully.

```
public class PayStationImpl implements PayStation {
  private int insertedSoFar;
  public void addPayment( int coinValue )
    throws IllegalCoinException {
    switch ( coinValue ) {
    case 5: break;
    case 10: break;
    case 25: break;
    default: throw new IllegalCoinException("Invalid coin: "+coinValue);
    }
    insertedSoFar += coinValue;
  }
  public int readDisplay() {
    return insertedSoFar / 5 * 2;
  }
  public Receipt buy() {
    return new Receipt() {
      public int value() { return (5+10+25) / 5 * 2; }
    };
  }
  public void cancel() {
  }
}
```

Step 4

Pass. Remember to ensure **Triangulation** down the road – update the test list:

~~insert legal coin~~
insert illegal coin, exception
~~5 cents should give 2 minutes parking time.~~
~~readDisplay~~
buy for 40 cents
buy for 100 cents
cancel
~~25 cents = 10 minutes~~
receipt value

Iteration 5: Receipt

Step 1

Step 1 actually involves design in the small. How do I construct receipts?

TDD Principle: **Assert First**

When should you write the asserts? Try writing them first.

Step 3

TDD Principle: **Obvious Implementation**

How do you implement simple operations? Just implement them.

The [Fake It, Triangulation] cousins are great for 'driving' algorithm development. However the really trivial algorithms we simply code

- Simple =
  - set/get methods
  - add a parameter in constructor that is assigned an instance variable
  - that is code of 3-5 lines simple complexity ☺

Iteration 6: Buy (Real)

Step 1

Buy for 100 cent.

Exercise:

But how to enter 100 cent?

- add 5, add 5, add 5, add 10, add ...
- for (int i = 0; i <= 20; i++) { add 5; }
- private method add2Quarters()

Evident Tests

TDD Principle: **Evident Tests**

How do we avoid writing defective tests? By keeping the testing code evident, readable, and as simple as possible.

Avoid loops, conditionals, recursion, **complexity** in your testing code.

Testing code is as **dumb** as possible.

Because testing code is **code** and you make mistakes in code!

- assignment, creation, private method calls
- and not much else!

Step 5

Refactoring

```
public int readDisplay() {
  return insertedSoFar * 2 / 5;
}
public Receipt buy() {
  return new ReceiptImpl(insertedSoFar * 2 / 5);
}
```

I introduce a new instance variable:

- int timeBought
- to hold the time bought so far

But how do I ensure that I do this reliably?

The Power of Tests

Well – I can do so without fear due to all the test cases I have made so far...

This is important!

- Tests are *not* something the QA team and customers own
- They help me as programmer to dare touch my code!

... and so on

Remember the important principles

TDD Principle: Break

What do you do when you feel tired or stuck? Take a break.

TDD Principle: Do Over

What do you do when you are feeling lost? Throw away the code and start over.

Conclusion

Clean code that works

The Test-Driven Development process in one short statement is

Clean code that works

But not in that order.

- First – you make it work
  - quickly; making small steps; sometimes faking it
- Next – you make it clean
  - refactoring, remove duplication

Confidence and reliability

TDD promises confidence for us as developers

- Green bar gives confidence
- Failing test cases tell exactly where to look
- We dare refactor and experiment because tests tell us if our ideas are OK.
- We have taken small steps, so getting back is easy (put it under version control !!!)

Reliability

- Code that is tested by good test cases is *much better* than code that is not ☺

To stay in control

Test code is an *asset* that must be maintained!

- *All* unit tests run *all* the time!
  - If you change production code API you update the test cases as well!!! You do *not* throw them away!!!
- Green bar Friday afternoon means go home, play ball with the kids and have a beer!

Bug report from customer site?

- A) Make a test case that demonstrate the failure
- B) Correct the defect

Programming process

Principles define language!

- I can say what I do when I code !!!
  - Can you explain to your mate why you code it like this and not like that? And – is this better than that???
- It is *reflected practice* instead of *divine inspiration* and *black magic...*

Reversing order of implementation

Many program ‘bottom-up’

- My client needs class X so I
  - write class X
  - write the client code that uses X

This often gives inferior software because the order is “wrong” – you only find out what X should really have done and how the API should really have been *after* you have made X. Thus X is either rewritten or we just live with stupid method names, unused parameters, and odd calling sequences...

TDD focus on the client’s *usage* of X *before* writing X. This clarifies the requirements better!

Tool support

xUnit will help us to

- Write test cases and test suites
  - Using annotations like [Test] and [SetUp] / @Test and @Before
- Organize and group test cases
  - In hierarchical test suites
    - does not work with JUnit 4.x any more ☹
- Execute test suites
  - Hit the ‘run’ button
- Diagnose bugs
  - Examine the failed test case and correct

Literature

Henrik Bærbak Christensen:

- Flexible, Reliable Software, CRC Press 2010

Kent Beck:

- Test-Driven Development by Example, Addison-Wesley 2003.
  - A whole book on money conversion ☺

Martin Fowler

- Refactoring – Improving the Design of Existing Code, Addison-Wesley 1999

Outlook

Behaviour-Driven Development

Idea: Get rid of the testing terminology and think *requirements* and *accept tests instead!!!*

Classes *should do something* (feature). Express it directly in the test case *names!!!*

```
public class PayStationAcceptTest {
  @Test
  public void ShouldDisplay2MinParkingFor5Cent()
    throws IllegalCoinException {
    PayStation ps = new PayStationImpl();
```

New JUnit 4.4 Features

Much more complex (and perhaps more readable?) assertions:

```
@Test
public void ShouldDisplay2MinParkingFor5Cent()
  throws IllegalCoinException {
  PayStation ps = new PayStationImpl();
  assertThat( ps.readDisplay(), is(0) );

  ps.addPayment(5);

  assertThat( ps.readDisplay(), is(2) );

}
```

Principle Summary

TDD Principle: **One Step Test**

Which test should you pick next from the test list? Pick a test that will teach you something and that you are confident you can implement.

TDD Principle: **Fake It ('Til You Make It)**

What is your first implementation once you have a broken test? Return a constant. Once you have your tests running, gradually transform it.

TDD Principle: **Triangulation**

How do you most conservatively drive abstraction with tests? Abstract only when you have two or more examples.

TDD Principle: **Obvious Implementation**

How do you implement simple operations? Just implement them.

Principle Summary

TDD Principle: **Isolated Test**

How should the running of tests affect one another? Not at all.

TDD Principle: **Evident Tests**

How do we avoid writing defective tests? By keeping the testing code evident, readable, and as simple as possible.

TDD Principle: **Evident Data**

How do you represent the intent of the data? Include expected and actual results in the test itself, and make their relationship apparent. You are writing tests for the reader, not just for the computer.

TDD Principle: **Representative Data**

What data do you use for your tests? Select a small set of data where each element represents a conceptual aspect or a special computational processing.