Refactoring and Integration Testing *or* *Strategy,* *introduced reliably by TDD*

The power of automated tests

Two product variants

Alphatown and Betatown

- Four models to handle this
  - compositional proposal has nice properties...

How do we introduce it?

Change by addition

I state:

Change by addition, not modification

because

- addition
  - little to test, little to review
  - little chance of introducing ripple-effects
- modification
  - more to test, more to review
  - high risk of ripples leading to side effects (bugs!)

The Problem Statement

But I have to *modify* the pay station implementation in order to prepare it for the new compositional design that uses a Strategy pattern

☹ *Change by modification*

Problem:

- How to reliably modify PayStationImpl?
- How can I stay confident that I do not accidentally introduce any defects?

Take Small Steps

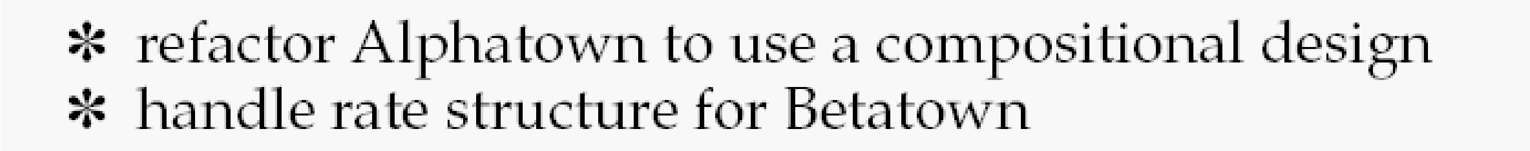
I will *stay focused* and *take small steps!*

I have **two** tasks

- Redesign the current implementation to introduce the Strategy and make AlphaTown work with new design
- Add a new strategy to handle Betatown requirements

... *and I will do it in that order – small steps!*

Refactoring



- ✻ refactor Alphatown to use a compositional design
- ✻ handle rate structure for Betatown

Definition:

Refactoring is the process of changing a software system in such a way that is does not alter the external behaviour of the code yet improves its internal structure.

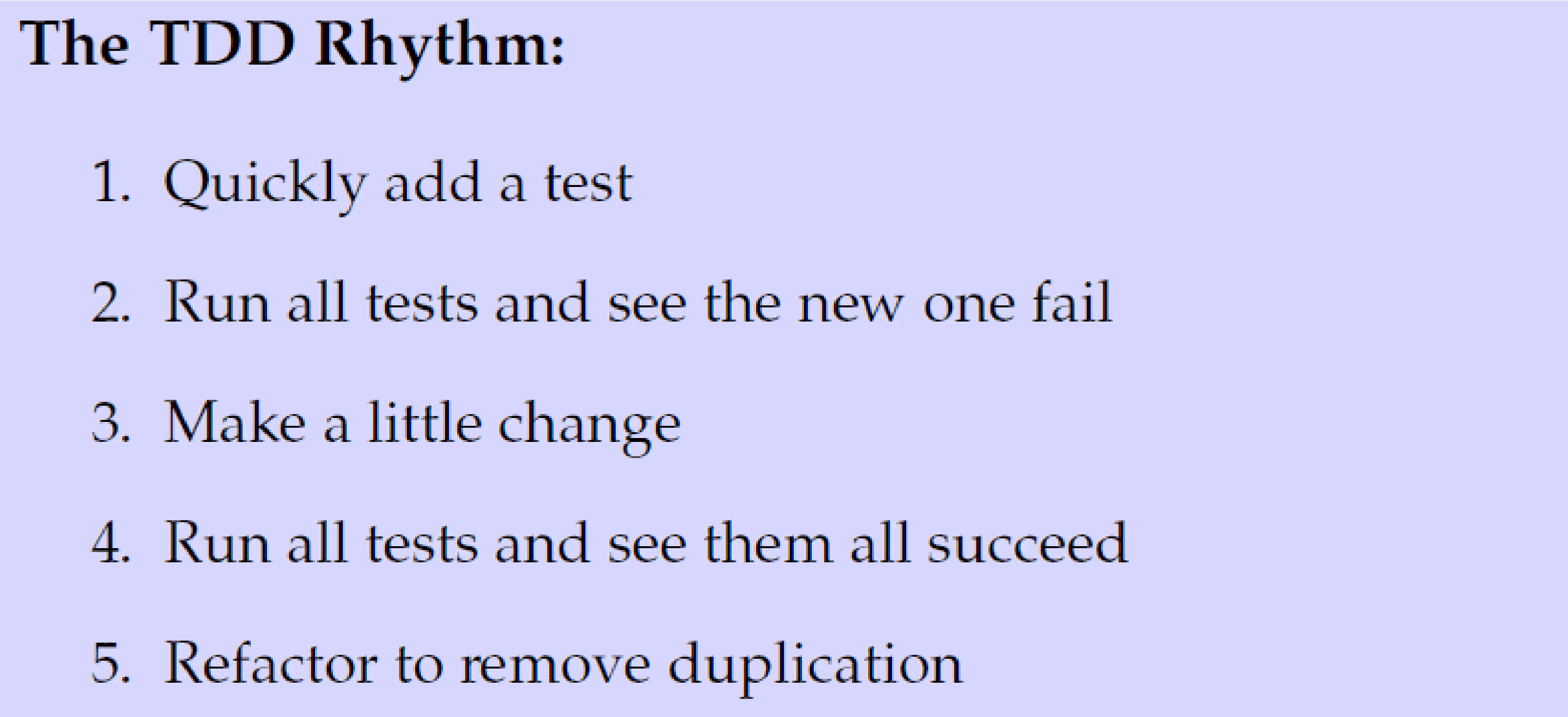
Fowler, 1999

Iteration 1

Refactoring step

The Rhythm

Refactoring and 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

Same spirit, but step 1: refactor

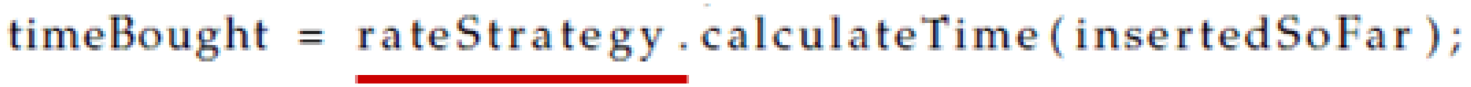
The new way ☺

Warning

The next slide section reflects the book and how I worked for many years. It works fine, but…

Today – I do it differently.

I simply enter the solution right away in Eclipse:



```
timeBought = rateStrategy.calculateTime(insertedSoFar);
```

And then what happens?

Eclipse will complain about a compile error, and come with suggestions to how to fix it.

So – I just choose all the right fixes!

Result: Eclipse does most of the typing ☺

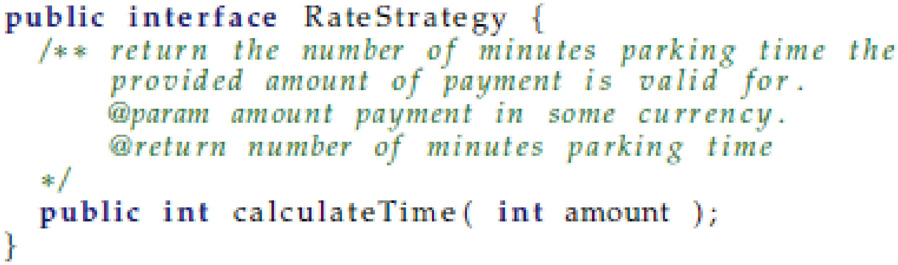
Morale: Start with what you want ☺

(If you can – sometime you cannot ☺)

The old way ☺

Step i

Introduce RateStrategy interface



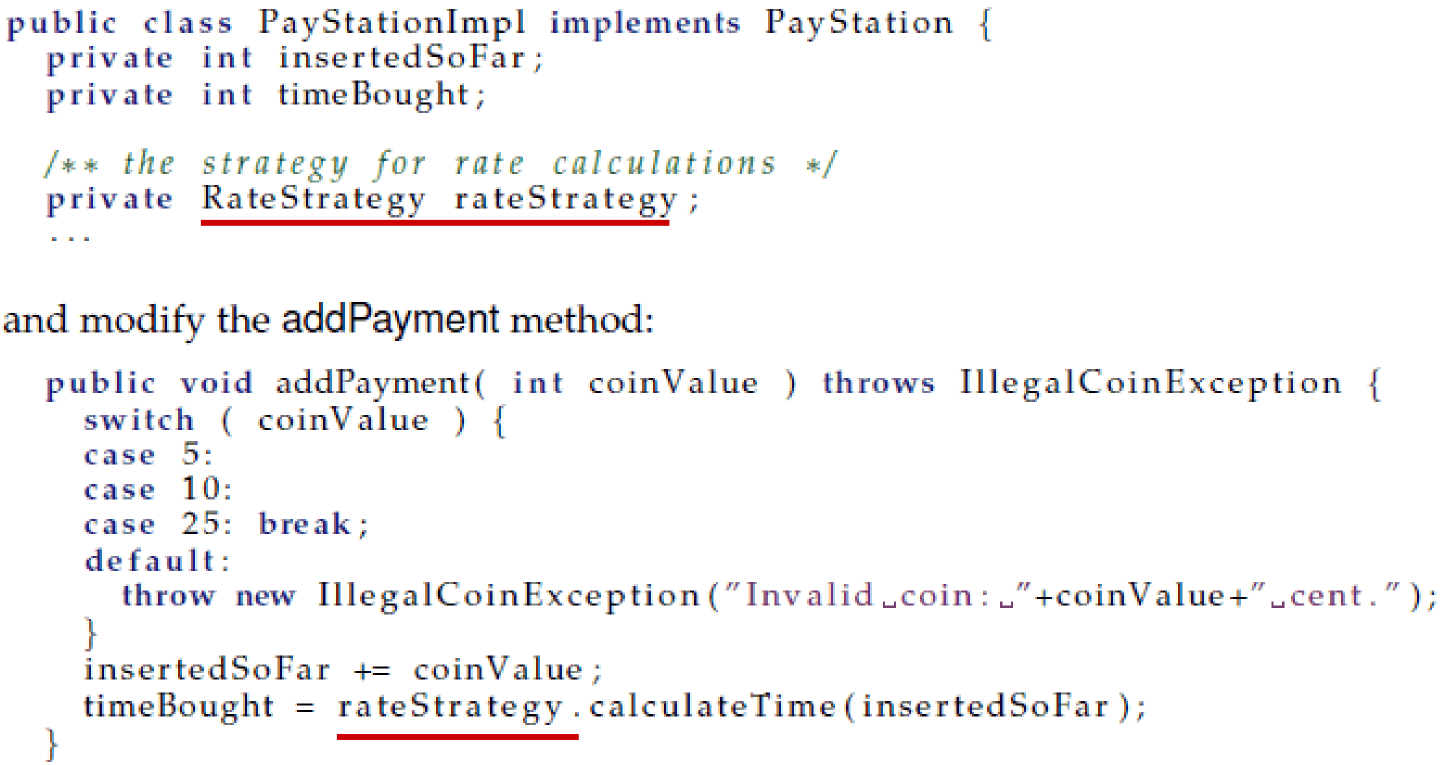
```
public interface RateStrategy {
  /** return the number of minutes parking time the
      provided amount of payment is valid for.
      @param amount payment in some currency.
      @return number of minutes parking time
  */
  public int calculateTime( int amount );
}
```

- – green bar = new interface compiles
- – (compiling is also a success ☺)

Step ii

Refactor to delegate to RateStrategy instance

– introduce reference + refactor calculation



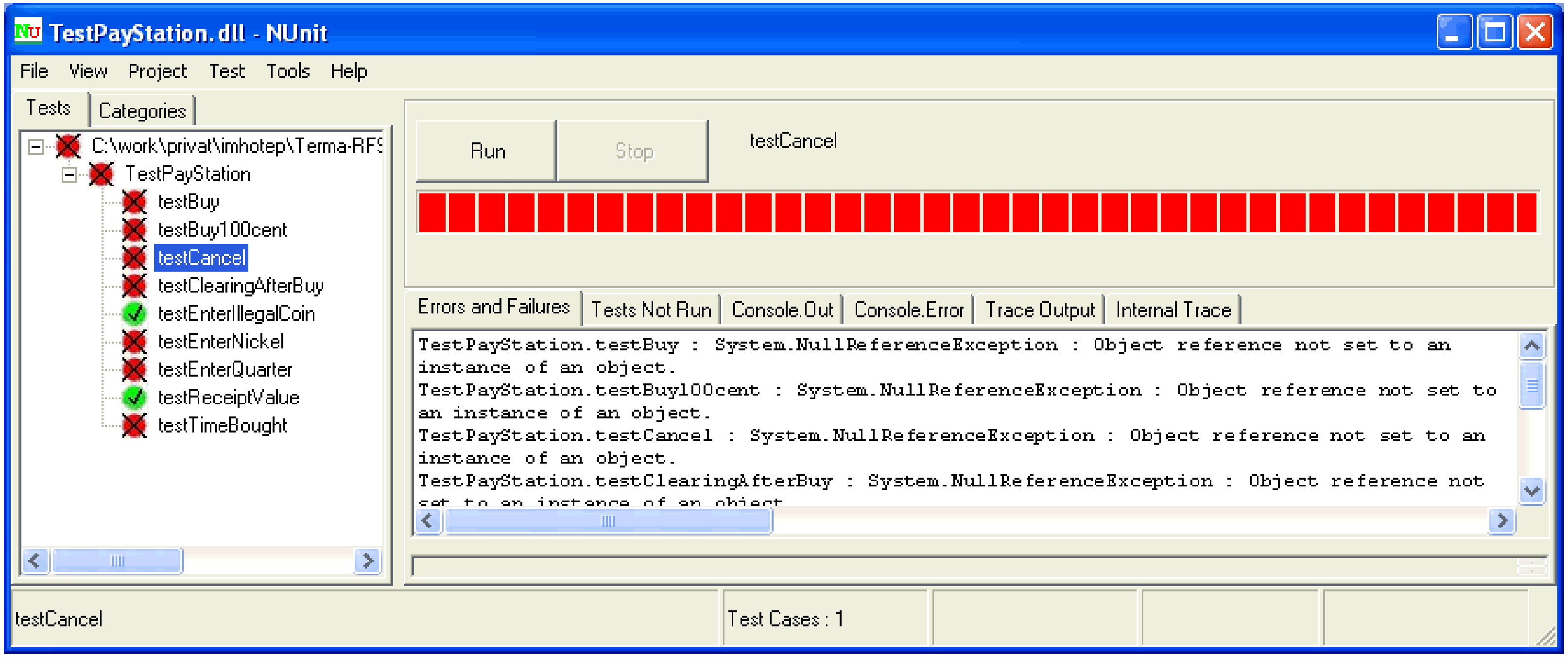
```
public class PayStationImpl implements PayStation {
  private int insertedSoFar;
  private int timeBought;

  /** the strategy for rate calculations */
  private RateStrategy rateStrategy;
  ...
```

and modify the addPayment method:

```
  public void addPayment( int coinValue ) throws IllegalCoinException {
    switch ( coinValue ) {
    case 5:
    case 10:
    case 25: break;
    default:
      throw new IllegalCoinException("Invalid␣coin:␣"+coinValue+"␣cent.");
    }
    insertedSoFar += coinValue;
    timeBought = rateStrategy.calculateTime(insertedSoFar);
  }
```

2. See that it fails



```
TestPayStation.testBuy : System.NullReferenceException : Object reference not set to an
instance of an object.
TestPayStation.testBuy100cent : System.NullReferenceException : Object reference not set to
an instance of an object.
TestPayStation.testCancel : System.NullReferenceException : Object reference not set to an
instance of an object.
TestPayStation.testClearingAfterBuy : System.NullReferenceException : Object reference not
set to an instance of an object
```

Why is it important to see it fail?

3. Make a little change

Introduce a proper constructor and

Refactor the tests code to invoke it...

– (Now it is *really good* that I refactored the creation of the pay station instance into a @Before method!)

However – no implementation of LinearRateStrategy...

What TDD pattern would you suggest?

4. See the green bar

Now…

- I have refactored the pay station production code because
  - the internal structure has been modified
  - but the external behaviour is unchanged

- The reason that this is a *refactoring* and not a *scary production code modification* is that I have high confidence in the external behaviour being the same as before.

Discussion

Why TDD?

Traditionally, developers see *tests* as

- boring
- time consuming

Why? Because of the stakeholders that benefit from tests are not the developers

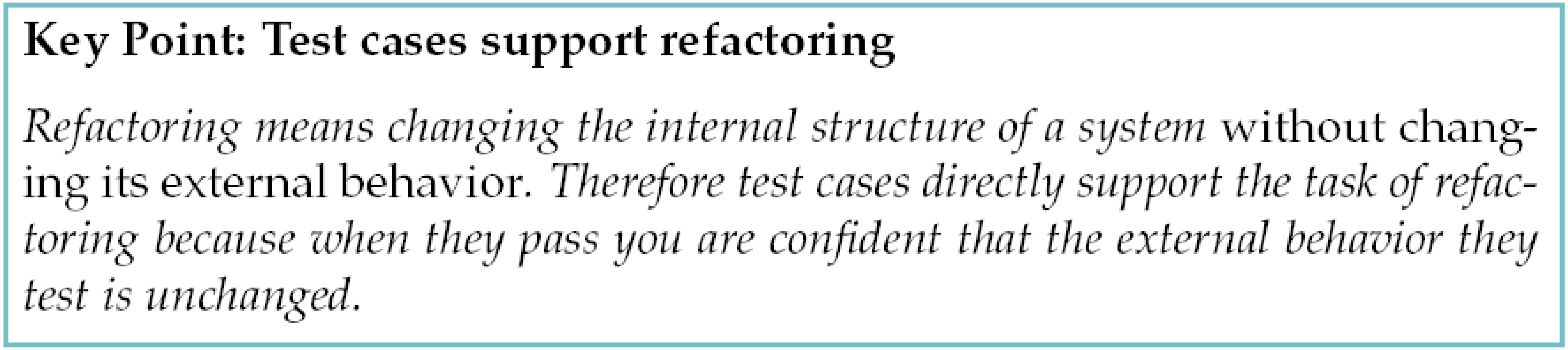
- customers: ensure they get right product ☺
- management: measure developer productivity ☺
- test department: job security ☺
- developers: *no benefit at all* ☹

If it ain't broke...

If it ain't broke, don't fix it

is the old saying of fear-driven programming

Developers and programmers do not dare doing drastic design and architecture changes in fear of odd side-effects.



Key Point: Test cases support refactoring

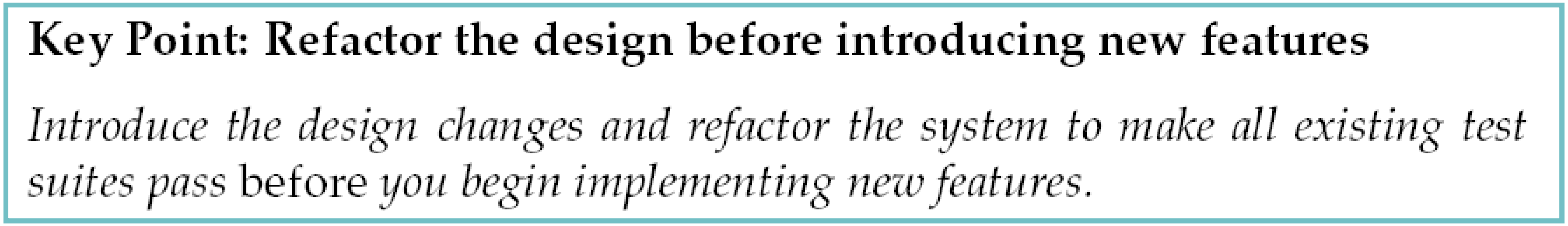
Refactoring means changing the internal structure of a system without changing its external behavior. *Therefore test cases directly support the task of refactoring because when they pass you are confident that the external behavior they test is unchanged.*

Test Ownership

Refactoring make developers want to have ownership of the tests:

Automatic tests is the developers' means to be courageous and dare modify existing production code.

When redesigning....



Key Point: Refactor the design before introducing new features

Introduce the design changes and refactor the system to make all existing test suites pass before *you begin implementing new features.*

TDD often seems like a nuisance to students and developers until the first time they realize that they dare do things they previously never dreamed of!

The first time a major refactoring is required – the light bulb turns on ☺

War story

Kent and associates tell a story

- Business software with a many-to-many relation
  - tedious to maintain and often give rise to errors
- However, is it really necessary?
  - change to one-to-many relation and run all functional tests (costumer owned, defining end-user behaviour)
  - **All pass!**

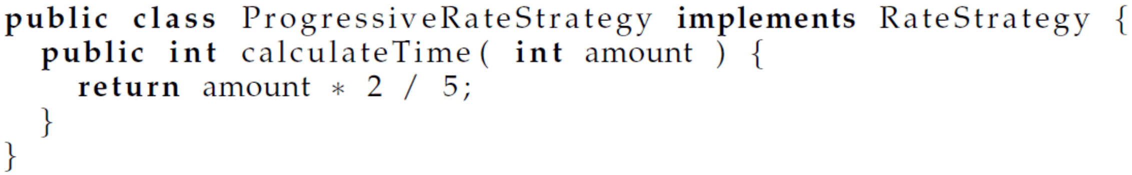
Iteration 2

Betatown Rate Policy

Triangulation at Algorithm Level

Introducing the real BetaTown rate policy is a nice example of using **Triangulation**

- Iteration 2
  - Add test case for first hour => production code



```
public class ProgressiveRateStrategy implements RateStrategy {
  public int calculateTime( int amount ) {
    return amount * 2 / 5;
  }
}
```

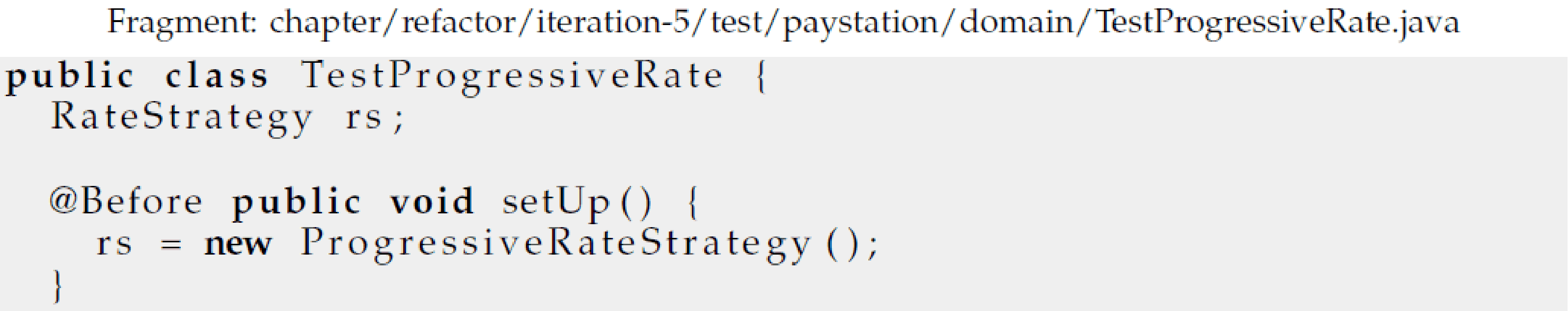
- Iteration 3: Add test case for second hour
  - Add just enough complexity to the rate policy algorithm
- Iteration 4: Add test case for third (and following) hour
  - Add just enough more complexity

Iteration 5

Unit and IntegrationTesting

Separate Testing

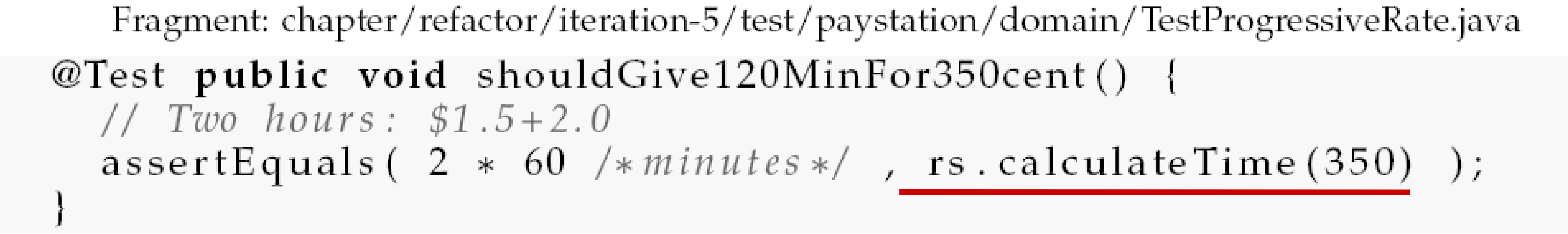
I can actually test the new rate policy without using the pay station at all



Fragment: chapter/refactor/iteration-5/test/paystation/domain/TestProgressiveRate.java

```
public class TestProgressiveRate {
  RateStrategy rs;

  @Before public void setUp() {
    rs = new ProgressiveRateStrategy();
  }
```

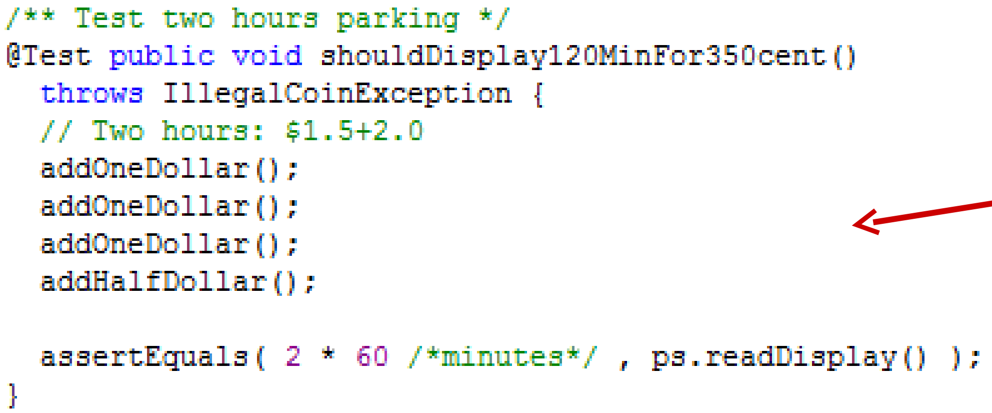


Fragment: chapter/refactor/iteration-5/test/paystation/domain/TestProgressiveRate.java

```
@Test public void shouldGive120MinFor350cent() {
  // Two hours: $1.5+2.0
  assertEquals( 2 * 60 /*minutes*/ , rs.calculateTime(350) );
}
```

Advantages

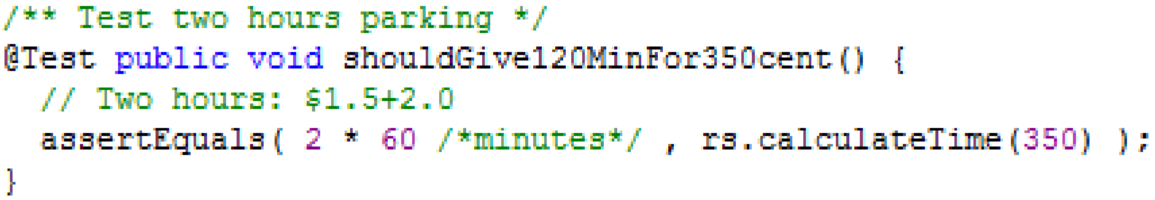
The unit testing of the progressive rate strategy is much simpler than the corresponding test case, using the strategy integrated into the pay station.



```
/** Test two hours parking */
@Test public void shouldDisplay120MinFor350cent()
  throws IllegalCoinException {
  // Two hours: $1.5+2.0
  addOneDollar();
  addOneDollar();
  addOneDollar();
  addHalfDollar();

  assertEquals( 2 * 60 /*minutes*/ , ps.readDisplay() );
}
```

Compare



```
/** Test two hours parking */
@Test public void shouldGive120MinFor350cent() {
  // Two hours: $1.5+2.0
  assertEquals( 2 * 60 /*minutes*/ , rs.calculateTime(350) );
}
```

Testing Types

Now

- I test the ProgressiveRateStrategy *in isolation* of the pay station (Unit testing)
- The pay station is tested *integrated* with the LinearRateStrategy (Integration testing)

Thus the two rate strategies are tested by *two* approaches

- In isolation (unit)
- As part of another unit

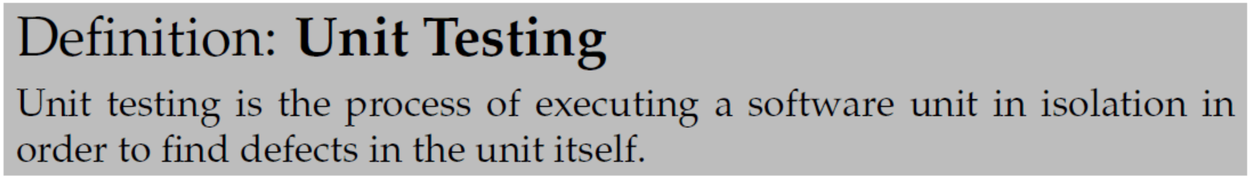
And

- The actual Betatown pay station is never tested!

Definitions

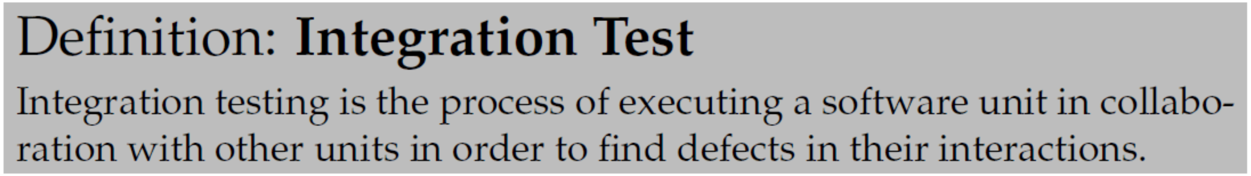
Experience tells us that *testing the parts does not mean that the whole is tested!*

- Often defects are caused by interactions between units for wrong configuration of units!



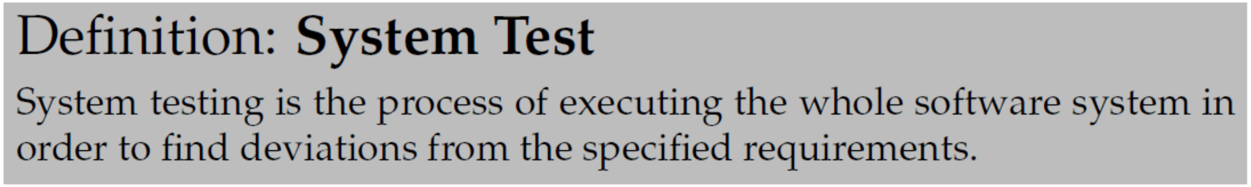
Definition: **Unit Testing**

Unit testing is the process of executing a software unit in isolation in order to find defects in the unit itself.



Definition: **Integration Test**

Integration testing is the process of executing a software unit in collaboration with other units in order to find defects in their interactions.



Definition: **System Test**

System testing is the process of executing the whole software system in order to find deviations from the specified requirements.

Exercise

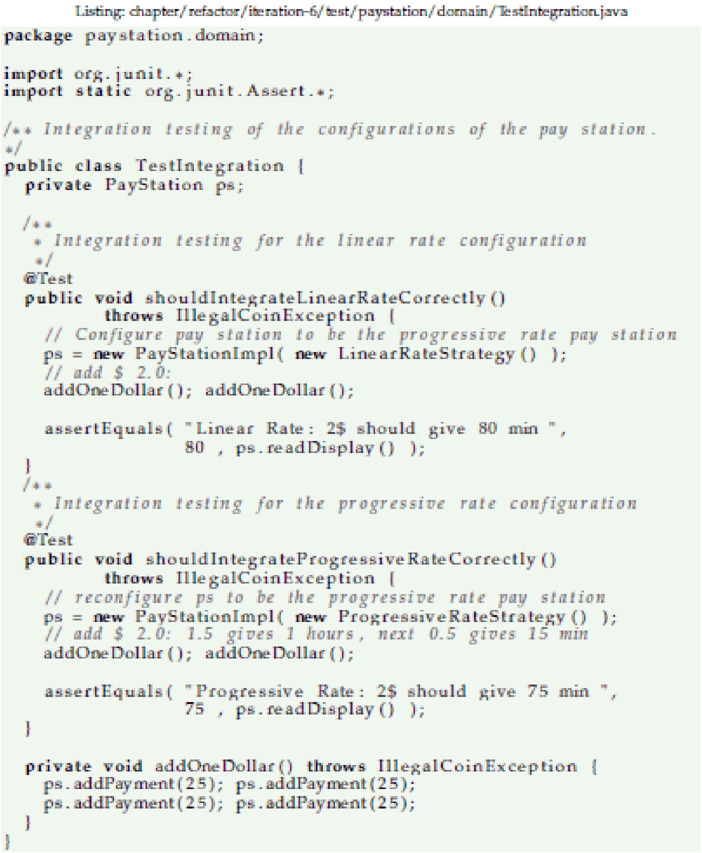
Tricky – but

- Give me a concrete example where having tested all the units in isolation does not guaranty that the system works correctly!

- The Mars Climate Orbiter...

Integration Testing the Pay Station

I must add a testcase that validate that the AlphaTown and BetaTown products are correctly configured!



Listing: chapter/refactor/iteration-6/test/paystation/domain/TestIntegration.java

```java
package paystation.domain;

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

/** Integration testing of the configurations of the pay station.
*/
public class TestIntegration {
  private PayStation ps;

  /**
   * Integration testing for the linear rate configuration
   */
  @Test
  public void shouldIntegrateLinearRateCorrectly()
          throws IllegalCoinException {
    // Configure pay station to be the progressive rate pay station
    ps = new PayStationImpl( new LinearRateStrategy() );
    // add $ 2.0:
    addOneDollar(); addOneDollar();

    assertEquals( "Linear Rate: 2$ should give 80 min ",
                  80 , ps.readDisplay() );
  }
  /**
   * Integration testing for the progressive rate configuration
   */
  @Test
  public void shouldIntegrateProgressiveRateCorrectly()
          throws IllegalCoinException {
    // reconfigure ps to be the progressive rate pay station
    ps = new PayStationImpl( new ProgressiveRateStrategy() );
    // add $ 2.0: 1.5 gives 1 hours, next 0.5 gives 15 min
    addOneDollar(); addOneDollar();

    assertEquals( "Progressive Rate: 2$ should give 75 min ",
                  75 , ps.readDisplay() );
  }

  private void addOneDollar() throws IllegalCoinException {
    ps.addPayment(25); ps.addPayment(25);
    ps.addPayment(25); ps.addPayment(25);
  }
}
```

More advanced integration testing

The pay station case’s integration is pretty simple as it is all a single process application.

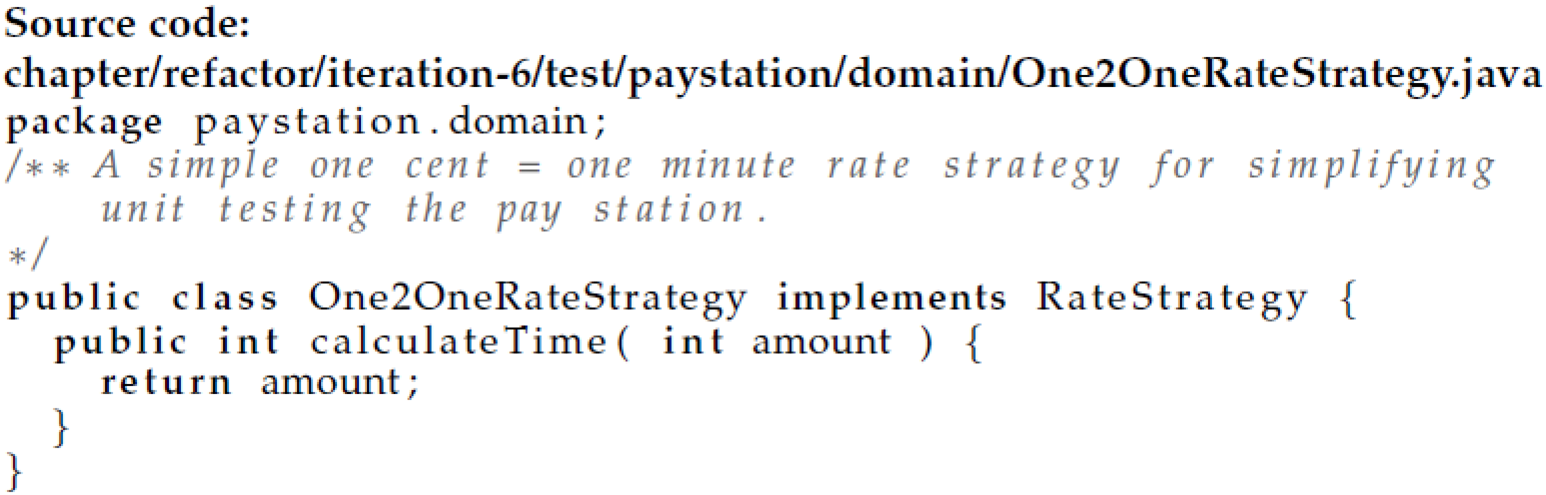
Net4Care case

- Maven integration testing involves starting web server and running tests against it to validate realistic deployment.

Iteration 6: Unit Testing Pay Station

Separate Testing

I can actually also apply *Evident Test* to the testing of the pay station by introducing a very simple rate policy

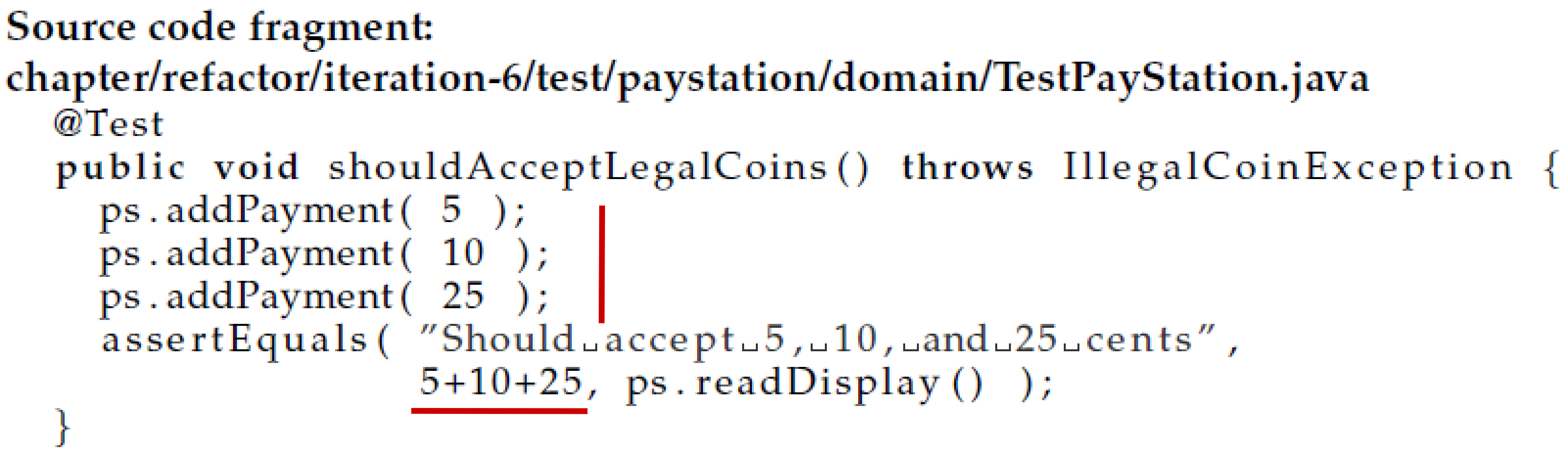


Source code:
chapter/refactor/iteration-6/test/paystation/domain/One2OneRateStrategy.java

```
package paystation.domain;
/** A simple one cent = one minute rate strategy for simplifying
    unit testing the pay station.
*/
public class One2OneRateStrategy implements RateStrategy {
  public int calculateTime( int amount ) {
    return amount;
  }
}
```

Resulting test cases

Using this rate policy makes reading pay station test cases much easier!



Source code fragment:
chapter/refactor/iteration-6/test/paystation/domain/TestPayStation.java

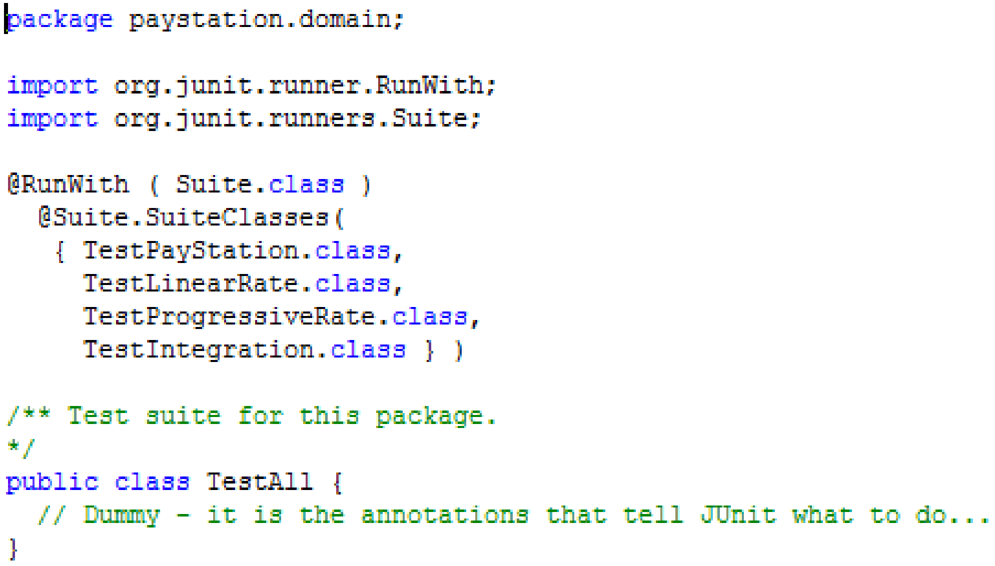
```
@Test
public void shouldAcceptLegalCoins() throws IllegalCoinException {
  ps.addPayment( 5 );
  ps.addPayment( 10 );
  ps.addPayment( 25 );
  assertEquals( "Should␣accept␣5,␣10,␣and␣25␣cents",
                5+10+25, ps.readDisplay() );
}
```

Iteration 7: Suites in JUnit

Suite

Suite = Set of test cases.

JUnit has special syntax to handle this, if you do not want to provide *very* long argument lists to JUnit.

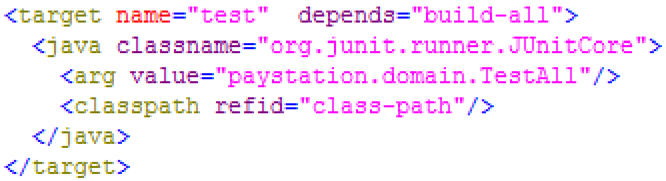


```
package paystation.domain;

import org.junit.runner.RunWith;
import org.junit.runners.Suite;

@RunWith ( Suite.class )
  @Suite.SuiteClasses(
   { TestPayStation.class,
     TestLinearRate.class,
     TestProgressiveRate.class,
     TestIntegration.class } )

/** Test suite for this package.
*/
public class TestAll {
  // Dummy - it is the annotations that tell JUnit what to do...
}
```

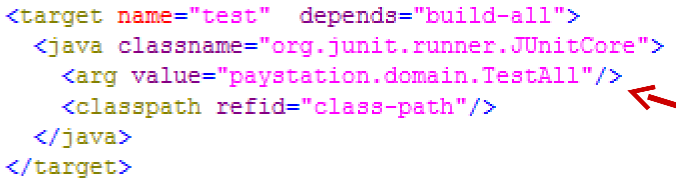


```
<target name="test"  depends="build-all">
  <java classname="org.junit.runner.JUnitCore">
    <arg value="paystation.domain.TestAll"/>
    <classpath refid="class-path"/>
  </java>
</target>
```

My preference today

I find the suite syntax of JUnit really weird.

Nowadays I prefer simply adding arguments to the ant target...



```
<target name="test"  depends="build-all">
  <java classname="org.junit.runner.JUnitCore">
    <arg value="paystation.domain.TestAll"/>
    <classpath refid="class-path"/>
  </java>
</target>
```

Maven

- Will run *all* java classes in folder ‘test’ as JUnit test cases...
  - Exercise for the reader: Make Ant do the same ☺

Outlook

System Testing

Testing levels

The normal classification is

- Unit testing: Individual software units
- Integration testing: unit collaborations
- System testing: User requirements

In our simple pay station, system testing is actually equal to the unit test of interface PayStation.

- In TDD all units tests pass all the time after an interation!!!
- System tests do not, but moves towards 100% as the product emerges over many iterations.

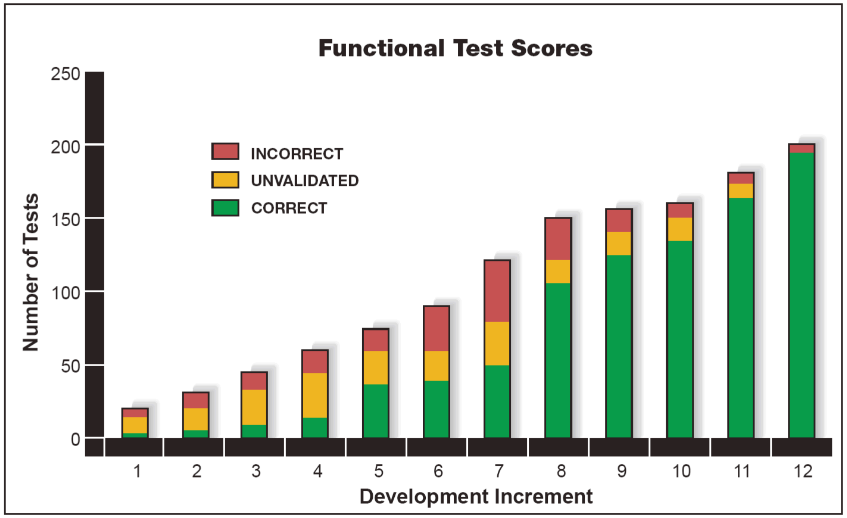
XP and TDD

Traditional Testing Terminology

- Unit testing, Integration Testing, System Testing

TDD:

- **Unit tests**
  - Owned by developers
  - Automatic
  - Runs 100% all the time!
- **Functional tests**
  - Owned by costumers
  - Automatic
  - Comprehensive and timely
  - Public, to show progress



Conclusion

Advice

Do not code in anticipation of need, code when need arise...

Automatic tests allow you to react when need arise

- because you dare refactor your current architecture...

Refactoring

When ’architecture refactoring’ need arise then

A) Use the old functional tests to refactor the architecture **without** adding new or changing existing behaviour

B) When everything is green again then proceed to introduce new/modified behaviour

C) Review again to see if there is any dead code lying around or other refactorings to do.

Discussion

These refactorings shown here are very local, so the ‘architecture decisions’ are also local.

However sometimes you need to make larger architectural changes that invalidate the test cases ☹

– Changing API or the way units are used
– Ex: Changing persistence from file to RDB based

What to do in this case?