Theme 2
Program Design and Testing

Test-Driven Development

Learning Objectives

- become familiar with the rhythm of test-driven development.
- learn a set of principles that form a list of potential actions that can be taken in each step in the rhythm.
- learn the definition of refactoring
- learn about the liabilities and benefits of a test-driven development process.

Table of content

Test-Driven Development

- *Working software*
  - ensure reliable software
  - ensure fast development process
  - ensure restructuring software and its architecture

TDD Values

- **Keep focus**
  - Make one thing only, at a time!
  - Often
    - *"Fixing this, requires fixing that, hey this could be smarter, fixing it, …"*
- **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…"*

TDD Values (c'td)

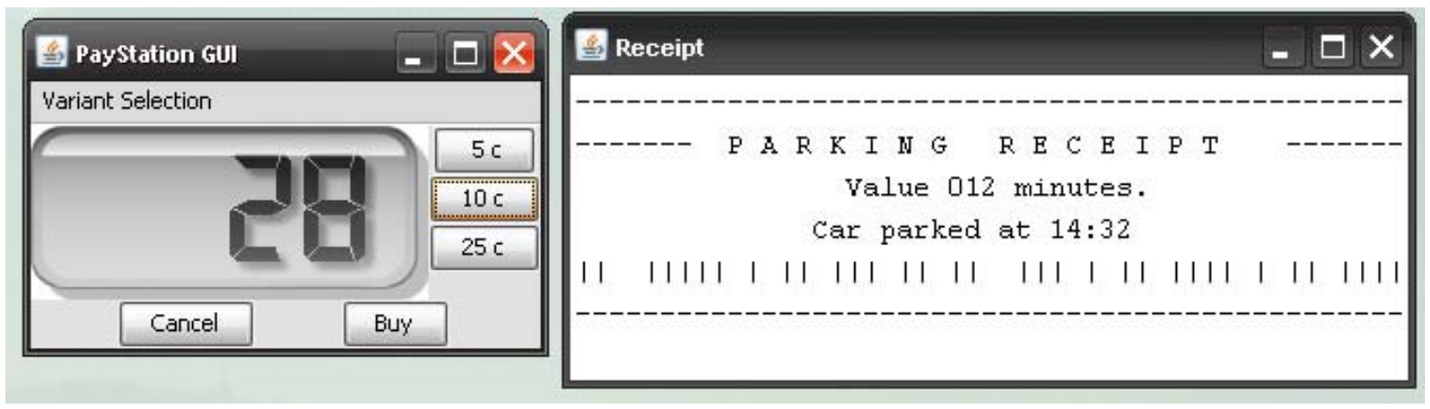
- **Speed**
  - You are what you do! Deliver every 14 days!!!
  - Often
    - *After two years, half the system is delivered, but works quite in another way than the user anticipate/wants…*
  - Speed, not by being sloppy but by making less functionality of superior quality!
- **Simplicity**
  - Maximize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

Before we can continue, we introduce a case study.

The Pay Station Case

The Pay Station Case

- You are all employees of PayStation Ltd.
- You will develop the main software to run pay stations



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

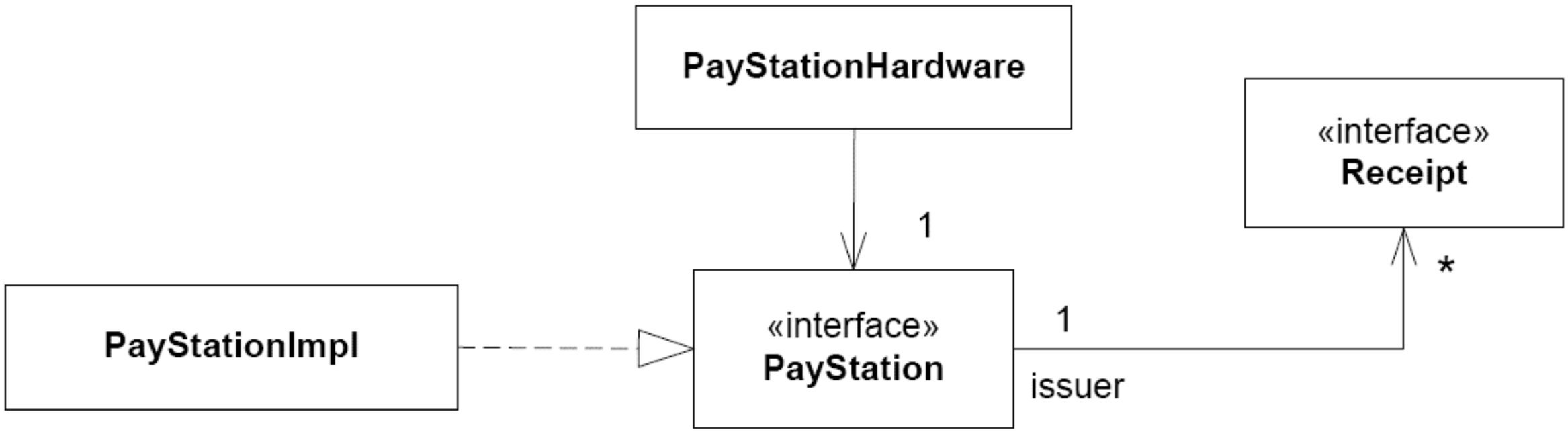
Stories

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

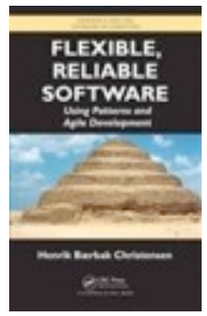


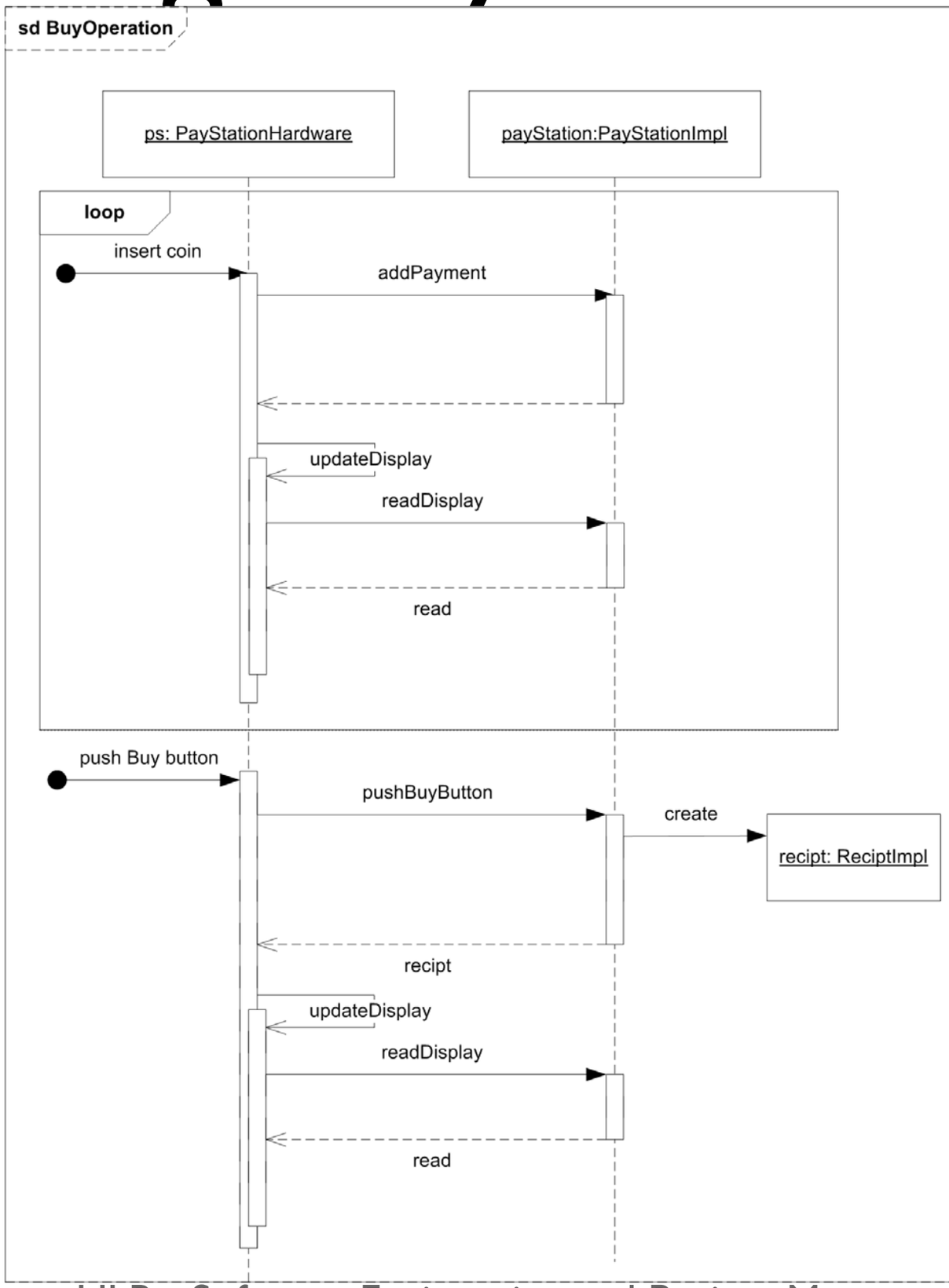
Design: Code View


```
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



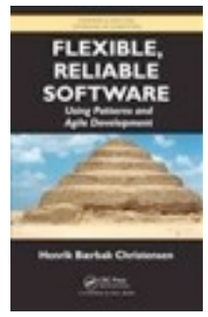


Back to TDD...

The fundamental TDD principles

Ragnhild Van Der Straeten - ULB - Software Engineering and Project Management - 2012/2013

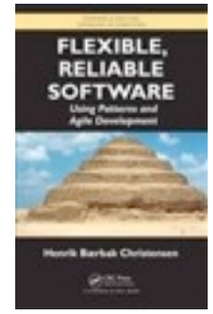
Principle: Test first



- **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 afterwards!!

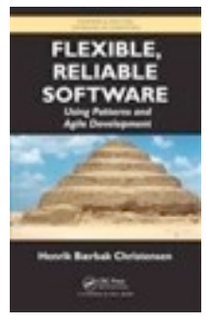
Principle: Automated Test



- **TDD principle: Automated Test**
  How do you test your software? Write an automated test.

- Kent Beck:
  "Software features that can't be demonstrated by automated tests simply don't exist."

Principle: Test List



- **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 ???

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 and remove code duplication

Refactoring?

- "Refactoring is the process of changing a software system in such a way that it does not alter the external behaviour of the code yet improves its internal structure" Martin Fowler
  www.martinfowler.com
  www.refactoring.com

- improving (should) means: easier to maintain

Size of an iteration

- An iteration is **small**
  typically adding a very very small increment of behaviour 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*!

Exercise

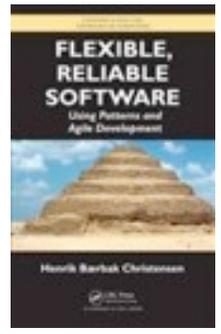
Generate the Test List for these stories.

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.

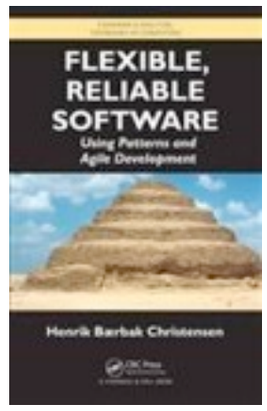
Story 2: Cancel a transaction. A driver has entered several coins but realize that the accumulated parking time shown in the display exceeds what she needs. She presses the button marked "Cancel" and her coins are returned. The display is cleared to prepare for another transaction.

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.

Possible Answer



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



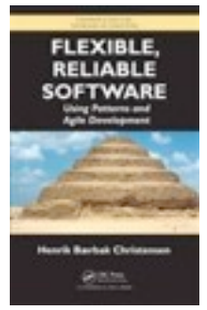
Iteration 0:

Setting Up

Which one to pick?

- Where do I start??

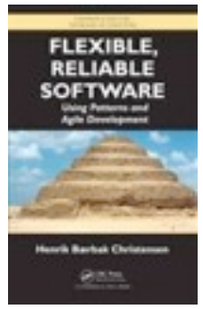
- **TDD Principle: One Step Test**
  Which test should you pick next from the test list?
  Pick a test that will teach you something and that you are confident you can implement.



Iteration 1:

5c = 2 min Parking

Step 1: Quickly add a test

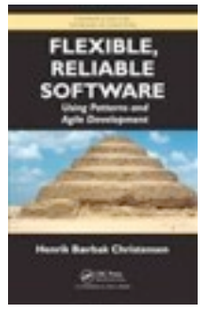


- The test case
  - ps.addPayment(5);
  - ps.readDisplay() == 2;

JUnit 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() );
  }
}
```

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



- Requires the implementation of a *PayStationImpl* ***Temporary Test Stub***
  - All methods are *empty* or *return null*

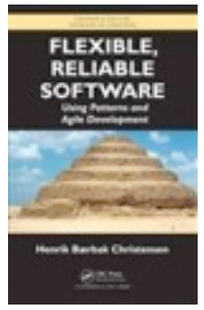
```
public class PayStationImpl implements PayStation {
  private int insertedSoFar;
  private int timeBought;

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

Step 3: Make a little change

- Exercise: what should you do?
- Remember:
  - Keep focus
  - Take small steps

```
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() {
  }
}
```

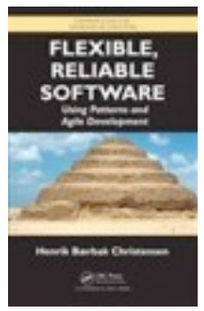
Fake it!



- **TDD Principle: Fake It!**
  What is your first implementation once you have broken a test? Return a constant.Once you have the tests running, **gradually transform it!!!**

- Might sound controversial: *Implement a solution that is known to be wrong and that must be deleted in two seconds.*
- *Why???*

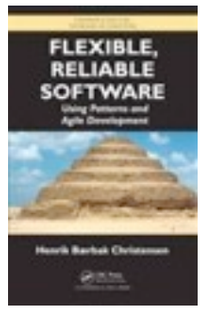
Test-*driven!!*



- ***Key point: no a single character is ever put into the production code if there is no test case defined to drive it into existence!***

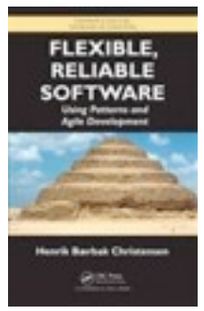
- We only have one test case, 5c = 2 min, and the simplest possible implementation is 'return 2;'. No other test case forces us to anything more!

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...

Step 4: Run all tests and see them all succeed

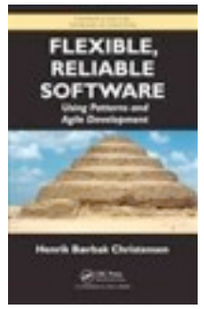


- Remember to note the success on the test list

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

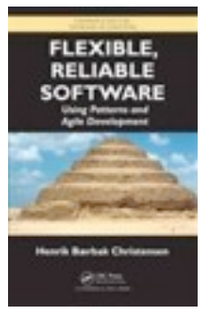
- But – of course I am concerned! The implementation is wrong!

Step 4 c'td: Triangulation



- ***Key point: one test case is not enough to ensure a reliable implementation of the rate calculation! Need more test cases to drive the implementation!!***
- **TDD Principle: Triangulation**
  How do you conservatively drive abstraction with tests? Abstract only when you have two or more examples.

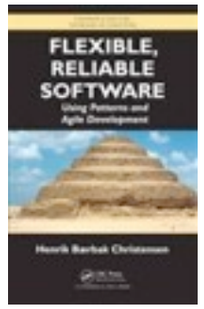
Triangulation



- The point is that *return 2* is actually the correct implementation of the *readDisplay* if the only occurring 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!

Triangulation



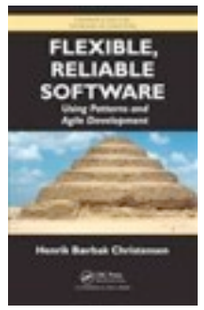
- So I simply remind myself that *Fake It* is playing around in the production code by adding it to the test list:

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

Iteration 2: Rate Calculation

25 cents = 10 minutes

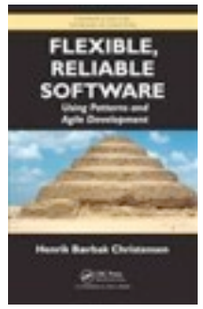
Step 1: Quickly add a test



- Where?
- **TDD Principle: Isolated Test**
  How should the running of tests affect one another? Not at all.

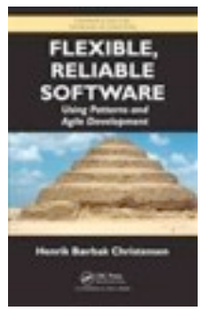
- Exercise: why??

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.
- **!!Isolate in order to overview failure consequences!!**

Do It...

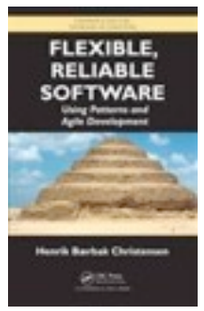


- Step 2: *write an isolated test for 25 cent parking.*
- Run all tests and see the new one fails...
- Step 3: *make a little change*

```
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() {
  }
}
```

- This is wrong?? No??
  - What should you do?

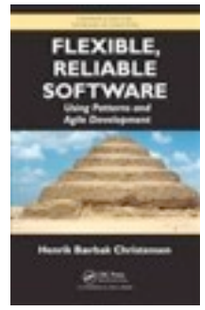
Test-*driven* Implementation



* ~~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: Refactoring

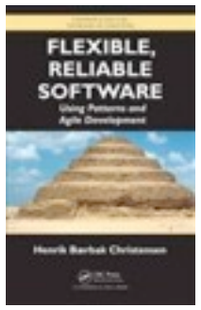
- Testing code is also best maintained!!
- Duplicated code that sets up the pay station object.
  - Move this to a Fixture which you define using the @Before annotation (JUnit)




```
/**
 * 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() );
}
```

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

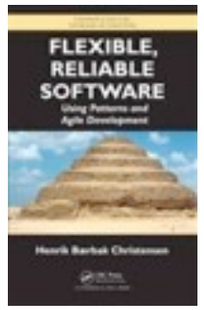
@Before




```
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

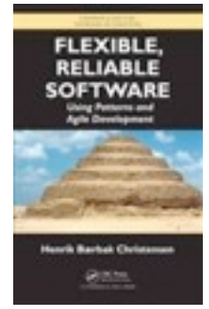



```
1  import org.junit.*;
2  import static org.junit.Assert.*;
3
4  /** Testcases for the Pay Station system.
5      Author: (c) Henrik Bærbak Christensen 2006 */
6
7  public class TestPayStation {
8    PayStation ps;
9    /** Fixture for pay station testing. */
10   @Before
11   public void setUp() {
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   }
```

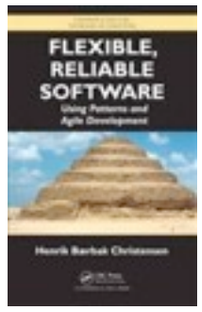
TDD Principle: Evident Data

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

Evident Data

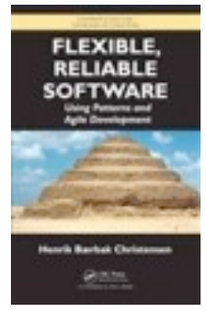



```
/**
 * Entering 5 cents should make the display report 2 minutes
 * parking time.
*/
@Test
public void shouldDisplay2MinFor5Cents() throws IllegalCoinException {
  ps.addPayment( 5 );
  assertEquals( "Should display 2 min for 5 cents",
                5 / 5 * 2, ps.readDisplay() );
}

/**
 * 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:

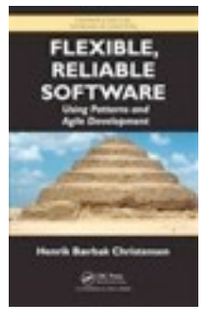
Illegal Coin




```java
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();
}
```

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

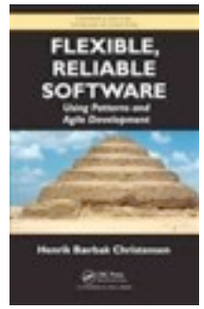
```java
/** Testing for illegal coin entry. */
@Test(expected=IllegalCoinException.class)
public void testEnterIllegalCoin() throws IllegalCoinException {
  ps.addPayment(17);
}
```



Iteration 4:

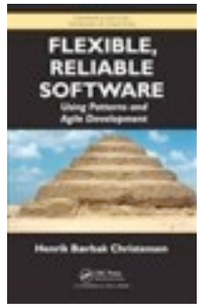
Valid Coins

Representative Data



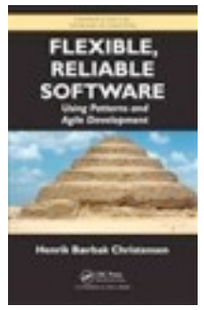
- What coins to use???
  - used a 5 and 25 cent, but have not tried 10 cent yet.
- **TDD Principle: Representative Data**

  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.
  - Buy story states "5, 10, 25 coins are valid"
  - So add a dime as a valid coin.



* ~~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~~ a 10 and 25 coin*

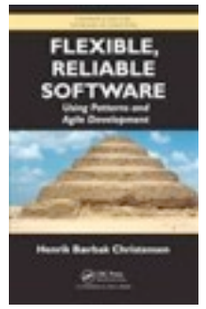
The Rhythm



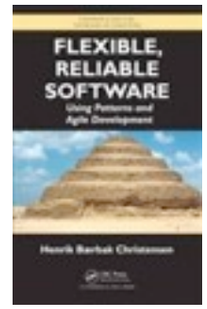
- Step 1: *Quickly add a test*

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

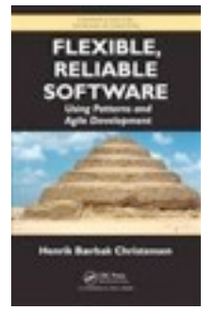
- Step 2: *Run all tests and see the new one fails*
- Step 3: *Make a little change*
  add missing case in the switch statement
- Step 4: *Run all tests*
  FAIL!! Why???




```
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;
}
```



* ~~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~~



Iteration 5 :

Buy It

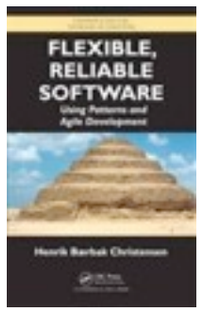
The Rhythm

- Step 1: *Quickly add a test*

```
/**
 * 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() );
}
```

- Step 2: *Run all tests* => new one fails
- Step 3: *Make a little change.... but*

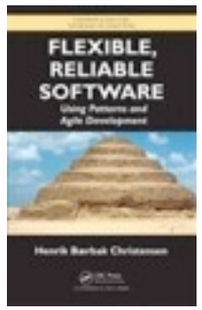
Step 3



- We need two changes:
  - an implementation of Receipt
  - implementing the buy method

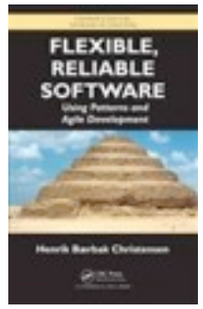
- Small steps? What are the options?
  1. The old way: Do both in one go!
  2. Fix receipt first, buy next...
  3. Fix buy first, implement receipt later...

Answer...



- Take small steps tells us either 2 or 3 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 can I do that given the dependency structure?
- What is your answer?

Fake It!!!



- I can complete buy by making a fake receipt.
- I keep focus!

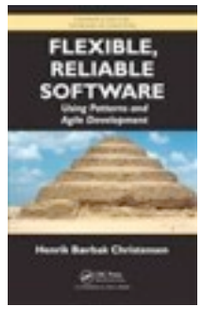
```
public class PayStationImpl implements PayStation {
  private int insertedSoFar;
  public void addPayment( int coinValue )
    throws IllegalCoinException {
    switch ( coinValue ) {
    case 5: break;
    case 10: break;
    case 25: break;
    default: throw new IllegalCoinException("Invalid coin: "+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() {
  }
}
```

anonymous inner class,
but will be removed later on!!!

Step 4

- Pass
- Update the test list!! Triangulation!!!

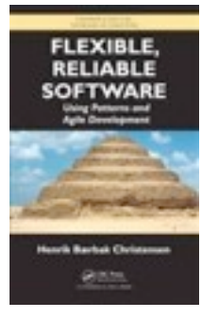
~~insert legal coin~~
~~insert illegal coin, exception~~
~~5 cents should give 2 minutes parking time.~~
~~readDisplay~~
buy for 40 cents
buy for 100 cents
cancel
~~25 cents = 10 minutes~~
receipt value



Iteration 6:

Receipt

The Rhythm



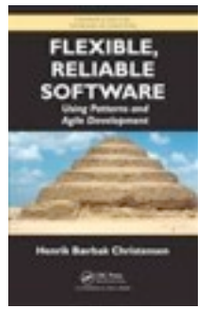
- Step 1

```
@Test
  public void shouldStoreTimeInReceipt() {
    Receipt receipt = new ReceiptImpl(30);
    assertEquals( "Receipt can store 30 minute value",
                  30, receipt.value() );
  }
```

- Step 2: FAIL
- Step 3

```
public class ReceiptImpl implements Receipt {
  private int value;
  public ReceiptImpl(int value) { this.value = value; }
  public int value() { return value;}
}
```

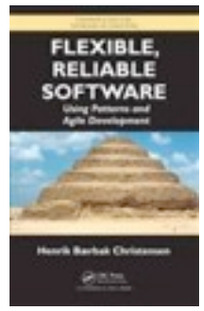
- Step 4: PASS



Iteration 7:

Buy

Step 1



- Buy for 100 cents

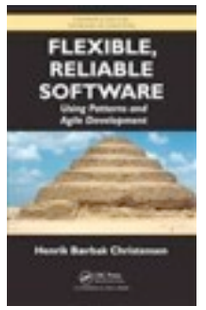
- Exercise:
  - How to enter 100 cent?
    - add 5, add 5, add 5, add10, ...
    - for (int i = 0; i <= 20; i++) { add 5; }
    - private method add2Quarters()

Evident Tests

- **TDD Principle: Evident Tests**
  How do we avoid writing defective tests? By keeping the test code evident, readable, and as simple as possible.

- avoid loops, conditionals, recursion, complexity in your testing code.
- because testing code is code and you make mistakes in code!
  - assignment, creation, private method calls
  - and not much else!

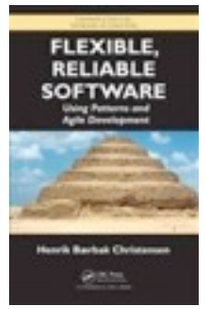
Step 3, 4 and 5




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

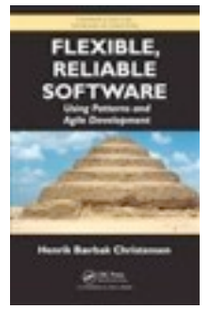
- Code Duplication
- Refactoring: several possibilities
  - introduce a new instance variable:
    - int *timeBought*
    - to hold the time bought so far
- How can I ensure that I do this reliably??

Implementation of the Pay Station so far...



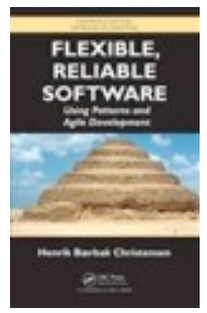

```
public class PayStationImpl implements PayStation {
  private int insertedSoFar;
  private int timeBought;

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

Do It Yourself!



- clearing after a buy operation
- cancel resets pay station

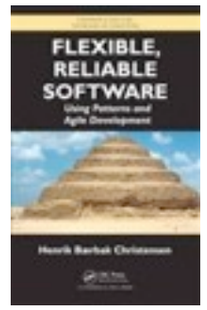
Conclusions



- Test-driven development process is
  - ***Clean code that works!!***

- First - make it work
  - quickly, taking small steps; sometimes faking it
- Next - make it clean
  - refactoring (removing duplication)

Conclusions



- TDD promises confidence for the developers
  - Tests that pass give confidence
  - Failing test cases tell exactly where to look
  - Developers dare to refactor and experiment because tests tell them if their ideas are OK.
  - Developers 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.

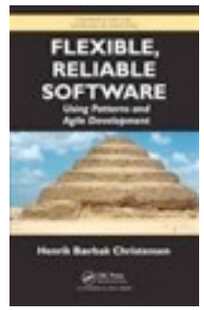
Conclusions

- 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!!!
- Bug report from customer site?

  A) Make a test case that demonstrate the failure

  B) Correct the defect

Conclusions



- Programming Process
  - Developers can tell what they do when they code.
    - Can you explain to your friend why you code it that way and another way? And why your way is better?
- It is reflected practice instead of divine inspiration or black magic...