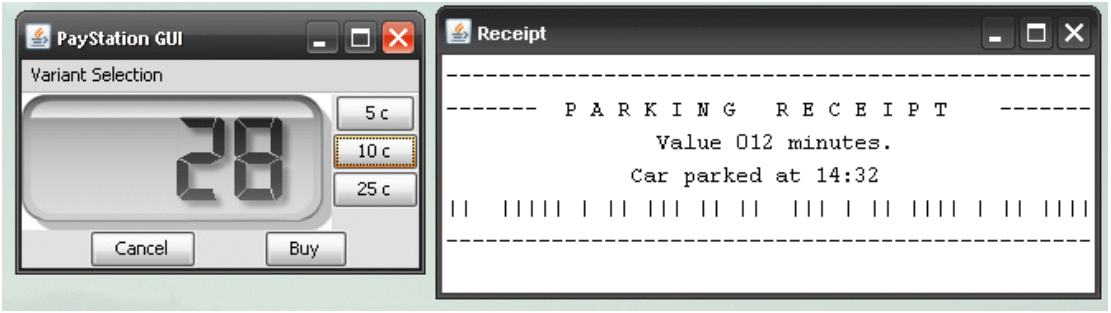
You are all employed today

- Welcome to *PayStation Ltd.*
- We will develop the main software to run pay stations.
  - [Demo]



Case Study: 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

User 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.

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!)

Refactoring?

Refactoring	Refactoring is the process of changing a software system in such a way that is does not alter the external behavior of the code yet improves its internal structure. Martin Fowler [?]

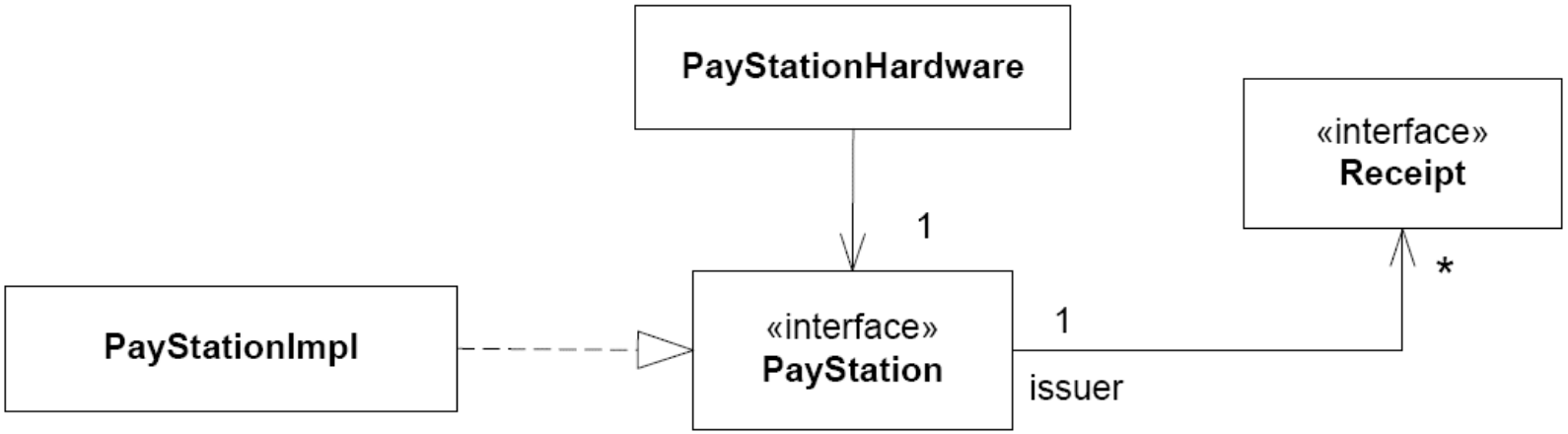
- Improving (usually) means
  - *Easier to understand ("Readability")*
  - *Easier to maintain ("duplication")*
- Defined in terms of software module input-output relationship

Test List

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

Design: Static View (Class Diagram)

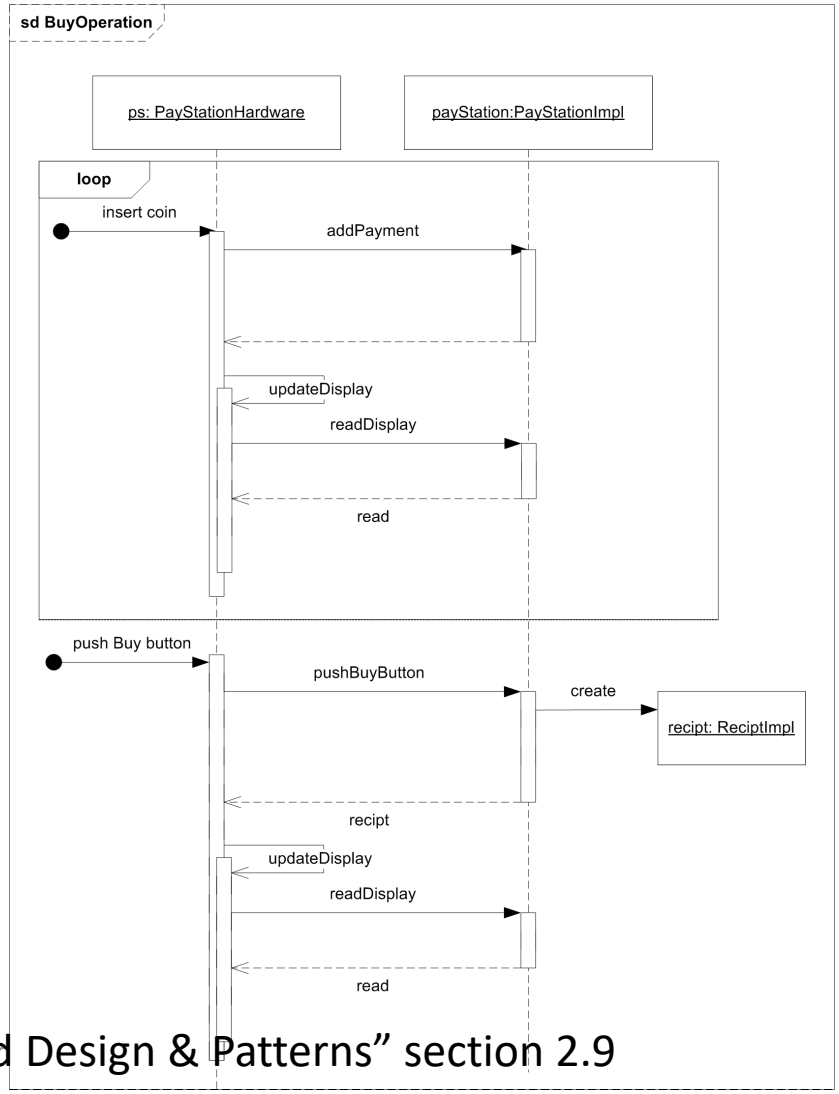
- Program to interfaces, not implementations
- Central *interface* **PayStation**



See textbook “Object-Oriented Design & Patterns” section 2.8

Design: Dynamic View (Seq. Diagram)

- The envisioned dynamics…



See textbook “Object-Oriented Design & Patterns” section 2.9

Paystation Interface

```
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();
}
```

IllegalCoinException.java

```
/** Exception representing illegal coin entry.

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

   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 IllegalCoinException extends Exception {
  public IllegalCoinException( String e ) { super(e); }
}
```

Receipt Interface

```
/** The receipt returned from a pay station.
   Responsibilities:

   1) Know the minutes parking time the receipt represents

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

   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 interface Receipt {

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

PaystationImpl.java

```
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() {
  }
}
```

Minimum implementation to satisfy compiler, nothing works now!

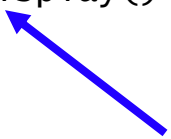
Testing Code in JUnit

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

/**
 *  Testcases for the Pay Station system.
 */
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

Fake it till you make it

```
public class PayStationImpl implements PayStation {

  public void addPayment( int coinValue )
          throws IllegalCoinException {
  }

  public int readDisplay() {
    return 0;
    return 2;
  }

  public Receipt buy() {
    return null;
  }

  public void cancel() {
  }
}
```

Implement a solution that is known to be wrong and must be deleted in minutes

Why Fake?

- Why we implement a fake solution that is known to be wrong and must be deleted in minutes?
- 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...

Second Test

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

/** Testcases for the Pay Station system.

*/
public class TestPayStation {
  PayStation ps;
  /** Fixture for pay station testing. */
  @Before
  public void setUp() {
    ps = new PayStationImpl();
  }

  /**
   * 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 {
    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
  }
}
```

Pass Second Test

```
public class PayStationImpl implements PayStation {
  private int insertedSoFar;
  public void addPayment( int coinValue )
          throws IllegalCoinException {
    insertedSoFar = coinValue;
  }

  public int readDisplay() {
    return 2;
    return insertedSoFar / 5 * 2;
  }

  public Receipt buy() {
    return null;
  }

  public void cancel() {
  }
}
```

Test for Illegal Coin

```
/**
 * Verify that illegal coin values are rejected.
 */

@Test(expected=IllegalCoinException.class)
public void shouldRejectIllegalCoin() throws
   IllegalCoinException {
   ps.addPayment(17);
}
```

Pass Test

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

Enter Two Coins

```
/**
  * Entering 10 and 25 cents should be valid and
  * return 14 minutes parking
  */
@Test
public void shouldDisplay14MinFor10And25Cents()
        throws IllegalCoinException {
  ps.addPayment( 10 );
  ps.addPayment( 25 );
  assertEquals( "Should display 14 min for 10+25 cents",
                14, ps.readDisplay() );
}
```

Pass Test

```
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;
    insertedSoFar += coinValue;
  }
  ⋮
}
```

Add Three Coins and Get a Receipt

```
/**
  * 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.",
                16 , receipt.value() );
```

Pass Test -- Fake It

```
public class PayStationImpl implements PayStation {
  private int insertedSoFar;
  ⋮
  public Receipt buy() {
    return new Receipt() {
        public int value() { return 16; }
      };
  }
  public void cancel() {
  }
}
```

ReceiptImpl.java

```
/** Implementation of Receipt.
*/

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

Pass Test -- Fake It (2)

```
public class PayStationImpl implements PayStation {
  private int insertedSoFar;
  ⋮
  public Receipt buy() {
    return new Receipt() {
        public int value() { return 16; }
    };
    return new ReceiptImpl(16);
  }
  public void cancel() {
  }
}
```

Try 100 cents

```
/**
  * Buy for 100 cents and verify the receipt
  */
@Test
public void shouldReturnReceiptWhenBuy100c()
  throws IllegalCoinException {
  ps.addPayment(10);
  ps.addPayment(10);
  ps.addPayment(10);
  ps.addPayment(10);
  ps.addPayment(10);
  ps.addPayment(25);
  ps.addPayment(25);

  Receipt receipt;
  receipt = ps.buy();
  assertEquals(40 , receipt.value() );
}
```

PaystationImpl.java

```
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 insertedSoFar / 5 * 2;
    return timeBought;
  }
  public Receipt buy() {
    return new ReceiptImpl(16);
    return new ReceiptImpl(timeBought);
  }
  public void cancel() {
  }
}
```

Test to see if Pay Station is Cleared After Buy

```
/**
  * Verify that the pay station is cleared after a buy scenario
  */
@Test
public void shouldClearAfterBuy()
        throws IllegalCoinException {
  ps.addPayment(25);
  ps.buy(); // I do not care about the result
  // verify that the display reads 0
  assertEquals( "Display should have been cleared",
                0 , ps.readDisplay() );
  // verify that a following buy scenario behaves properly
  ps.addPayment(10); ps.addPayment(25);
  assertEquals( "Next add payment should display correct time",
                14, ps.readDisplay() );
  Receipt r = ps.buy();
  assertEquals( "Next buy should return valid receipt",
                14, r.value() );
  assertEquals( "Again, display should be cleared",
                0 , ps.readDisplay() );
}
```

Pass Test

```
public class PayStationImpl implements PayStation {
  private int insertedSoFar;
  ⋮
  public Receipt buy() {
    return new ReceiptImpl(timeBought);
    Receipt r = new ReceiptImpl(timeBought);
    timeBought = insertedSoFar = 0;
    return r;
  }
  public void cancel() {
  }
}
```

Test Cancel

```
/**
  * Verify that cancel clears the pay station
  */
@Test
public void shouldClearAfterCancel()
        throws IllegalCoinException {
  ps.addPayment(10);
  ps.cancel();
  assertEquals( "Cancel should clear display",
                0 , ps.readDisplay() );
  ps.addPayment(25);
  assertEquals( "Insert after cancel should work",
                10 , ps.readDisplay() );
}
```

Pass Test

```
public class PayStationImpl implements PayStation {
  private int insertedSoFar;
  ⋮
  public Receipt buy() {
    Receipt r = new ReceiptImpl(timeBought);
    timeBought = insertedSoFar = 0;
    reset();
    return r;
  }
  public void cancel() {
    reset();
  }
  private void reset() {
    timeBought = insertedSoFar = 0;
  }
}
```

Testing at Different Levels

- 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.

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>
```