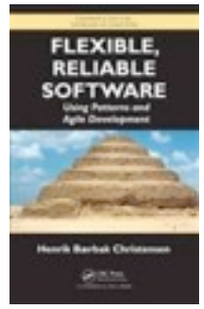
Theme 2
Program Design

Towards a Strategy Pattern

Learning Objectives

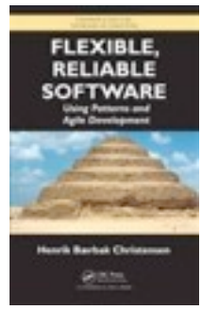
- learn several different solutions to the problem of one product, several versions and see their respective benefits and liabilities.
- learn that one of the solutions, the compositional one, is actually the strategy design pattern.

Remember: Pay Station



- The pay station must:
  - accept coins for payment
  - show time bought
  - print parking time receipts
  - 2 minutes cost 5 cent
  - handle buy and cancel
  - maintenance (empty it)

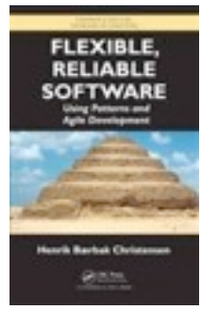
New Requirement



- New progressive pricing model for a new client
  1. first hour: $1.5 (5 cent gives 2 minutes)
  2. second hour: $2 (5 cent gives 1.5 minutes)
  3. third and following hours: $3 (5 cents gives 1 minute)

- How can we handle these different products?

Code to Change

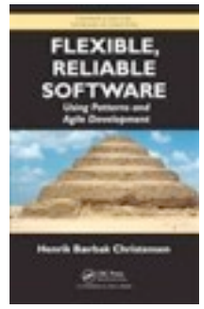



```
public void addPayment( int coinValue ) throws IllegalCoinException {
  switch ( coinValue ) {
  case 5:
  case 10:
  case 25: break;
  default:
    throw new IllegalCoinException("Invalid coin: "+coinValue+" cent.");
  }
  insertedSoFar += coinValue;
  timeBought = insertedSoFar / 5 * 2;
}
```

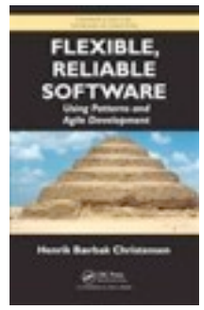
Variability point!!

How do I introduce having two different behaviors of the rate calculation variability point such that both the cost and the risk of introducing defects are low?

Exercise!



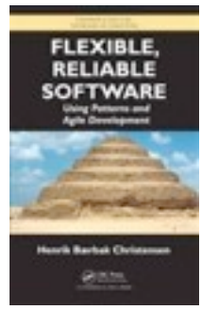
- Propose some solutions to handle variations in the code. Think about these issues:
  - most of the code is the same in the 2 products
  - what about 20 or more variants?
- Find different solutions!!!



Possible Solutions

- Model 1:
  - Make a copy of the source tree
- Model 2:
  - Parameterization
- Model 3:
  - Polymorphic proposal
- Model 4:
  - Compositional proposal

Source Tree Copying



- Deep copy production code source tree
- Code the new variant by replacing the code at the variability point.
- **Benefits**
  - simple
  - fast
  - variant decoupling
- **Liabilities**
  - Multiple maintenance problem
    - changes in common code must be propagated to all copies, duplicated test cases

Parametrization

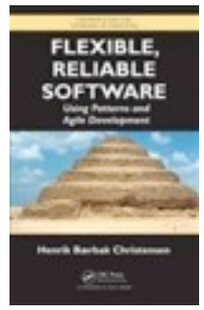
Rehearsal

Throw in some 'if'-statements

```
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;
  if ( town == Town.ALPHATOWN ) {
    timeBought = insertedSoFar * 2 / 5;
  } else if ( town == Town.BETATOWN ) {
    [the progressive rate policy code]
  }
}
```

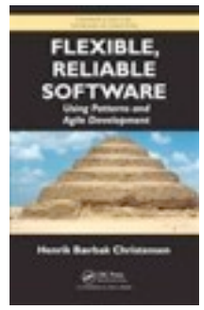
1. Introduce a parameter (which town)
2. Switch on the parameter each time town specific behaviour is needed.

Parametrisation

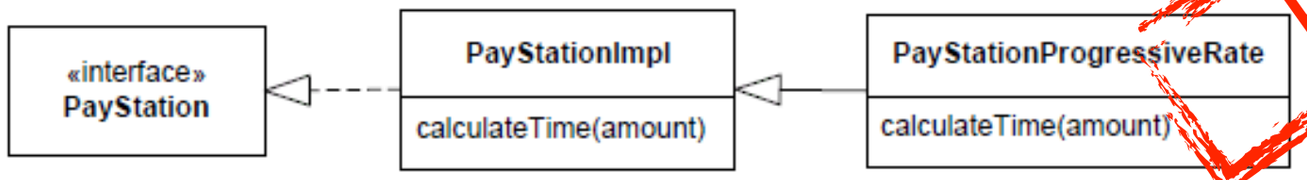


- **Benefits**
  - Simple
  - Avoid the multiple maintenance problem
- **Liabilities**
  - Reliability concerns: adding a new rate model => adding code to the existing *PayStationImpl* class, potential of introducing errors, complete regression testing
  - Readability concerns: switching code becomes long!
  - Responsibility erosion: new requirement: handle variations for different towns
  - Composition problem: a rate model that is a combination of existing ones leads to code duplication
- CONCLUSION: it is tempting but it should turn on the alarm bell!

Polymorphic Solution



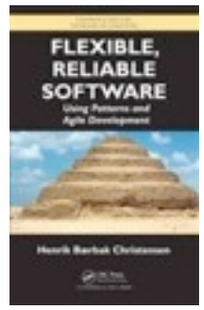
Subclass and override



Rehearsal

```
public void addPayment( int coinValue )
        throws IllegalCoinException {
  switch ( coinValue ) {
  case 5:
  case 10:
  case 25: break;
  default:
    throw new IllegalCoinException("Invalid coin: "+coinValue);
  }
  insertedSoFar += coinValue;
  timeBought = calculateTime(insertedSoFar);
}
/** calculate the parking time equivalent to the amount of
    cents paid so far
    @param paidSoFar the amount of cents paid so far
    @return the parking time this amount qualifies for
*/
protected int calculateTime(int paidSoFar) {
  return paidSoFar * 2 / 5;
}
```

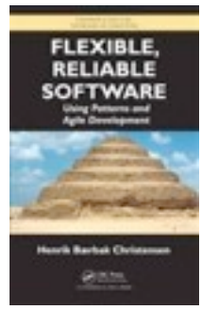
Polymorphic Solution



- **Benefits**
  - Avoid multiple maintenance
  - Reliability concern: new requirements regarding rate policies by adding new subclasses not by modifying existing classes.
  - Code readability: no code bloating

- **Liabilities**
  - Increased number of classes: one new class for each rate policy
  - Inheritance relation spent on single variation type: ending up with *"PayStationProgressivePriceButLiniarInWeekendsWithOracleDataBaseAccessDebuggingVersionAndBothCoinAndMobilePhonePaymentOptions" ???*
  - Reuse across variants difficult: new requirement:"We want a rate policy similar to Alphatown during weekdays but similar to Betatown during weekends."

Compositional Solution

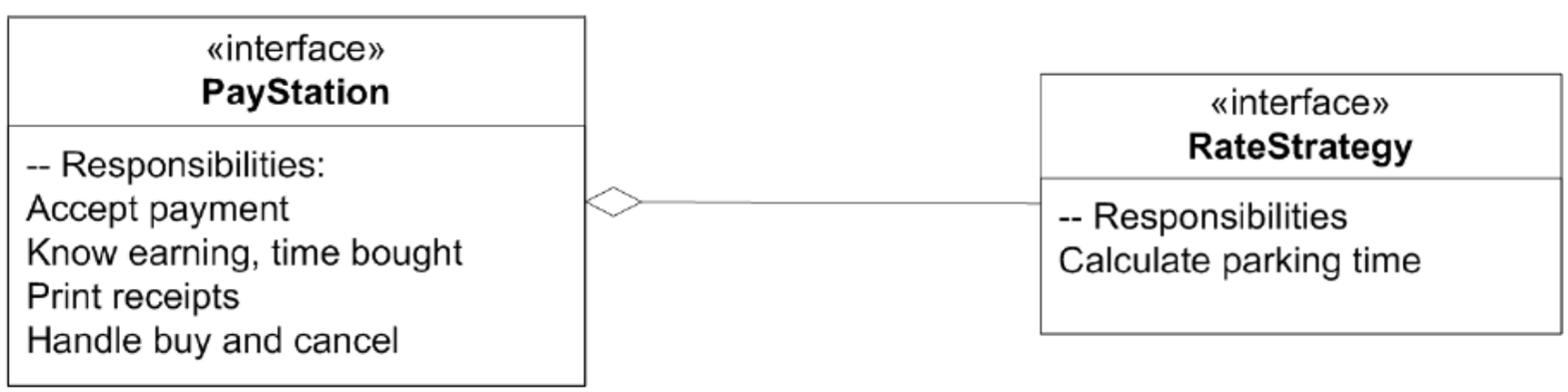


Responsibilities of original system:

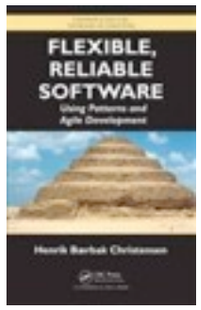
PayStation

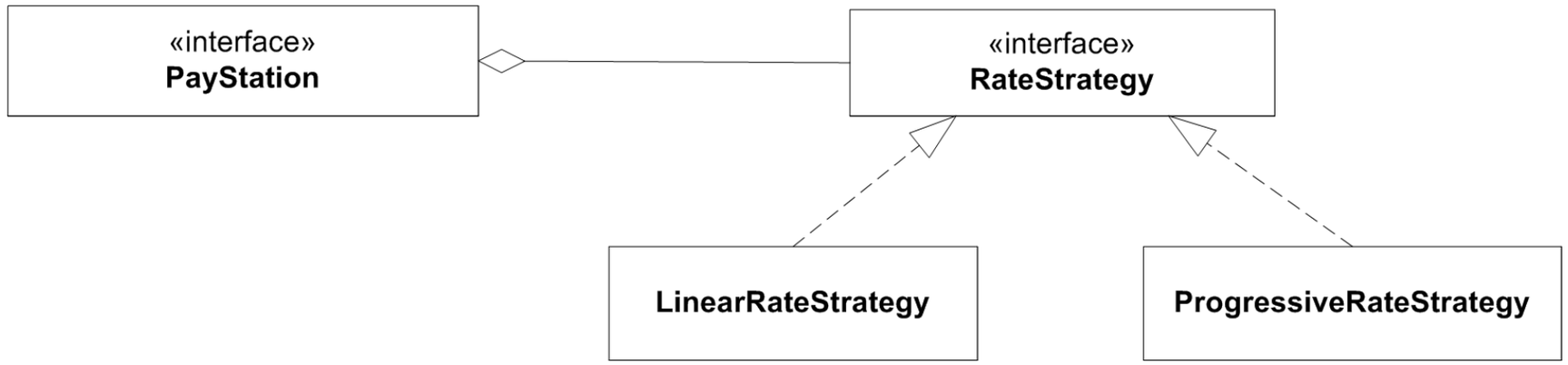
- Accept payment
- Calculate parking time based on payment
- Know earning, parking time bought
- Print receipts
- Handle buy and cancel transactions

Divide the responsibilities over a set of objects and let them collaborate: *delegation*

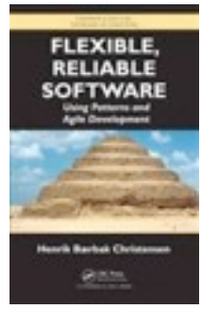


Concrete Behaviour





Compositional Solution

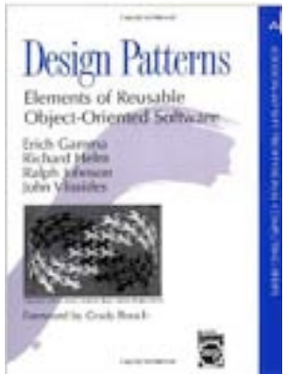


- **Benefits**
  - Readability: no code bloat of conditional statements
  - Run-time binding: change the rate policy while the system is running
  - Separation of responsibilities: responsibilities clearly stated in interfaces.
    - Variant selection is localized: There is only one place in the code where we decide which rate policy to take
    - Combinatorial: no inheritance used, can still be used to provide new behaviour.
- **Liabilities**
  - Increased number of interfaces, objects
  - Clients must be aware of strategies

The Compositional Process

- We have **identified some behaviour** that is likely to **change**…
  - rate policies
- We have clearly **stated a responsibility** that **covers this behaviour** and expressed it in an interface.
- We get the full pay station behaviour by **delegating the rate calculation** responsibility to a delegate.

Strategy Pattern

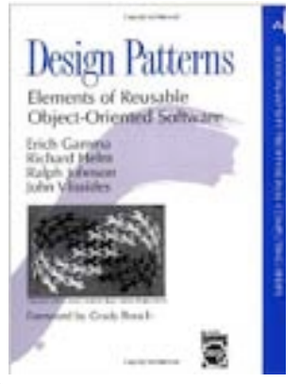


- **Intent**
  define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

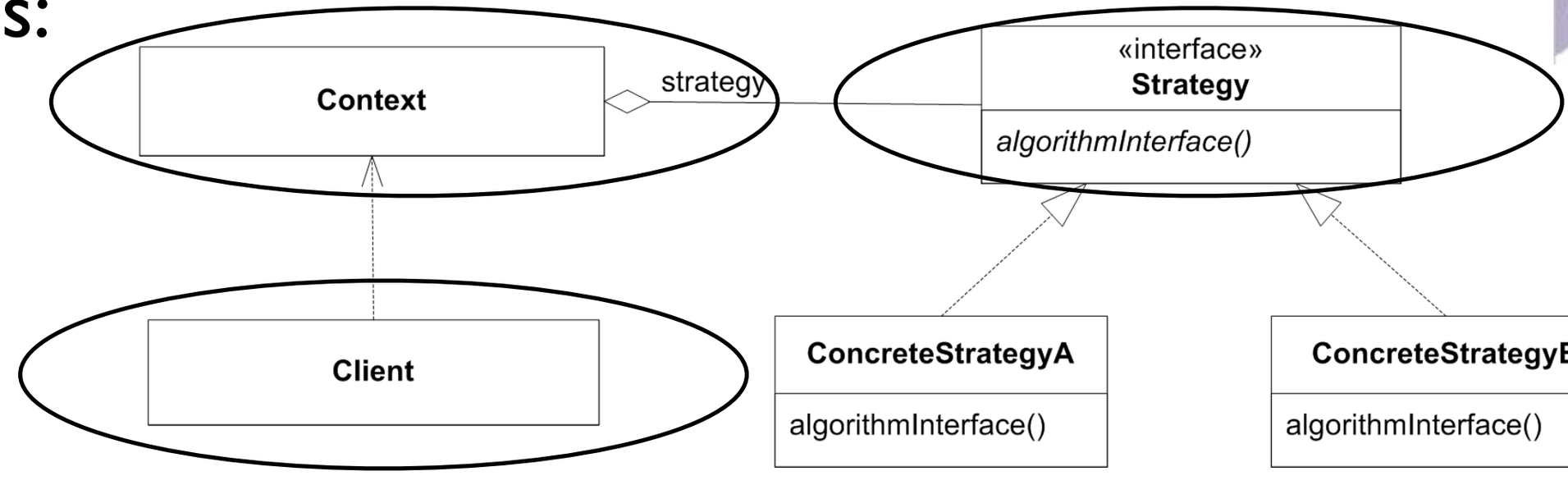
- **Problem**
  Your product must support variable algorithms or business rules and you want a flexible and reliable way of controlling the variability.

- **Solution**
  Separate the selection of algorithm from its implementation by expressing the algorithm's responsibilities in an interface and let each implementation of the algorithm realize this interface.

Strategy Pattern



Roles:



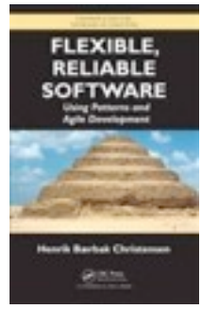
Consequences:

+ Hierarchies of Strategy classes define a family of algorithms or behaviors to reuse.
+ Is alternative to subclassing.
+ Is an alternative to using conditional statements for selecting desired behavior
+ Can offer a choice of implementations for the same behavior

- Clients must be aware of different Strategies.
- Some communication overhead between Strategy and Context.
- The number of objects in the application increases.

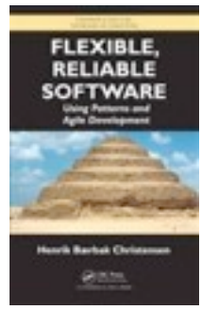
Theme 2
Program Design
Refactoring and Integration Testing

Learning Objectives



- learn a *small steps approach* to refactoring as well as see the *Triangulation principle* in action on a more complex problem.
- learn about the different levels in testing and the difference between unit and integration testing in particular.

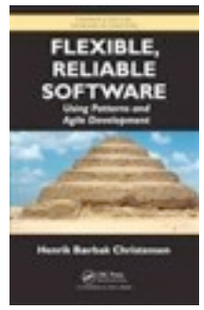
Problem Statement



- How to introduce the Betatown rate structure?
- Two options:
  - Write test cases that tests the behavior of Betatown rate structure and get the new compositional design as well as the rate calculation algorithm in place.
  - Refactor the existing pay station to use a compositional design first for Alphatown. Next write test cases and algorithm for the Betatown rate structure.

Take small steps

Take Small Steps

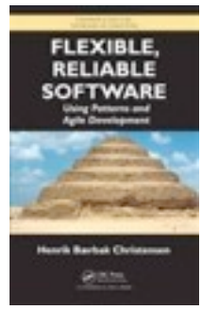


* refactor Alphatown to use a compositional design
* handle rate structure for Betatown

REMEMBER

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

Iteration 1: Refactoring



1. Introduce the *RateStrategy* interface.
2. Refactor *PayStationImpl* to use a reference to a *RateStrategy* instance for calculating rate (test to see it fail.)
3. Move the rate calculation algorithm to a class implementing the *RateStrategy* interface.
4. Refactor the pay station setup to use a concrete *RateStrategy* instance.

Step 1: 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 );
}
```

Step 2: introduce a reference in the *PayStationImpl*

```
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);
  }
```

Change the setup in the test case class:


```
@Before
  public void setUp() {
    ps = new PayStationImpl( new LinearRateStrategy() );
  }
```

Change the constructor:

```
/** Construct a pay station instance with the given
      rate calculation strategy.
      @param rateStrategy the rate calculation strategy to use
  */
  public PayStationImpl( RateStrategy rateStrategy ) {
    this.rateStrategy = rateStrategy;
  }
```

Fake it! Define a stub:

```
public class LinearRateStrategy implements RateStrategy {
  public int calculateTime( int amount ) {
    return 0;
  }}
```

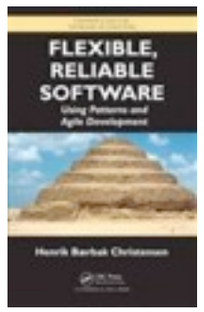
Change stub into a real application:

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

Key Points

- **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.*
- **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*

Iteration 2: BetaTown Rate Policy

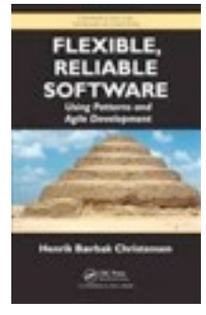


- Introducing rate policy is an example of **Triangulation**:
  - Iteration 2: add test case for first hour

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

  - Iteration 3: add test case for second hour
  - Iteration 4: add test case for third (and following) hour.

Iteration 5: Unit and Integration Testing




```
public class TestProgressiveRate {
  RateStrategy rs;

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

```
/** 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

- The ProgressiveRateStrategy is tested 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, as part of another unit and
- The actual Betatown pay station is never tested!

Three levels of Testing

- Unit Testing
- **Integration Testing**
  Integration testing is the process of executing a software unit in collaboration with other units in order to find defects in their interactions.
- **System Testing**
  System testing is the process of executing a whole software system in order to find deviations from the specified requirements.


```java
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);
  }
}
```

Test Suites


```
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 } )

public class TestAll {
  // Dummy - it is the annotations that tell JUnit
what to do...
}
```

Overview of the Iterations

1. refactor to introduce a rate strategy, make all existing test cases pass
2. triangulate the first hour rate calculation into the rate algorithm.
3. triangulate the second hour rate.
4. triangulate the third and following hours rate.
5. discover that the rate strategies can be tested as separate software units, refactor the test cases for the Betatown's rate algorithm.
6. discover that the analyzability of the pay station test code can be improved by introducing a simple rate strategy. Refactored test cases. Introduced integration testing of the pay station with the individual rate strategies.
7. introduce the JUnit annotations for handling test suites.

Conclusion

- Use the old functional tests to refactor without adding new or changing existing behaviour
- When everything is green again then proceed to introduce new/modified behaviour
- Review again to see if there is any dead code lying around or other refactorings to do