Lecture 07

ECE 1145: Software Construction and Evolution

Refactoring and Integration Testing (CH 8)

Announcements

- Iteration 2 due Sept. 26
  - For the screencast, choose just a few “interesting” test cases and **show each step** in the TDD process (1. add a test, 2. see it fail, 3. make a small change, 4. see the test pass, 5. refactor if necessary)
- Relevant Exercises: **7.5** 7.6 9.1 9.3
- Iteration 3 (Strategy) will be posted next week, due Oct. 3

Questions for Today

How do we safely modify existing code?

How do we implement a compositional design?

How do we adequately test a design with multiple parts?

Review

Pay Station so far:

- Initial implementation for AlphaTown – linear rate calculation
- BetaTown requested a progressive rate strategy

→ We discussed several models for implementation and decided to go forward with **compositional design**

③ identify behavior likely to change

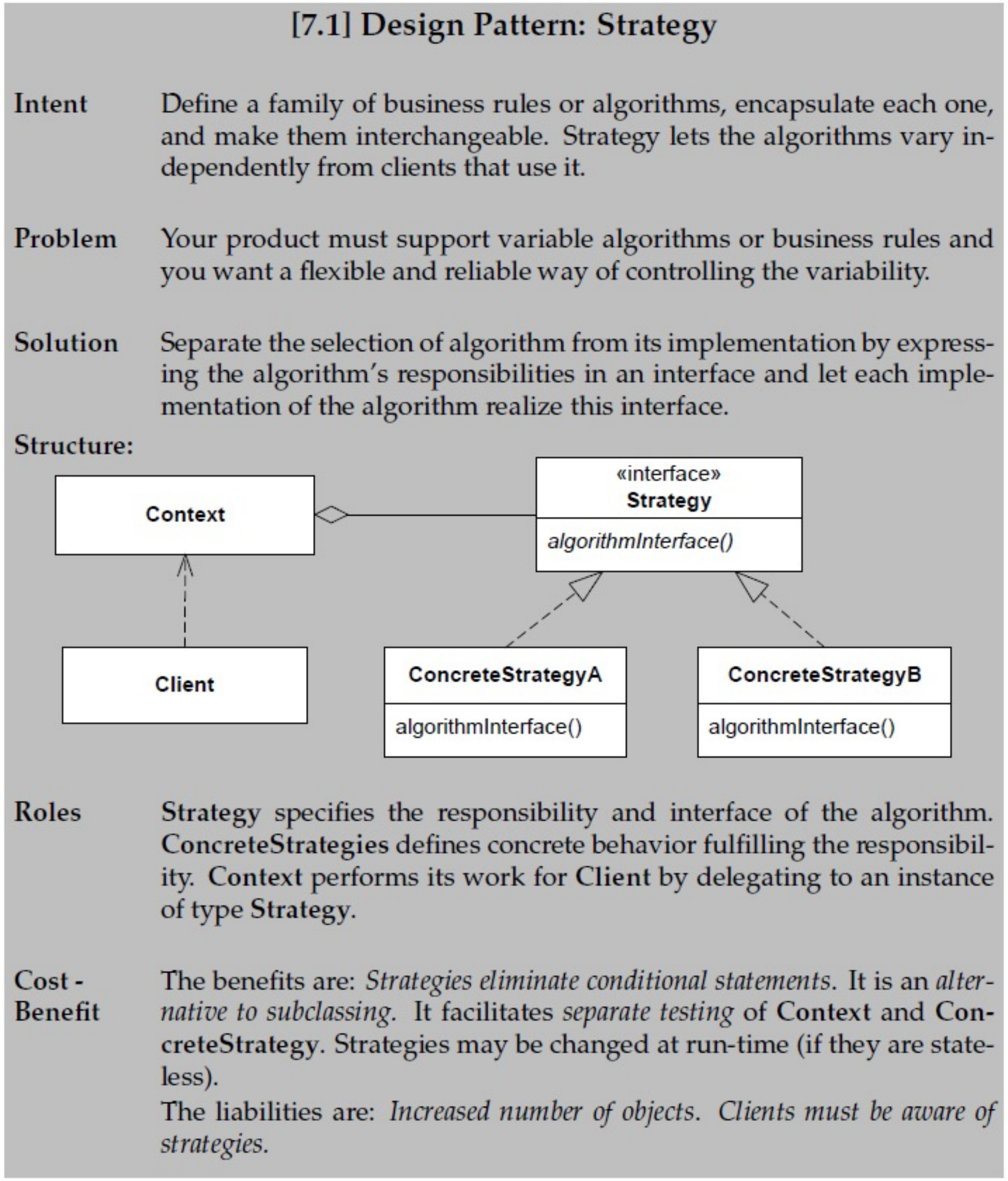
① express responsibility for behavior as interfaces

② use delegation to support behavior

→ **Strategy pattern**

Review

Strategy addresses the problem of encapsulating a family of algorithms / business rules and allows implementations to vary independently from the client that uses them.



[7.1] Design Pattern: Strategy

Intent Define a family of business rules or algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithms 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.

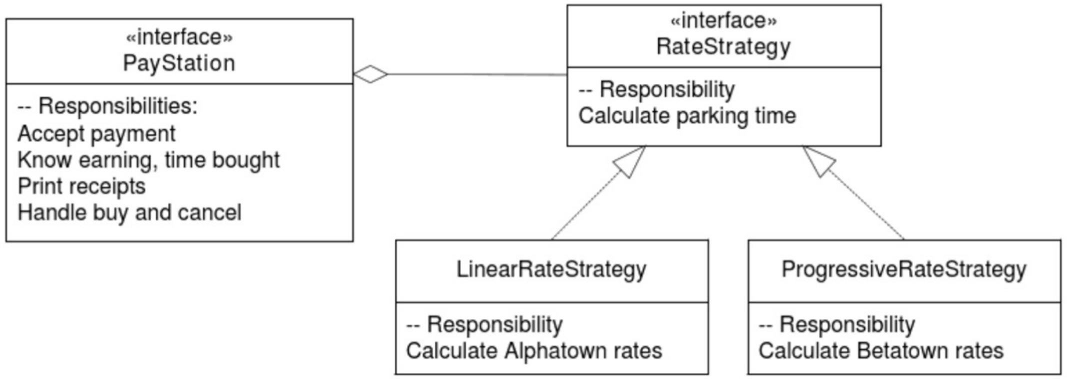
Structure:

Roles **Strategy** specifies the responsibility and interface of the algorithm. **ConcreteStrategies** defines concrete behavior fulfilling the responsibility. **Context** performs its work for **Client** by delegating to an instance of type **Strategy**.

Cost - Benefit The benefits are: *Strategies eliminate conditional statements*. It is an *alternative to subclassing*. It facilitates *separate testing* of **Context** and **ConcreteStrategy**. Strategies may be changed at run-time (if they are stateless).
The liabilities are: *Increased number of objects. Clients must be aware of strategies.*

Review

Strategy addresses the problem of encapsulating a family of algorithms / business rules and allows implementations to vary independently from the client that uses them.



«interface»
PayStation
-- Responsibilities:
Accept payment
Know earning, time bought
Print receipts
Handle buy and cancel

«interface»
RateStrategy
-- Responsibility
Calculate parking time

LinearRateStrategy
-- Responsibility
Calculate Alphatown rates

ProgressiveRateStrategy
-- Responsibility
Calculate Betatown rates

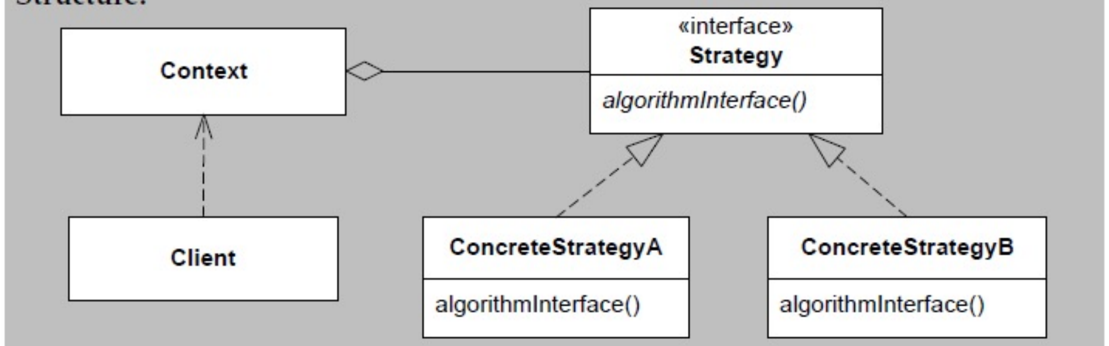
[7.1] Design Pattern: Strategy

Intent Define a family of business rules or algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithms 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.

Structure:



Context

«interface»
Strategy
algorithmInterface()

Client

ConcreteStrategyA
algorithmInterface()

ConcreteStrategyB
algorithmInterface()

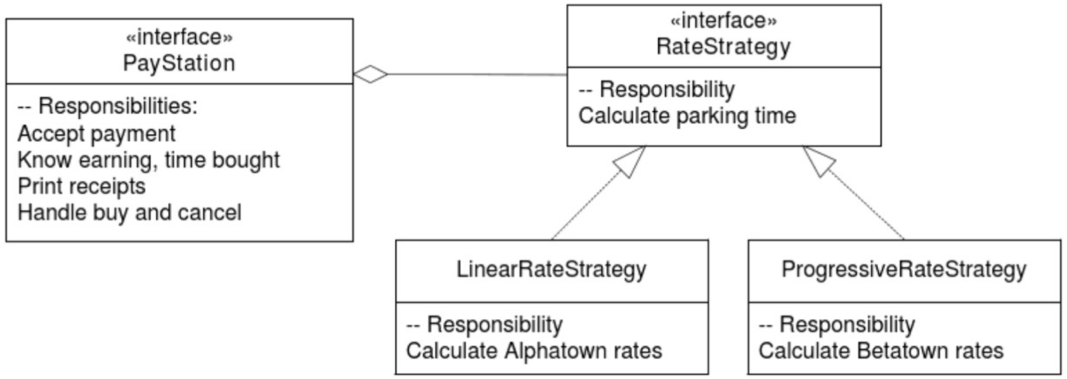
Roles **Strategy** specifies the responsibility and interface of the algorithm. **ConcreteStrategies** defines concrete behavior fulfilling the responsibility. **Context** performs its work for **Client** by delegating to an instance of type **Strategy**.

Cost - Benefit The benefits are: *Strategies eliminate conditional statements*. It is an *alternative to subclassing*. It facilitates *separate testing* of **Context** and **ConcreteStrategy**. Strategies may be changed at run-time (if they are stateless).
The liabilities are: *Increased number of objects*. *Clients must be aware of strategies*.

Next Steps

We need to **modify** the pay station software to implement the Strategy pattern.

- How can we **reliably** modify PayStationImpl?
- How do we ensure that we do not introduce **defects** during modification?



«interface»
PayStation

-- Responsibilities:
Accept payment
Know earning, time bought
Print receipts
Handle buy and cancel

«interface»
RateStrategy

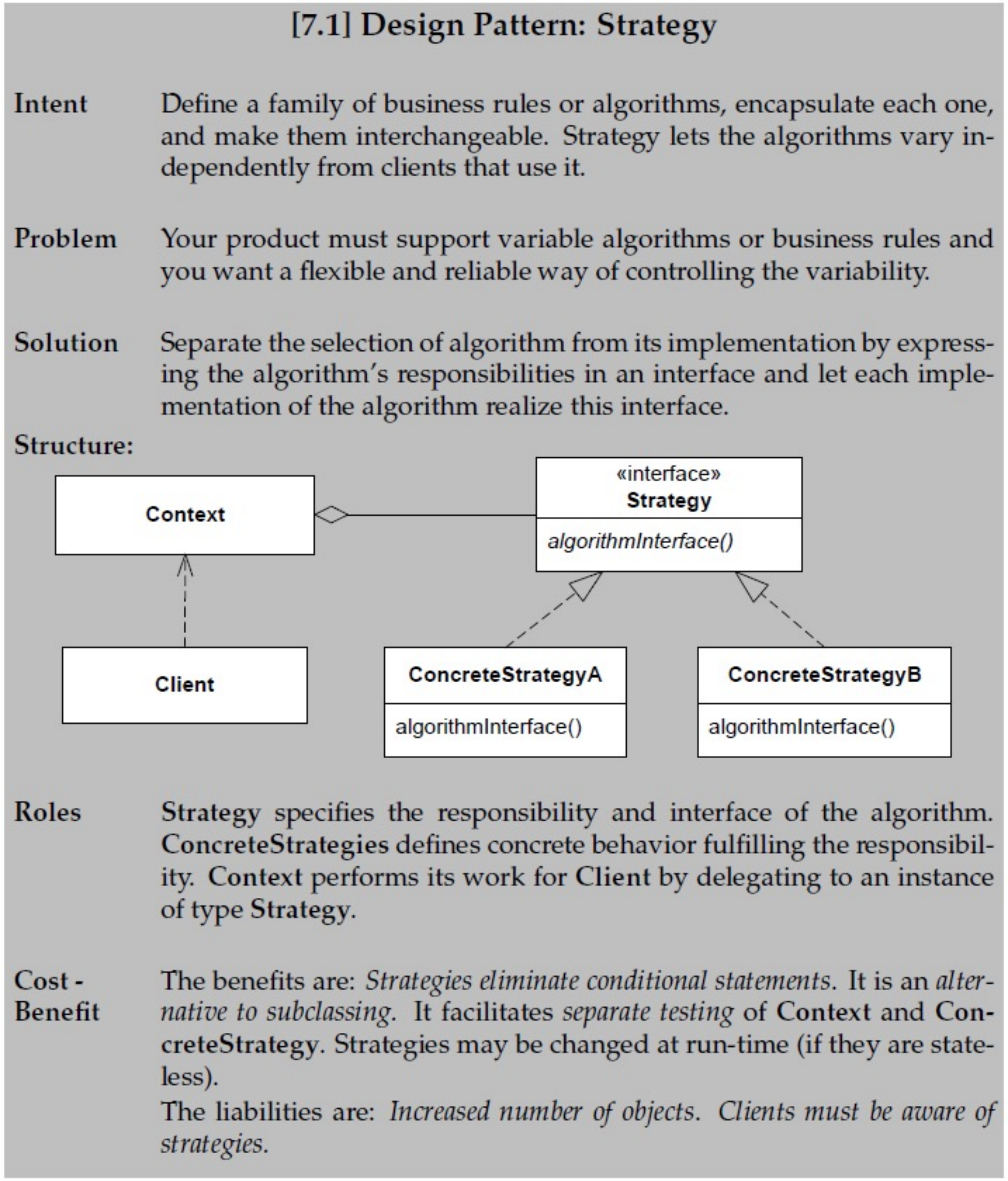
-- Responsibility
Calculate parking time

LinearRateStrategy

-- Responsibility
Calculate Alphatown rates

ProgressiveRateStrategy

-- Responsibility
Calculate Betatown rates



[7.1] Design Pattern: Strategy

Intent Define a family of business rules or algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithms 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.

Structure:

Roles **Strategy** specifies the responsibility and interface of the algorithm. **ConcreteStrategies** defines concrete behavior fulfilling the responsibility. **Context** performs its work for **Client** by delegating to an instance of type **Strategy**.

Cost - Benefit The benefits are: *Strategies eliminate conditional statements*. It is an *alternative to subclassing*. It facilitates *separate testing* of **Context** and **ConcreteStrategy**. Strategies may be changed at run-time (if they are stateless).
The liabilities are: *Increased number of objects*. *Clients must be aware of strategies*.

Refactoring

Stay focused, take **small steps** (recall TDD principles)

1. **Refactor** the current AlphaTown implementation to use compositional design (Strategy) using **existing test cases**
2. **Add** code to handle BetaTown’s rate policy (in addition to preserving AlphaTown’s)

Refactoring

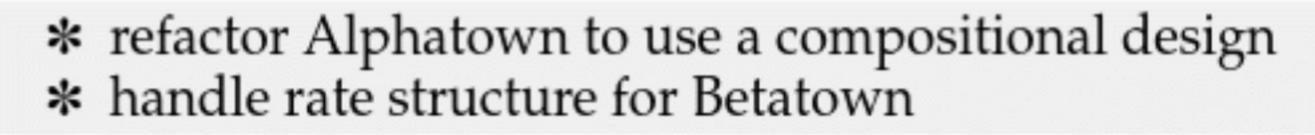
Stay focused, take **small steps** (recall TDD principles)

1. **Refactor** the current AlphaTown implementation to use compositional design (Strategy) using **existing test cases**
2. **Add** code to handle BetaTown's rate policy (in addition to preserving AlphaTown's)

> **Refactoring** is the process of changing a software system in such a way that **does not alter the external behavior** of the code yet **improves its internal structure**.
>
> Fowler, 1999

Refactoring

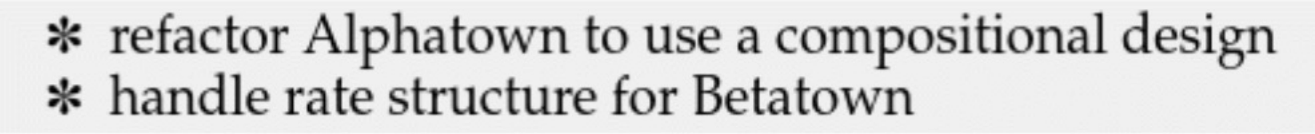
Stay focused, take **small steps** (recall TDD principles)



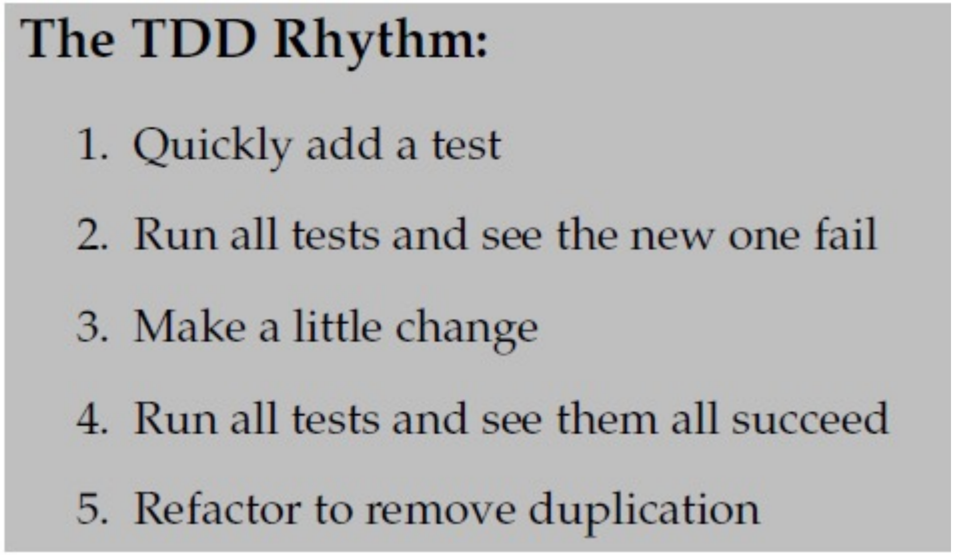
- * refactor Alphatown to use a compositional design
- * handle rate structure for Betatown

Refactoring

Stay focused, take **small steps** (recall TDD principles)



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



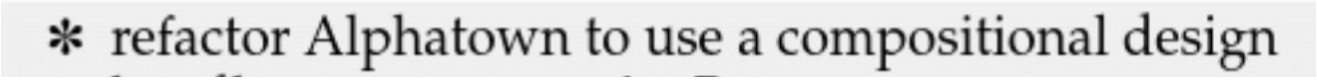
The TDD Rhythm:

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

Difference when refactoring is that we start by modifying code instead of writing a new test.

But, we still use test cases as we make changes.

Refactoring



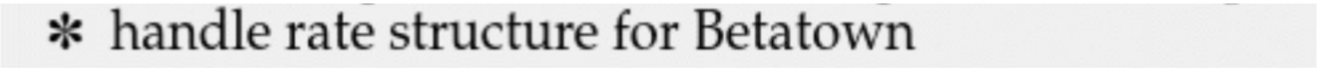
∗ refactor Alphatown to use a compositional design

i. Introduce the RateStrategy interface (so that the next step compiles)

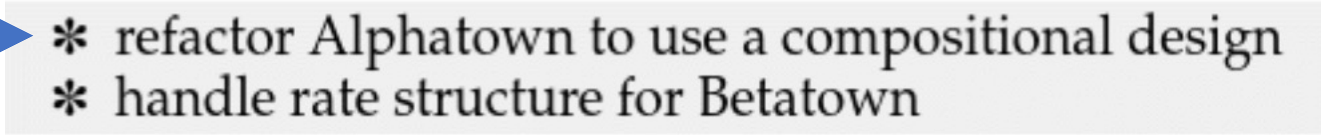
ii. Refactor PayStationImpl to use a reference to a RateStrategy instance for calculating rate (run tests to see them fail)

iii. Refactor PayStationImpl to use a concrete RateStrategy instance

iv. Move rate calculation algorithm to a class implementing the RateStrategy interface



∗ handle rate structure for Betatown

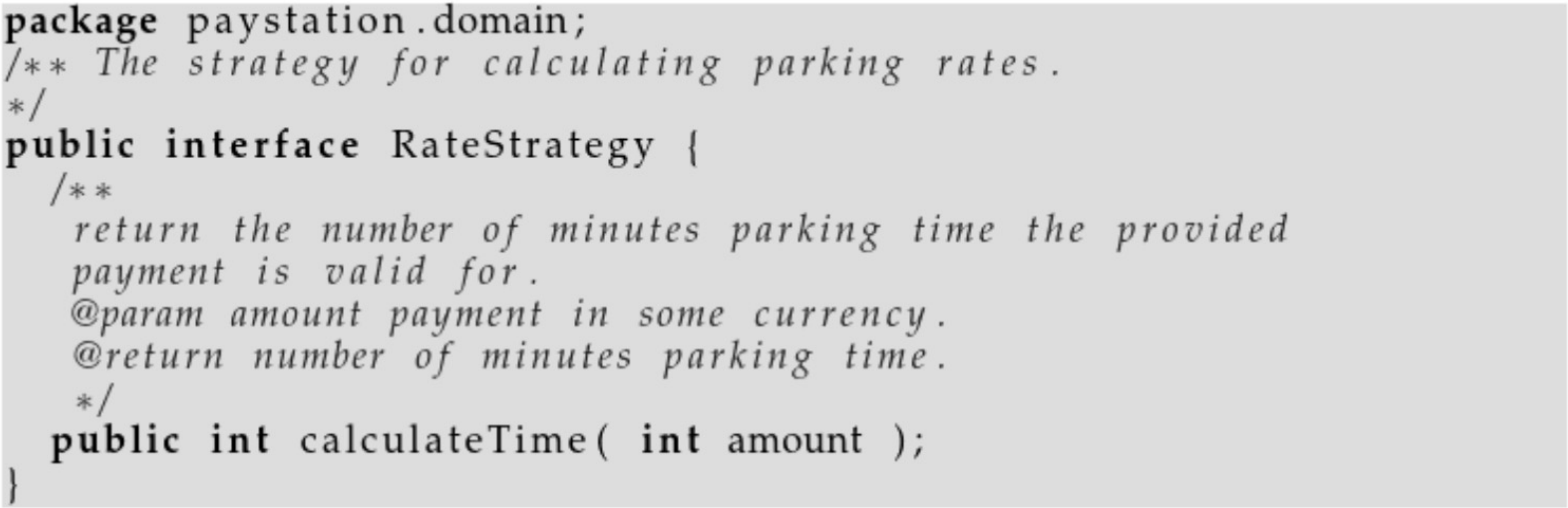


- ✻ refactor Alphatown to use a compositional design
- ✻ handle rate structure for Betatown

Refactoring

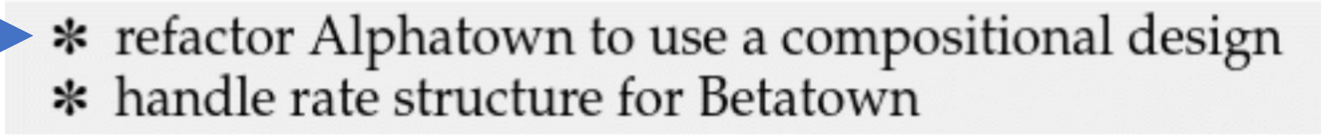
i. Introduce the RateStrategy interface

RateStrategy.java



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

Refactoring

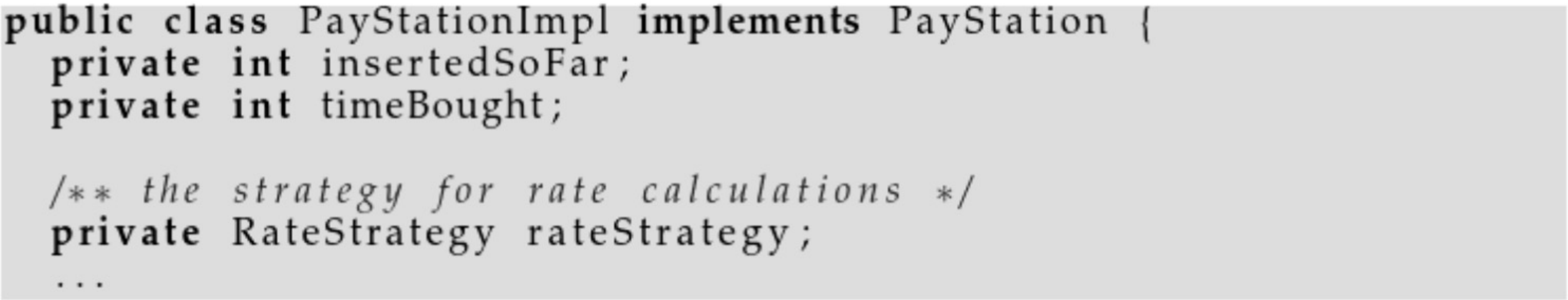


- ✻ refactor Alphatown to use a compositional design
- ✻ handle rate structure for Betatown

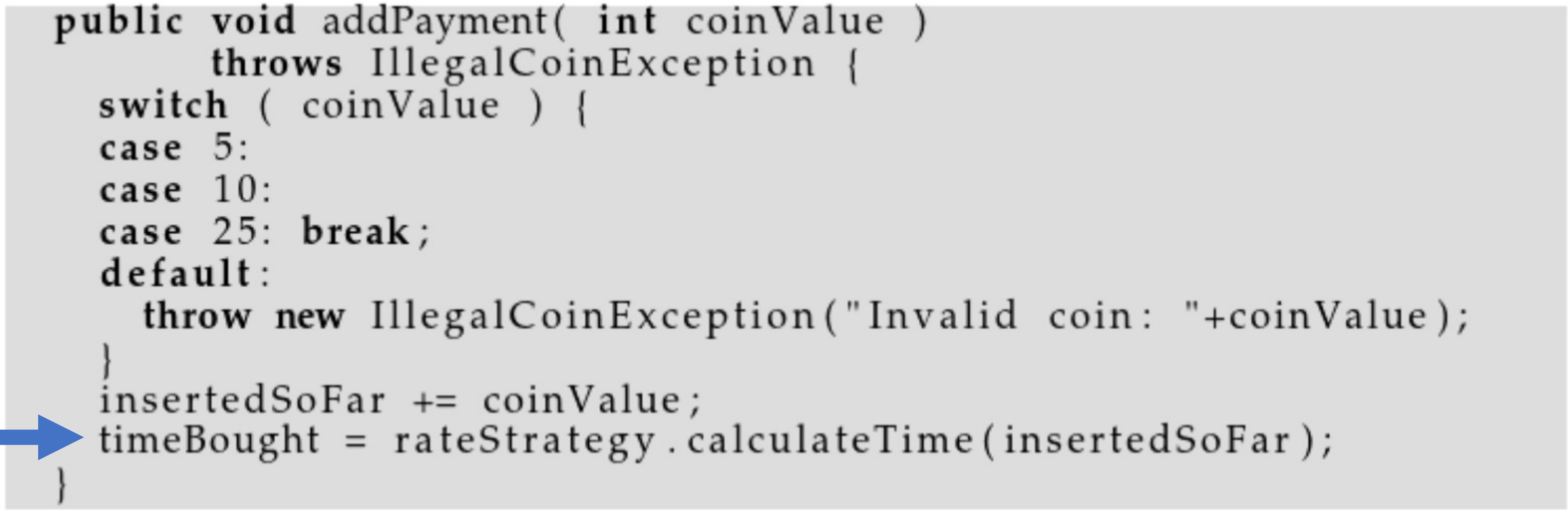
i. Introduce the RateStrategy interface (so that the next step compiles)

ii. Refactor PayStationImpl to use a reference to a RateStrategy instance for calculating rate (run tests to see them fail)

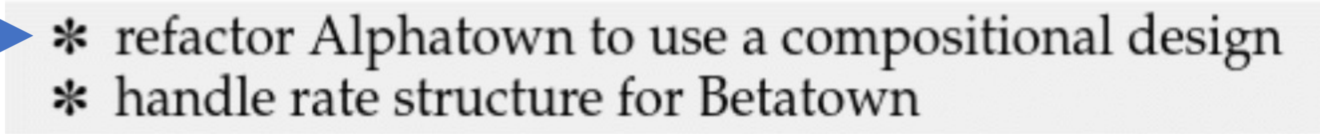
Delegate



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

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



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

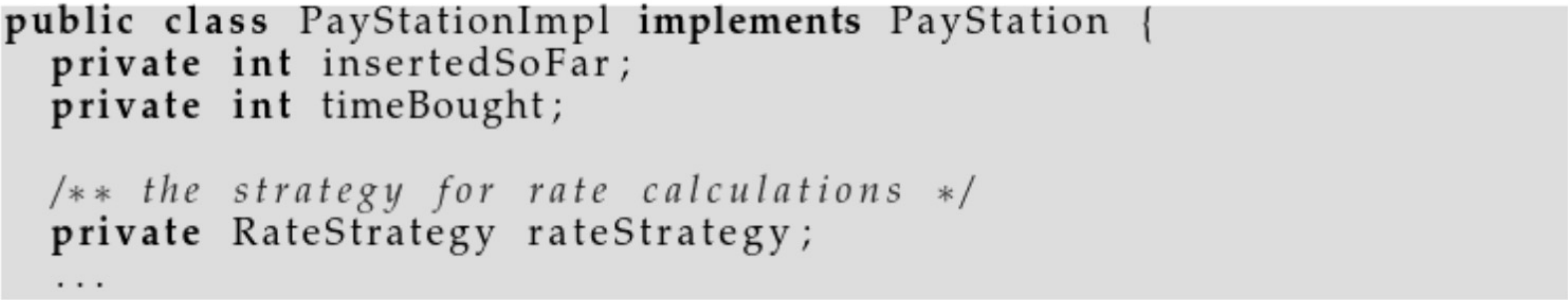

- refactor Alphatown to use a compositional design
- handle rate structure for Betatown

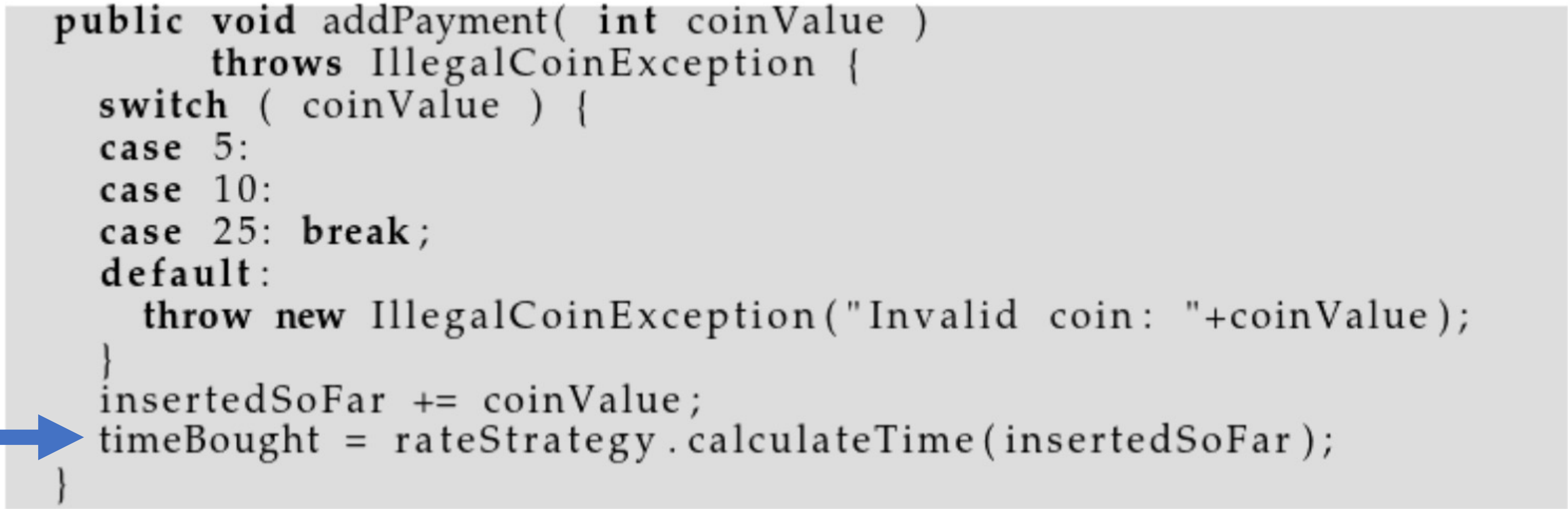
Refactoring

i. Introduce the RateStrategy interface (so that the next step compiles)

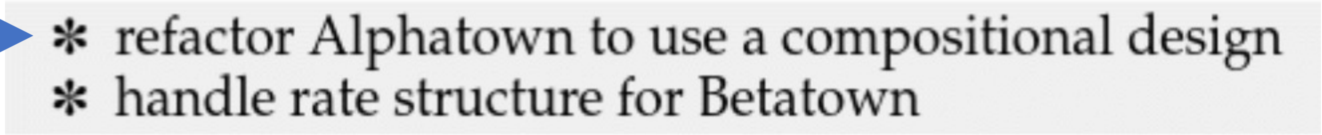
ii. Refactor PayStationImpl to use a reference to a RateStrategy instance for calculating rate (run tests to see them fail)

Delegate


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

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


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

Using existing test cases
→ Tests fail after this change

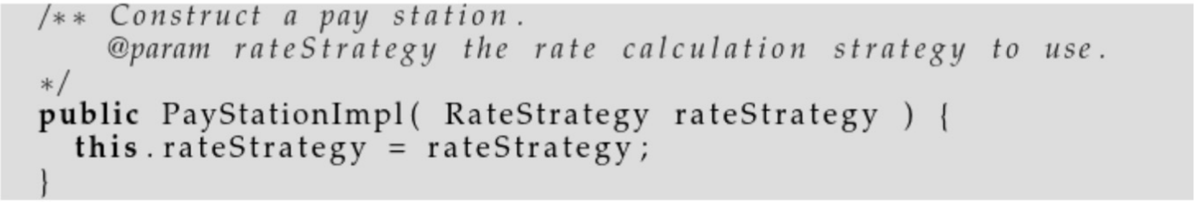


- ✻ refactor Alphatown to use a compositional design
- ✻ handle rate structure for Betatown

Refactoring

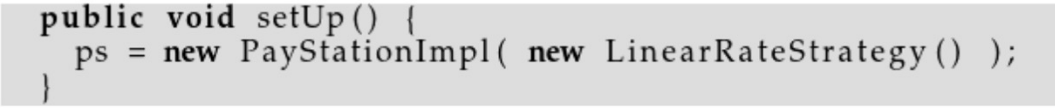
iii. Refactor PayStationImpl to use a concrete RateStrategy instance

PayStationImpl.java



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

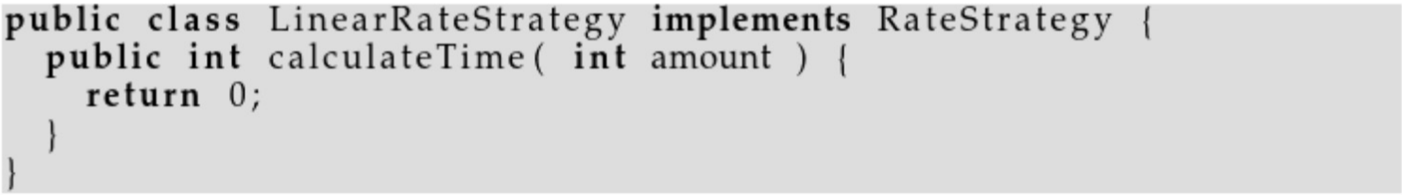
TestPayStation.java



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

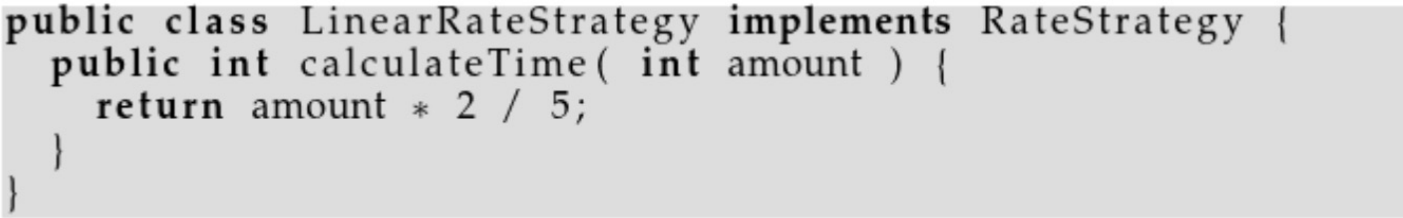
Also update test case setup

iv. Move rate calculation algorithm to a class implementing the RateStrategy interface



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

→ Tests fail by value

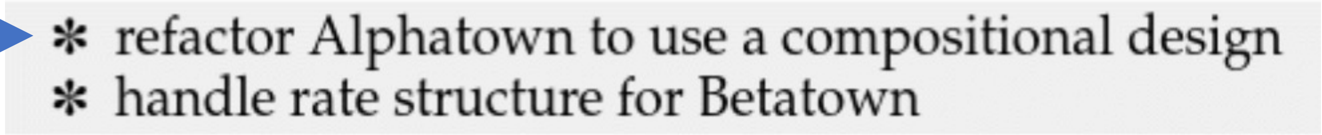


```
public class LinearRateStrategy implements RateStrategy {
  public int calculateTime( int amount ) {
    return amount * 2 / 5;
  }
}
```

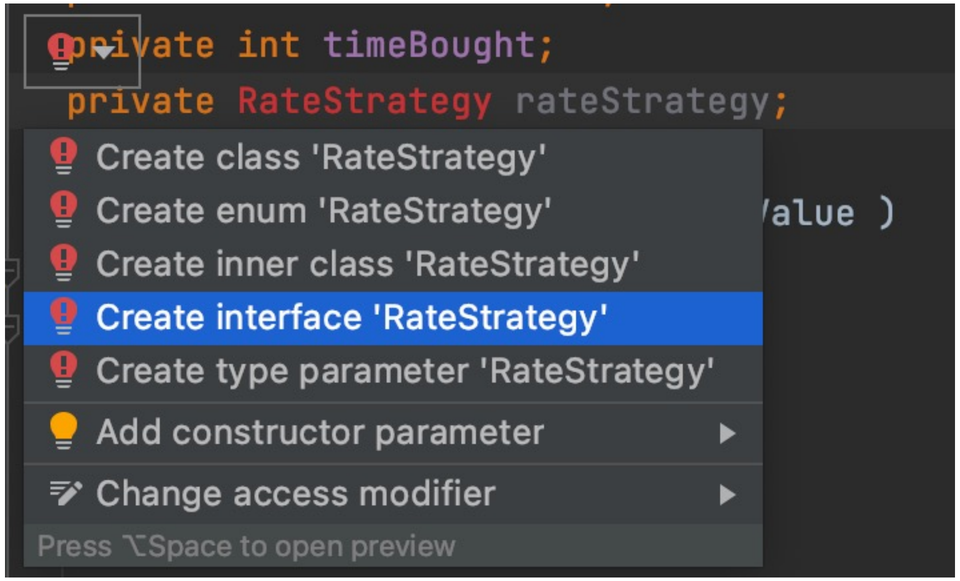
→ Tests pass

→ Refactoring complete!

Refactoring

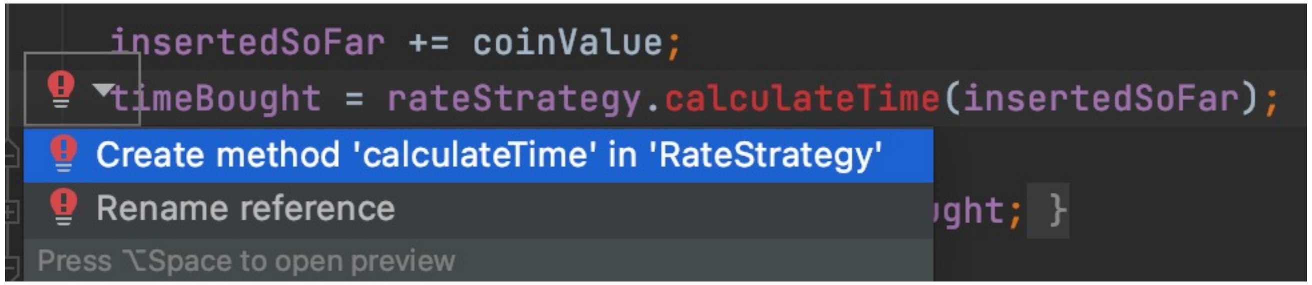


- ✻ refactor Alphatown to use a compositional design
- ✻ handle rate structure for Betatown



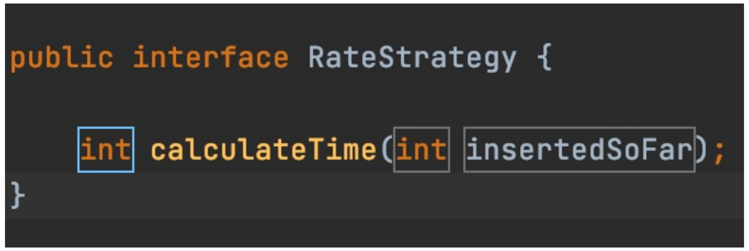
```
private int timeBought;
private RateStrategy rateStrategy;
```

Create class 'RateStrategy'
Create enum 'RateStrategy'
Create inner class 'RateStrategy'
Create interface 'RateStrategy'
Create type parameter 'RateStrategy'
Add constructor parameter
Change access modifier
Press ⌥Space to open preview
/alue)



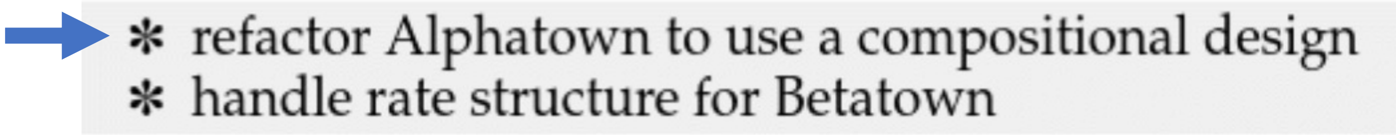
```
insertedSoFar += coinValue;
timeBought = rateStrategy.calculateTime(insertedSoFar);
```

Create method 'calculateTime' in 'RateStrategy'
Rename reference
Press ⌥Space to open preview
ight; }

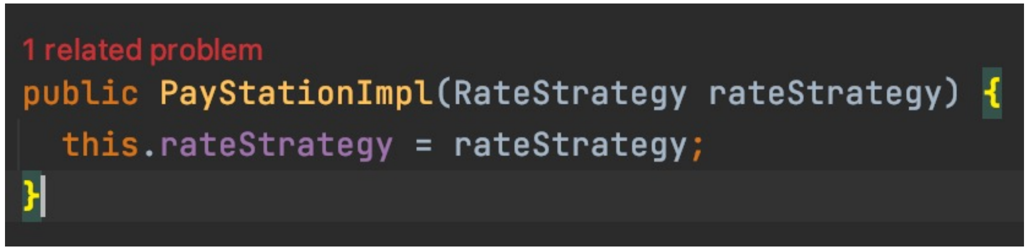


```
public interface RateStrategy {

    int calculateTime(int insertedSoFar);
}
```

Refactoring

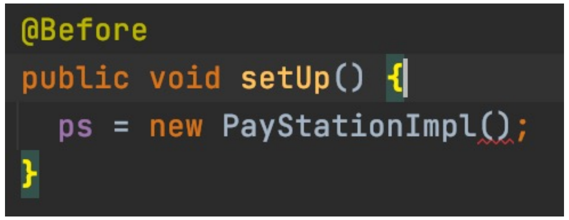


- ✻ refactor Alphatown to use a compositional design
- ✻ handle rate structure for Betatown

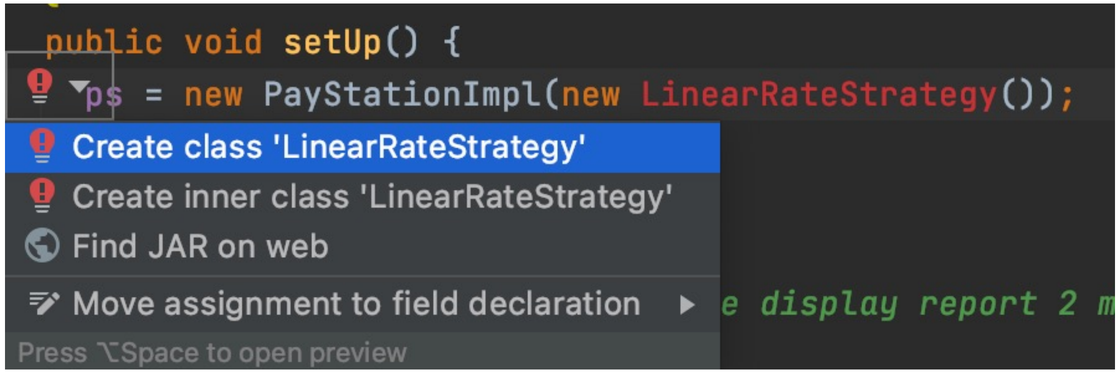


1 related problem

```
public PayStationImpl(RateStrategy rateStrategy) {
  this.rateStrategy = rateStrategy;
}
```

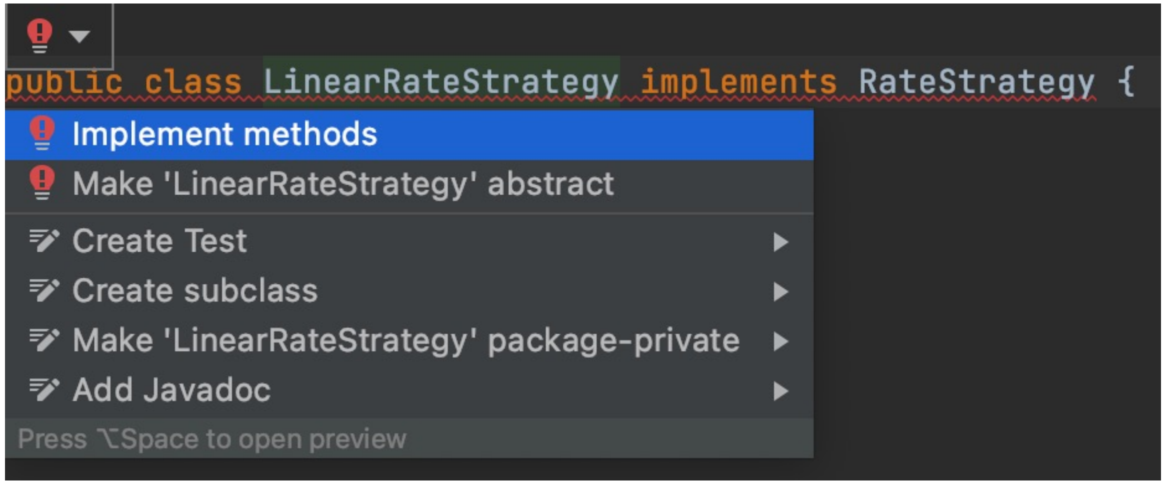


```
@Before
public void setUp() {
  ps = new PayStationImpl();
}
```



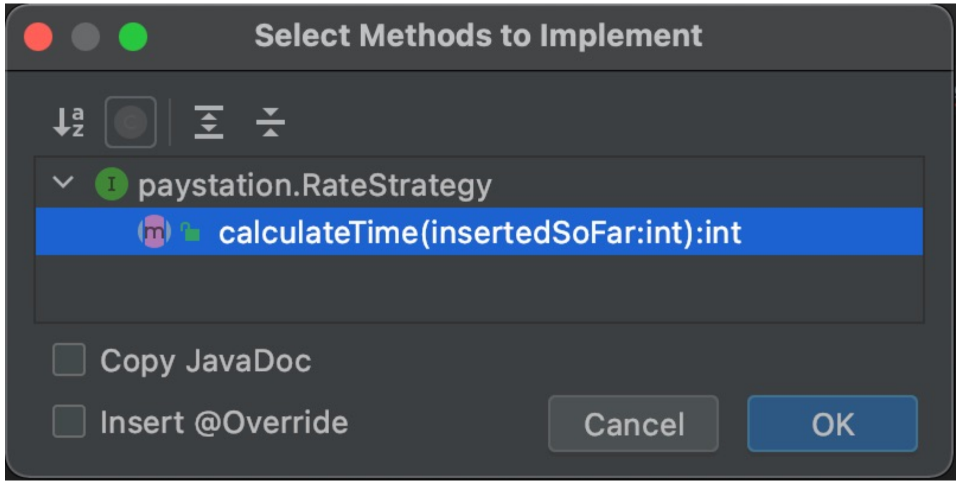
```
public void setUp() {
  ps = new PayStationImpl(new LinearRateStrategy());
```

Create class 'LinearRateStrategy'
Create inner class 'LinearRateStrategy'
Find JAR on web
Move assignment to field declaration ▸
e display report 2 m
Press ⌥Space to open preview

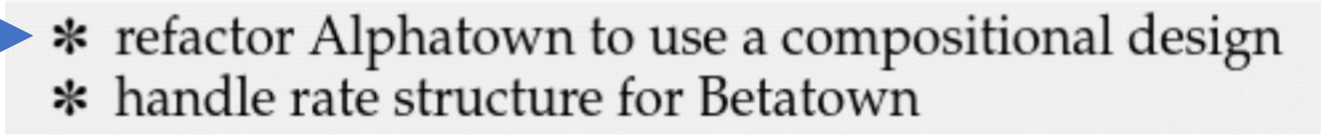


```
public class LinearRateStrategy implements RateStrategy {
```

Implement methods
Make 'LinearRateStrategy' abstract
Create Test ▸
Create subclass ▸
Make 'LinearRateStrategy' package-private ▸
Add Javadoc ▸
Press ⌥Space to open preview

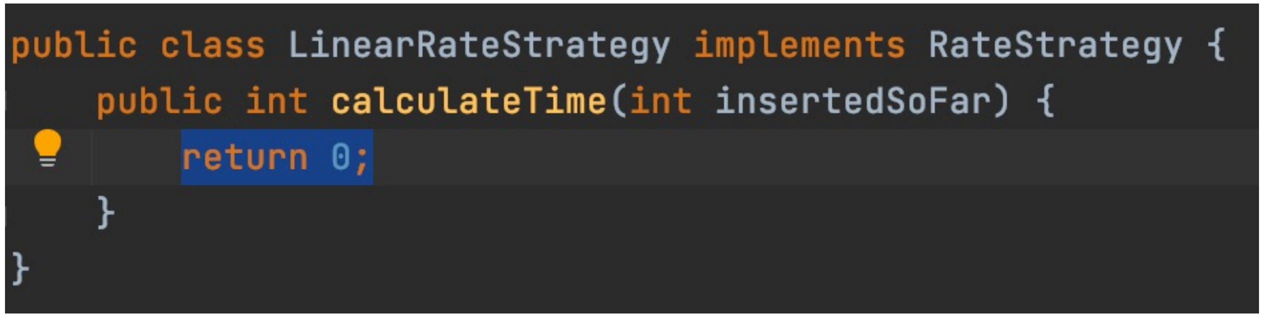


Select Methods to Implement

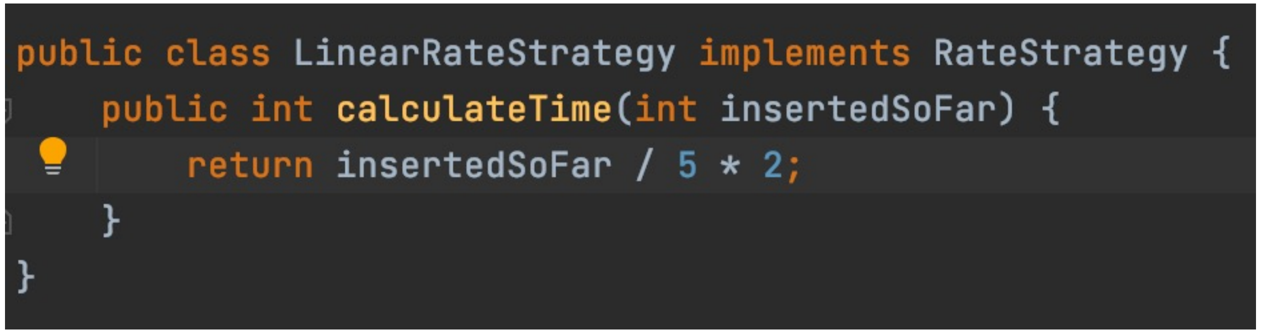


- ✻ refactor Alphatown to use a compositional design
- ✻ handle rate structure for Betatown

Refactoring

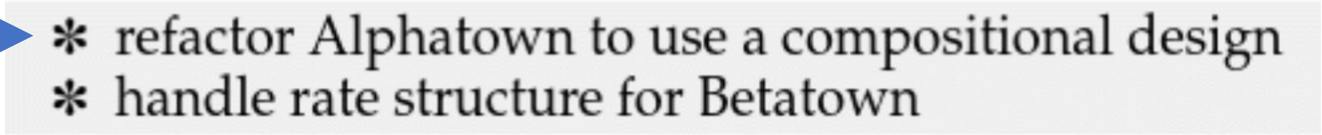


```
public class LinearRateStrategy implements RateStrategy {
    public int calculateTime(int insertedSoFar) {
        return 0;
    }
}
```

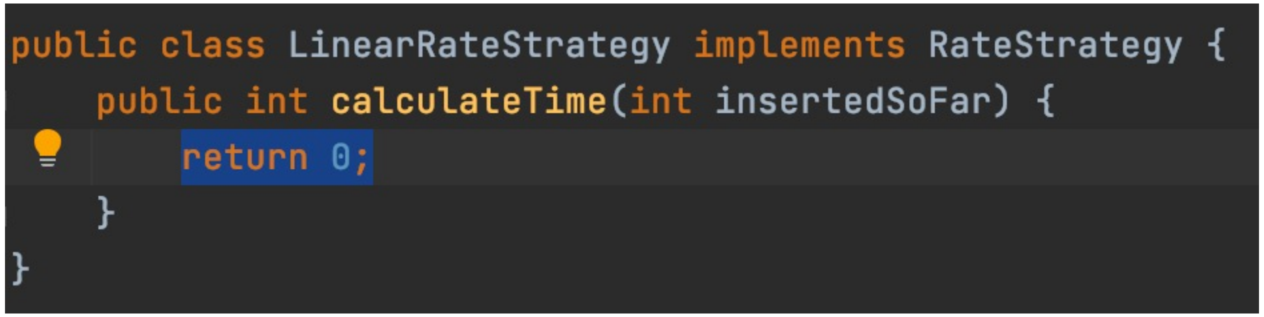


```
public class LinearRateStrategy implements RateStrategy {
    public int calculateTime(int insertedSoFar) {
        return insertedSoFar / 5 * 2;
    }
}
```

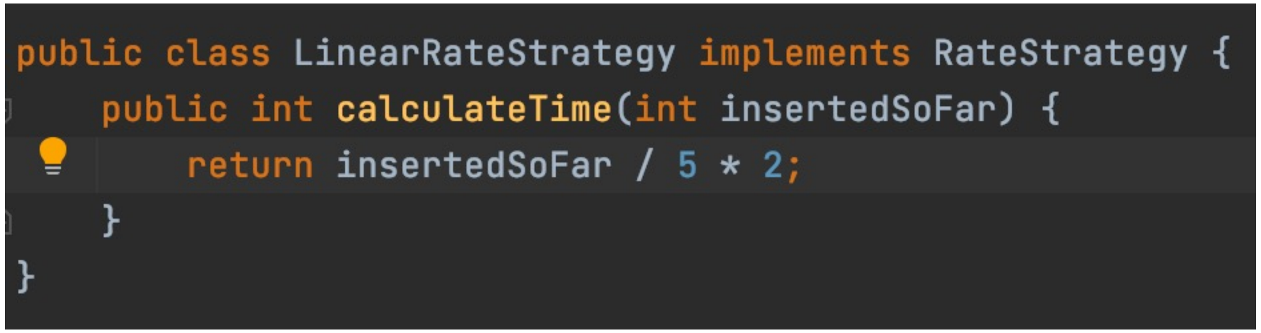
Refactoring



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



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



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

Solution-first programming: Statements that represent the solution are written first. They may refer to unknown interfaces, classes, methods, etc. – so use IDE suggestions to fill in the blanks.

Refactoring

Refactor the design **before** introducing new features
→ Make sure all existing tests pass!

Test cases should support refactoring
→ Refactoring is changing the implementation without changing external behavior
→ Test cases should not rely on implementation details

Do this: assertThat(game.getCityAt(p), is….)

Not this: assertThat(game.getInternalDataStruture().getAsArray()[47], is …)

Refactoring

Refactor the design **before** introducing new features
→ Make sure all existing tests pass!

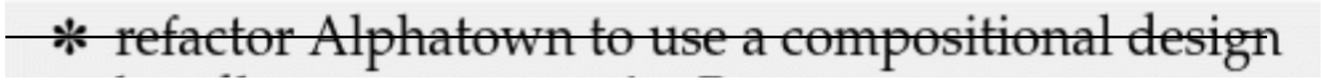
Test cases should support refactoring
→ Refactoring is changing the implementation without changing external behavior
→ Test cases should not rely on implementation details

Do this: assertThat(game.getCityAt(p), is….)

Not this: assertThat(game.getInternalDataStruture().getAsArray()[47], is …)

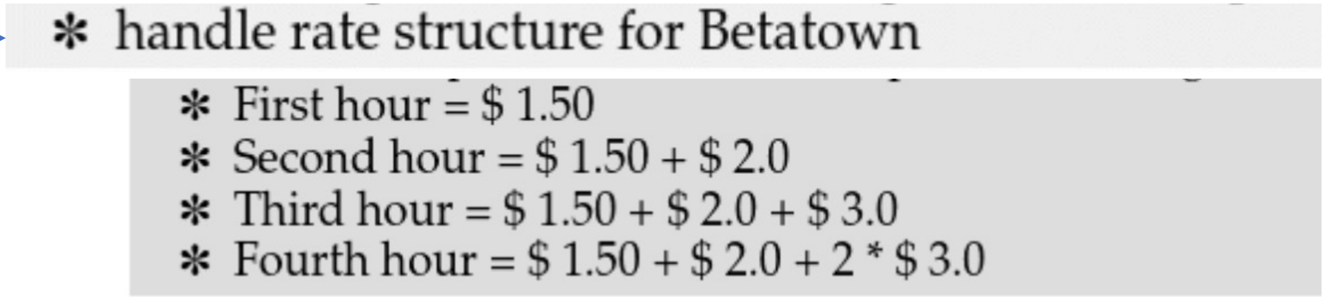
TDD may seem like a nuisance when developing…
But, it ensures you have tests written to enable refactoring!

Refactoring



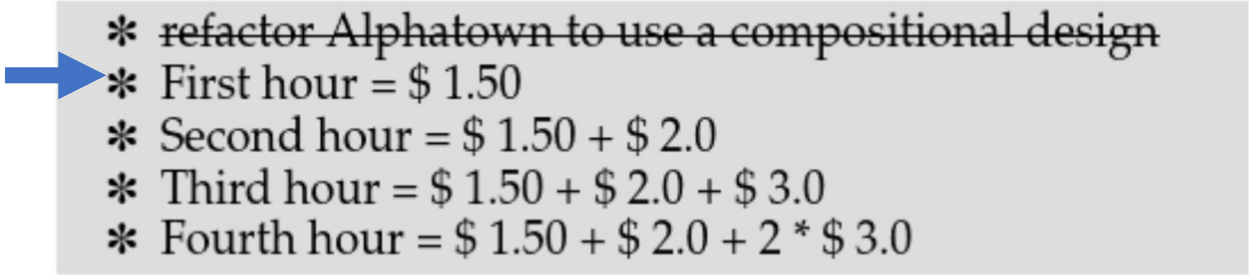
~~* refactor Alphatown to use a compositional design~~

i. ~~Introduce the RateStrategy interface (so that the next step compiles)~~
ii. ~~Refactor PayStationImpl to use a reference to a RateStrategy instance for calculating rate (run tests to see them fail)~~
iii. ~~Refactor PayStationImpl to use a concrete RateStrategy instance~~
iv. ~~Move rate calculation algorithm to a class implementing the RateStrategy interface~~



* handle rate structure for Betatown
  * First hour = $ 1.50
  * Second hour = $ 1.50 + $ 2.0
  * Third hour = $ 1.50 + $ 2.0 + $ 3.0
  * Fourth hour = $ 1.50 + $ 2.0 + 2 * $ 3.0

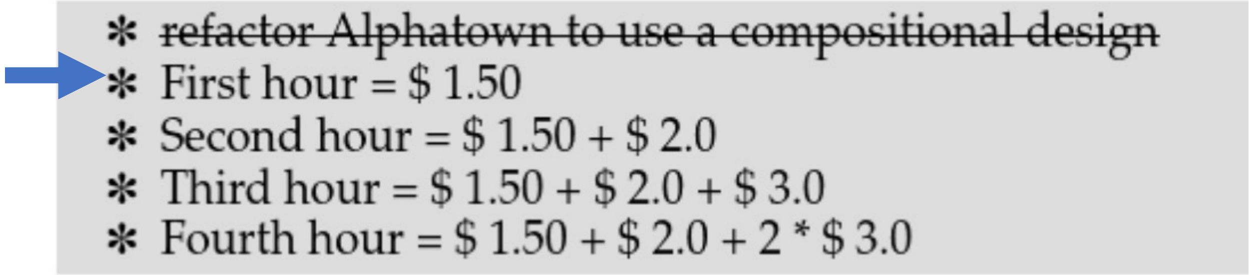
BetaTown



- ~~refactor Alphatown to use a compositional design~~
- First hour = $ 1.50
- Second hour = $ 1.50 + $ 2.0
- Third hour = $ 1.50 + $ 2.0 + $ 3.0
- Fourth hour = $ 1.50 + $ 2.0 + 2 * $ 3.0

To introduce the real BetaTown rate policy, we will use **Triangulation** (Abstract only when you have two or more examples)

BetaTown



1 ✻ ~~refactor Alphatown to use a compositional design~~

2 ✻ First hour = $ 1.50

3 ✻ Second hour = $ 1.50 + $ 2.0

4 ✻ Third hour = $ 1.50 + $ 2.0 + $ 3.0

✻ Fourth hour = $ 1.50 + $ 2.0 + 2 * $ 3.0

To introduce the real BetaTown rate policy, we will use **Triangulation** (Abstract only when you have two or more examples)

Iteration **2:** Add test case for the first hour

Add just enough production code to make the test pass

Iteration **3**: Add test case for second hour

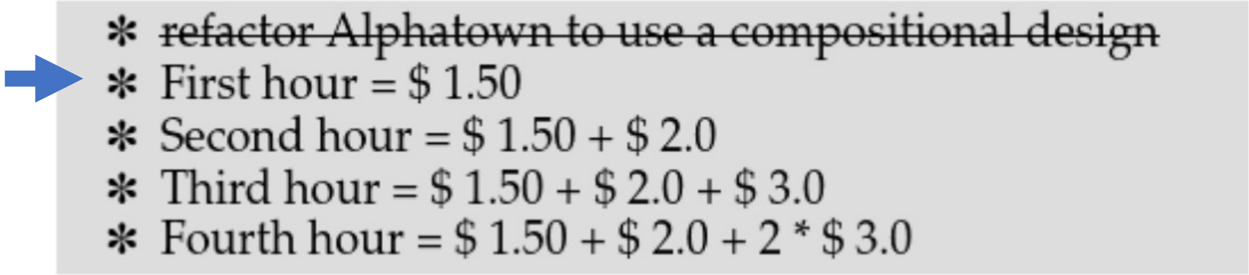
Add just enough complexity to the rate policy algorithm

Iteration **4:** Add test case for third and following hours

Add just enough more complexity

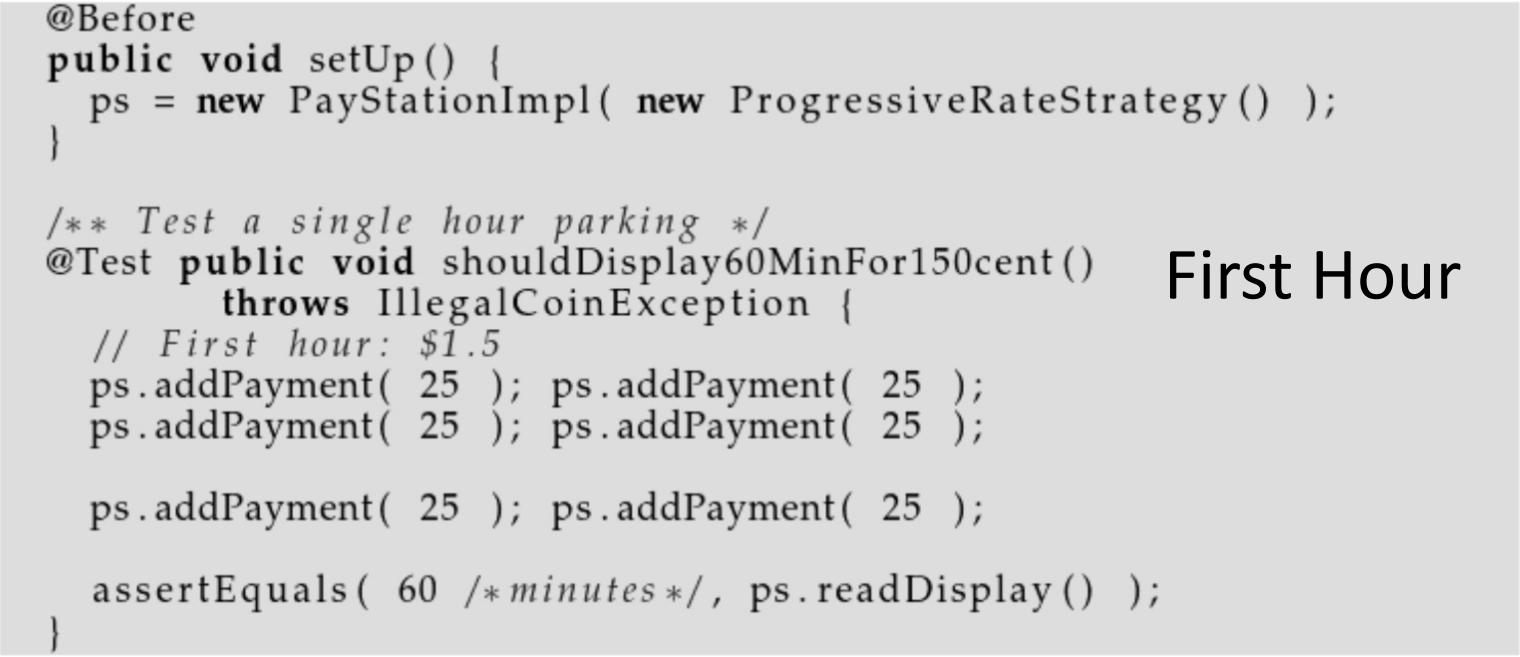
Iteration 2

BetaTown



- ~~refactor Alphatown to use a compositional design~~
- First hour = $ 1.50
- Second hour = $ 1.50 + $ 2.0
- Third hour = $ 1.50 + $ 2.0 + $ 3.0
- Fourth hour = $ 1.50 + $ 2.0 + 2 * $ 3.0

New test file: TestProgressiveRate.java

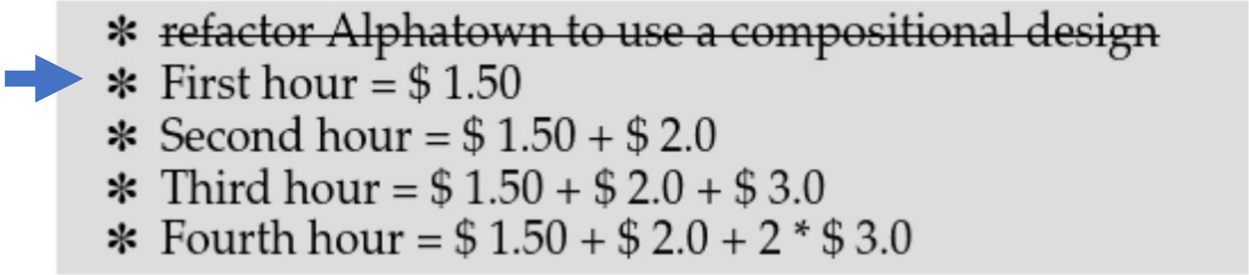


```
@Before
public void setUp() {
  ps = new PayStationImpl( new ProgressiveRateStrategy() );
}

/** Test a single hour parking */
@Test public void shouldDisplay60MinFor150cent()
        throws IllegalCoinException {
  // First hour: $1.5
  ps.addPayment( 25 ); ps.addPayment( 25 );
  ps.addPayment( 25 ); ps.addPayment( 25 );

  ps.addPayment( 25 ); ps.addPayment( 25 );

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

First Hour

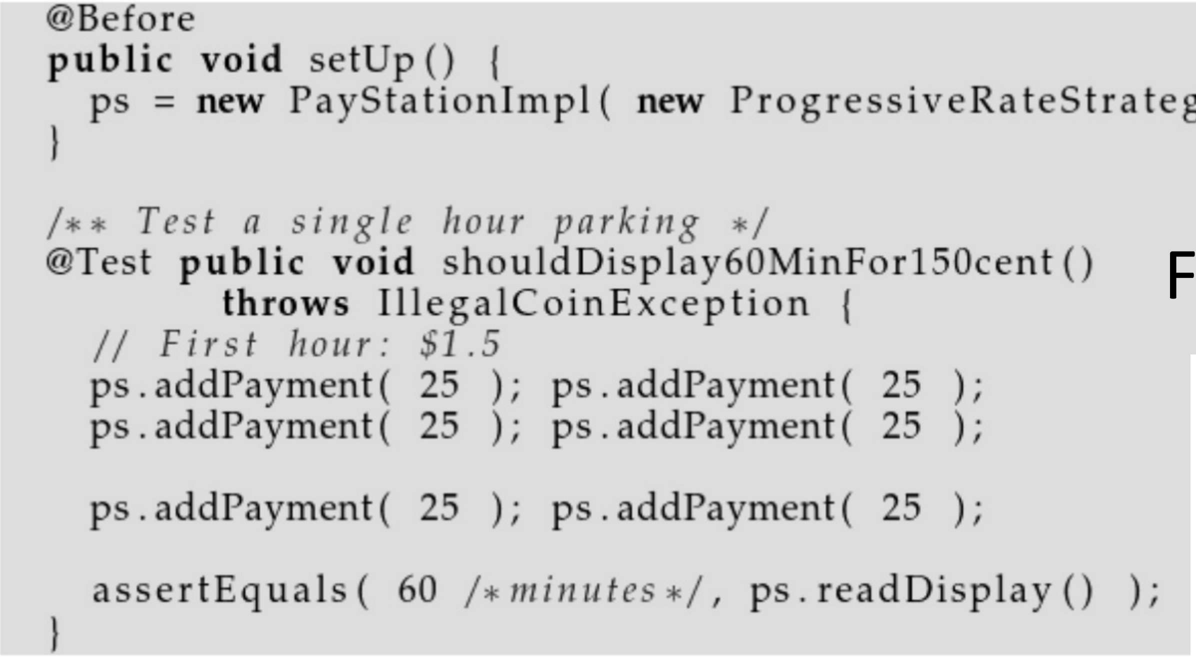
Iteration 2

BetaTown



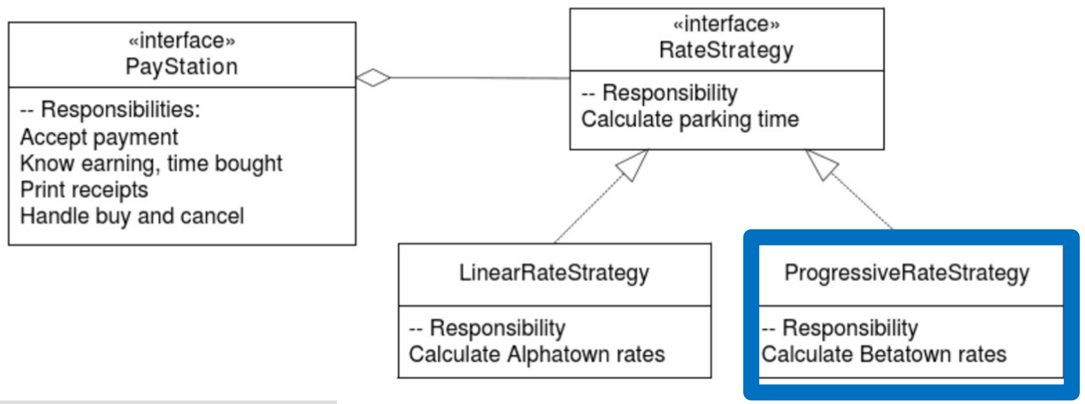
- ~~refactor Alphatown to use a compositional design~~
- First hour = $ 1.50
- Second hour = $ 1.50 + $ 2.0
- Third hour = $ 1.50 + $ 2.0 + $ 3.0
- Fourth hour = $ 1.50 + $ 2.0 + 2 * $ 3.0

New test file: TestProgressiveRate.java

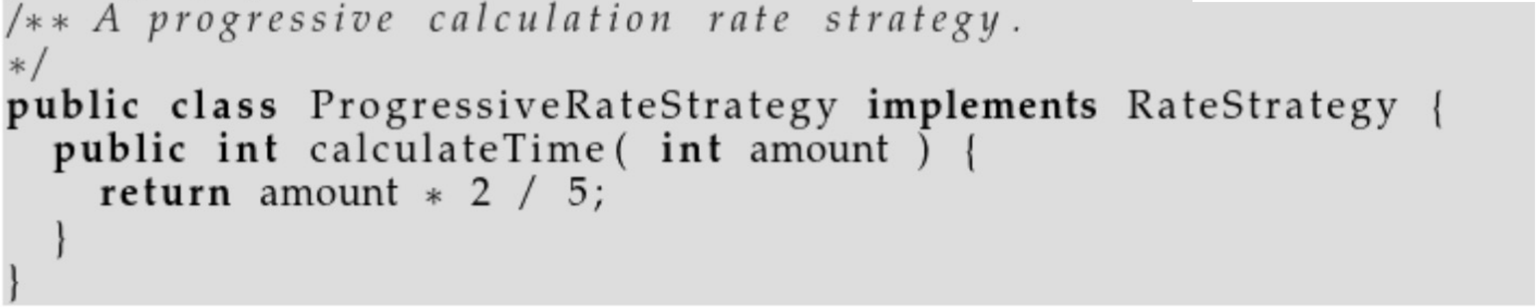


```
@Before
public void setUp() {
  ps = new PayStationImpl( new ProgressiveRateStrategy() );
}

/** Test a single hour parking */
@Test public void shouldDisplay60MinFor150cent()
        throws IllegalCoinException {
  // First hour: $1.5
  ps.addPayment( 25 ); ps.addPayment( 25 );
  ps.addPayment( 25 ); ps.addPayment( 25 );

  ps.addPayment( 25 ); ps.addPayment( 25 );

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

First Hour



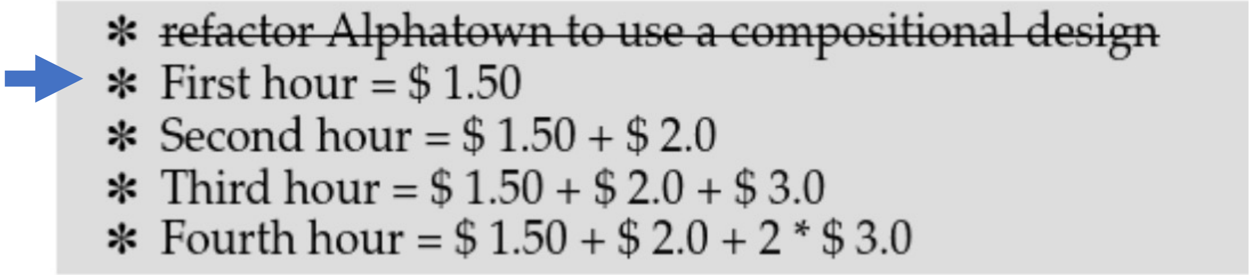
Create ProgressiveRateStrategy.java



```
package paystation.domain;
/** A progressive calculation rate strategy.
*/
public class ProgressiveRateStrategy implements RateStrategy {
  public int calculateTime( int amount ) {
    return amount * 2 / 5;
  }
}
```

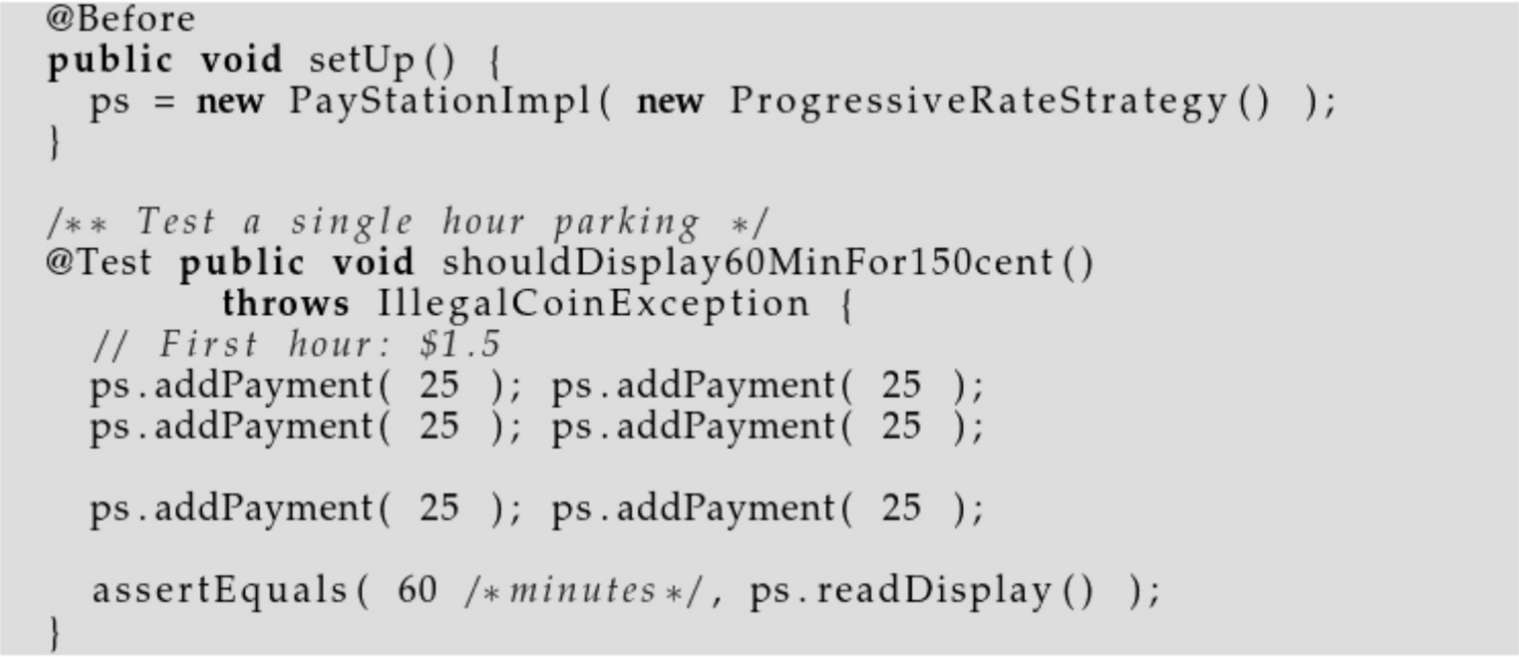
Iteration 2

BetaTown



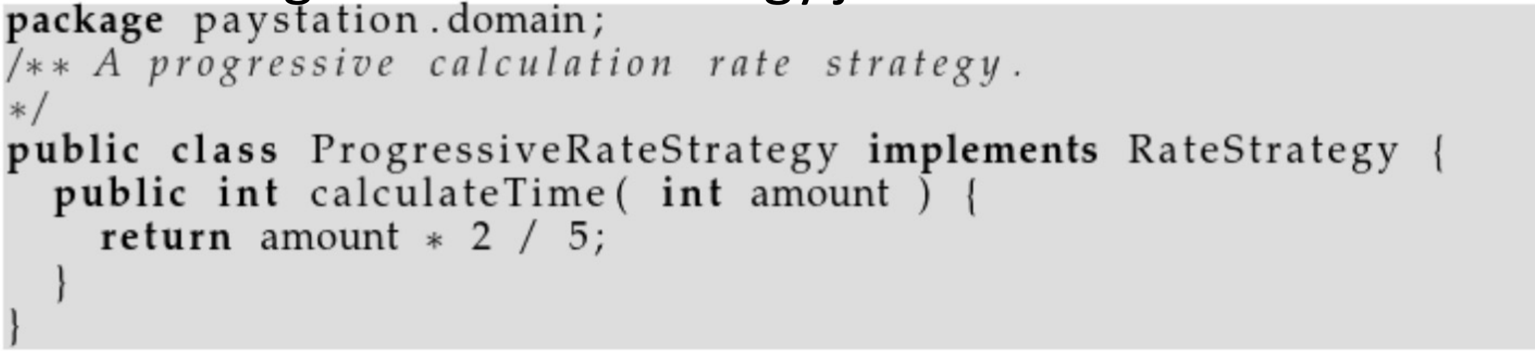
- ~~refactor Alphatown to use a compositional design~~
- First hour = $ 1.50
- Second hour = $ 1.50 + $ 2.0
- Third hour = $ 1.50 + $ 2.0 + $ 3.0
- Fourth hour = $ 1.50 + $ 2.0 + 2 * $ 3.0

New test file: TestProgressiveRate.java



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

/** Test a single hour parking */
@Test public void shouldDisplay60MinFor150cent()
        throws IllegalCoinException {
  // First hour: $1.5
  ps.addPayment( 25 ); ps.addPayment( 25 );
  ps.addPayment( 25 ); ps.addPayment( 25 );

  ps.addPayment( 25 ); ps.addPayment( 25 );

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

Create ProgressiveRateStrategy.java

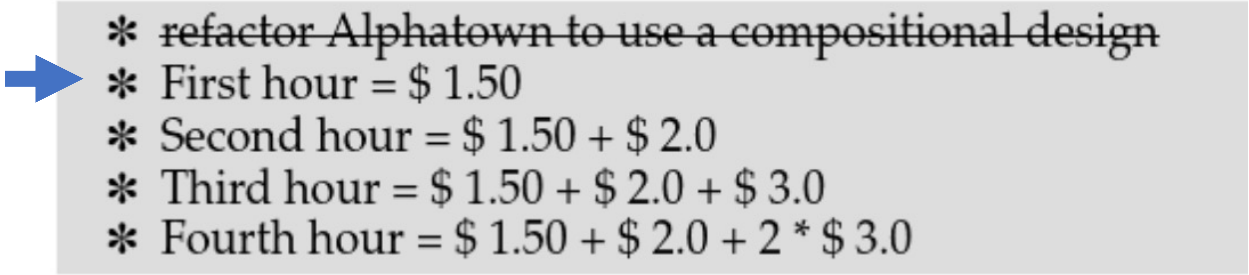


```
package paystation.domain;
/** A progressive calculation rate strategy.
*/
public class ProgressiveRateStrategy implements RateStrategy {
  public int calculateTime( int amount ) {
    return amount * 2 / 5;
  }
}
```

Same as AlphaTown!
("Obvious Implementation")

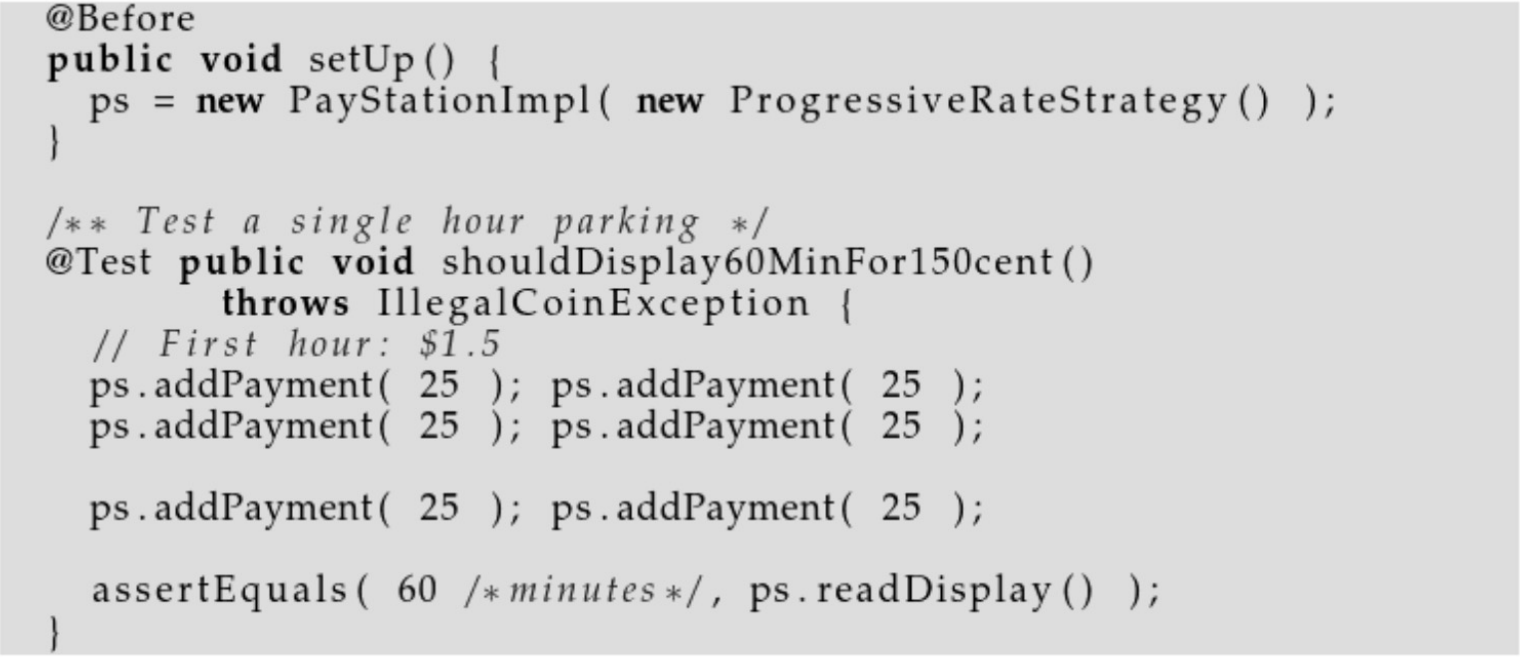
Iteration 2

BetaTown



- ~~refactor Alphatown to use a compositional design~~
- First hour = $ 1.50
- Second hour = $ 1.50 + $ 2.0
- Third hour = $ 1.50 + $ 2.0 + $ 3.0
- Fourth hour = $ 1.50 + $ 2.0 + 2 * $ 3.0

New test file: TestProgressiveRate.java



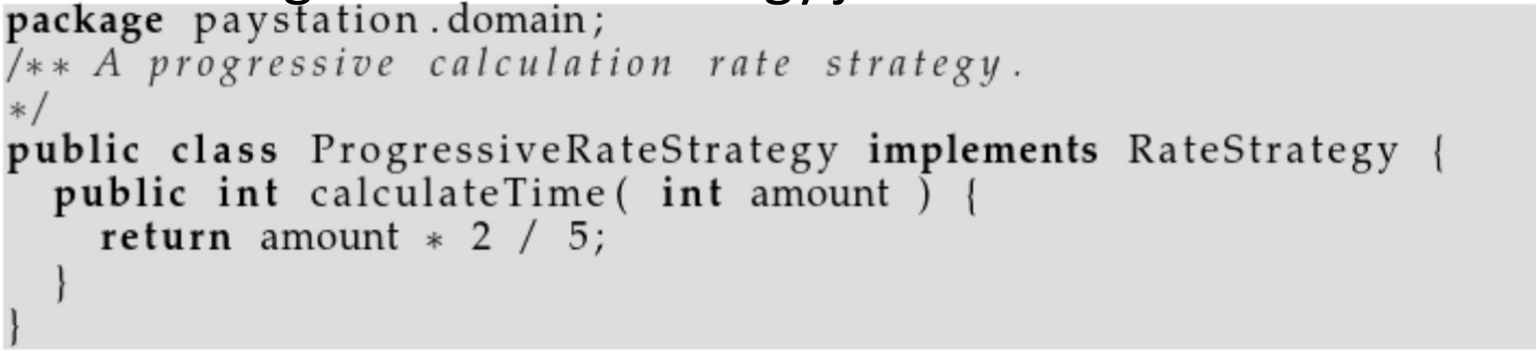
```
@Before
public void setUp() {
  ps = new PayStationImpl( new ProgressiveRateStrategy() );
}

/** Test a single hour parking */
@Test public void shouldDisplay60MinFor150cent()
        throws IllegalCoinException {
  // First hour: $1.5
  ps.addPayment( 25 ); ps.addPayment( 25 );
  ps.addPayment( 25 ); ps.addPayment( 25 );

  ps.addPayment( 25 ); ps.addPayment( 25 );

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

Tests pass! But, it is good practice to make the test case fail on purpose

→ Make sure it is executed!

→ e.g., return 0

Create ProgressiveRateStrategy.java

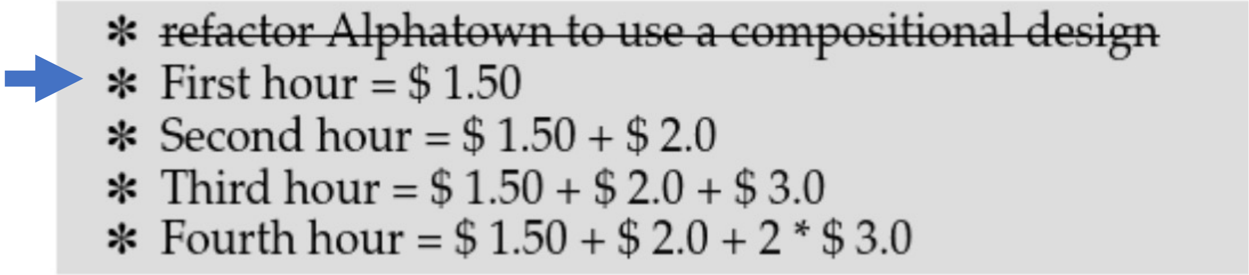


```
package paystation.domain;
/** A progressive calculation rate strategy.
*/
public class ProgressiveRateStrategy implements RateStrategy {
  public int calculateTime( int amount ) {
    return amount * 2 / 5;
  }
}
```

Same as AlphaTown!
("Obvious Implementation")

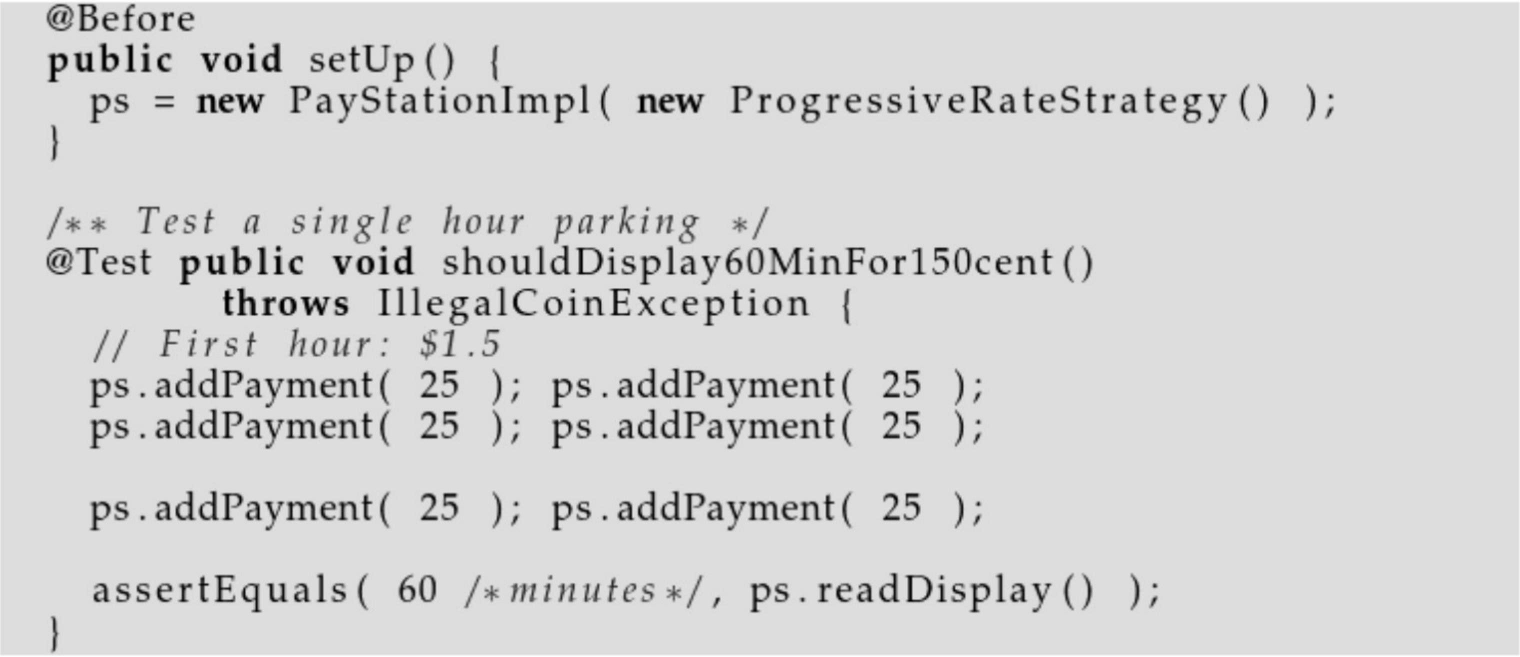
Iteration 2

BetaTown



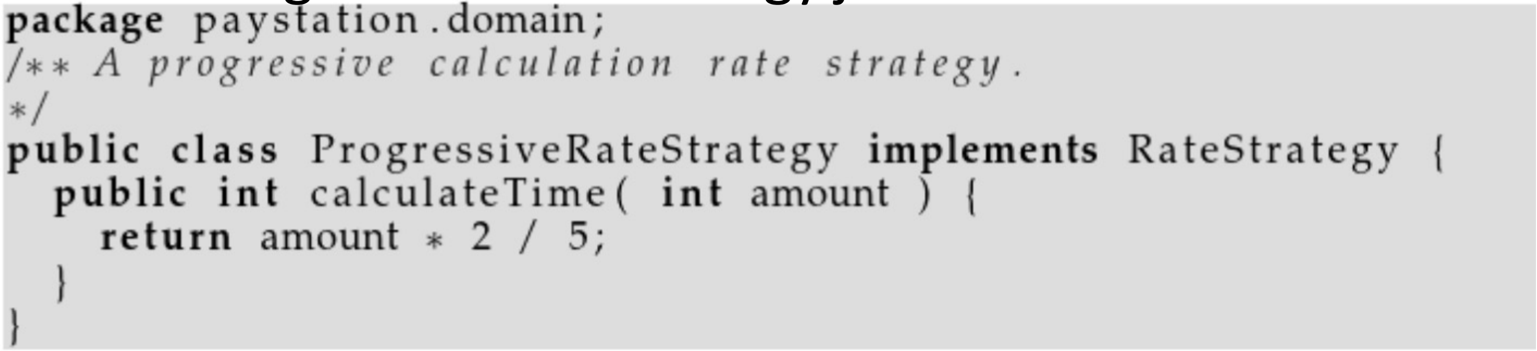
- ~~refactor Alphatown to use a compositional design~~
- First hour = $ 1.50
- Second hour = $ 1.50 + $ 2.0
- Third hour = $ 1.50 + $ 2.0 + $ 3.0
- Fourth hour = $ 1.50 + $ 2.0 + 2 * $ 3.0

New test file: TestProgressiveRate.java



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

/** Test a single hour parking */
@Test public void shouldDisplay60MinFor150cent()
        throws IllegalCoinException {
  // First hour: $1.5
  ps.addPayment( 25 ); ps.addPayment( 25 );
  ps.addPayment( 25 ); ps.addPayment( 25 );

  ps.addPayment( 25 ); ps.addPayment( 25 );

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

Create ProgressiveRateStrategy.java



```
package paystation.domain;
/** A progressive calculation rate strategy.
*/
public class ProgressiveRateStrategy implements RateStrategy {
  public int calculateTime( int amount ) {
    return amount * 2 / 5;
  }
}
```

Tests pass! But, it is good practice to make the test case fail on purpose

→ Make sure it is executed!

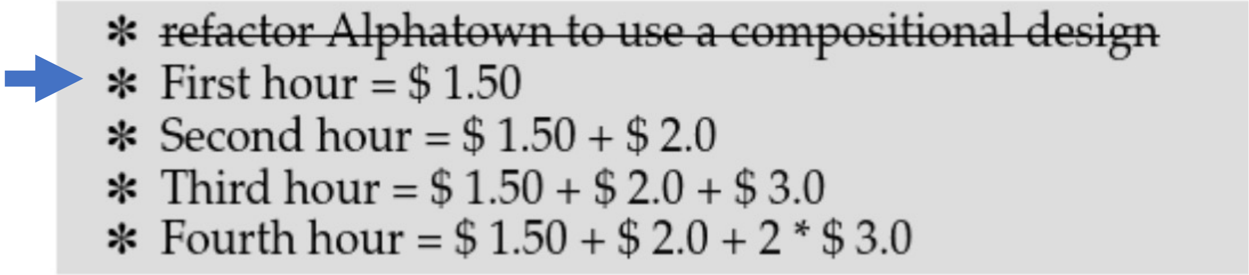
→ e.g., return 0

Weigh the costs of duplicated code vs. complexity

Same as AlphaTown! ("Obvious Implementation")

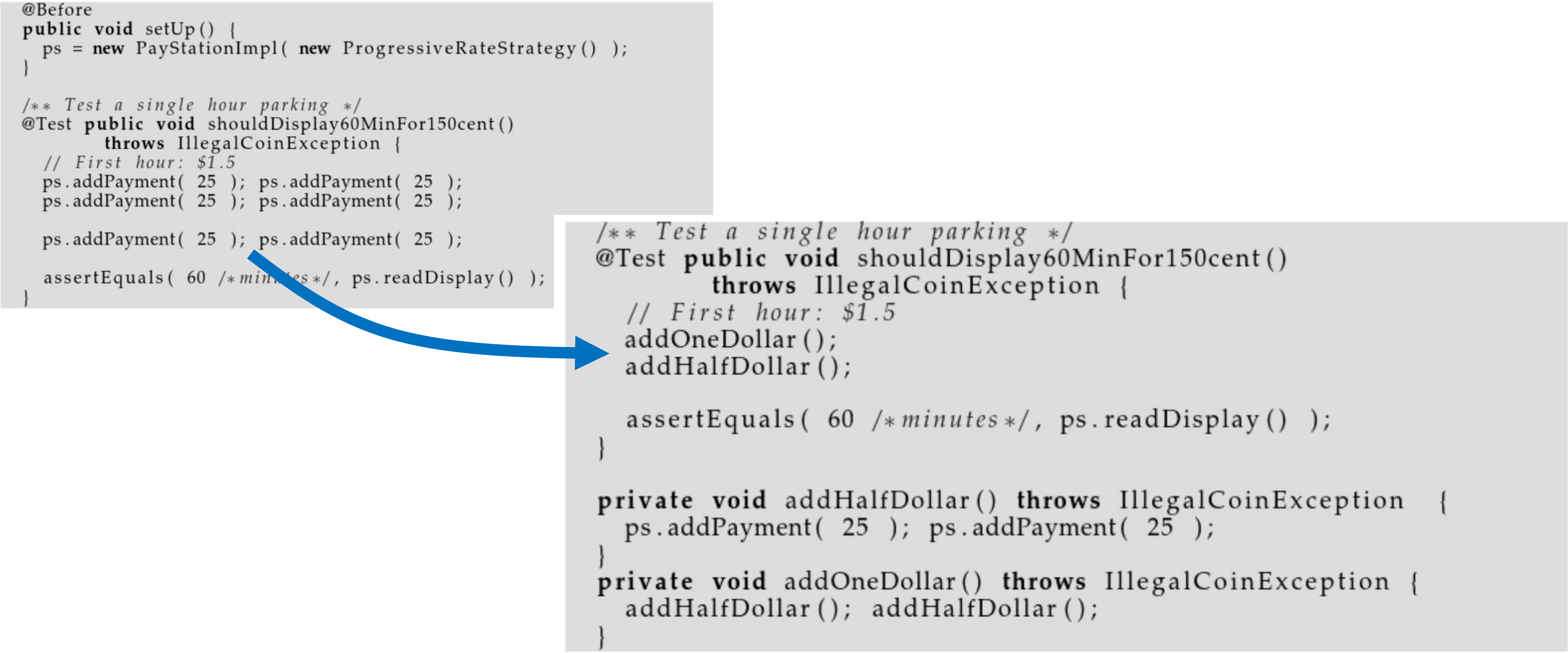
Iteration 2

BetaTown



- ~~refactor Alphatown to use a compositional design~~
- First hour = $ 1.50
- Second hour = $ 1.50 + $ 2.0
- Third hour = $ 1.50 + $ 2.0 + $ 3.0
- Fourth hour = $ 1.50 + $ 2.0 + 2 * $ 3.0

TestProgressiveRate.java



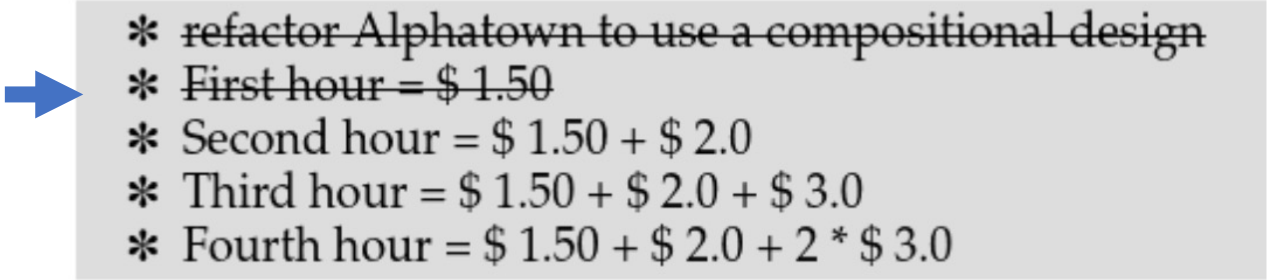
```
@Before
public void setUp() {
  ps = new PayStationImpl( new ProgressiveRateStrategy() );
}

/** Test a single hour parking */
@Test public void shouldDisplay60MinFor150cent()
        throws IllegalCoinException {
  // First hour: $1.5
  ps.addPayment( 25 ); ps.addPayment( 25 );
  ps.addPayment( 25 ); ps.addPayment( 25 );

  ps.addPayment( 25 ); ps.addPayment( 25 );

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

```
/** Test a single hour parking */
@Test public void shouldDisplay60MinFor150cent()
        throws IllegalCoinException {
  // First hour: $1.5
  addOneDollar();
  addHalfDollar();

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

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

Refactor to remove duplication in test code

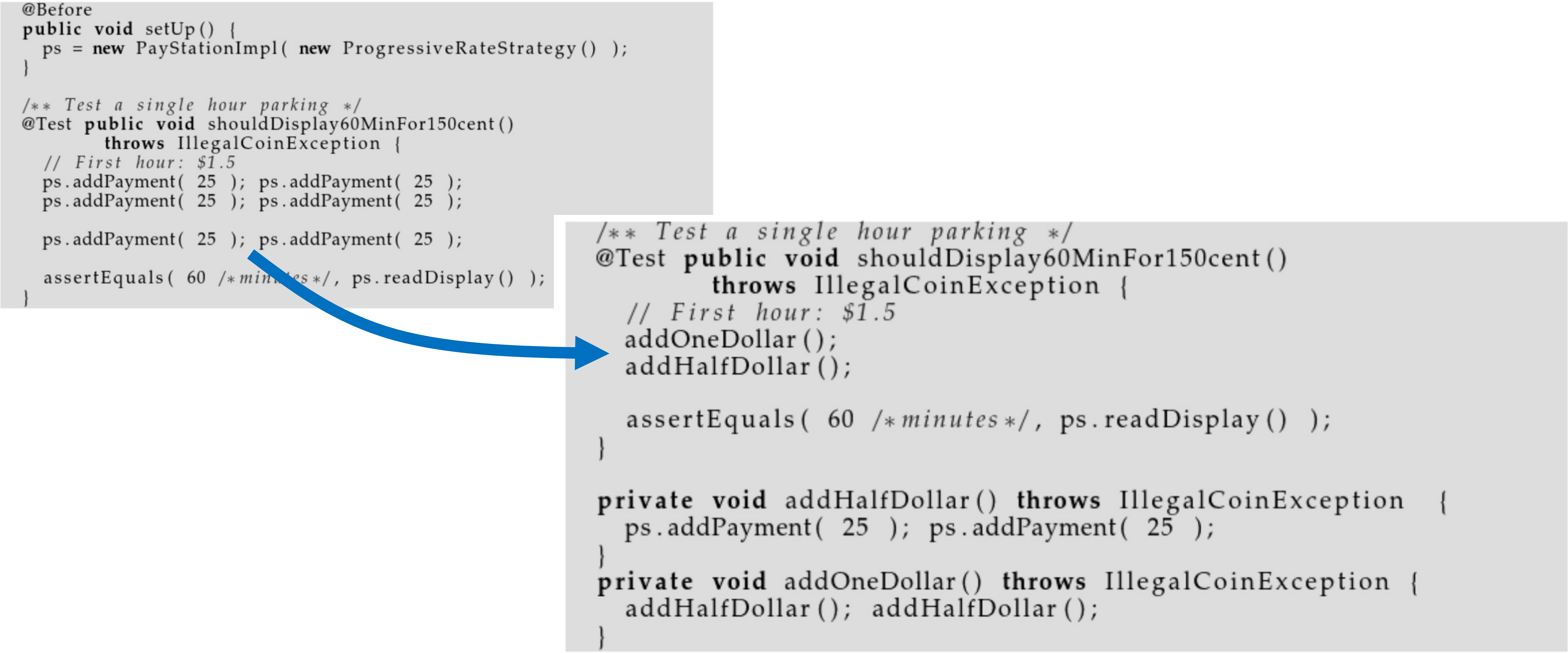
Iteration 2

BetaTown



- ~~refactor Alphatown to use a compositional design~~
- ~~First hour = $ 1.50~~
- Second hour = $ 1.50 + $ 2.0
- Third hour = $ 1.50 + $ 2.0 + $ 3.0
- Fourth hour = $ 1.50 + $ 2.0 + 2 * $ 3.0

TestProgressiveRate.java



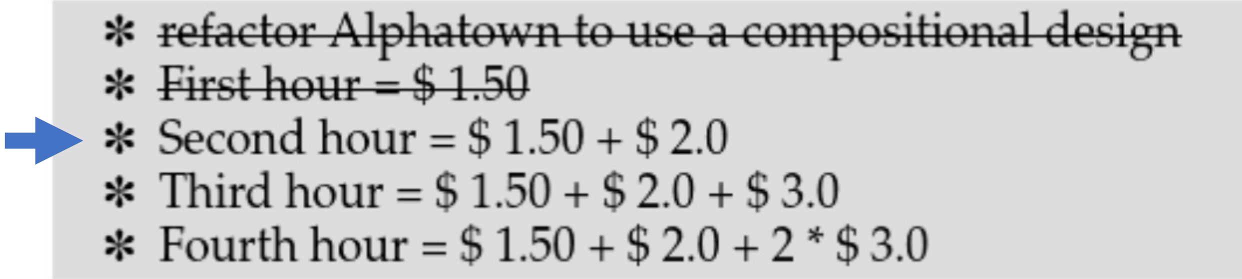
```java
@Before
public void setUp() {
  ps = new PayStationImpl( new ProgressiveRateStrategy() );
}

/** Test a single hour parking */
@Test public void shouldDisplay60MinFor150cent()
        throws IllegalCoinException {
  // First hour: $1.5
  ps.addPayment( 25 ); ps.addPayment( 25 );
  ps.addPayment( 25 ); ps.addPayment( 25 );

  ps.addPayment( 25 ); ps.addPayment( 25 );

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

```java
/** Test a single hour parking */
@Test public void shouldDisplay60MinFor150cent()
        throws IllegalCoinException {
  // First hour: $1.5
  addOneDollar();
  addHalfDollar();

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

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

Refactor to remove duplication in test code

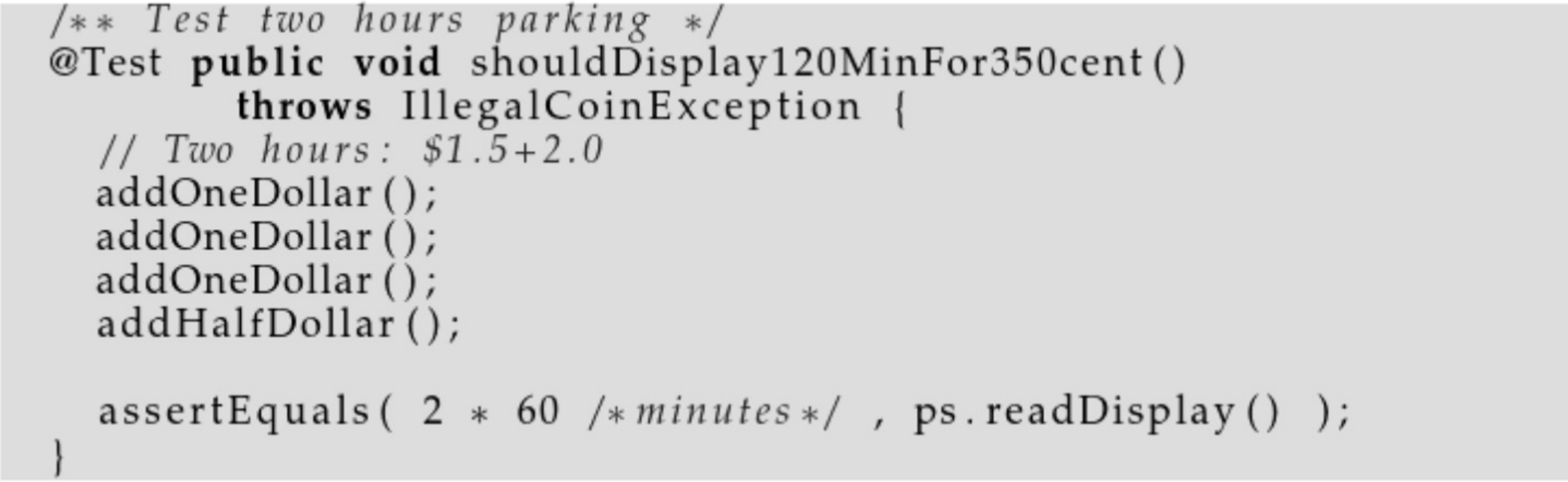
Iteration 3

BetaTown



- ~~refactor Alphatown to use a compositional design~~
- ~~First hour = $ 1.50~~
- Second hour = $ 1.50 + $ 2.0
- Third hour = $ 1.50 + $ 2.0 + $ 3.0
- Fourth hour = $ 1.50 + $ 2.0 + 2 * $ 3.0

TestProgressiveRate.java

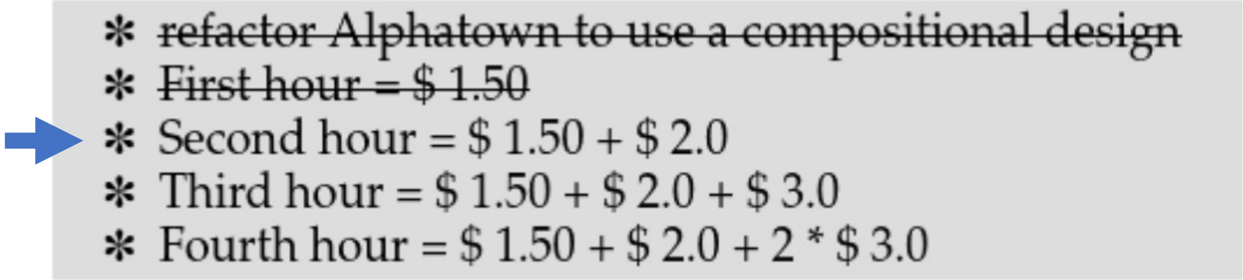


```
/** Test two hours parking */
@Test public void shouldDisplay120MinFor350cent()
        throws IllegalCoinException {
  // Two hours: $1.5+2.0
  addOneDollar();
  addOneDollar();
  addOneDollar();
  addHalfDollar();

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

→ Test fails

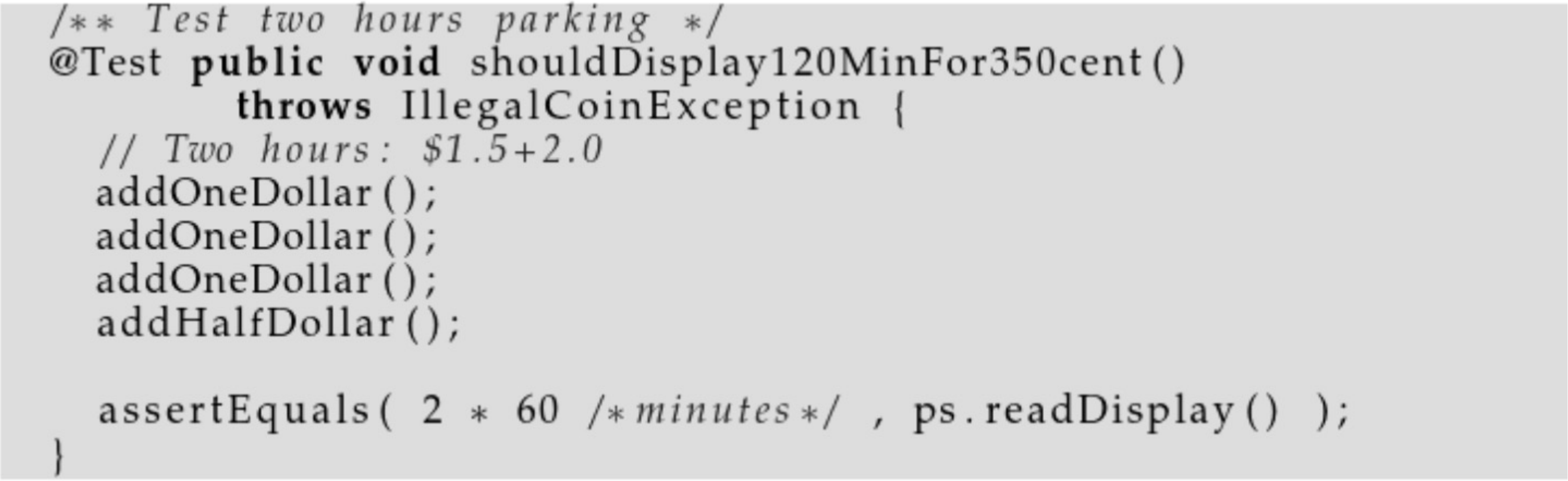
Iteration 3

BetaTown



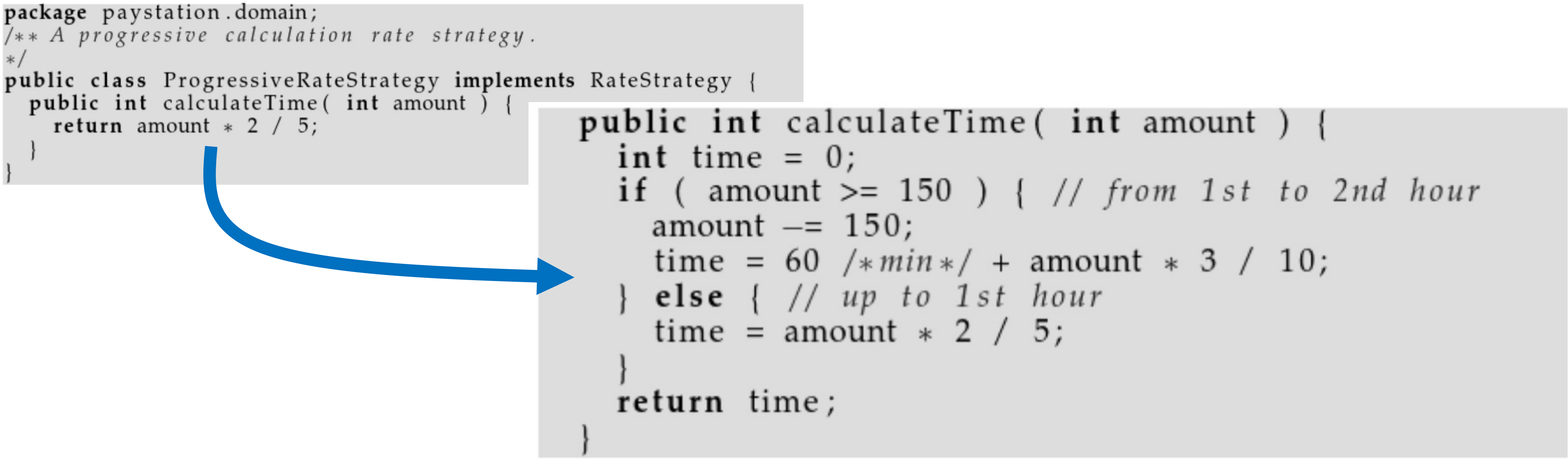
- ~~refactor Alphatown to use a compositional design~~
- ~~First hour = $ 1.50~~
- Second hour = $ 1.50 + $ 2.0
- Third hour = $ 1.50 + $ 2.0 + $ 3.0
- Fourth hour = $ 1.50 + $ 2.0 + 2 * $ 3.0

TestProgressiveRate.java



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

ProgressiveRateStrategy.java

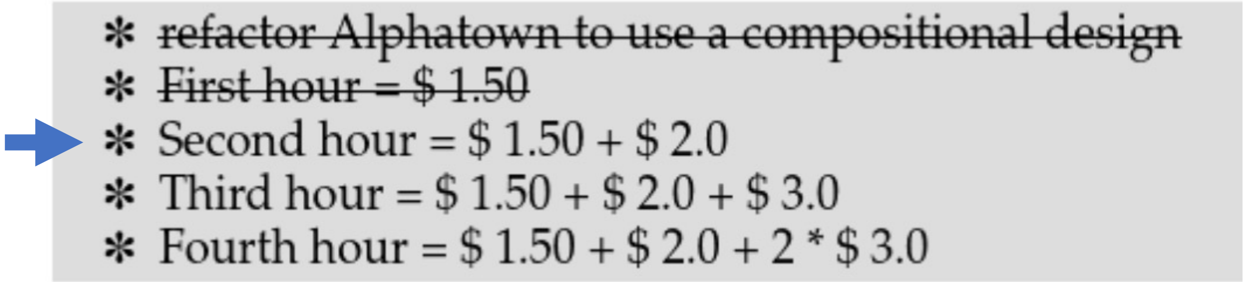


```
package paystation.domain;
/** A progressive calculation rate strategy.
*/
public class ProgressiveRateStrategy implements RateStrategy {
  public int calculateTime( int amount ) {
    return amount * 2 / 5;
  }
}
```

```
public int calculateTime( int amount ) {
  int time = 0;
  if ( amount >= 150 ) { // from 1st to 2nd hour
    amount -= 150;
    time = 60 /*min*/ + amount * 3 / 10;
  } else { // up to 1st hour
    time = amount * 2 / 5;
  }
  return time;
}
```

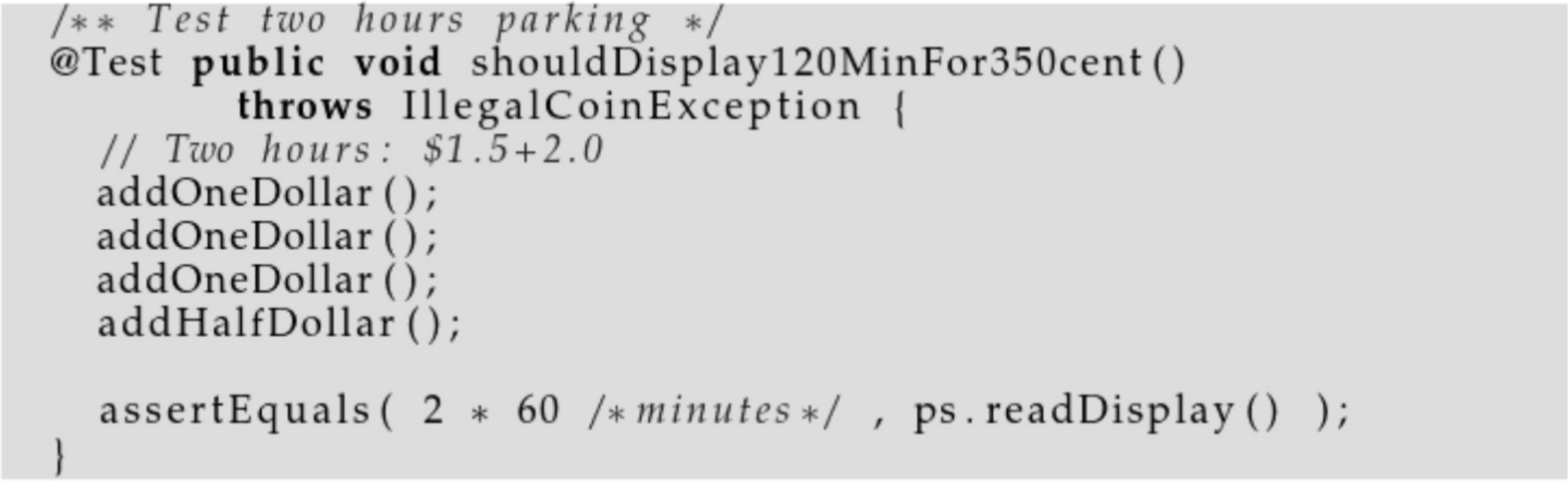
Iteration 3

BetaTown



- ✻ ~~refactor Alphatown to use a compositional design~~
- ✻ ~~First hour = $ 1.50~~
- ✻ Second hour = $ 1.50 + $ 2.0
- ✻ Third hour = $ 1.50 + $ 2.0 + $ 3.0
- ✻ Fourth hour = $ 1.50 + $ 2.0 + 2 * $ 3.0

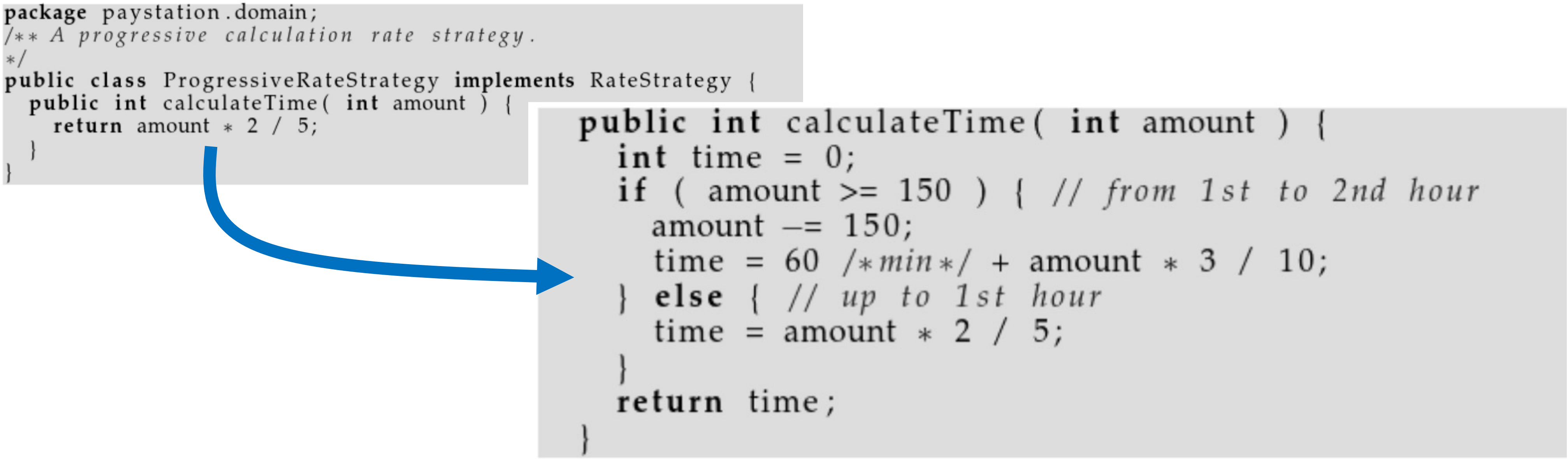
TestProgressiveRate.java



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

→ Test passes

ProgressiveRateStrategy.java

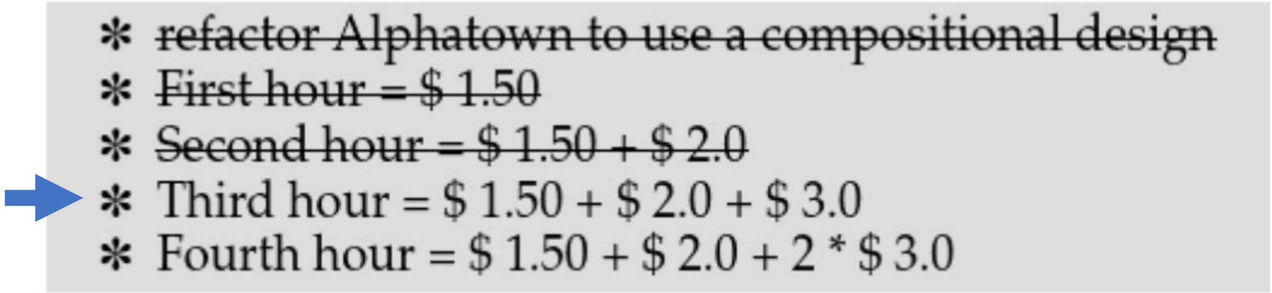


```
package paystation.domain;
/** A progressive calculation rate strategy.
*/
public class ProgressiveRateStrategy implements RateStrategy {
  public int calculateTime( int amount ) {
    return amount * 2 / 5;
  }
}
```

```
public int calculateTime( int amount ) {
  int time = 0;
  if ( amount >= 150 ) { // from 1st to 2nd hour
    amount -= 150;
    time = 60 /*min*/ + amount * 3 / 10;
  } else { // up to 1st hour
    time = amount * 2 / 5;
  }
  return time;
}
```

Iteration 4, etc.

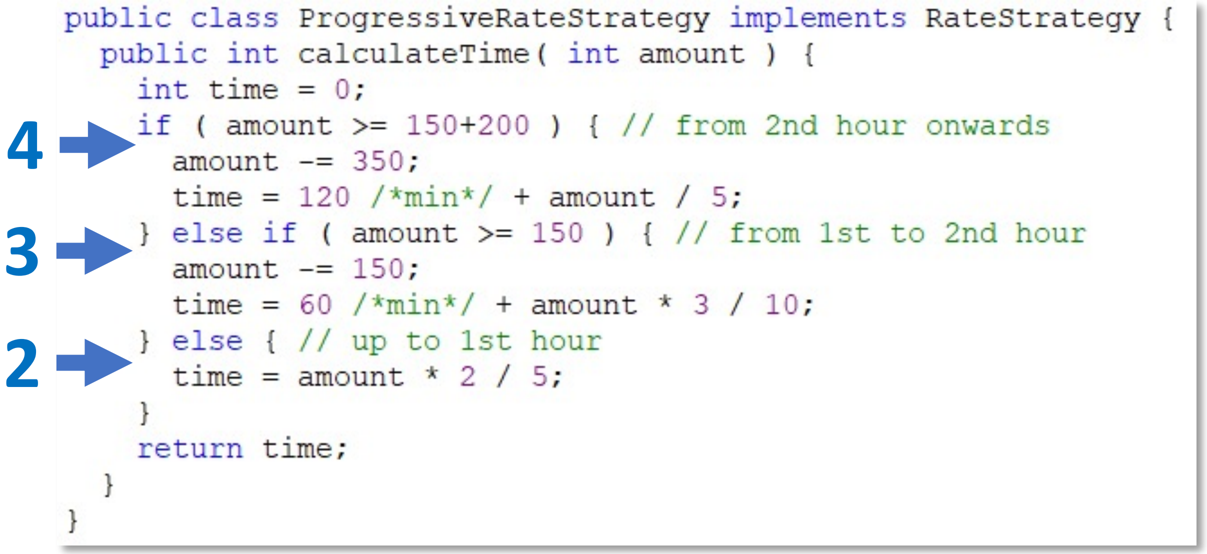
BetaTown



- ~~refactor Alphatown to use a compositional design~~
- ~~First hour = $ 1.50~~
- ~~Second hour = $ 1.50 + $ 2.0~~
- Third hour = $ 1.50 + $ 2.0 + $ 3.0
- Fourth hour = $ 1.50 + $ 2.0 + 2 * $ 3.0

And so on…

ProgressiveRateStrategy.java

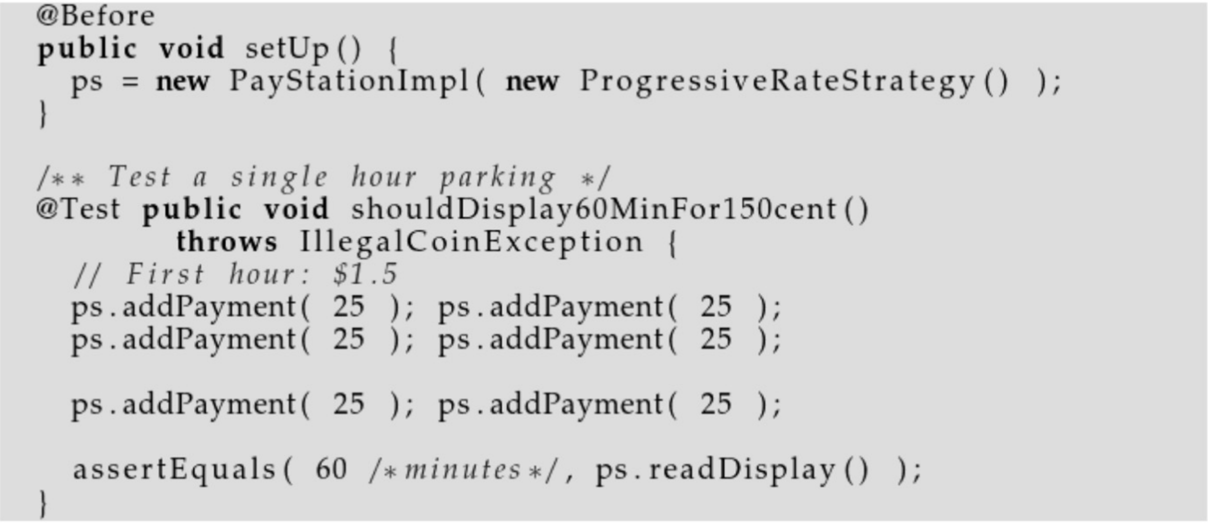


```java
public class ProgressiveRateStrategy implements RateStrategy {
  public int calculateTime( int amount ) {
    int time = 0;
    if ( amount >= 150+200 ) { // from 2nd hour onwards
      amount -= 350;
      time = 120 /*min*/ + amount / 5;
    } else if ( amount >= 150 ) { // from 1st to 2nd hour
      amount -= 150;
      time = 60 /*min*/ + amount * 3 / 10;
    } else { // up to 1st hour
      time = amount * 2 / 5;
    }
    return time;
  }
}
```

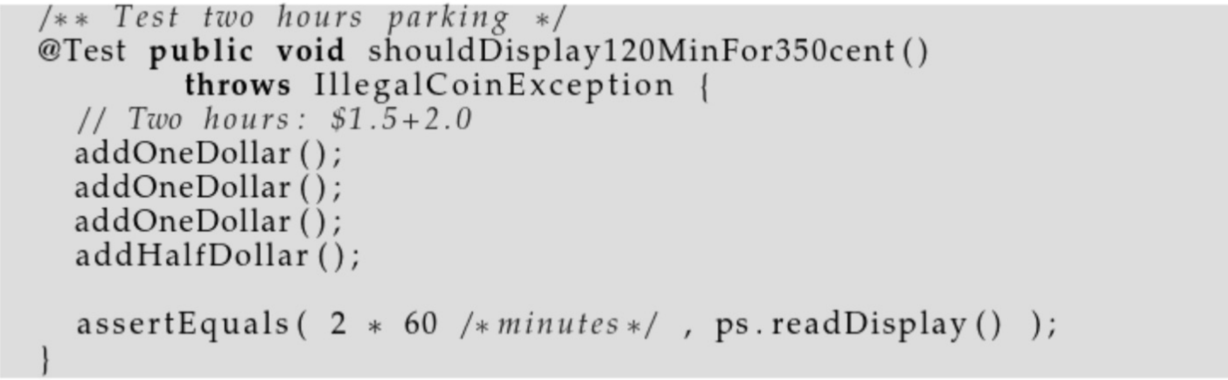
Unit and Integration Testing

We can actually test the new rate policy **independent of the Pay Station!**

TestProgressiveRate.java



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

/** Test a single hour parking */
@Test public void shouldDisplay60MinFor150cent()
        throws IllegalCoinException {
  // First hour: $1.5
  ps.addPayment( 25 ); ps.addPayment( 25 );
  ps.addPayment( 25 ); ps.addPayment( 25 );

  ps.addPayment( 25 ); ps.addPayment( 25 );

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

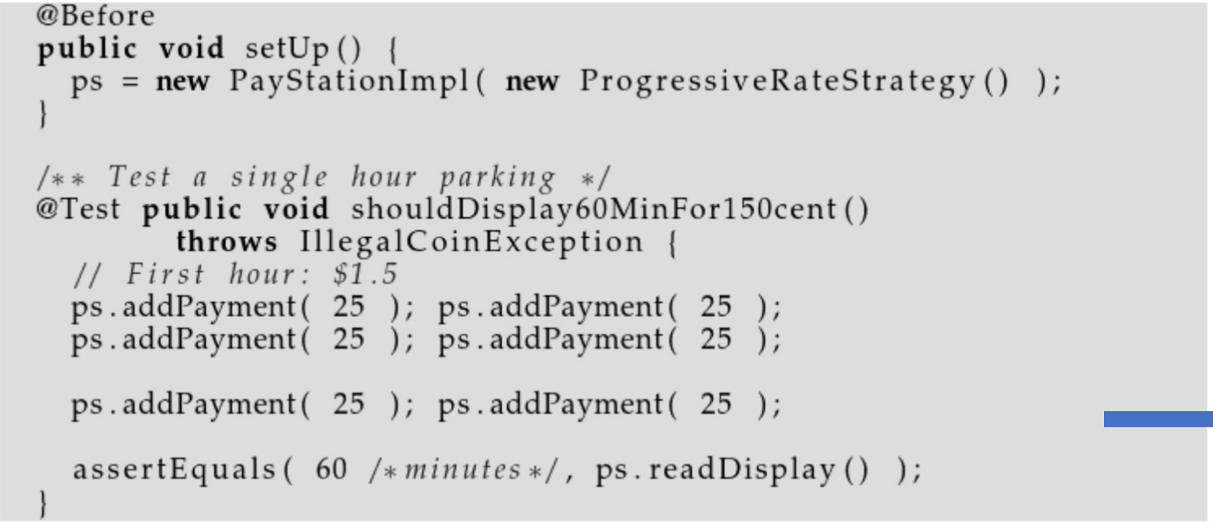


```
/** Test two hours parking */
@Test public void shouldDisplay120MinFor350cent()
        throws IllegalCoinException {
  // Two hours: $1.5+2.0
  addOneDollar();
  addOneDollar();
  addOneDollar();
  addHalfDollar();

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

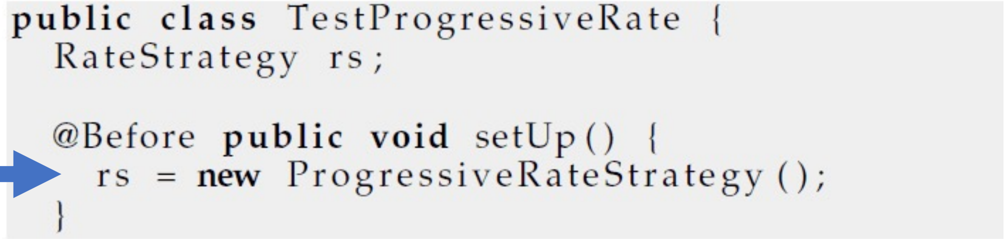
Unit and Integration Testing

We can actually test the new rate policy **independent of the Pay Station!**

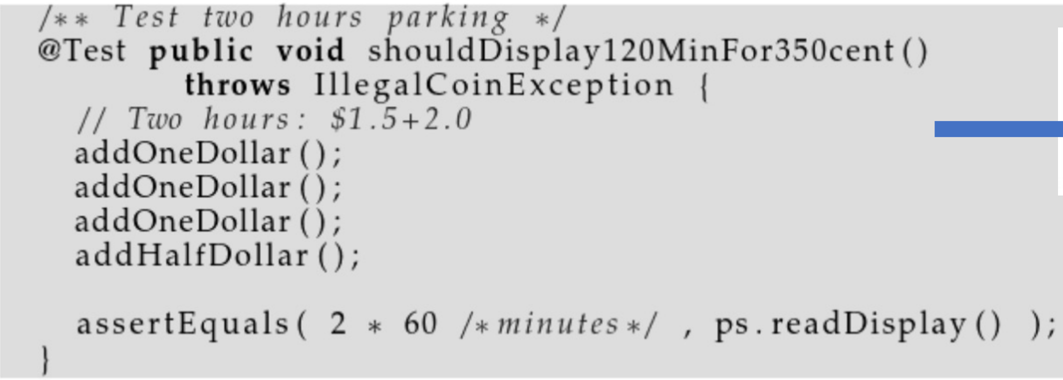
TestProgressiveRate.java



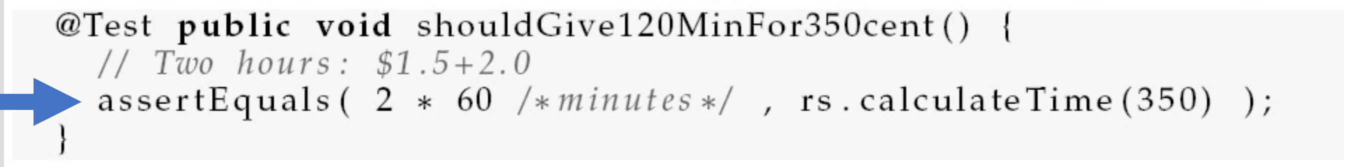
```
@Before
public void setUp() {
  ps = new PayStationImpl( new ProgressiveRateStrategy() );
}

/** Test a single hour parking */
@Test public void shouldDisplay60MinFor150cent()
        throws IllegalCoinException {
  // First hour: $1.5
  ps.addPayment( 25 ); ps.addPayment( 25 );
  ps.addPayment( 25 ); ps.addPayment( 25 );

  ps.addPayment( 25 ); ps.addPayment( 25 );

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



```
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() );
}
```

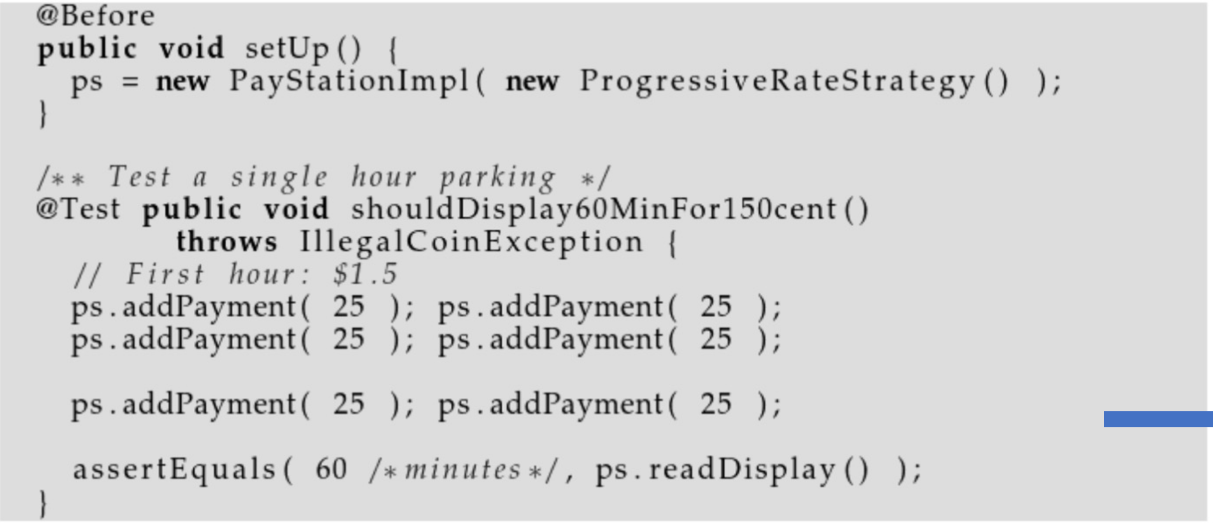


```
@Test public void shouldGive120MinFor350cent() {
  // Two hours: $1.5+2.0
  assertEquals( 2 * 60 /*minutes*/ , rs.calculateTime(350) );
}
```

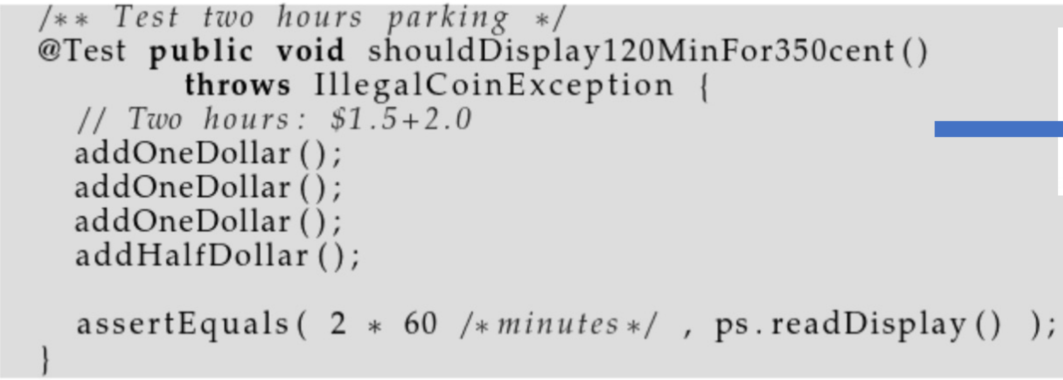
Unit and Integration Testing

We can actually test the new rate policy **independent of the Pay Station!**

TestProgressiveRate.java

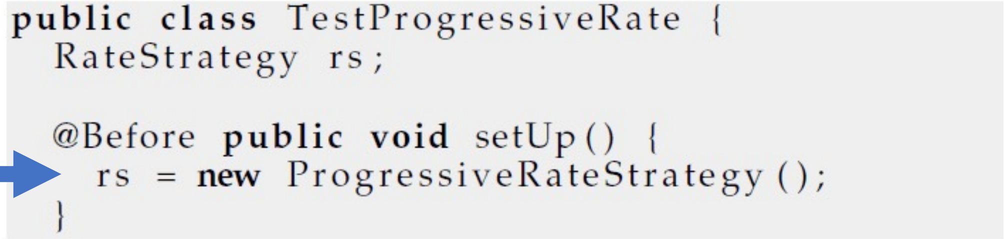


```
@Before
public void setUp() {
  ps = new PayStationImpl( new ProgressiveRateStrategy() );
}

/** Test a single hour parking */
@Test public void shouldDisplay60MinFor150cent()
        throws IllegalCoinException {
  // First hour: $1.5
  ps.addPayment( 25 ); ps.addPayment( 25 );
  ps.addPayment( 25 ); ps.addPayment( 25 );

  ps.addPayment( 25 ); ps.addPayment( 25 );

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

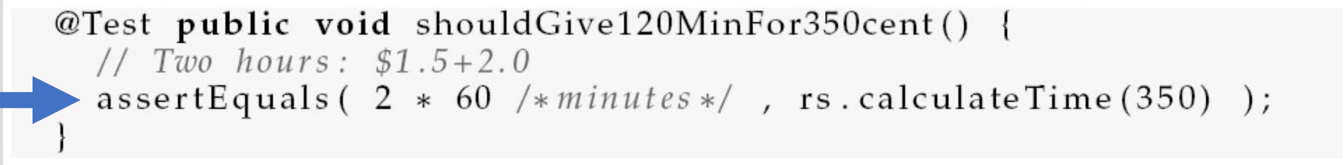


```
/** Test two hours parking */
@Test public void shouldDisplay120MinFor350cent()
        throws IllegalCoinException {
  // Two hours: $1.5+2.0
  addOneDollar();
  addOneDollar();
  addOneDollar();
  addHalfDollar();

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

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



```
public class TestProgressiveRate {
  RateStrategy rs;

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



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

Unit and Integration Testing

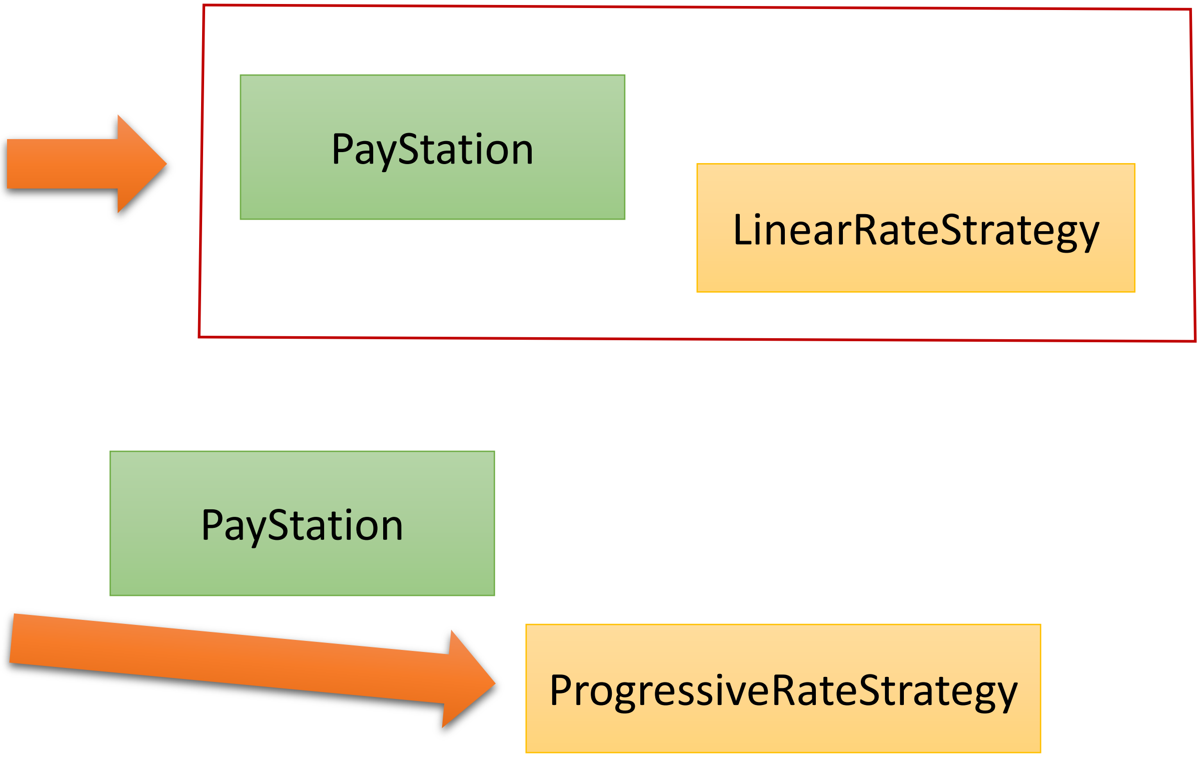
Do we test the BetaTown PayStation anywhere?

- TestPayStation tests use the LinearRateStrategy (AlphaTown)
- TestProgressiveRate **unit tests** ProgressiveRateStrategy

Unit and Integration Testing

Do we test the BetaTown PayStation anywhere?

- TestPayStation tests use the LinearRateStrategy (AlphaTown)
- TestProgressiveRate **unit tests** ProgressiveRateStrategy



Unit and Integration Testing

Testing the parts does not mean that the whole is tested (and vice versa)!

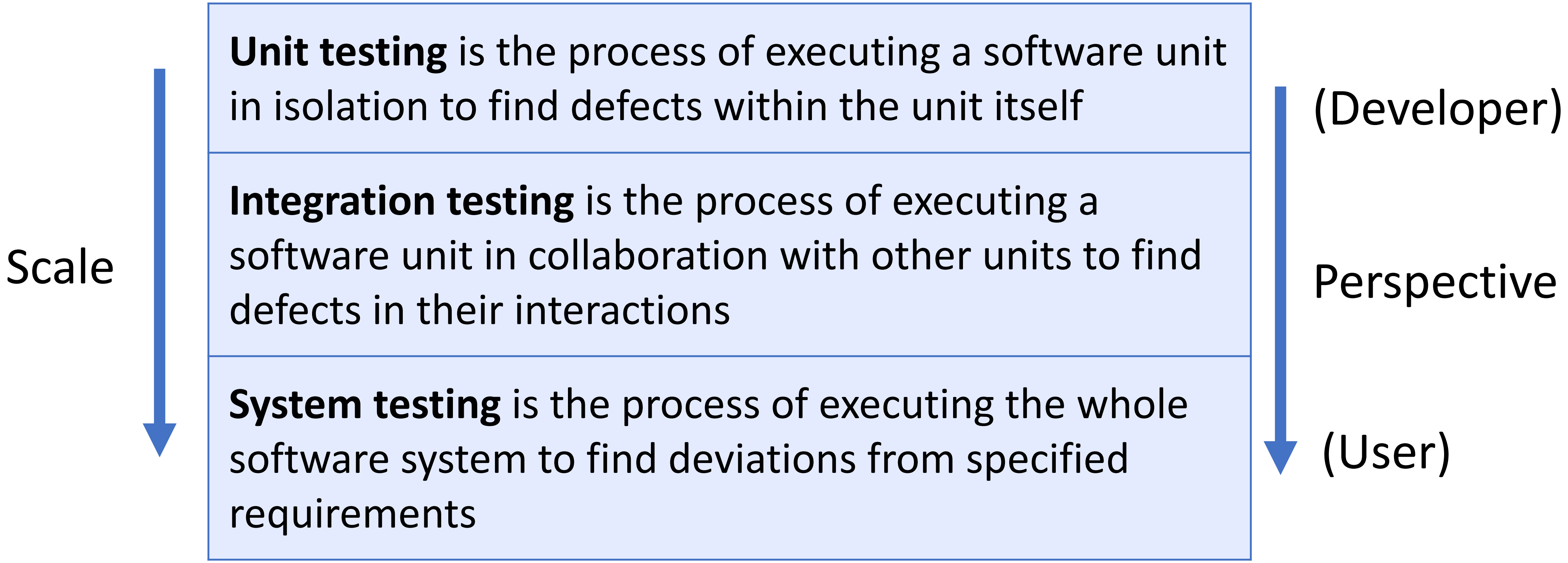
Unit testing is the process of executing a software unit in isolation to find defects within the unit itself

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

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

Unit and Integration Testing

Testing the parts does not mean that the whole is tested (and vice versa)!



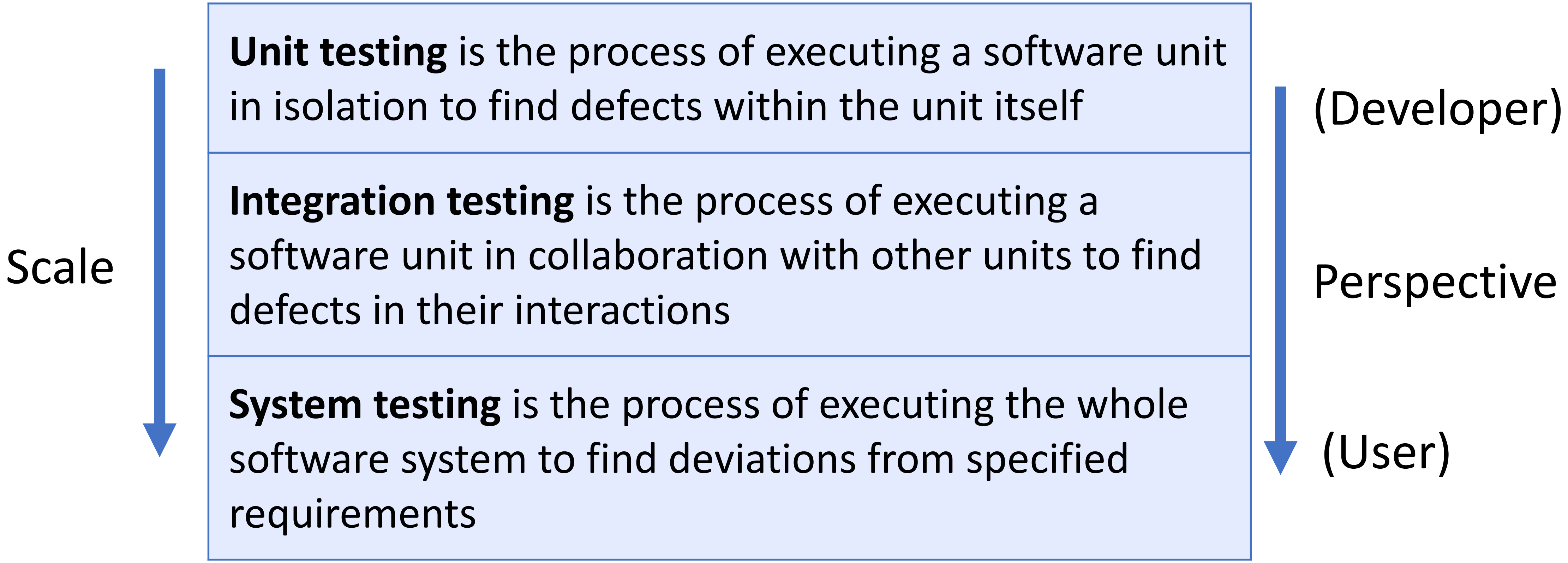
Unit testing is the process of executing a software unit in isolation to find defects within the unit itself

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

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

Unit and Integration Testing

Testing the parts does not mean that the whole is tested (and vice versa)!



Scale

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

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

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

(Developer)

Perspective

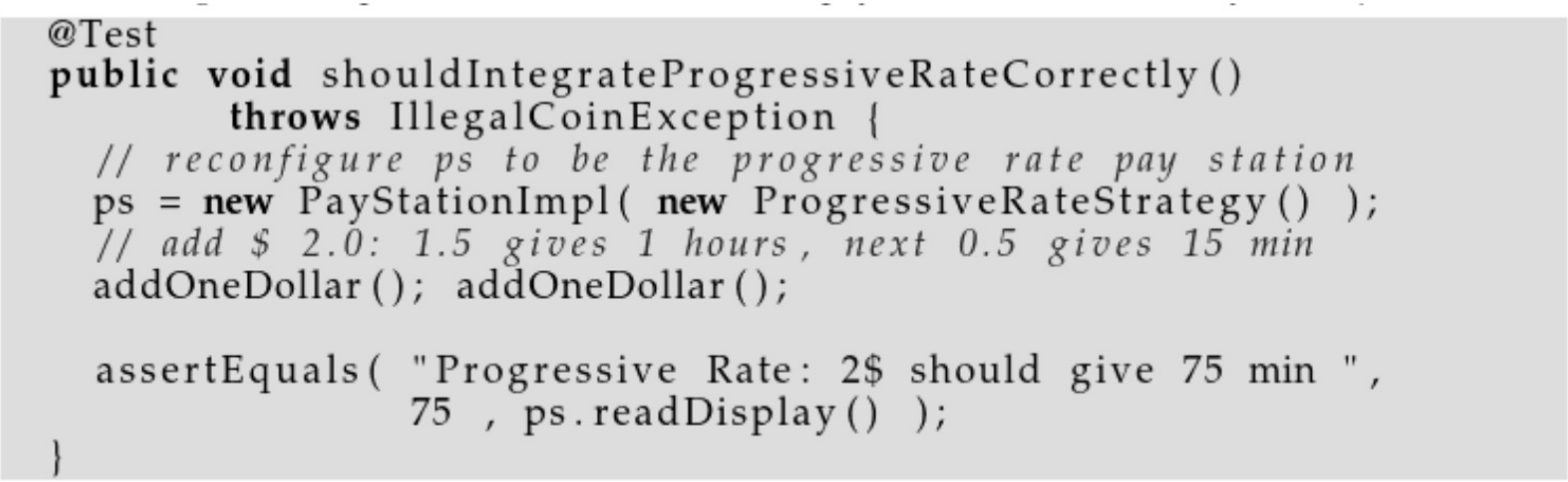
(User)

Defects can be caused by **interactions** between units with the wrong configuration!

Unit and Integration Testing

Progressive rate **integration testing** (BetaTown):

TestPayStation.java

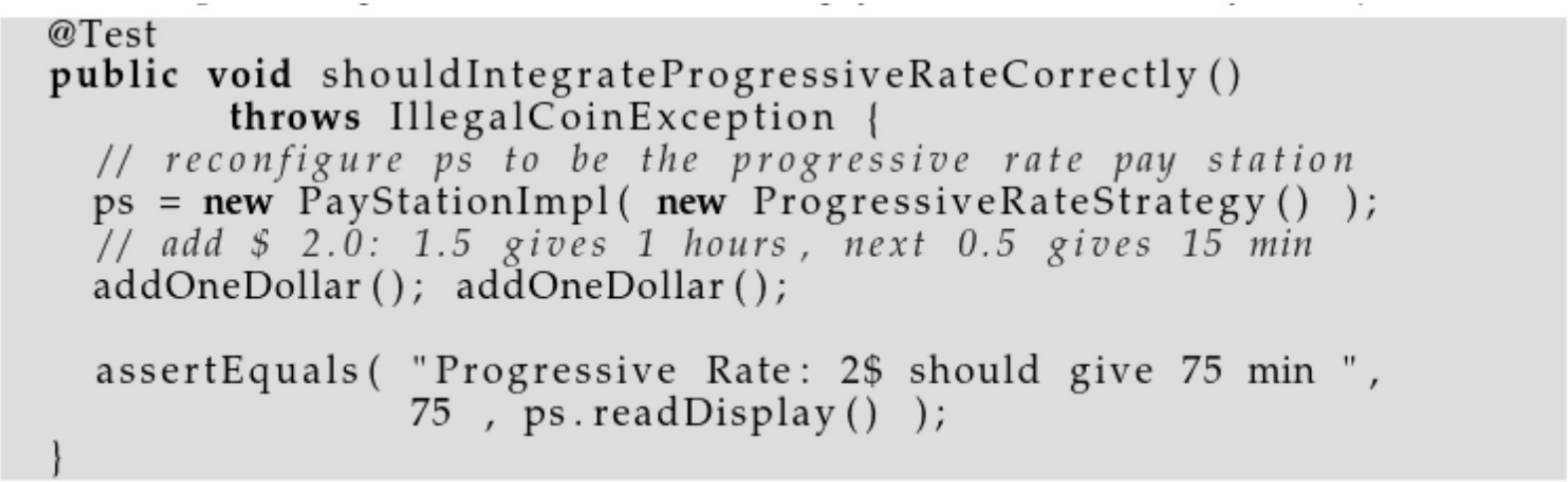


```
@Test
public void shouldIntegrateProgressiveRateCorrectly()
        throws IllegalCoinException {
  // reconfigure ps to be the progressive rate pay station
  ps = new PayStationImpl( new ProgressiveRateStrategy() );
  // add $ 2.0: 1.5 gives 1 hours, next 0.5 gives 15 min
  addOneDollar(); addOneDollar();

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

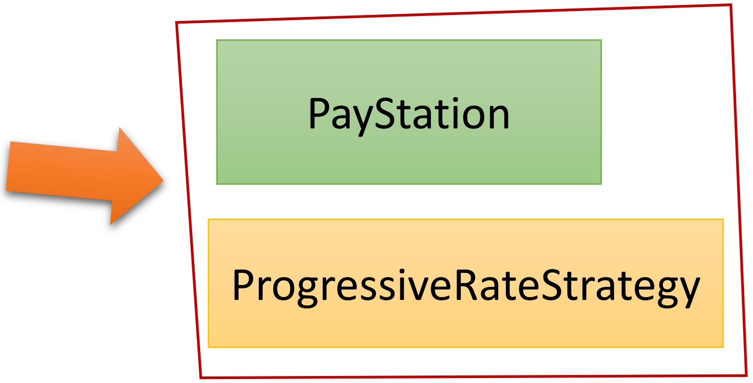
Unit and Integration Testing

Progressive rate **integration testing** (BetaTown):

TestPayStation.java



```
@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() );
}
```



Unit and Integration Testing

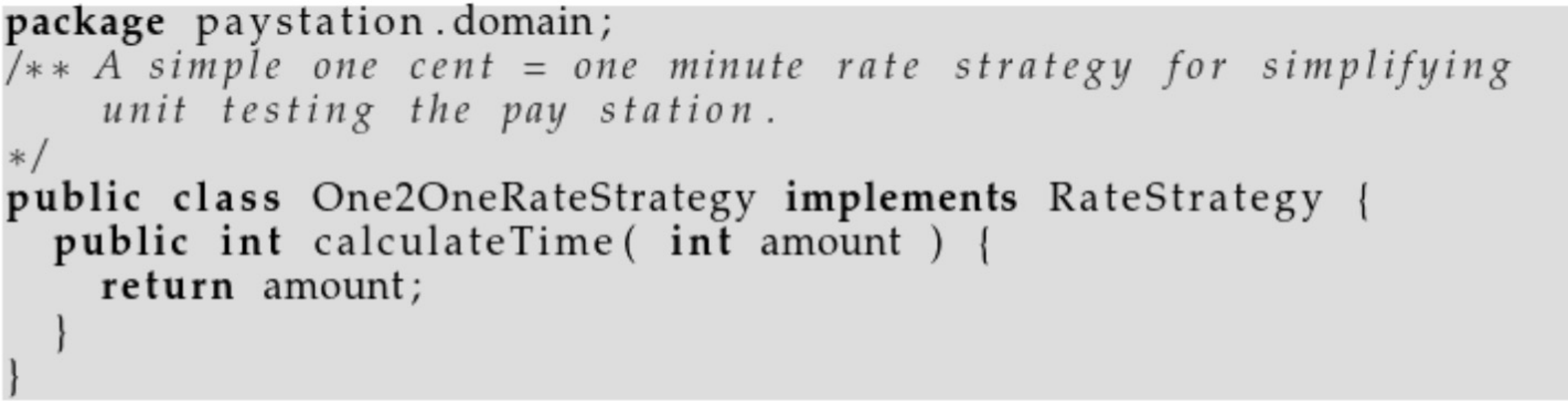
How should we unit test the Pay Station? Which rate strategy should we use?

Unit and Integration Testing

How should we unit test the Pay Station? Which rate strategy should we use?

→ Introduce a very simple rate strategy for unit testing the pay station

One2OneRateStrategy.java



```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;
  }
}
```

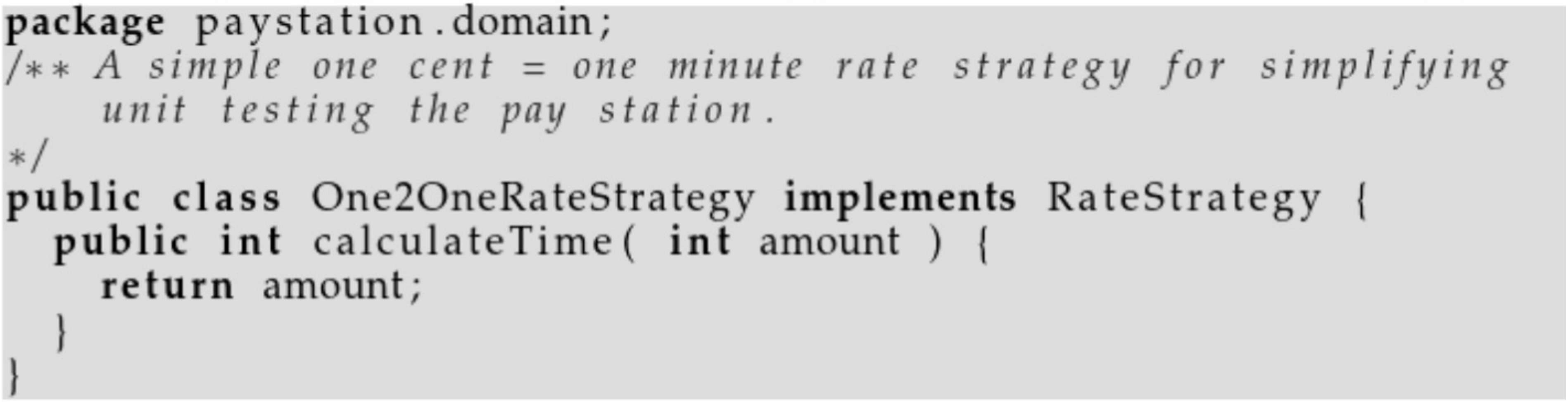
Unit and Integration Testing

How should we unit test the Pay Station? Which rate strategy should we use?

→ Introduce a very simple rate strategy for unit testing the pay station

Only in test code

src/**test**/java/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;
  }
}
```

Keep all testing related code in the test tree!

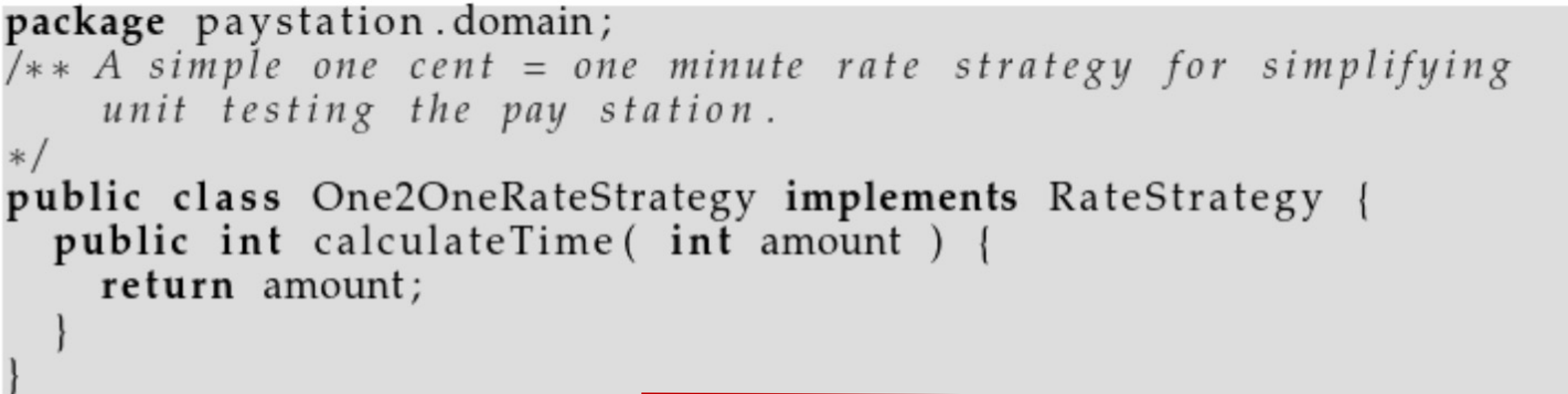
→ If a class is not used in the production code

Unit and Integration Testing

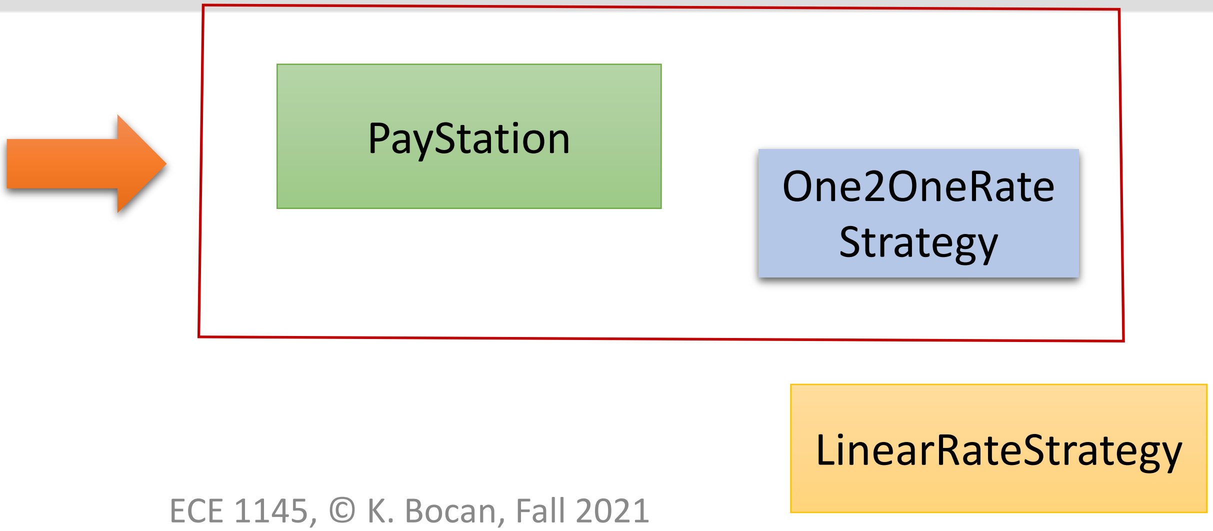
How should we unit test the Pay Station? Which rate strategy should we use?

→ Introduce a very simple rate strategy for unit testing the pay station

src/**test**/java/paystation/domain/One2OneRateStrategy.java



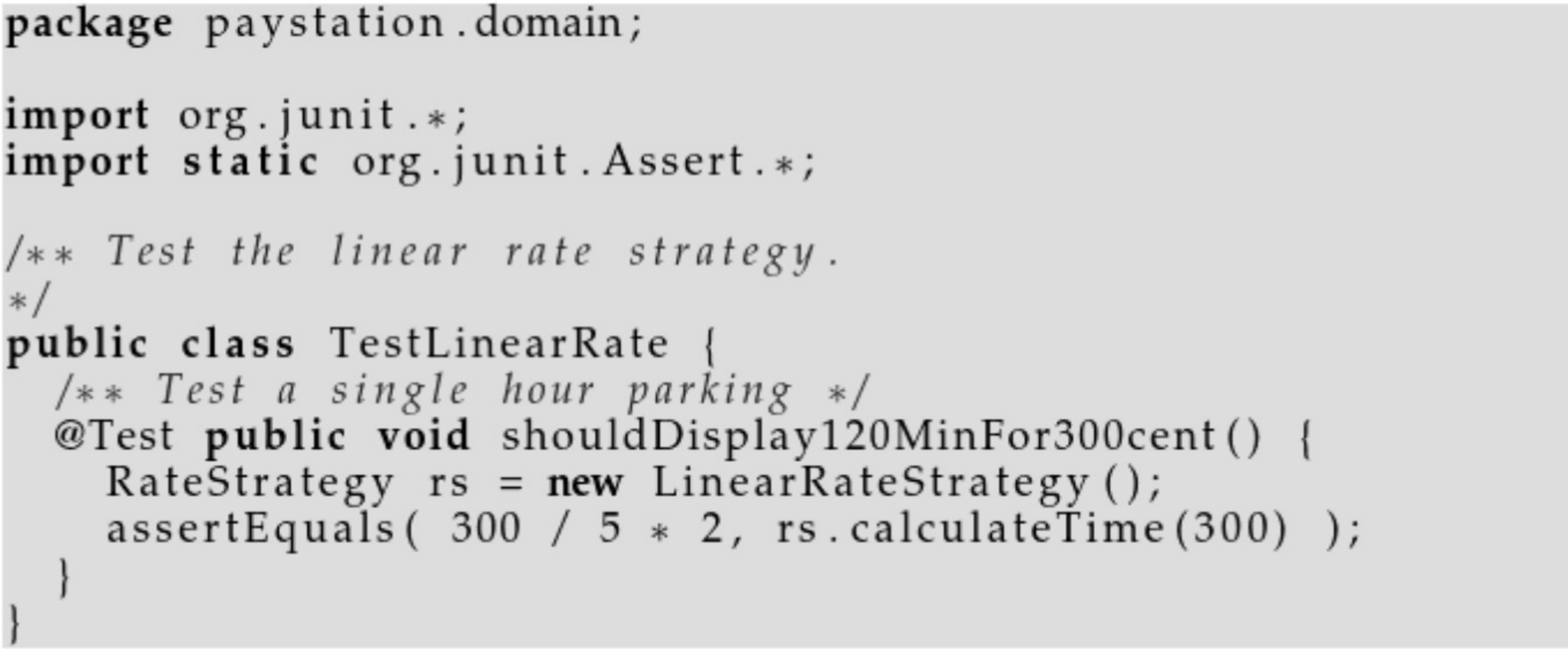
```
package paystation.domain;
/** A simple one cent = one minute rate strategy for simplifying
    unit testing the pay station.
*/
public class One2OneRateStrategy implements RateStrategy {
  public int calculateTime( int amount ) {
    return amount;
  }
}
```



Unit and Integration Testing

Unit test the linear rate policy (since we removed it from the pay station tests)

TestLinearRate.java

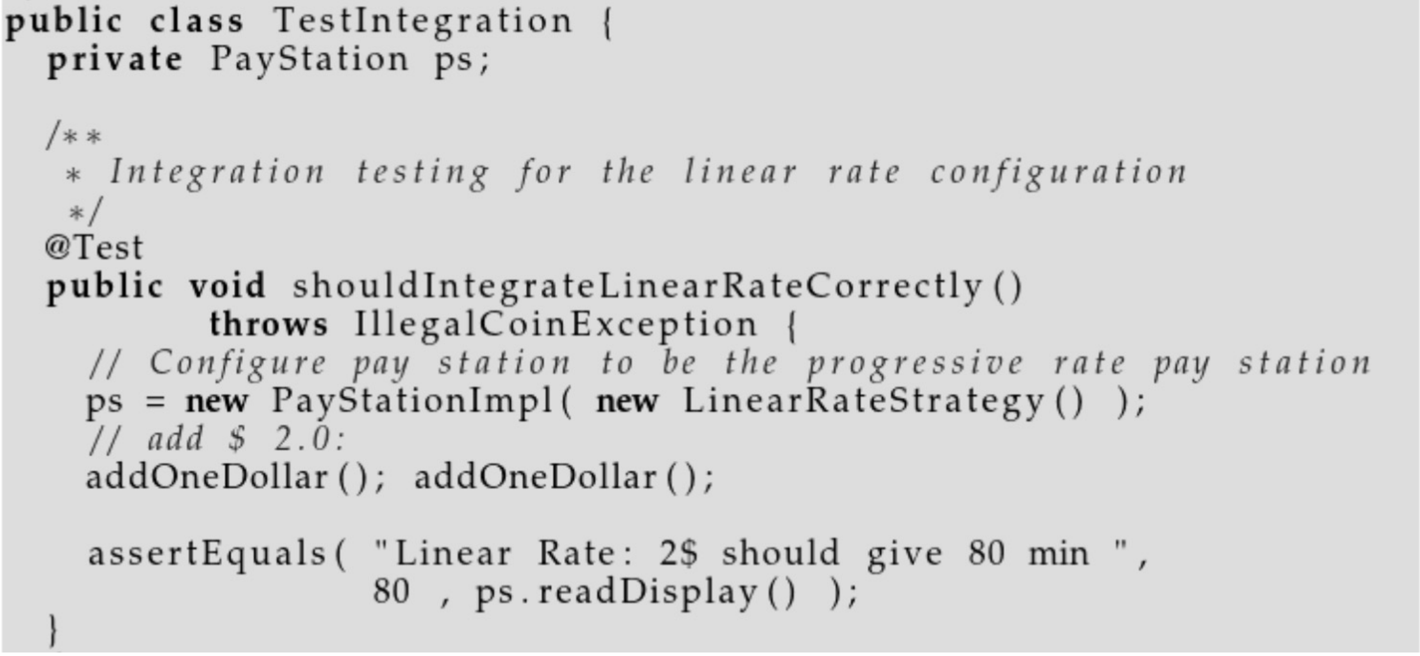


```
package paystation.domain;

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

/** Test the linear rate strategy.
*/
public class TestLinearRate {
  /** Test a single hour parking */
  @Test public void shouldDisplay120MinFor300cent() {
    RateStrategy rs = new LinearRateStrategy();
    assertEquals( 300 / 5 * 2, rs.calculateTime(300) );
  }
}
```

Unit and Integration Testing

AlphaTown (linear rate) integration testing:

TestIntegration.java

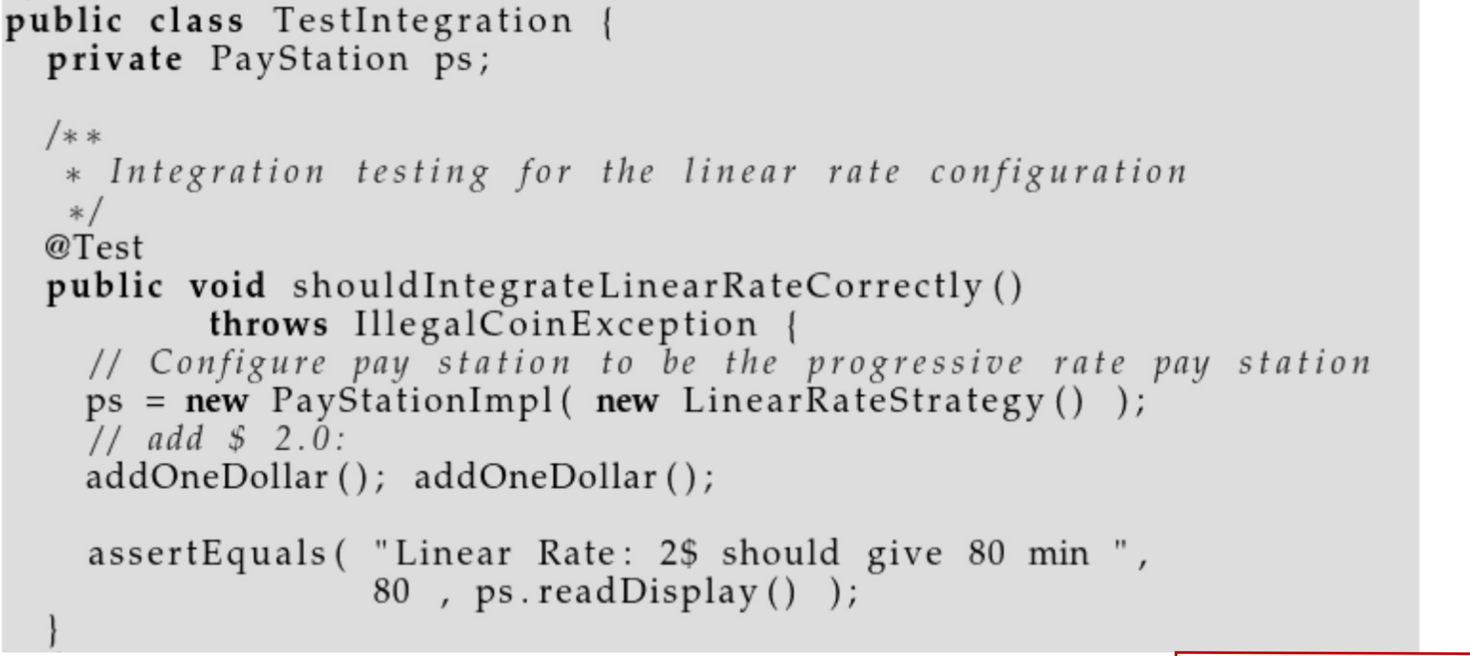


```
public class TestIntegration {
  private PayStation ps;

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

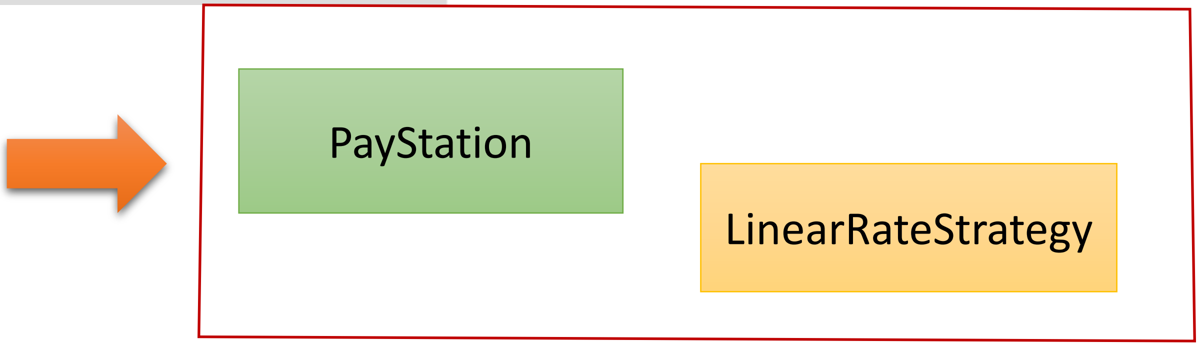
Unit and Integration Testing

AlphaTown (linear rate) integration testing:

TestIntegration.java



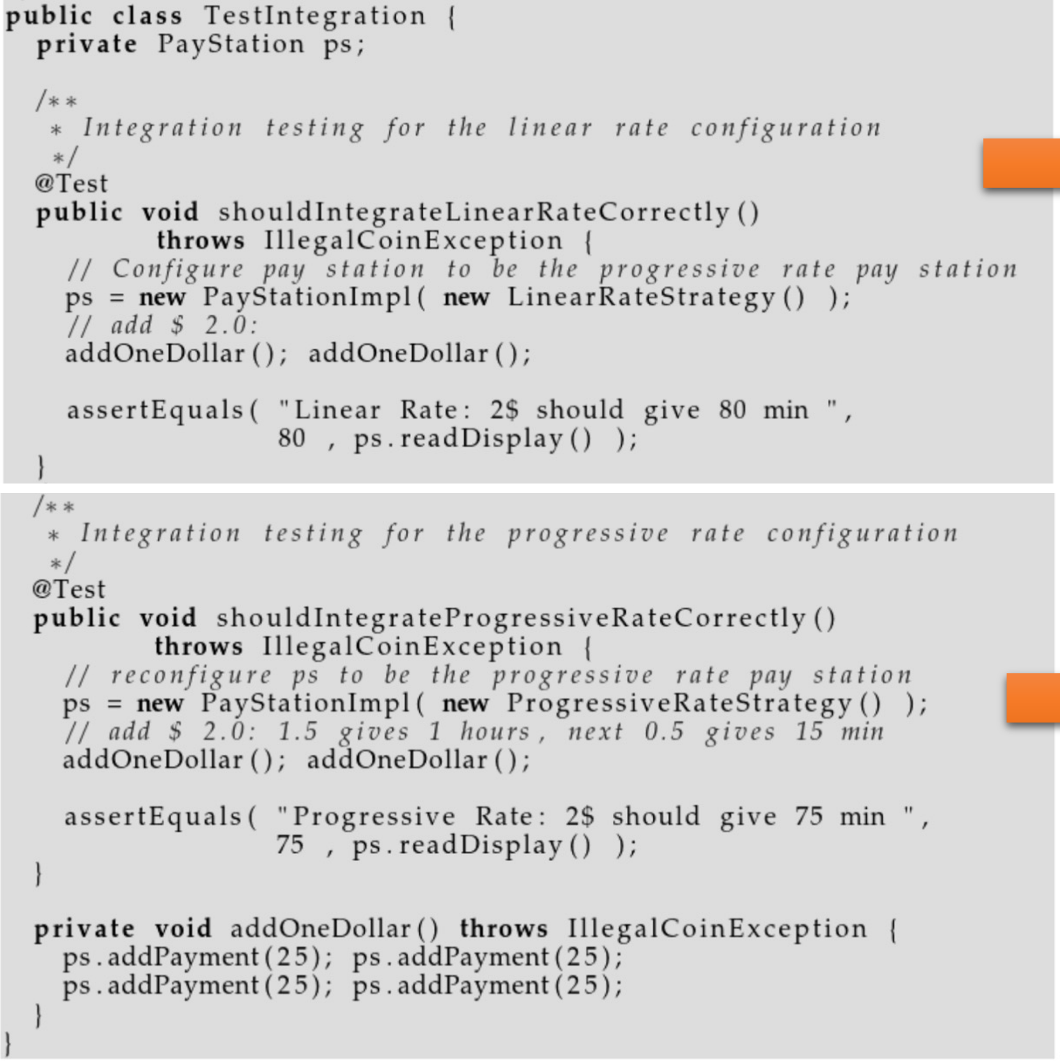
```
public class TestIntegration {
  private PayStation ps;

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



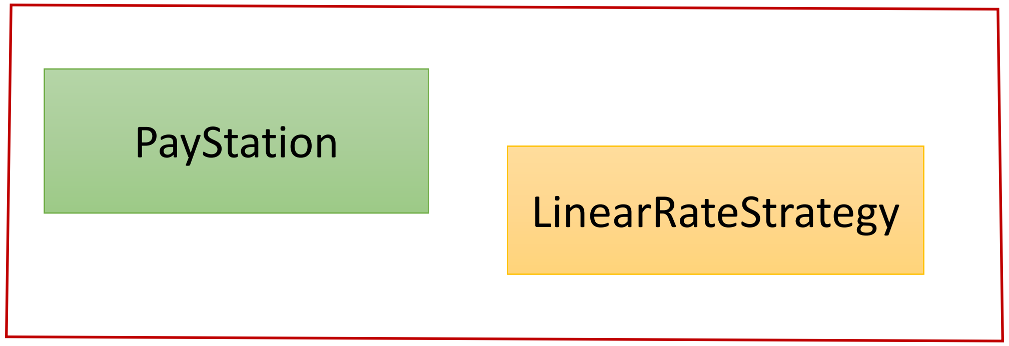
Unit and Integration Testing

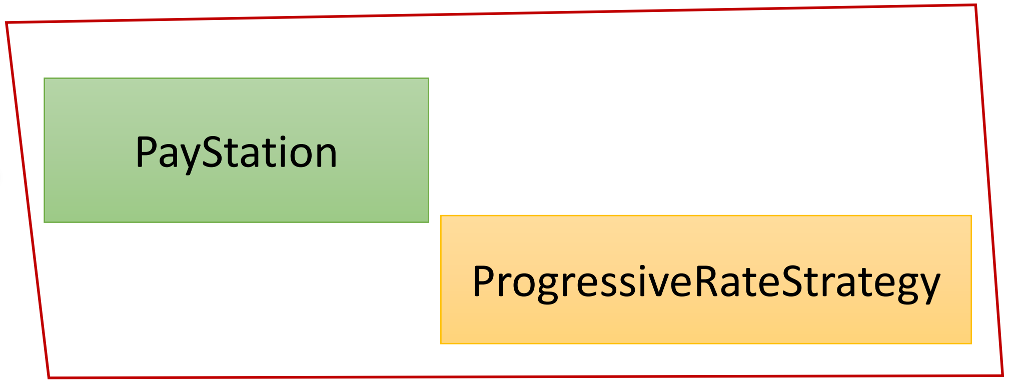
Consolidate integration testing code:

TestIntegration.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);
  }
}
```



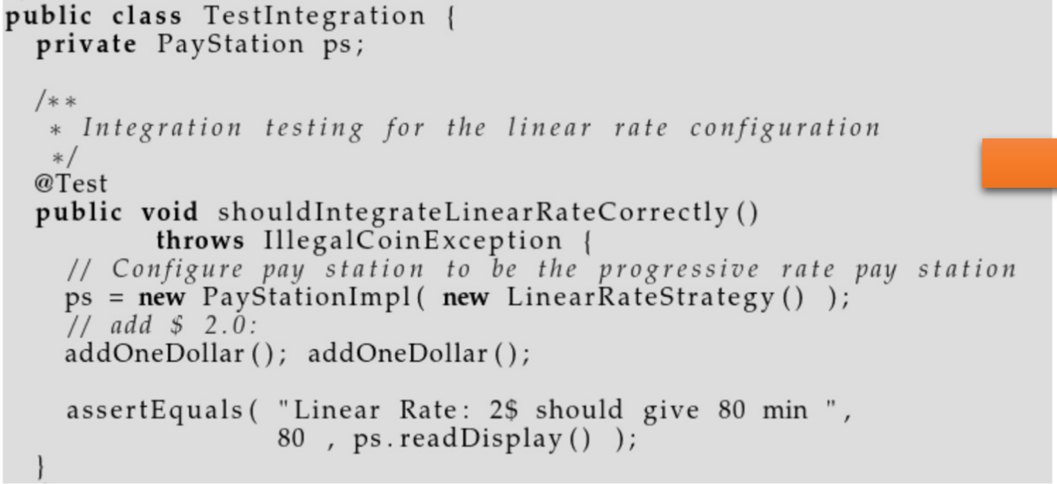


Unit and Integration Testing

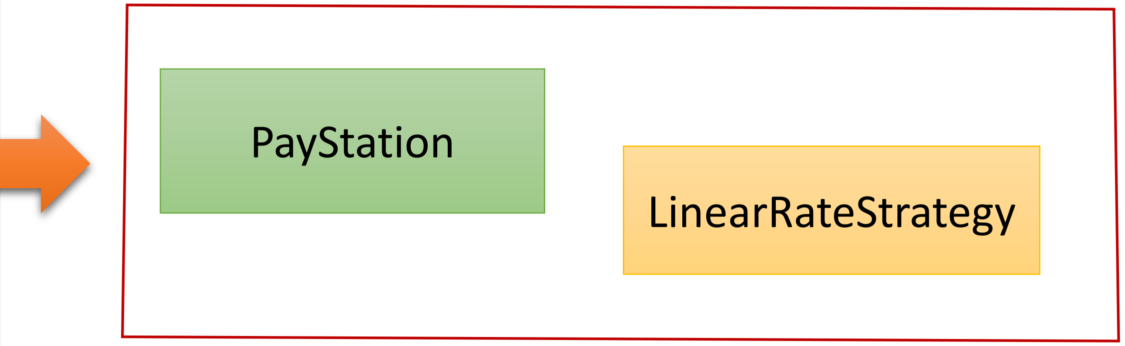
Consolidate integration testing code:

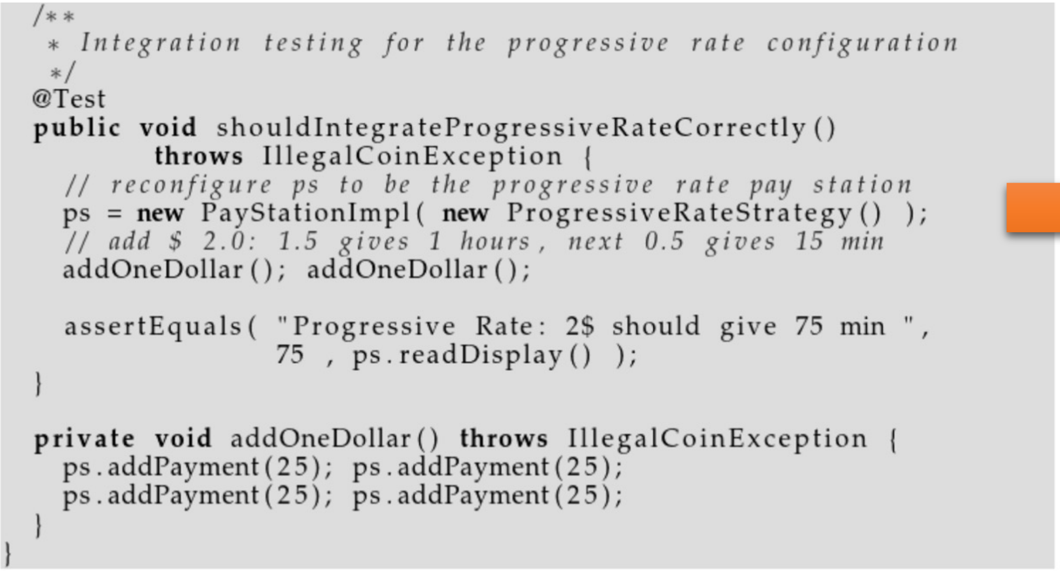
Note: If not using Gradle's convention-based build management, update the build script to include new test files!

TestIntegration.java

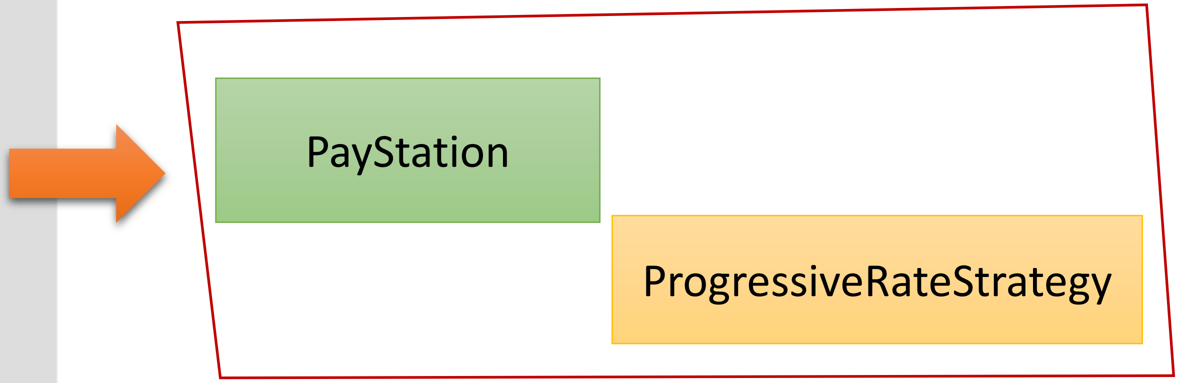


```
public class TestIntegration {
  private PayStation ps;

  /**
   * 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);
  }
}
```



Unit and Integration Testing

Summary:

1. Refactor to introduce rate strategy, ensure all existing test cases pass after refactoring
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. Notice that the rate strategies can be tested as separate units, refactor the test cases for progressive rate to become a unit test (improve analyzability)
6. Notice that analyzability of the pay station test code can be improved by introducing a simple rate strategy, just for testing the pay station. Refactor test cases and introduce integration testing of the pay station with each rate strategy

Was it worth it?

→ Changes risk introducing defects

→ Need to judge whether changes are worth the risk
(Version control is **very useful** here! Make changes on a branch)

Integration vs. System Testing

Integration testing is not system testing!

System testing is testing the **full system** (“Functional Testing” in XP):
Test that A works with **real** B, **real** C, **real** D, and **real** E units
(e.g., databases, servers, hardware, etc.)

We typically integration test with “stubs” representing real units – more later.

→ Use cases drive system testing (collaborate with customer)

Integration vs. System Testing

Integration testing is not system testing!

System testing is testing the **full system** (“Functional Testing” in XP):
Test that A works with **real** B, **real** C, **real** D, and **real** E units
(e.g., databases, servers, hardware, etc.)

We typically integration test with “stubs” representing real units – more later.

→ Use cases drive system testing (collaborate with customer)

While unit/integration tests must pass, **system tests** may not pass at 100% until project completion.

Summary

Automatic tests help us refactor without (much) fear!

When needs arise:

1. Use the old tests to **refactor** the architecture **without** adding new or changing existing behavior
2. Only after all tests pass, introduce new/modified behavior
3. Review again for any obsolete code or other things that need refactoring

Unit test the parts, integration test for interactions

Next time: Coding standards