Lecture 12

ECE 1145: Software Construction and Evolution

Abstract Factory Pattern

Pattern Fragility

(CH 13, 14)

Announcements

- Relevant Exercises: 13.6
- **Midterm Oct. 18 (take-home) – one week from today**
  - Open book, open notes, work individually
  - Access and submit via Canvas
  - 24 hour window
  - Lectures 1 – 9, project iterations 1 – 3 and code review
  - Midterm review on Wednesday Oct. 13
- Iteration 4 due **Oct. 24** - code quality improvements

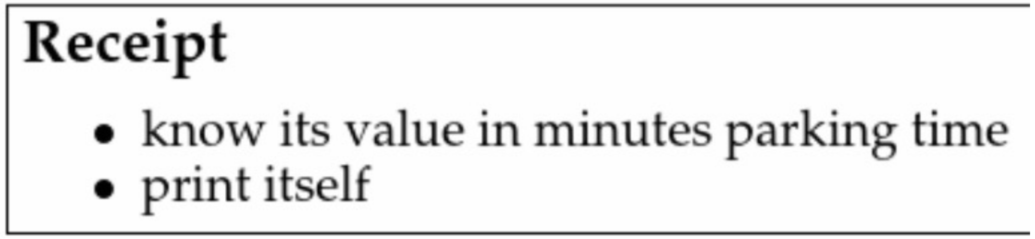
Questions for Today

How can we use compositional design for object creation?

What practices can lessen the effectiveness of design patterns?

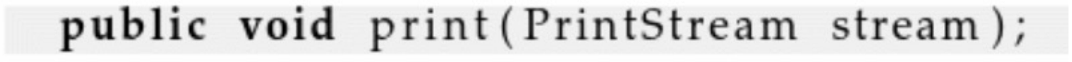
Pay Station Receipts

Suppose we have added the ability to print a receipt

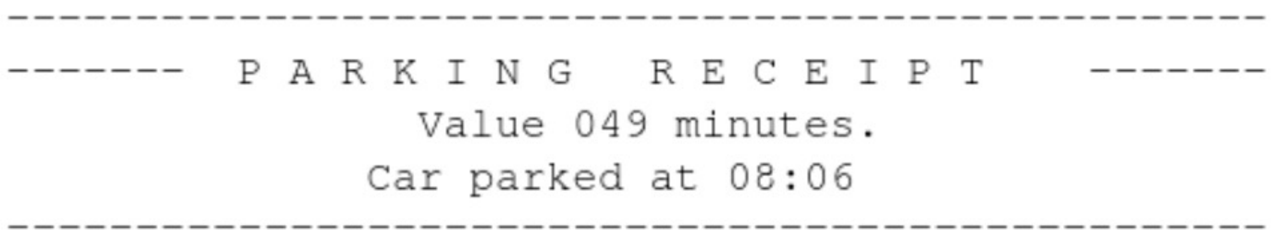


Receipt

- know its value in minutes parking time
- print itself



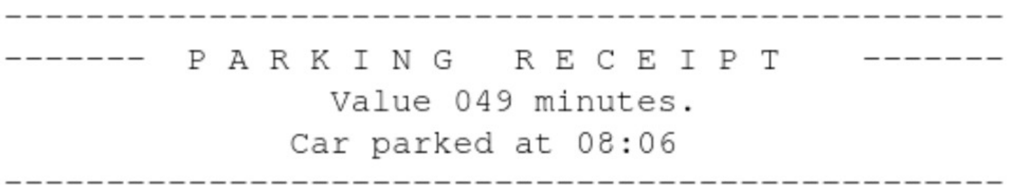
```
public void print(PrintStream stream);
```



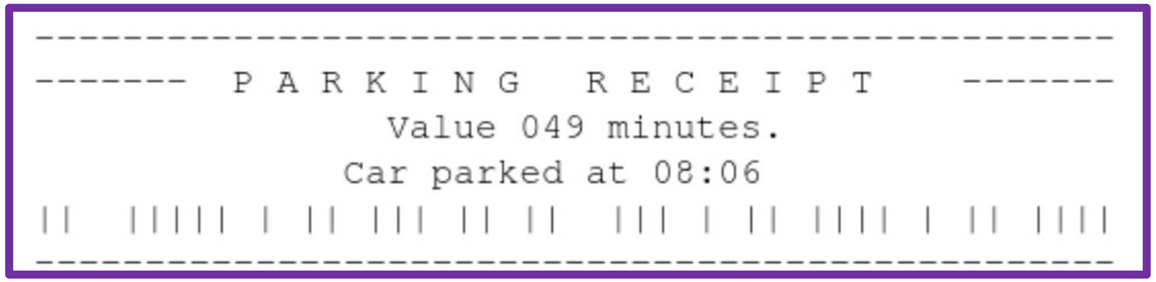
```
------------------------------------------------
-------  P A R K I N G   R E C E I P T   -------
               Value 049 minutes.
             Car parked at 08:06
------------------------------------------------
```

New Customer Request!

BetaTown wants printed receipts with parking statistics in the form of **bar codes**



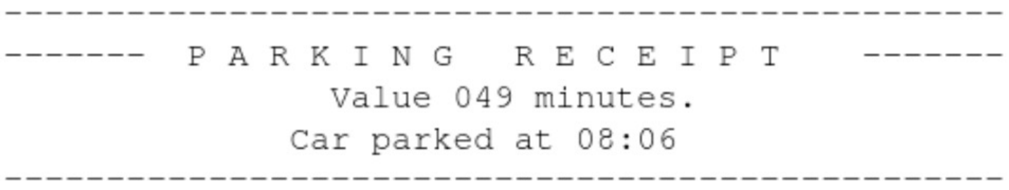
```
-------------------------------------------------
-------  P A R K I N G   R E C E I P T    -------
                Value 049 minutes.
              Car parked at 08:06
-------------------------------------------------
```



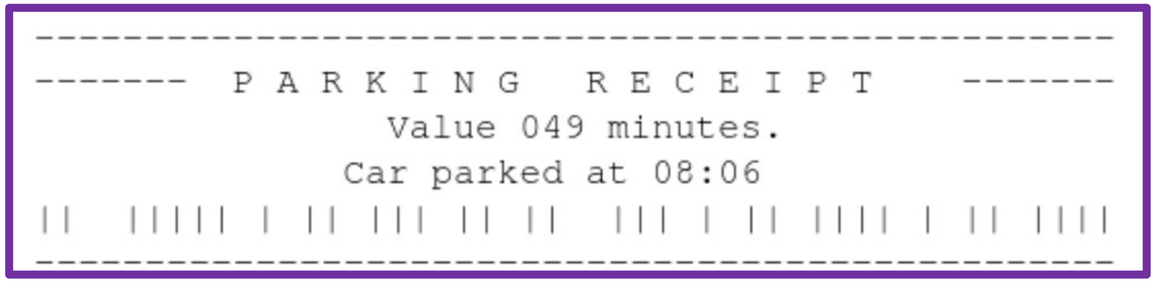
```
-------------------------------------------------
-------  P A R K I N G   R E C E I P T    -------
                Value 049 minutes.
              Car parked at 08:06
||  ||||| | || ||| || ||  ||| | || |||| | || ||||
-------------------------------------------------
```

New Customer Request!

BetaTown wants printed receipts with parking statistics in the form of **bar codes**



```
-------------------------------------------------
-------  P A R K I N G   R E C E I P T   -------
              Value 049 minutes.
            Car parked at 08:06
-------------------------------------------------
```



```
-------------------------------------------------
-------  P A R K I N G   R E C E I P T   -------
              Value 049 minutes.
            Car parked at 08:06
||  ||||| | || ||| || ||  ||| | || |||| | || ||||
-------------------------------------------------
```

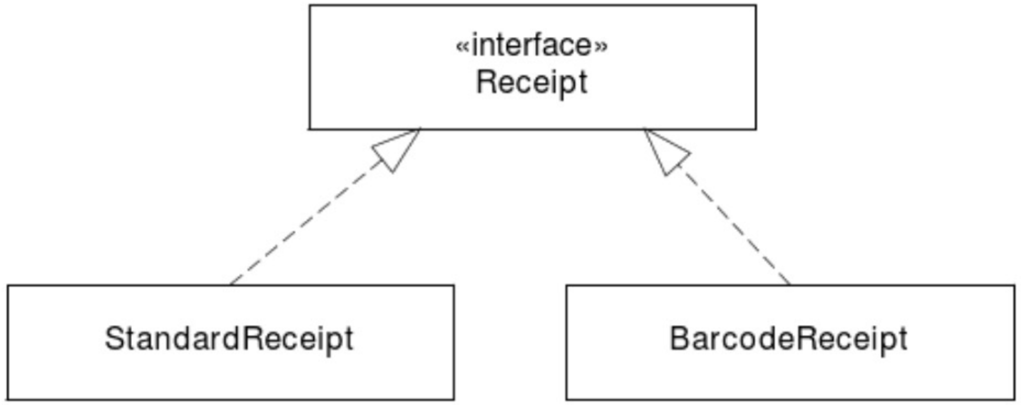
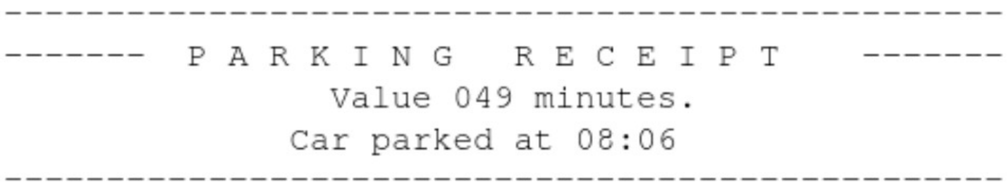


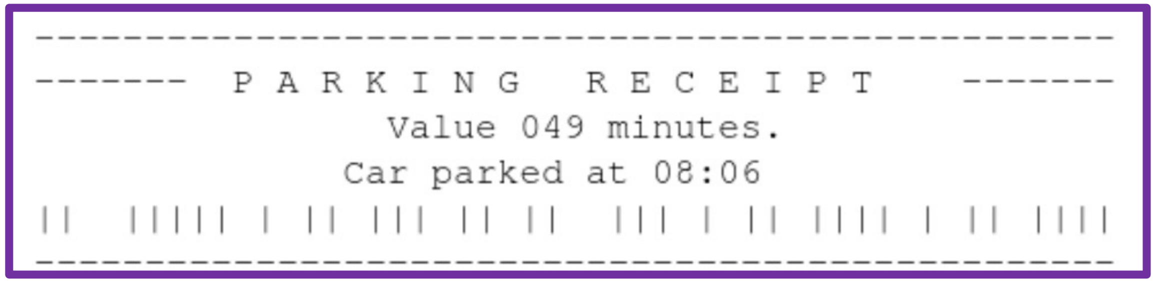
Figure 13.1: New types of Receipts.

New Customer Request!

BetaTown wants printed receipts with parking statistics in the form of **bar codes**



```
----------------------------------------------
-------  P A R K I N G   R E C E I P T   -------
              Value 049 minutes.
            Car parked at 08:06
----------------------------------------------
```



```
----------------------------------------------
-------  P A R K I N G   R E C E I P T   -------
              Value 049 minutes.
            Car parked at 08:06
||  ||||| | || ||| || ||  ||| | || |||| | || ||||
----------------------------------------------
```

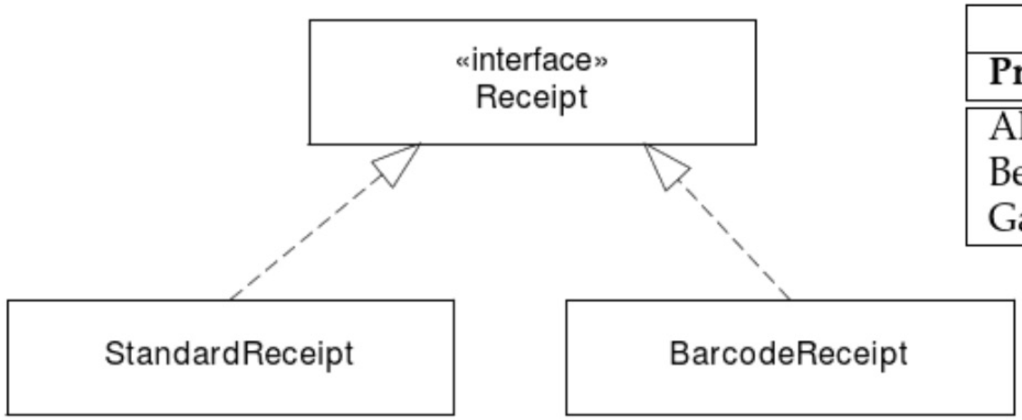
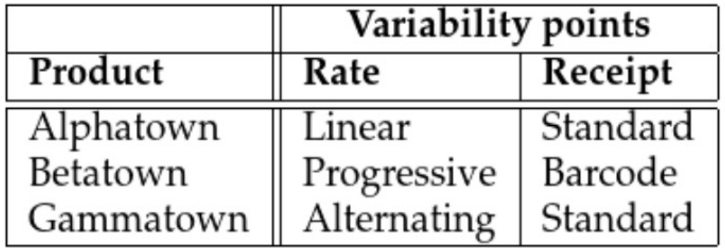


Figure 13.1: New types of Receipts.



	Variability points	
Product	**Rate**	**Receipt**
Alphatown	Linear	Standard
Betatown	Progressive	Barcode
Gammatown	Alternating	Standard

First Attempt: 3-1-2

(3) Identify a behavior that needs to vary

First Attempt: 3-1-2

(3) Identify a behavior that needs to vary

→ Printing of receipts; printing is instantiation, so instantiation needs to vary. AlphaTown and GammaTown Pay Stations should instantiate StandardReceipt, while BetaTown Pay Stations should instantiate BarcodeReceipt

First Attempt: 3-1-2

(3) Identify a behavior that needs to vary

→ Printing of receipts; printing is instantiation, so instantiation needs to vary. AlphaTown and GammaTown Pay Stations should instantiate StandardReceipt, while BetaTown Pay Stations should instantiate BarcodeReceipt

(1) State a responsibility that covers the behavior and express it as an interface

First Attempt: 3-1-2

(3) Identify a behavior that needs to vary

→Printing of receipts; printing is instantiation, so instantiation needs to vary. AlphaTown and GammaTown Pay Stations should instantiate StandardReceipt, while BetaTown Pay Stations should instantiate BarcodeReceipt

(1) State a responsibility that covers the behavior and express it as an interface

→We can define a ReceiptIssuer interface whose responsibility is to issue Receipts (that is, create Receipt objects). Currently this is handled by PayStation

First Attempt: 3-1-2

(3) Identify a behavior that needs to vary

→ Printing of receipts; printing is instantiation, so instantiation needs to vary. AlphaTown and GammaTown Pay Stations should instantiate StandardReceipt, while BetaTown Pay Stations should instantiate BarcodeReceipt

(1) State a responsibility that covers the behavior and express it as an interface

→ We can define a ReceiptIssuer interface whose responsibility is to issue Receipts (that is, create Receipt objects). Currently this is handled by PayStation

(2) Compose the resulting behavior by delegating

First Attempt: 3-1-2

(3) Identify a behavior that needs to vary

→Printing of receipts; printing is instantiation, so instantiation needs to vary. AlphaTown and GammaTown Pay Stations should instantiate StandardReceipt, while BetaTown Pay Stations should instantiate BarcodeReceipt

(1) State a responsibility that covers the behavior and express it as an interface

→We can define a ReceiptIssuer interface whose responsibility is to issue Receipts (that is, create Receipt objects). Currently this is handled by PayStation

(2) Compose the resulting behavior by delegating

→ The pay station should delegate to the receipt issuer to instantiate receipts

First Attempt: 3-1-2

(3) Identify a behavior that needs to vary

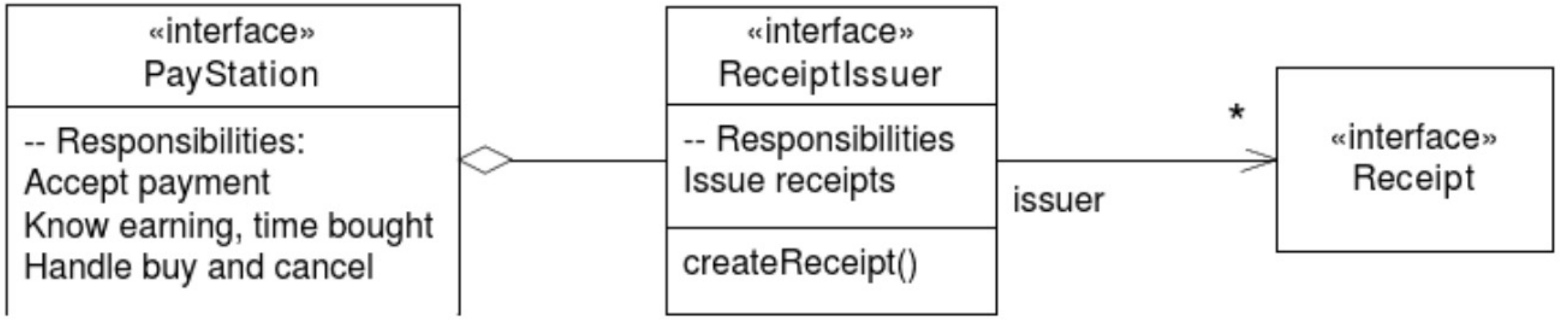
→ Printing of receipts; printing is instantiation, so instantiation needs to vary. AlphaTown and GammaTown Pay Stations should instantiate StandardReceipt, while BetaTown Pay Stations should instantiate BarcodeReceipt

(1) State a responsibility that covers the behavior and express it as an interface

→ We can define a ReceiptIssuer interface whose responsibility is to issue Receipts (that is, create Receipt objects). Currently this is handled by PayStation

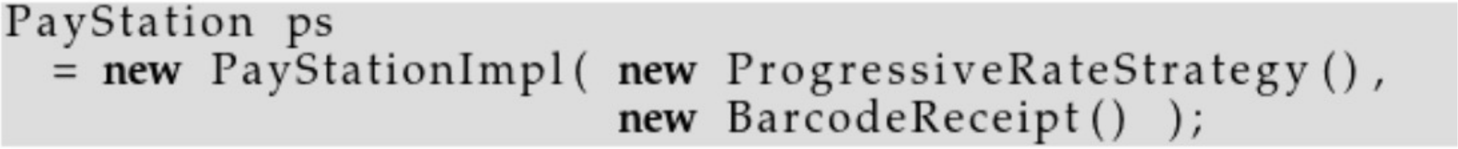
(2) Compose the resulting behavior by delegating

→ The pay station should delegate to the receipt issuer to instantiate receipts

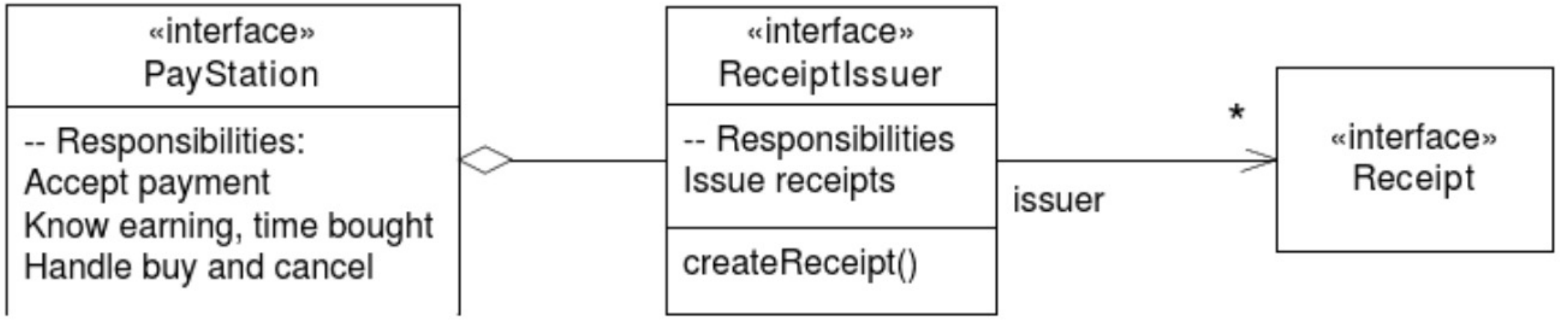


First Attempt: 3-1-2

Couldn’t we just provide the receipt type similarly to the rate strategy?

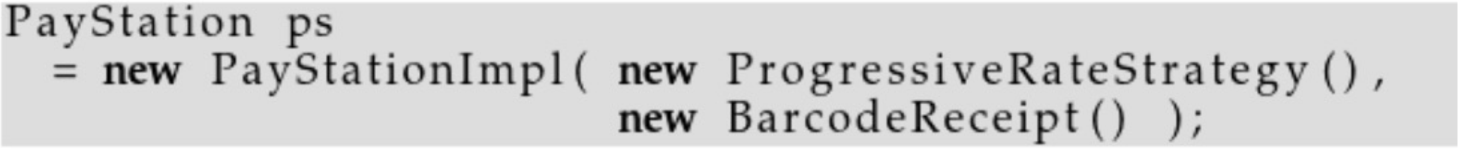


```
PayStation ps
  = new PayStationImpl( new ProgressiveRateStrategy(),
                        new BarcodeReceipt()  );
```

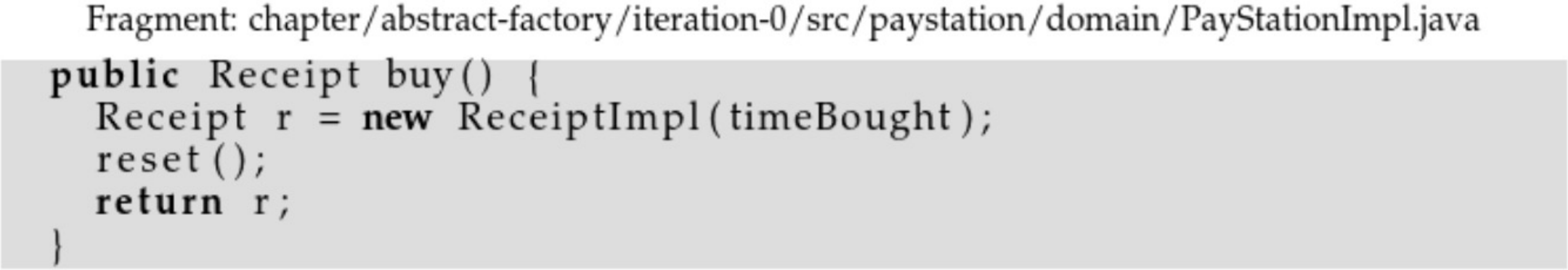


First Attempt: 3-1-2

Couldn’t we just provide the receipt type similarly to the rate strategy?

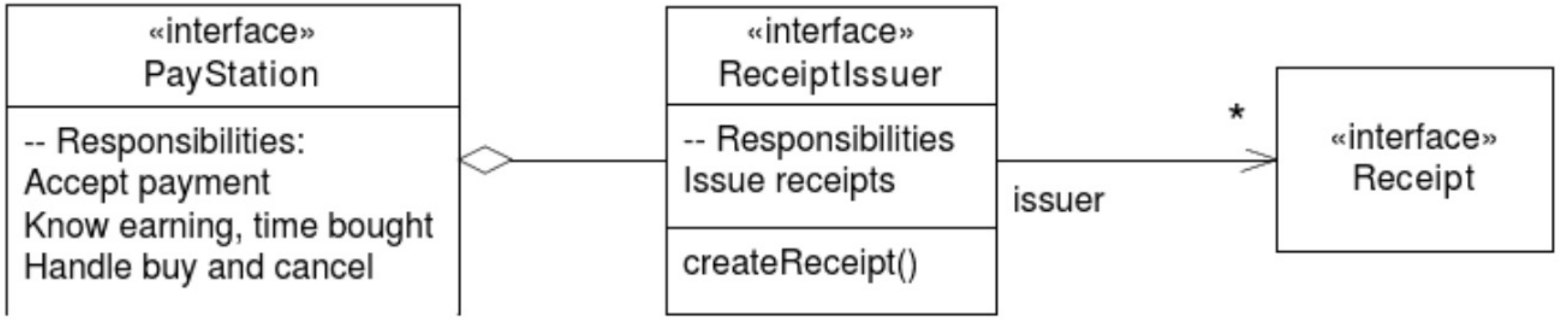


```
PayStation ps
  = new PayStationImpl( new ProgressiveRateStrategy(),
                        new BarcodeReceipt()  );
```



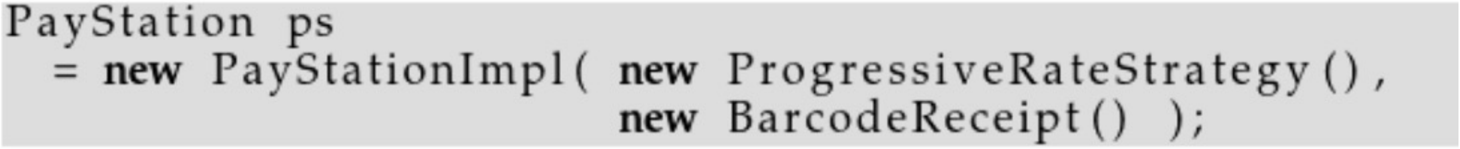
Fragment: chapter/abstract-factory/iteration-0/src/paystation/domain/PayStationImpl.java

```
  public Receipt buy() {
    Receipt r = new ReceiptImpl(timeBought);
    reset();
    return r;
  }
```



First Attempt: 3-1-2

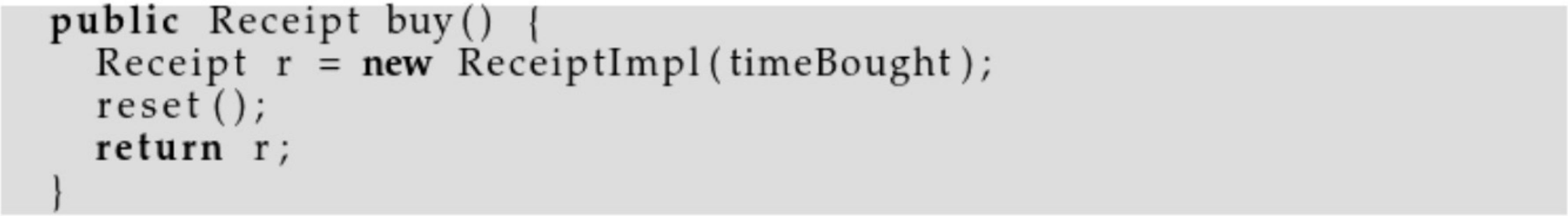
Couldn’t we just provide the receipt type similarly to the rate strategy?



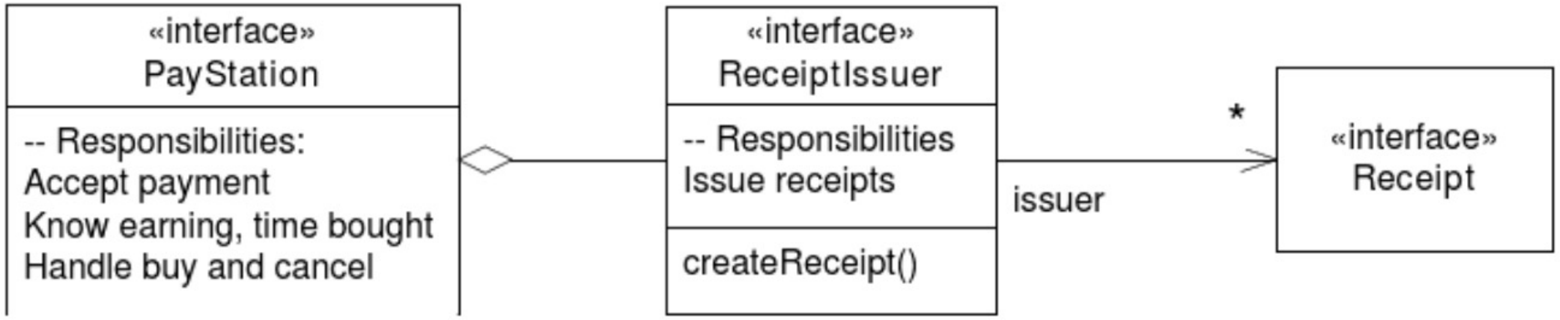
```
PayStation ps
  = new PayStationImpl( new ProgressiveRateStrategy(),
                        new BarcodeReceipt()  );
```

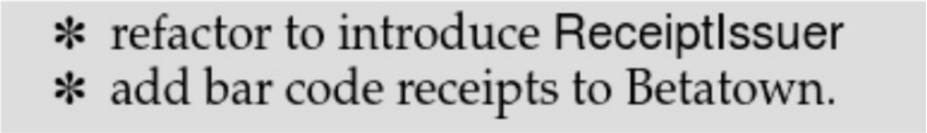
Problem: Pay Station **creates** Receipt! Each receipt is unique and has its own parking time. So, we need the issuer.

Fragment: chapter/abstract-factory/iteration-0/src/paystation/domain/PayStationImpl.java



```
  public Receipt buy() {
    Receipt r = new ReceiptImpl(timeBought);
    reset();
    return r;
  }
```

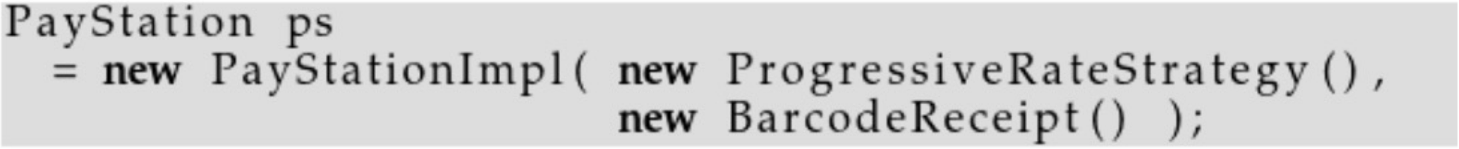




✻ refactor to introduce ReceiptIssuer
✻ add bar code receipts to Betatown.

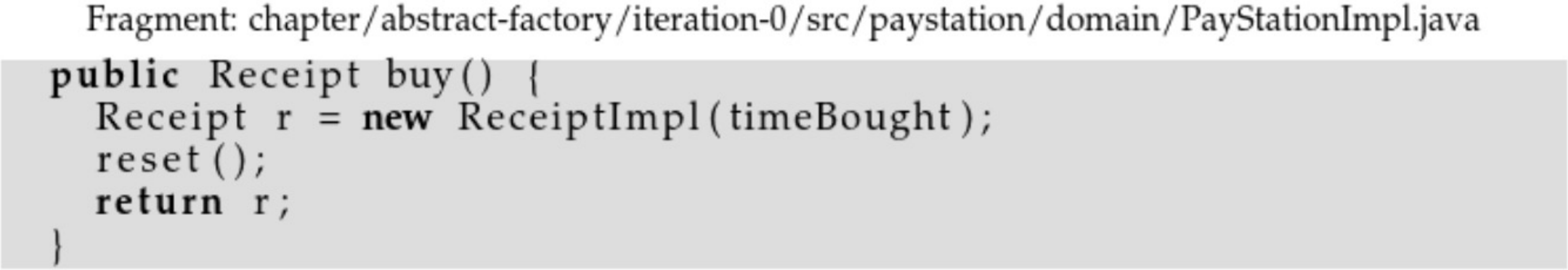
First Attempt: 3-1-2

Couldn’t we just provide the receipt type similarly to the rate strategy?



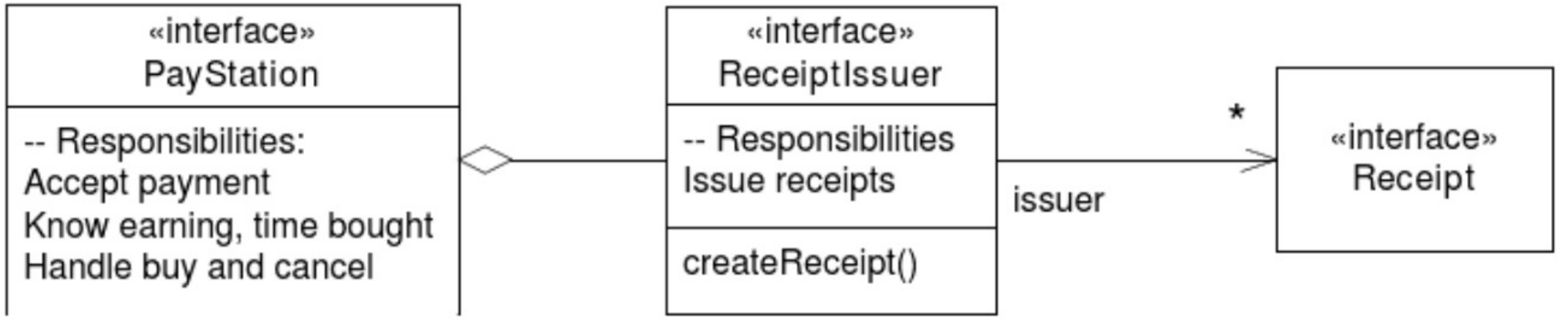
```
PayStation ps
  = new PayStationImpl( new ProgressiveRateStrategy(),
                        new BarcodeReceipt()  );
```

Problem: Pay Station **creates** Receipt! Each receipt is unique and has its own parking time. So, we need the issuer.

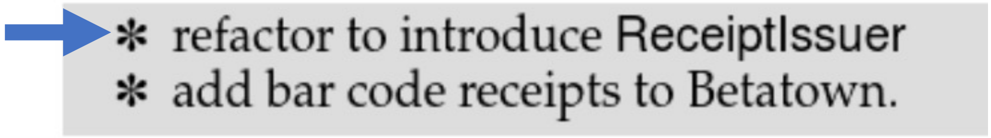


Fragment: chapter/abstract-factory/iteration-0/src/paystation/domain/PayStationImpl.java

```
  public Receipt buy() {
    Receipt r = new ReceiptImpl(timeBought);
    reset();
    return r;
  }
```

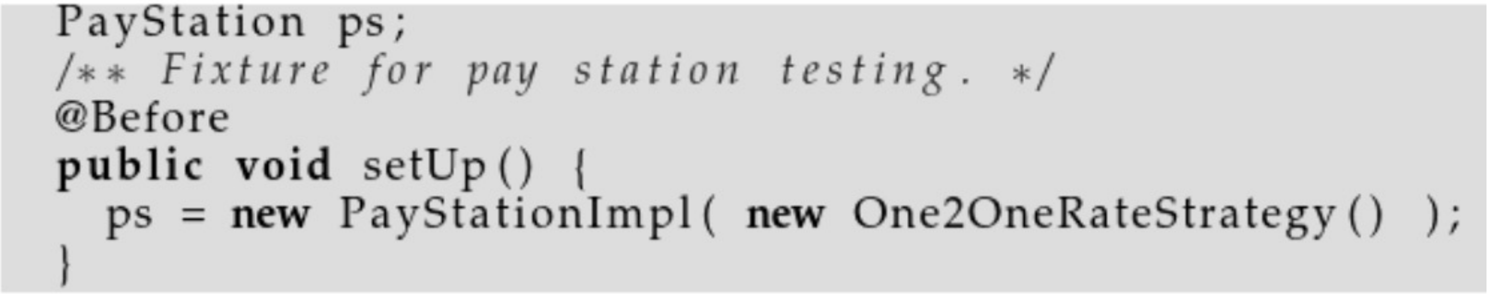


First Attempt: 3-1-2

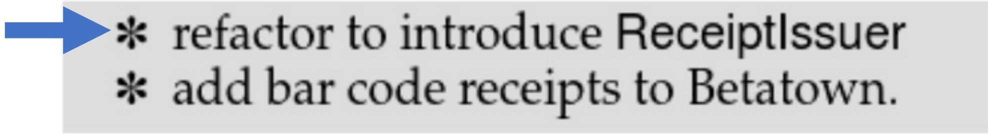


- ✻ refactor to introduce ReceiptIssuer
- ✻ add bar code receipts to Betatown.

Current test fixture:



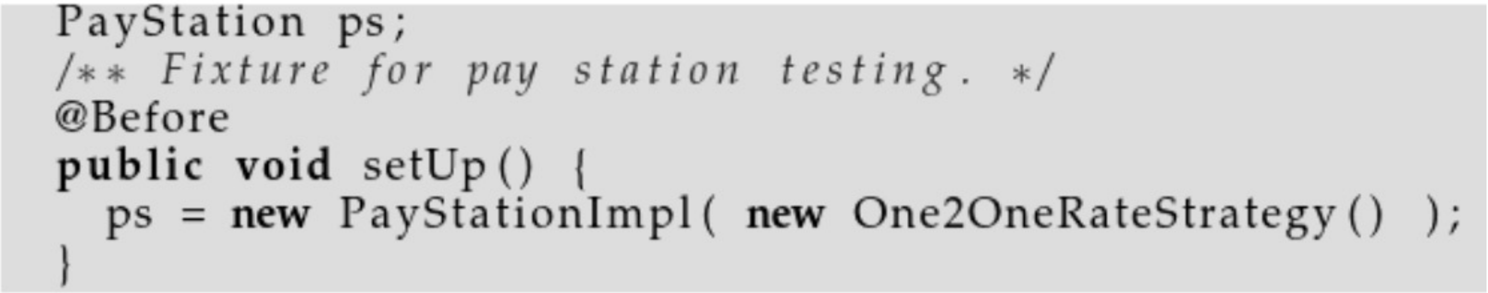
```
PayStation ps;
/** Fixture for pay station testing. */
@Before
public void setUp() {
  ps = new PayStationImpl( new One2OneRateStrategy() );
}
```



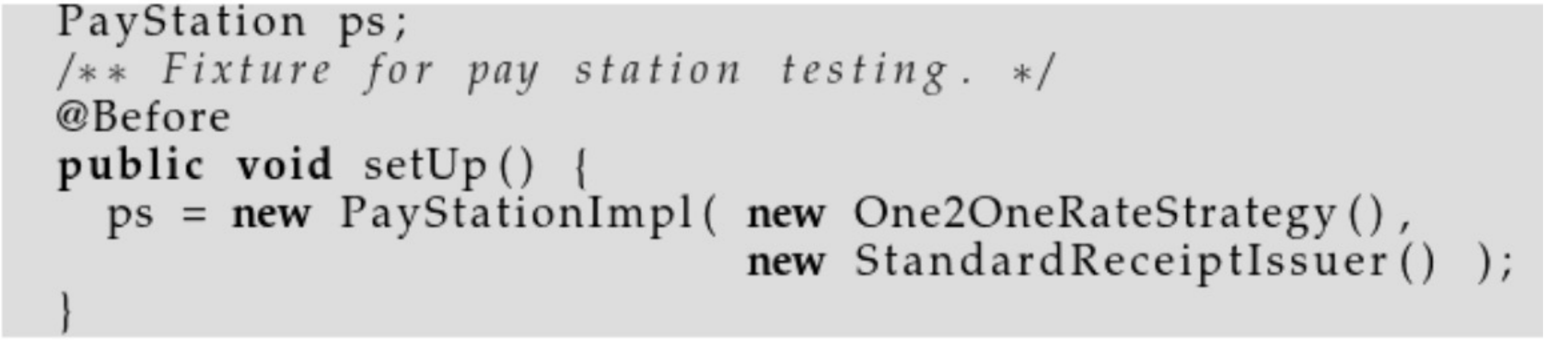
- ✱ refactor to introduce ReceiptIssuer
- ✱ add bar code receipts to Betatown.

First Attempt: 3-1-2

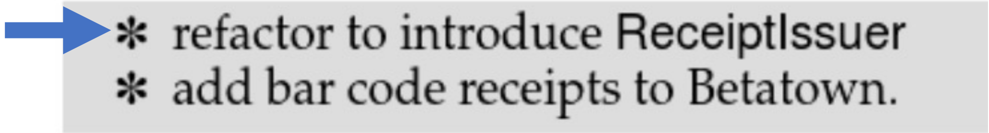
Current test fixture:



```java
PayStation ps;
/** Fixture for pay station testing. */
@Before
public void setUp() {
  ps = new PayStationImpl( new One2OneRateStrategy() );
}
```



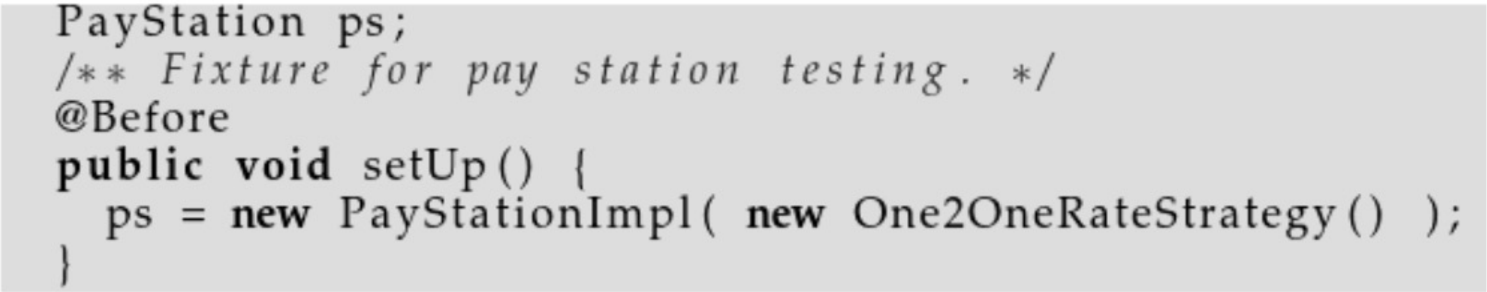
```java
PayStation ps;
/** Fixture for pay station testing. */
@Before
public void setUp() {
  ps = new PayStationImpl( new One2OneRateStrategy(),
                           new StandardReceiptIssuer() );
}
```



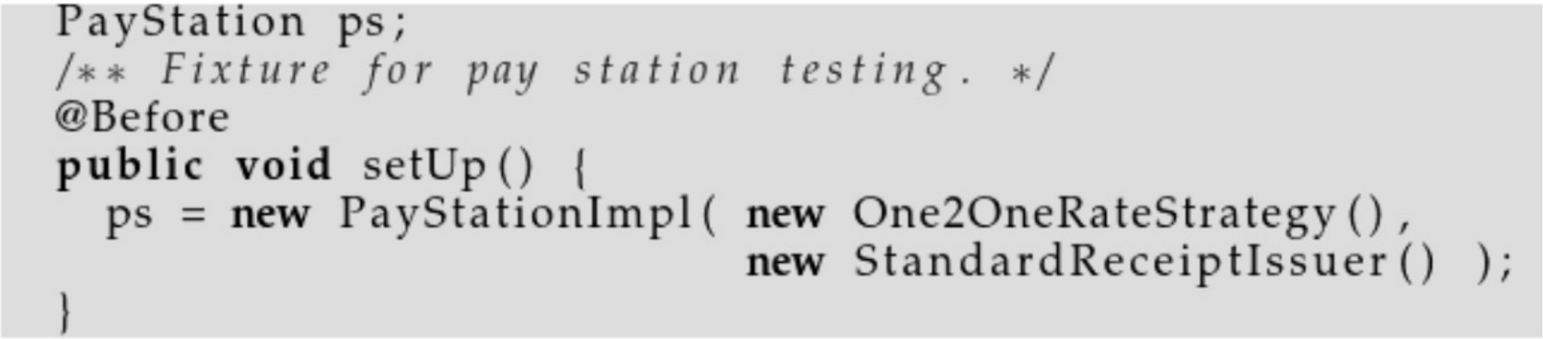
- ✻ refactor to introduce ReceiptIssuer
- ✻ add bar code receipts to Betatown.

First Attempt: 3-1-2

Current test fixture:



```
PayStation ps;
/** Fixture for pay station testing. */
@Before
public void setUp() {
  ps = new PayStationImpl( new One2OneRateStrategy() );
}
```

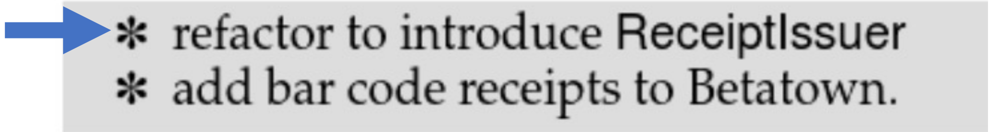


```
PayStation ps;
/** Fixture for pay station testing. */
@Before
public void setUp() {
  ps = new PayStationImpl( new One2OneRateStrategy(),
                           new StandardReceiptIssuer() );
}
```

Problem: Configuration behavior is split between two different objects, the pay station and receipt issuer

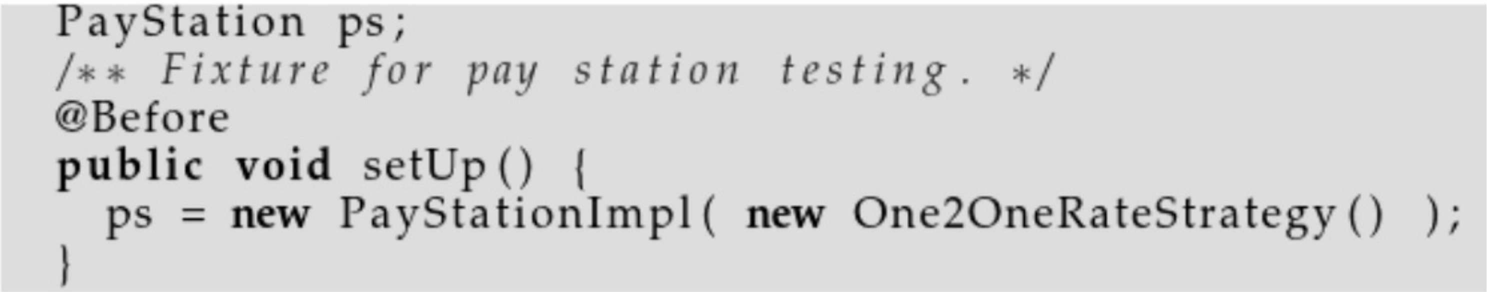
→ Not cohesive

First Attempt: 3-1-2

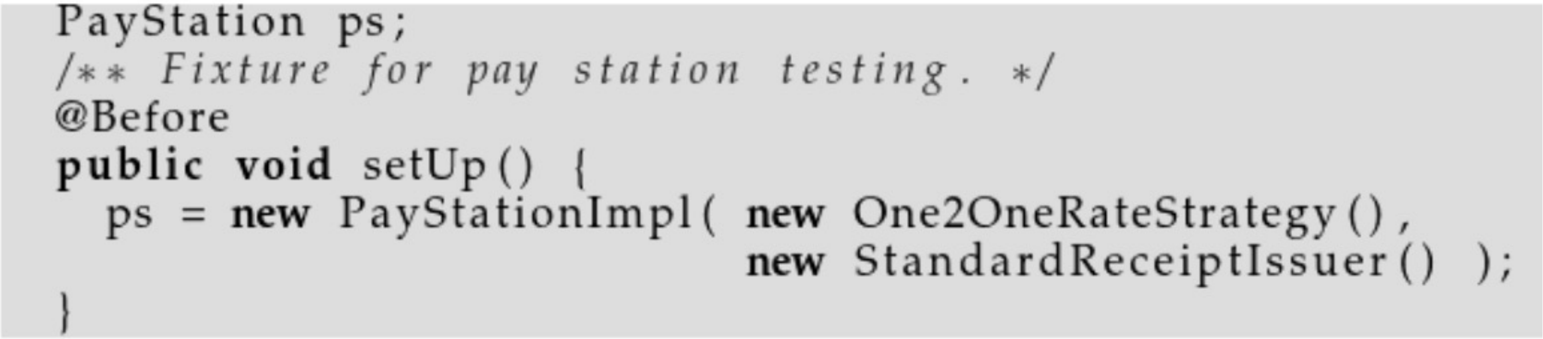


- ✻ refactor to introduce ReceiptIssuer
- ✻ add bar code receipts to Betatown.

Current test fixture:



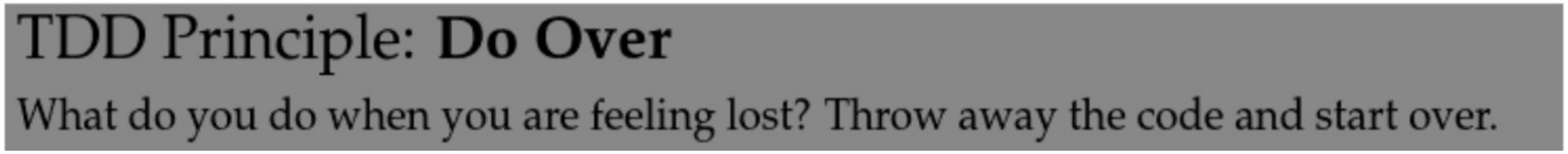
```
PayStation ps;
/** Fixture for pay station testing. */
@Before
public void setUp() {
  ps = new PayStationImpl( new One2OneRateStrategy() );
}
```



```
PayStation ps;
/** Fixture for pay station testing. */
@Before
public void setUp() {
  ps = new PayStationImpl( new One2OneRateStrategy(),
                           new StandardReceiptIssuer() );
}
```

Problem: Configuration behavior is split between two different objects, the pay station and receipt issuer

→ Not cohesive



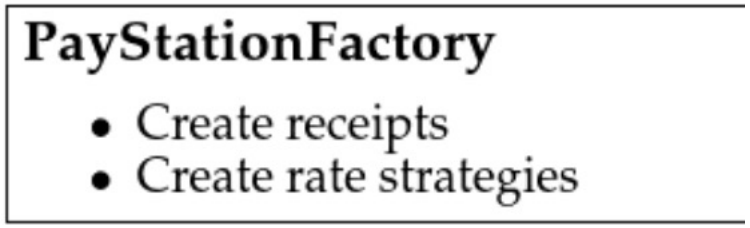
TDD Principle: Do Over

What do you do when you are feeling lost? Throw away the code and start over.

New Attempt

Create a **factory object**

→ Responsibility is to make objects



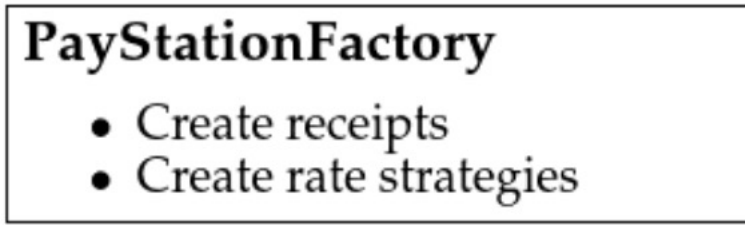
PayStationFactory

- Create receipts
- Create rate strategies

New Attempt

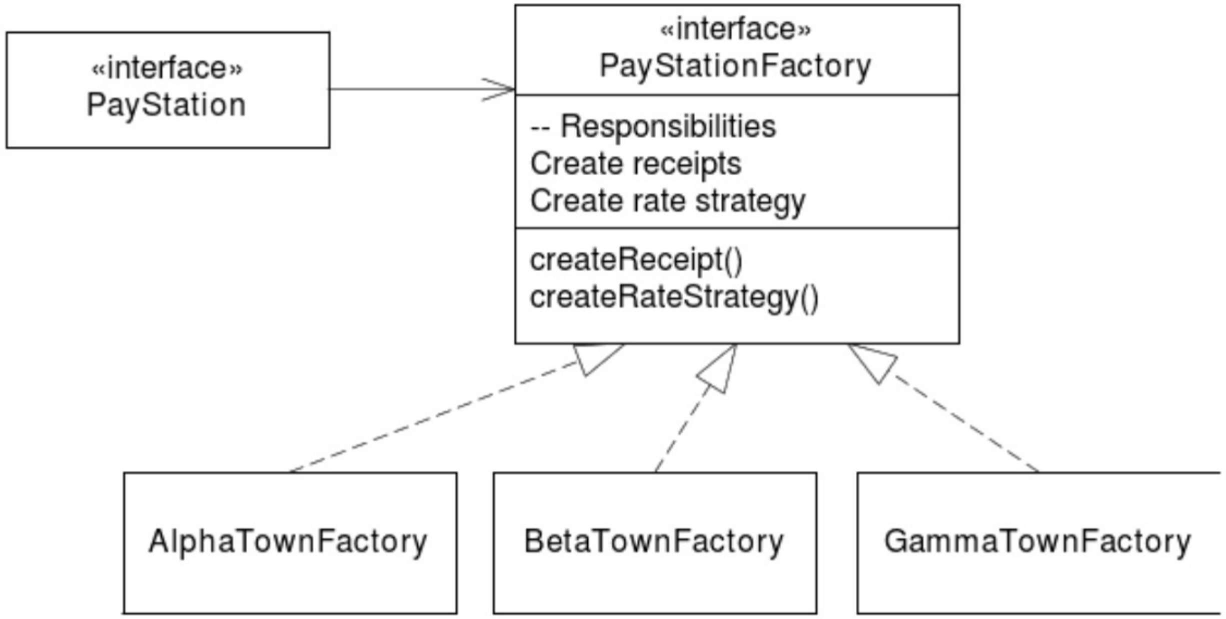
Create a **factory object**

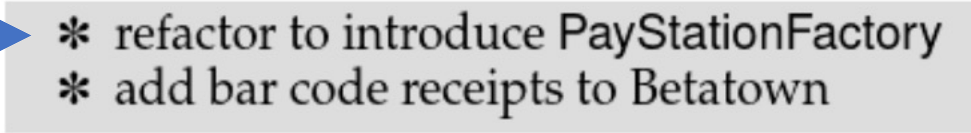
→ Responsibility is to make objects



PayStationFactory

- Create receipts
- Create rate strategies



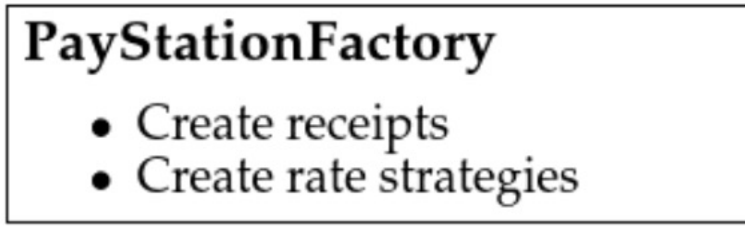


- ✻ refactor to introduce PayStationFactory
- ✻ add bar code receipts to Betatown

New Attempt

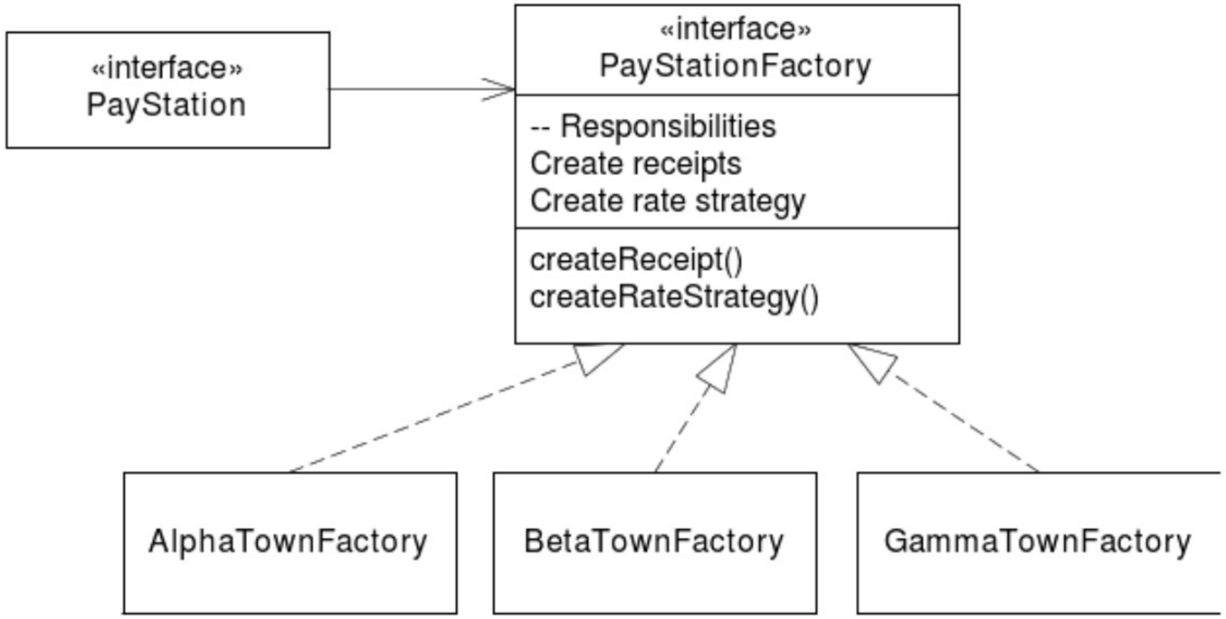
Create a **factory object**

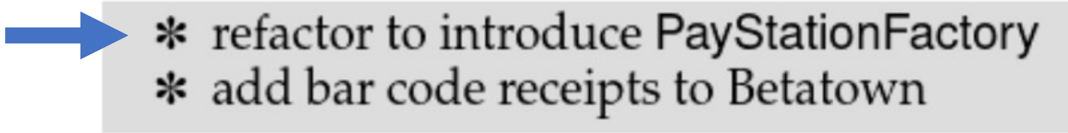
→ Responsibility is to make objects



PayStationFactory

- Create receipts
- Create rate strategies

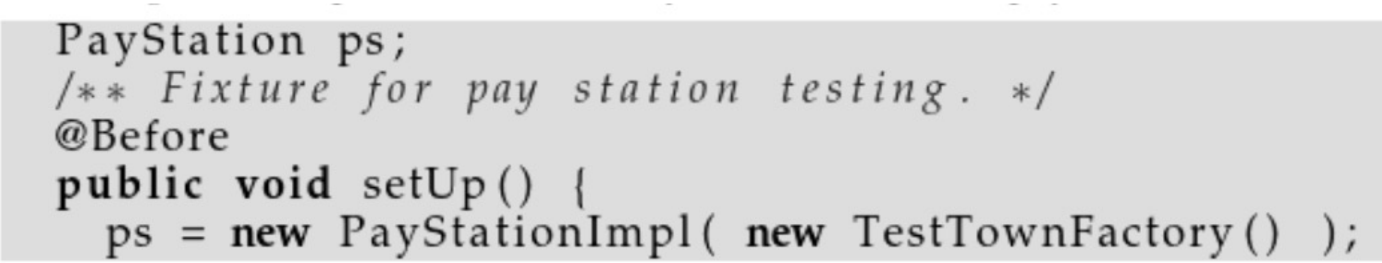




- ✻ refactor to introduce PayStationFactory
- ✻ add bar code receipts to Betatown

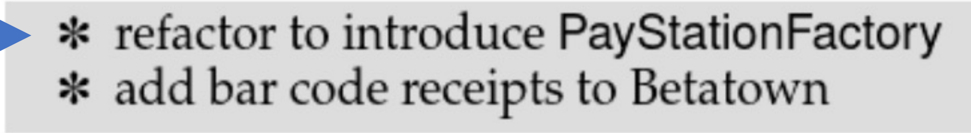
New Attempt

New test



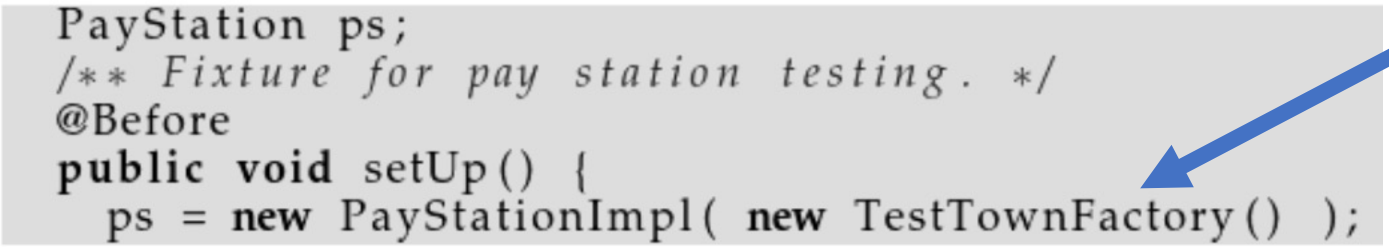
```
PayStation ps;
/** Fixture for pay station testing. */
@Before
public void setUp() {
  ps = new PayStationImpl( new TestTownFactory() );
```

New Attempt



- ✻ refactor to introduce PayStationFactory
- ✻ add bar code receipts to Betatown

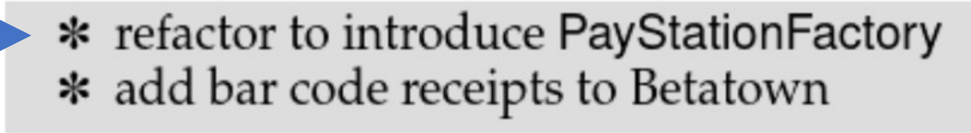
New test



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

One2OneRateStrategy

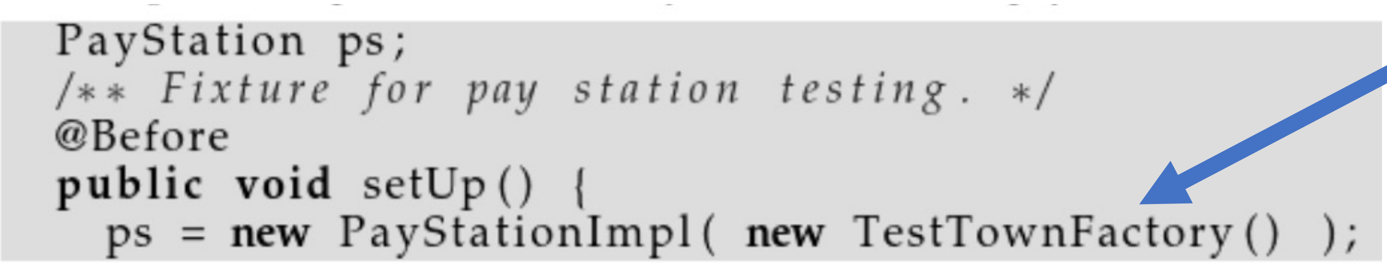
StandardReceipt



✻ refactor to introduce PayStationFactory
✻ add bar code receipts to Betatown

New Attempt

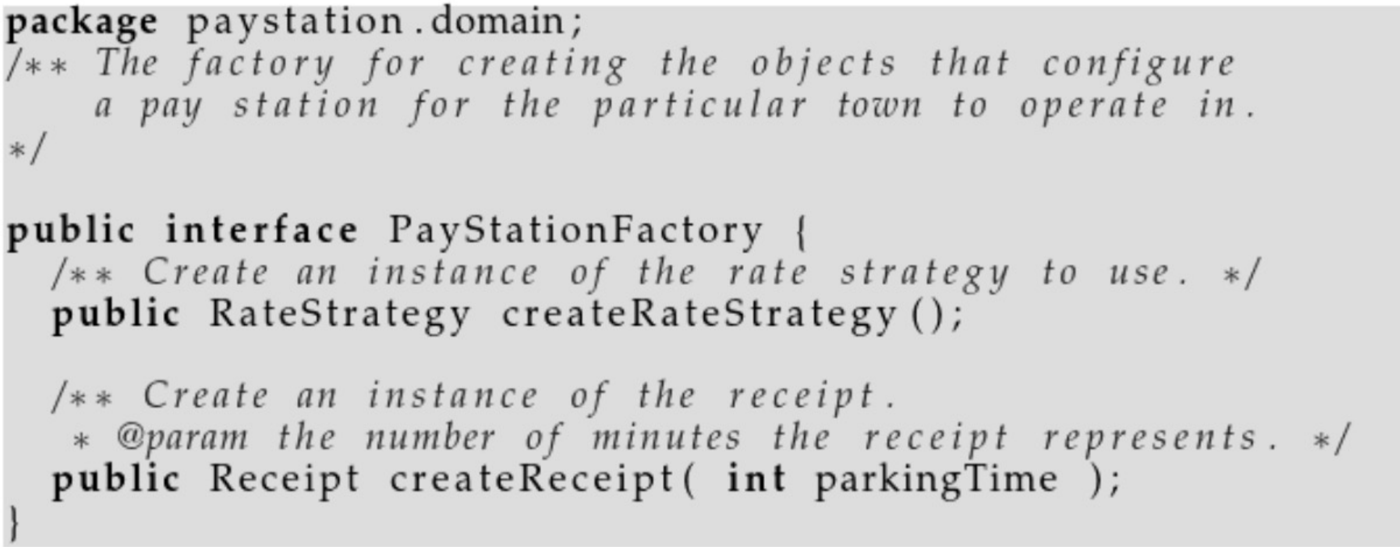
New test



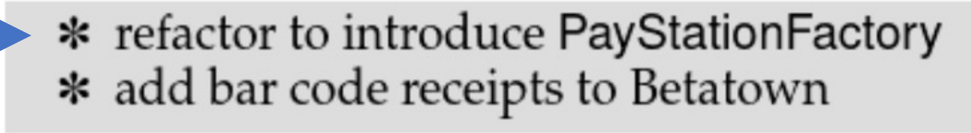
```
  PayStation ps;
  /** Fixture for pay station testing. */
  @Before
  public void setUp() {
    ps = new PayStationImpl( new TestTownFactory() );
```

One2OneRateStrategy
StandardReceipt

Add PayStationFactory interface



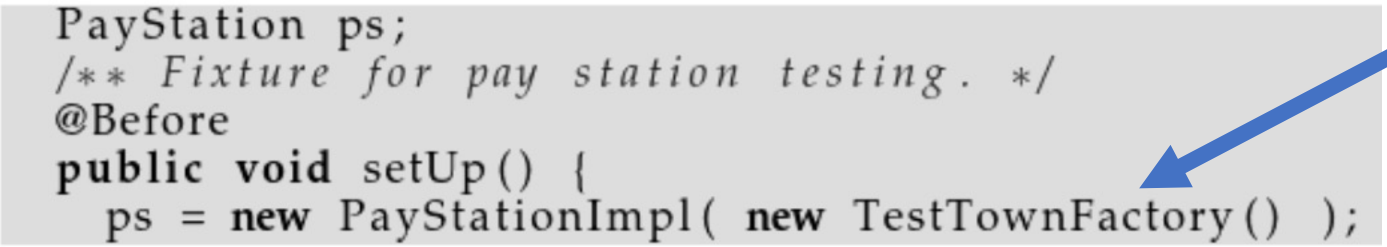
```
package paystation.domain;
/** The factory for creating the objects that configure
    a pay station for the particular town to operate in.
*/

public interface PayStationFactory {
  /** Create an instance of the rate strategy to use. */
  public RateStrategy createRateStrategy();

  /** Create an instance of the receipt.
   * @param the number of minutes the receipt represents. */
  public Receipt createReceipt( int parkingTime );
}
```



- ✻ refactor to introduce PayStationFactory
- ✻ add bar code receipts to Betatown

New Attempt

New test

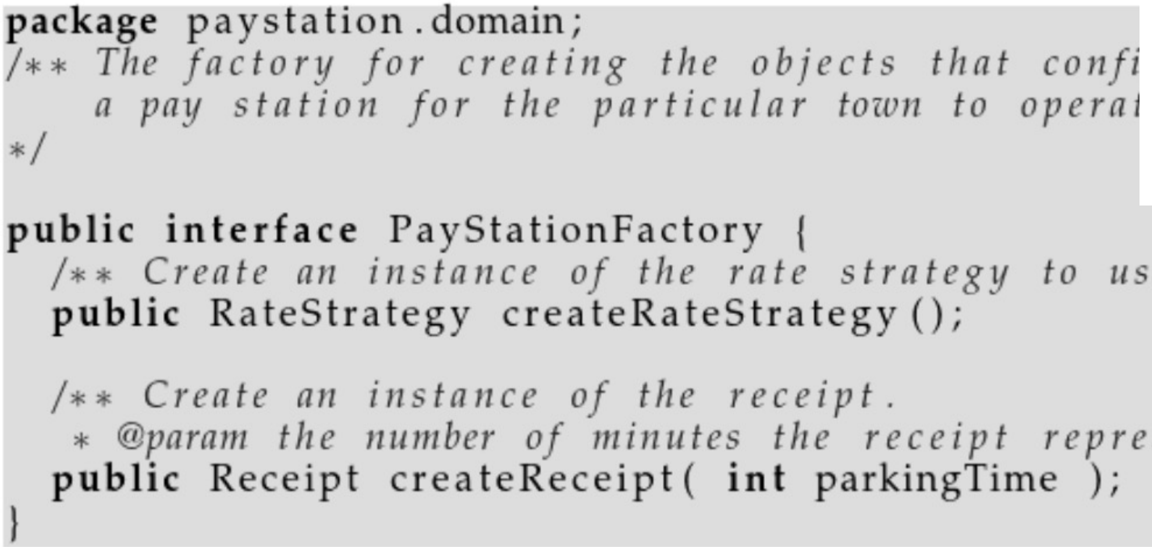


```
PayStation ps;
/** Fixture for pay station testing. */
@Before
public void setUp() {
  ps = new PayStationImpl( new TestTownFactory() );
```

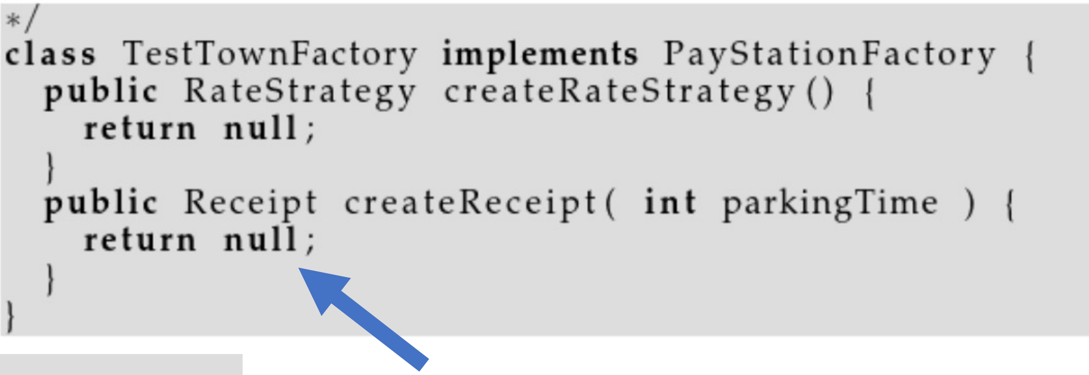
One2OneRateStrategy

StandardReceipt

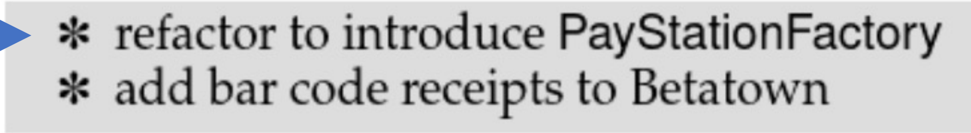
Add PayStationFactory interface



```
package paystation.domain;
/** The factory for creating the objects that confi
    a pay station for the particular town to operat
*/

public interface PayStationFactory {
  /** Create an instance of the rate strategy to use. */
  public RateStrategy createRateStrategy();

  /** Create an instance of the receipt.
   * @param the number of minutes the receipt represents. */
  public Receipt createReceipt( int parkingTime );
}
```



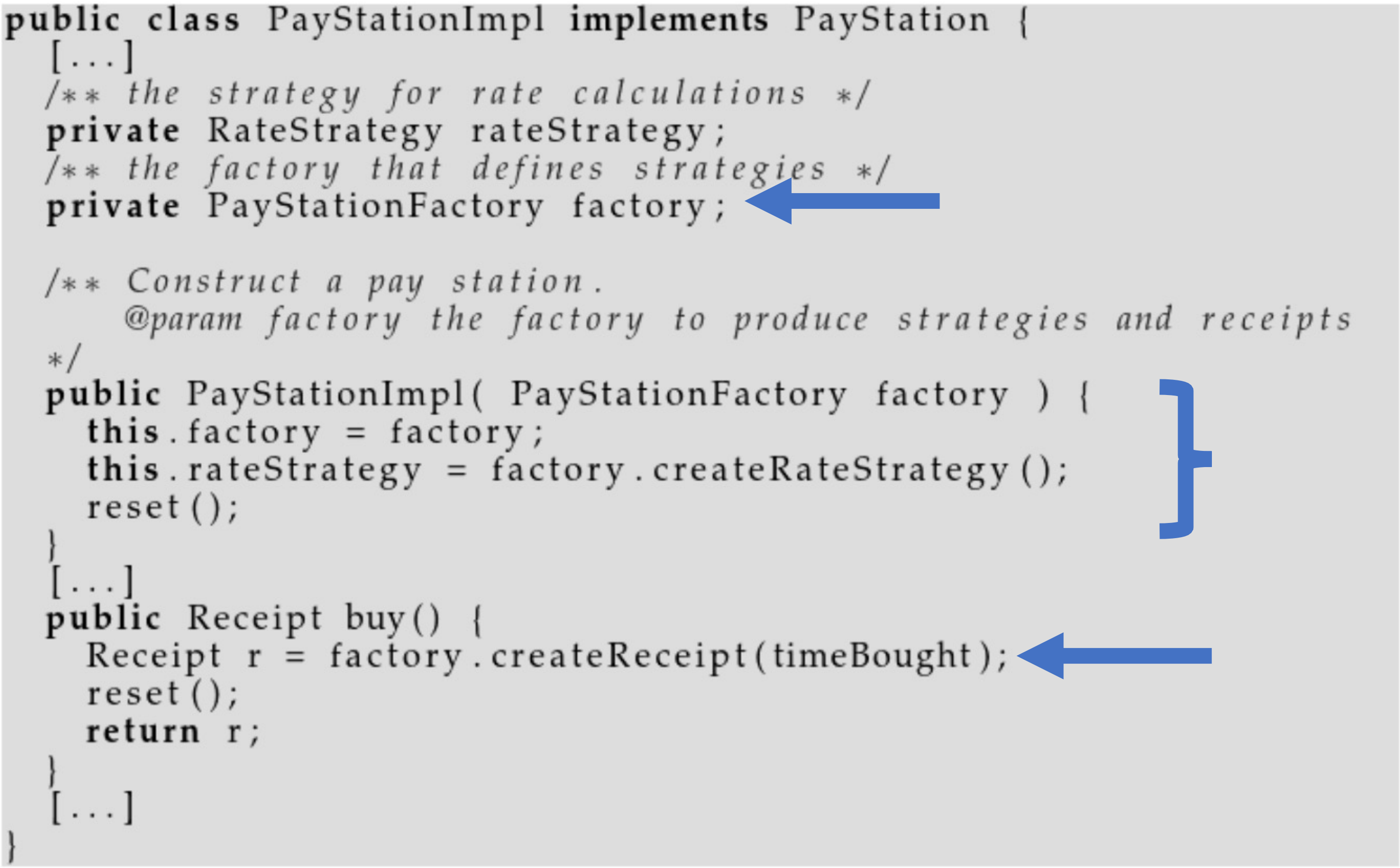
```
*/
class TestTownFactory implements PayStationFactory {
  public RateStrategy createRateStrategy() {
    return null;
  }
  public Receipt createReceipt( int parkingTime ) {
    return null;
  }
}
```



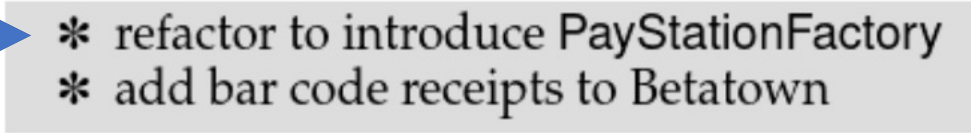
✻ refactor to introduce PayStationFactory
✻ add bar code receipts to Betatown

New Attempt

Refactor PayStationImpl to use the factory



```
public class PayStationImpl implements PayStation {
  [...]
  /** the strategy for rate calculations */
  private RateStrategy rateStrategy;
  /** the factory that defines strategies */
  private PayStationFactory factory;

  /** Construct a pay station.
      @param factory the factory to produce strategies and receipts
  */
  public PayStationImpl( PayStationFactory factory ) {
    this.factory = factory;
    this.rateStrategy = factory.createRateStrategy();
    reset();
  }
  [...]
  public Receipt buy() {
    Receipt r = factory.createReceipt(timeBought);
    reset();
    return r;
  }
  [...]
}
```

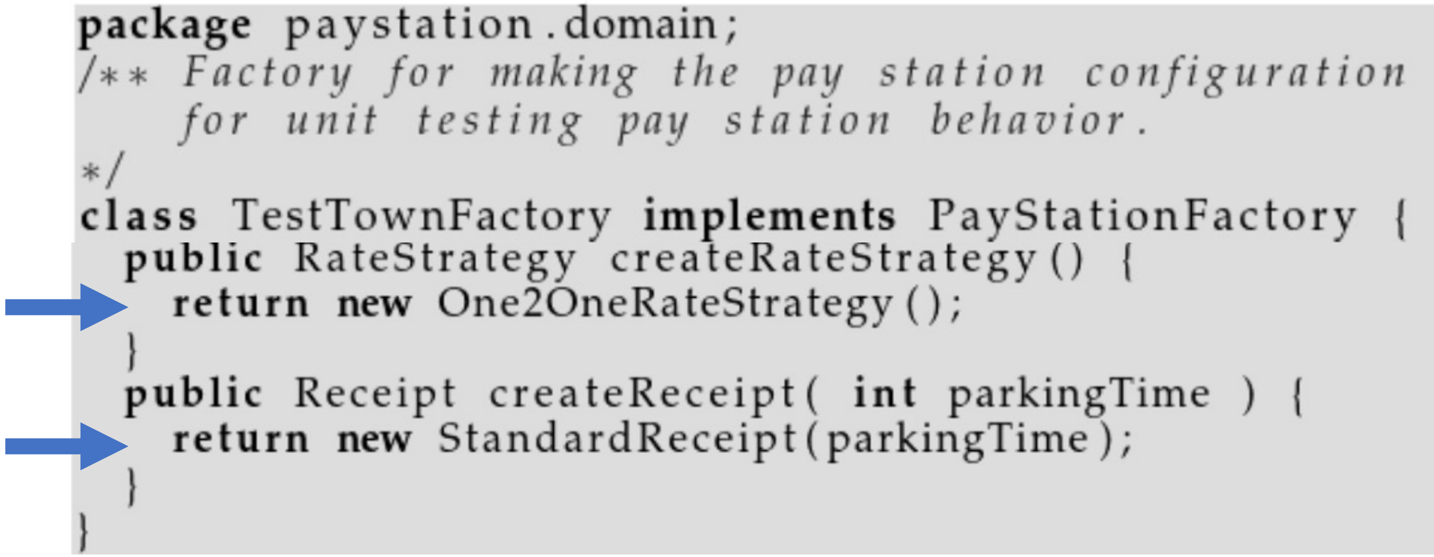


- ✻ refactor to introduce PayStationFactory
- ✻ add bar code receipts to Betatown

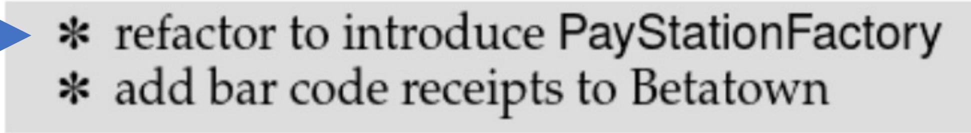
New Attempt

Implement factories

TestTownFactory



```
package paystation.domain;
/** Factory for making the pay station configuration
    for unit testing pay station behavior.
*/
class TestTownFactory implements PayStationFactory {
  public RateStrategy createRateStrategy() {
    return new One2OneRateStrategy();
  }
  public Receipt createReceipt( int parkingTime ) {
    return new StandardReceipt(parkingTime);
  }
}
```

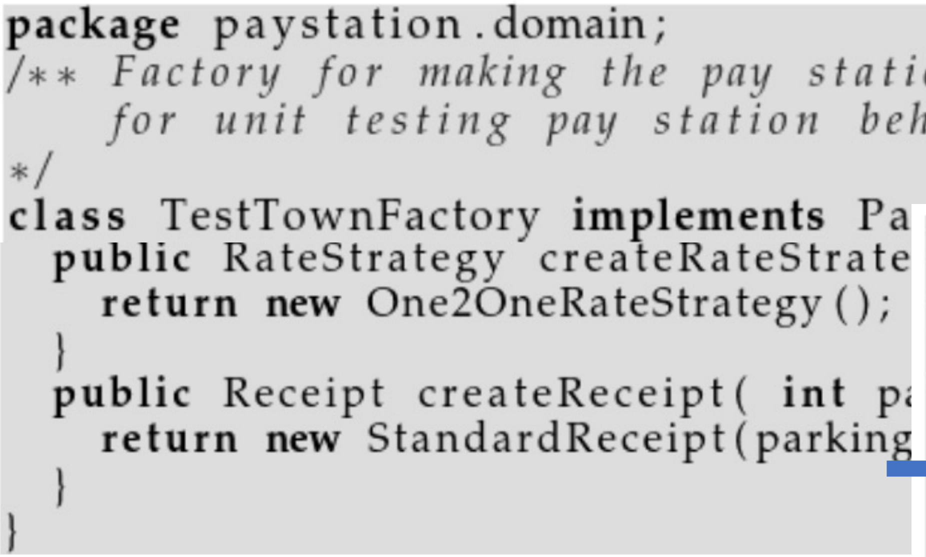


✻ refactor to introduce PayStationFactory
✻ add bar code receipts to Betatown

New Attempt

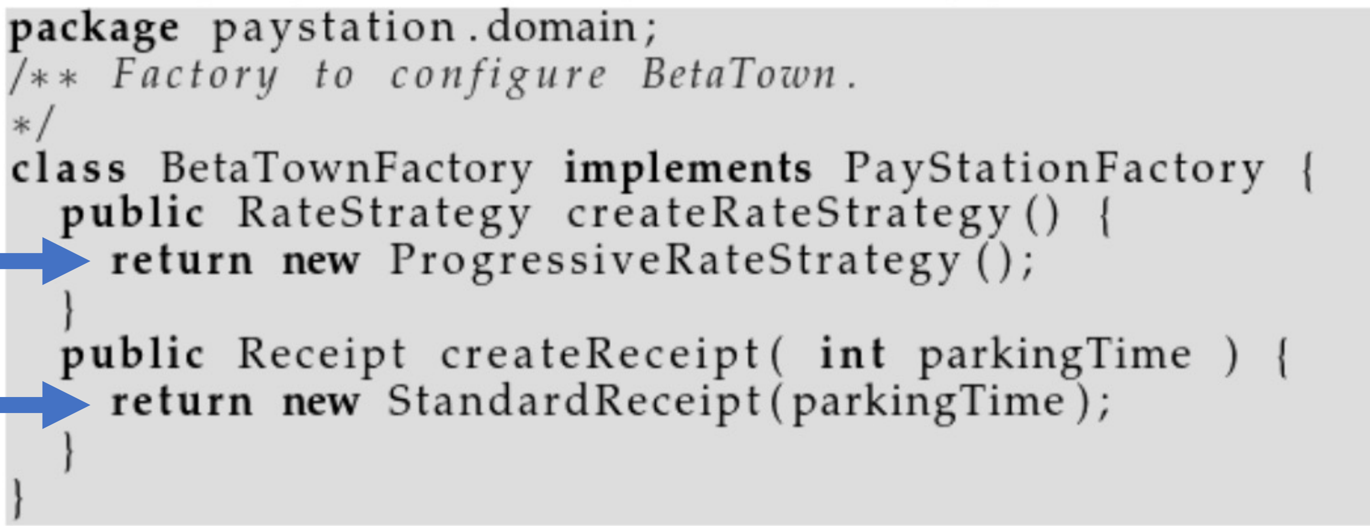
Implement factories

TestTownFactory



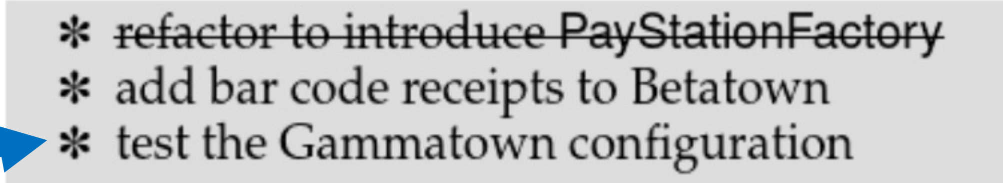
```
package paystation.domain;
/** Factory for making the pay station configuration
    for unit testing pay station behavior.
*/
class TestTownFactory implements Pa
  public RateStrategy createRateStrate
    return new One2OneRateStrategy();
  }
  public Receipt createReceipt( int pa
    return new StandardReceipt(parking
  }
}
```

BetaTownFactory



```
package paystation.domain;
/** Factory to configure BetaTown.
*/
class BetaTownFactory implements PayStationFactory {
  public RateStrategy createRateStrategy() {
    return new ProgressiveRateStrategy();
  }
  public Receipt createReceipt( int parkingTime ) {
    return new StandardReceipt(parkingTime);
  }
}
```

New Attempt



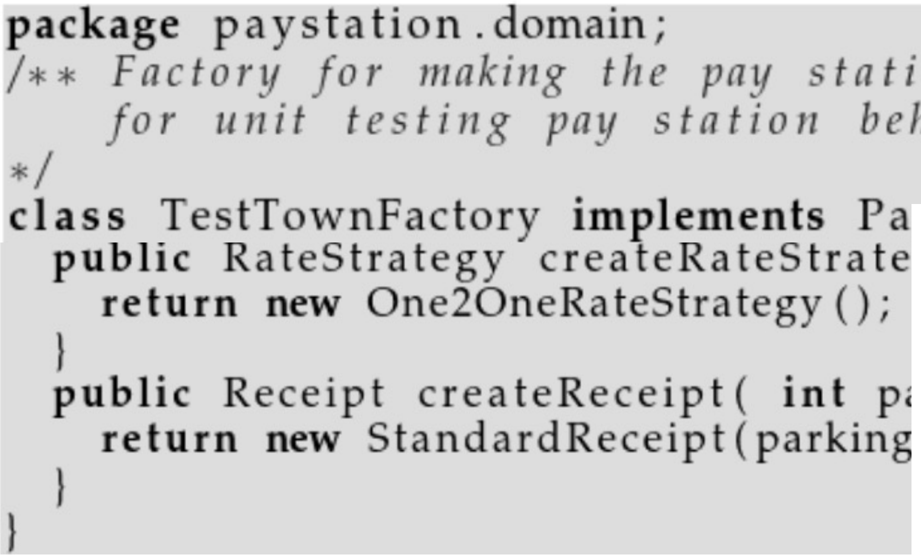
- ✻ ~~refactor to introduce PayStationFactory~~
- ✻ add bar code receipts to Betatown
- ✻ test the Gammatown configuration

Implement factories

AlphaTownFactory:
LinearRateStrategy, StandardReceipt

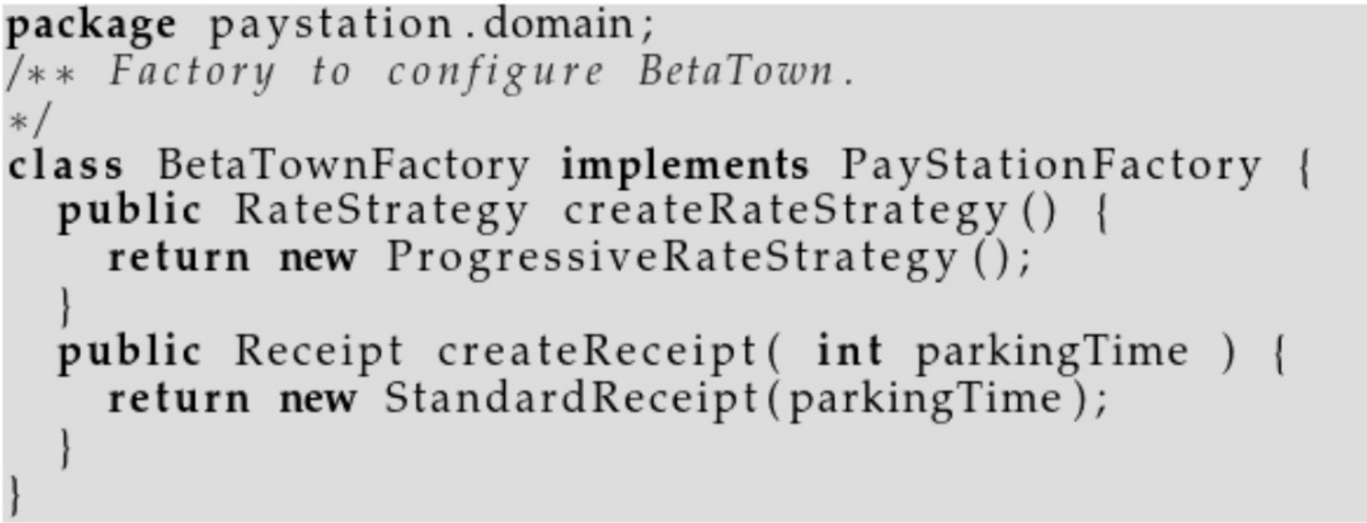
Remember to add a test for GammaTown factory!

TestTownFactory



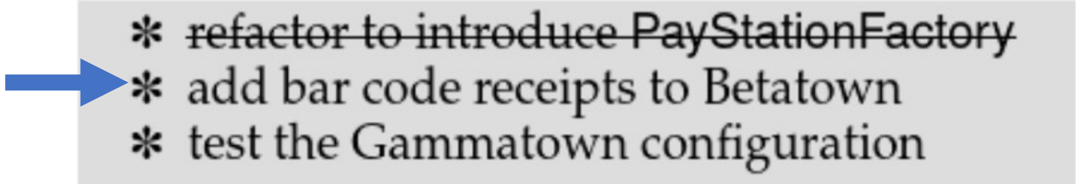
```
package paystation.domain;
/** Factory for making the pay station configuration
    for unit testing pay station behavior.
*/
class TestTownFactory implements Pa
  public RateStrategy createRateStrate
    return new One2OneRateStrategy();
  }
  public Receipt createReceipt( int pa
    return new StandardReceipt(parking
  }
}
```

BetaTownFactory



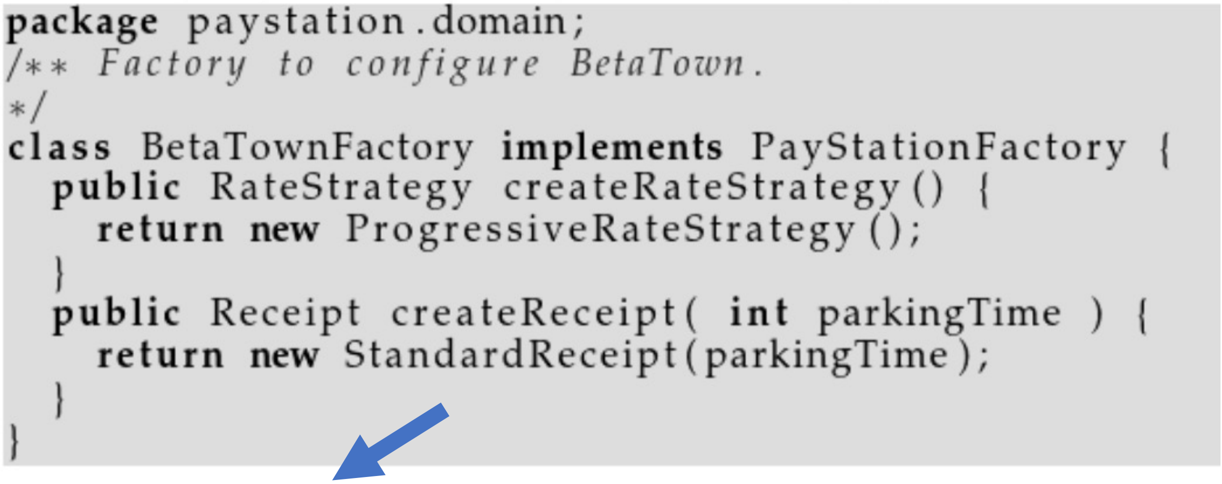
```
package paystation.domain;
/** Factory to configure BetaTown.
*/
class BetaTownFactory implements PayStationFactory {
  public RateStrategy createRateStrategy() {
    return new ProgressiveRateStrategy();
  }
  public Receipt createReceipt( int parkingTime ) {
    return new StandardReceipt(parkingTime);
  }
}
```

New Attempt



✻ ~~refactor to introduce PayStationFactory~~
✻ add bar code receipts to Betatown
✻ test the Gammatown configuration

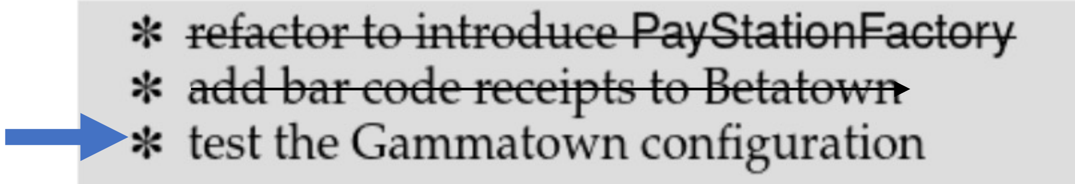
Add bar code receipts to BetaTown



```
package paystation.domain;
/** Factory to configure BetaTown.
*/
class BetaTownFactory implements PayStationFactory {
  public RateStrategy createRateStrategy() {
    return new ProgressiveRateStrategy();
  }
  public Receipt createReceipt( int parkingTime ) {
    return new StandardReceipt(parkingTime);
  }
}
```

BarcodeReceipt

New Attempt



✻ ~~refactor to introduce PayStationFactory~~
✻ ~~add bar code receipts to Betatown~~
✻ test the Gammatown configuration

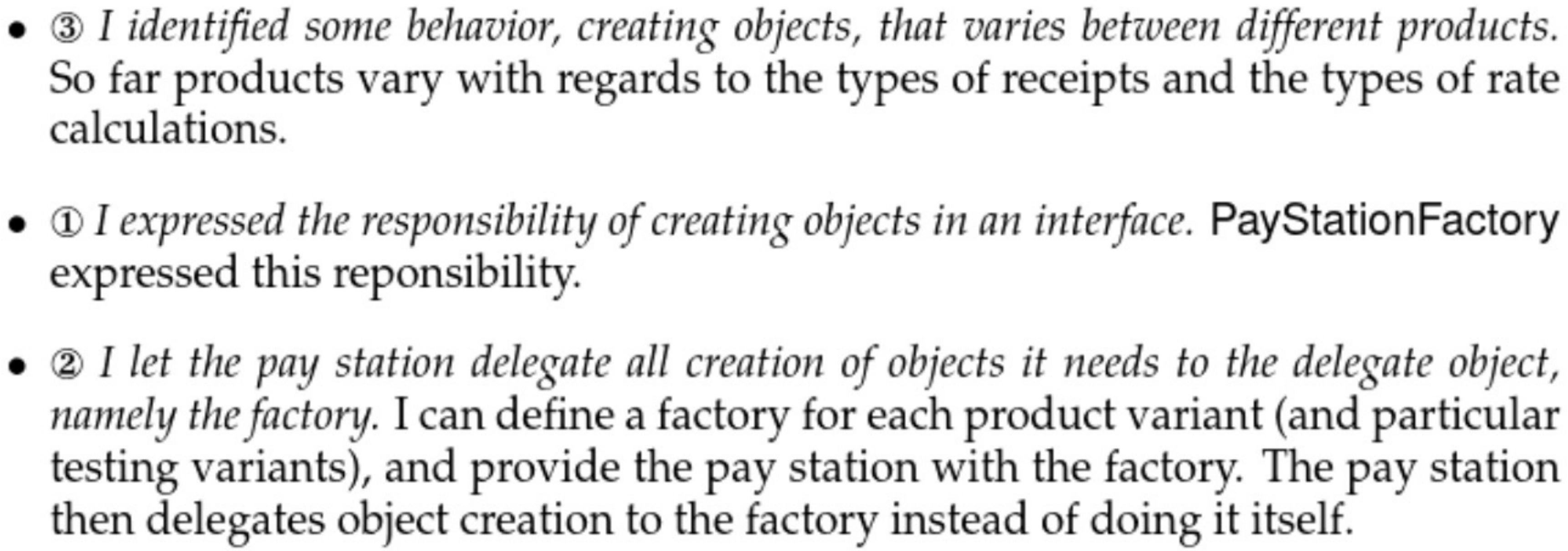
Integration tests

- Need a test for GammaTown – rate strategy and receipt type
- Also need to add to tests for AlphaTown and BetaTown to include receipts
  - Add to test list

Abstract Factory

The **Abstract Factory** pattern is a solution to the problem of creating variable types of objects.

→ Also consistent with compositional design



- ③ *I identified some behavior, creating objects, that varies between different products.* So far products vary with regards to the types of receipts and the types of rate calculations.
- ① *I expressed the responsibility of creating objects in an interface.* PayStationFactory expressed this reponsibility.
- ② *I let the pay station delegate all creation of objects it needs to the delegate object, namely the factory.* I can define a factory for each product variant (and particular testing variants), and provide the pay station with the factory. The pay station then delegates object creation to the factory instead of doing it itself.

Abstract Factory

The goal of Abstract Factory is to provide an **interface** for creating families of related or dependent objects **without specifying their concrete classes.**

Dependency inversion: high-level modules should only depend on interfaces, not low-level implementations

- Production code that is common to all pay station variants only collaborates with delegates through their **interfaces** (does not depend on concrete types)

Abstract Factory

The goal of Abstract Factory is to provide an **interface** for creating families of related or dependent objects **without specifying their concrete classes.**

Dependency inversion: high-level modules should only depend on interfaces, not low-level implementations

- Production code that is common to all pay station variants only collaborates with delegates through their **interfaces** (does not depend on concrete types)

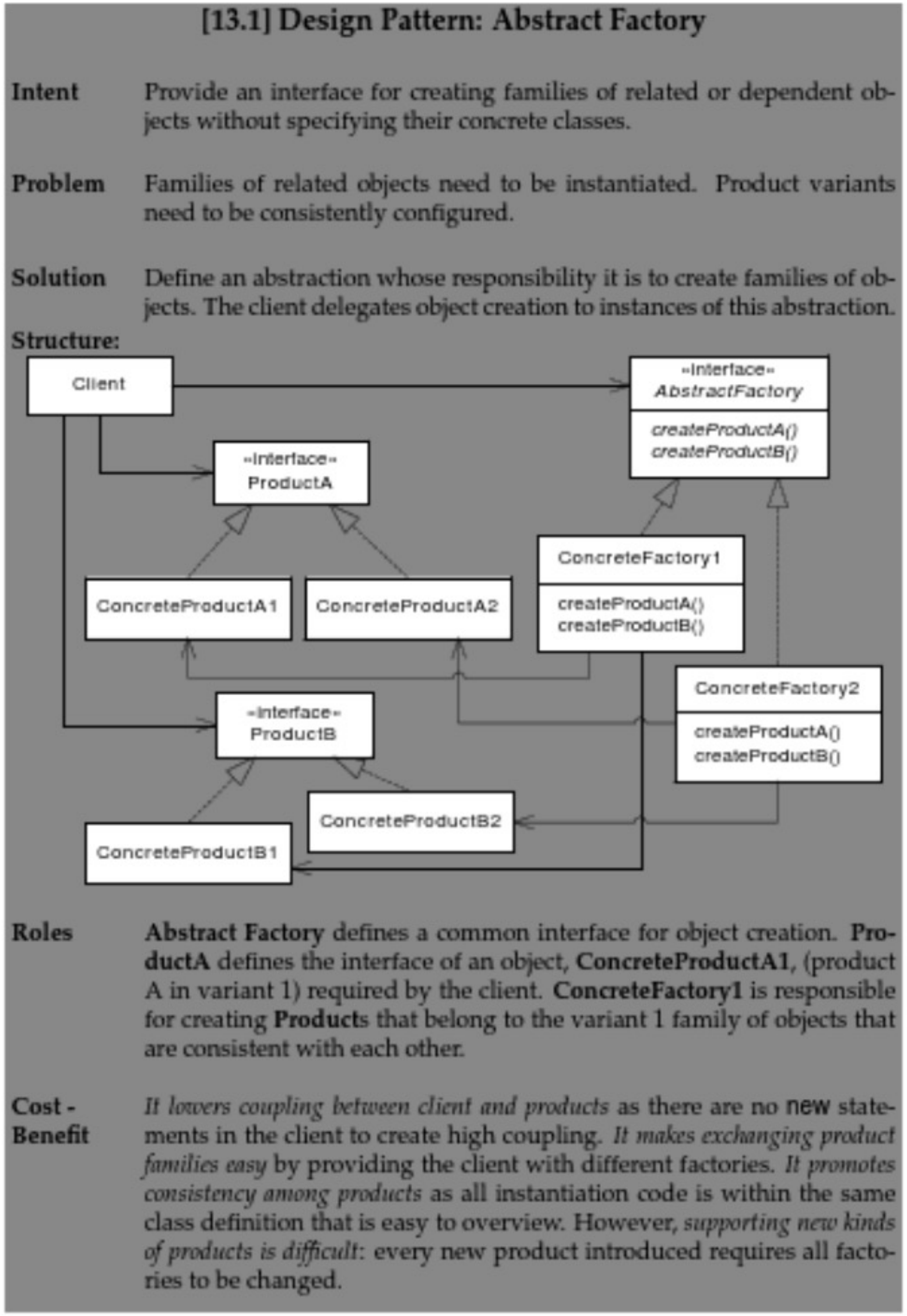
Dependency injection: dependencies should be established by client objects

- The factory object creates concrete objects to be used by the pay station
- The factory is passed into the pay station constructor

Abstract Factory

The goal of Abstract Factory is to provide an interface for creating families of related or dependent objects without specifying their concrete classes

- The **client** must create a consistent set of **products**
  - Client: pay station
  - Products: receipts, rate strategies



[13.1] Design Pattern: Abstract Factory

Intent Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

Problem Families of related objects need to be instantiated. Product variants need to be consistently configured.

Solution Define an abstraction whose responsibility it is to create families of objects. The client delegates object creation to instances of this abstraction.

Structure:

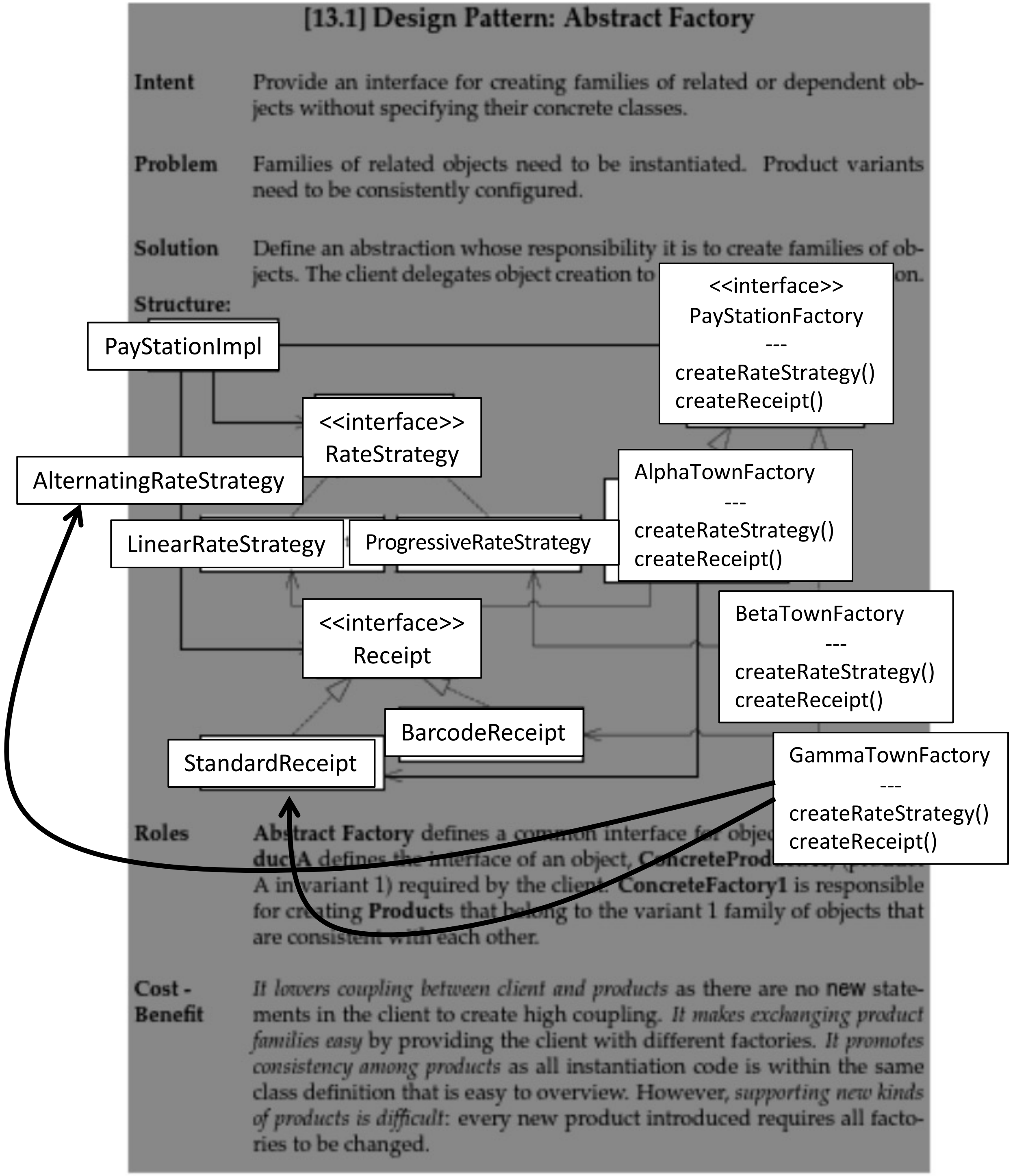
Roles **Abstract Factory** defines a common interface for object creation. **ProductA** defines the interface of an object, **ConcreteProductA1**, (product A in variant 1) required by the client. **ConcreteFactory1** is responsible for creating **Products** that belong to the variant 1 family of objects that are consistent with each other.

Cost - Benefit *It lowers coupling between client and products* as there are no new statements in the client to create high coupling. *It makes exchanging product families easy* by providing the client with different factories. *It promotes consistency among products* as all instantiation code is within the same class definition that is easy to overview. However, *supporting new kinds of products is difficult*: every new product introduced requires all factories to be changed.

Abstract Factory

The goal of Abstract Factory is to provide an interface for creating families of related or dependent objects without specifying their concrete classes

- The **client** must create a consistent set of **products**
  - Client: pay station
  - Products: receipts, rate strategies



[13.1] Design Pattern: Abstract Factory

Intent Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

Problem Families of related objects need to be instantiated. Product variants need to be consistently configured.

Solution Define an abstraction whose responsibility it is to create families of objects. The client delegates object creation to [illegible] on.

Structure:

Roles **Abstract Factory** defines a common interface for objec[illegible] ductA defines the interface of an object, **ConcreteProd**[illegible] A in variant 1) required by the client. **ConcreteFactory1** is responsible for creating **Products** that belong to the variant 1 family of objects that are consistent with each other.

Cost - Benefit *It lowers coupling between client and products* as there are no new statements in the client to create high coupling. *It makes exchanging product families easy* by providing the client with different factories. *It promotes consistency among products* as all instantiation code is within the same class definition that is easy to overview. However, *supporting new kinds of products is difficult*: every new product introduced requires all factories to be changed.

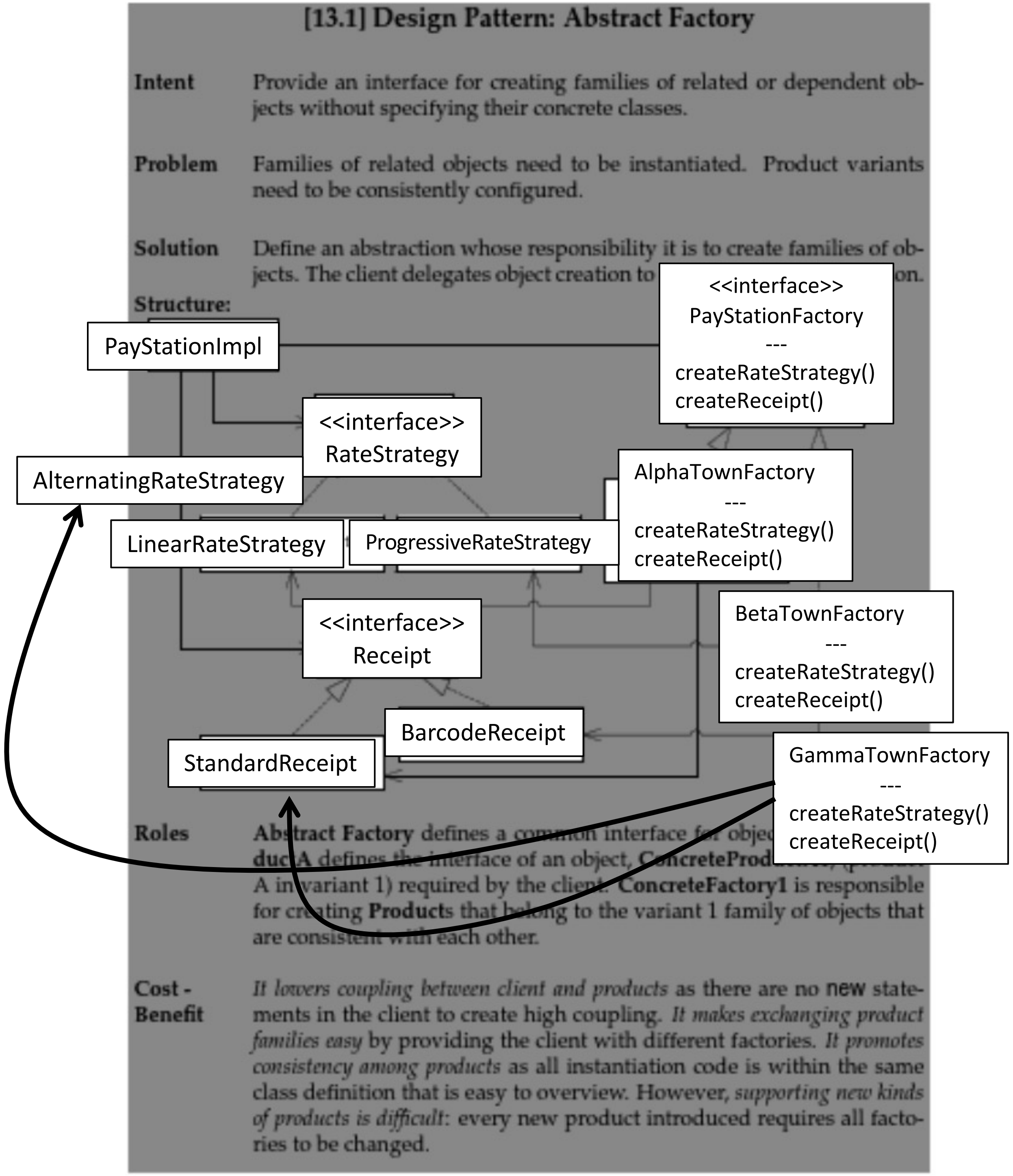
Abstract Factory

The goal of Abstract Factory is to provide an interface for creating families of related or dependent objects without specifying their concrete classes

- The **client** must create a consistent set of **products**
    - Client: pay station
    - Products: receipts, rate strategies

Strategy and State are **behavioral** patterns

Abstract Factory is a **creational** pattern



[13.1] Design Pattern: Abstract Factory

Intent Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

Problem Families of related objects need to be instantiated. Product variants need to be consistently configured.

Solution Define an abstraction whose responsibility it is to create families of objects. The client delegates object creation to on.

Structure:

Roles **Abstract Factory** defines a common interface for objec duc A defines the interface of an object, **ConcreteProd** A in variant 1) required by the client. **ConcreteFactory1** is responsible for creating **Products** that belong to the variant 1 family of objects that are consistent with each other.

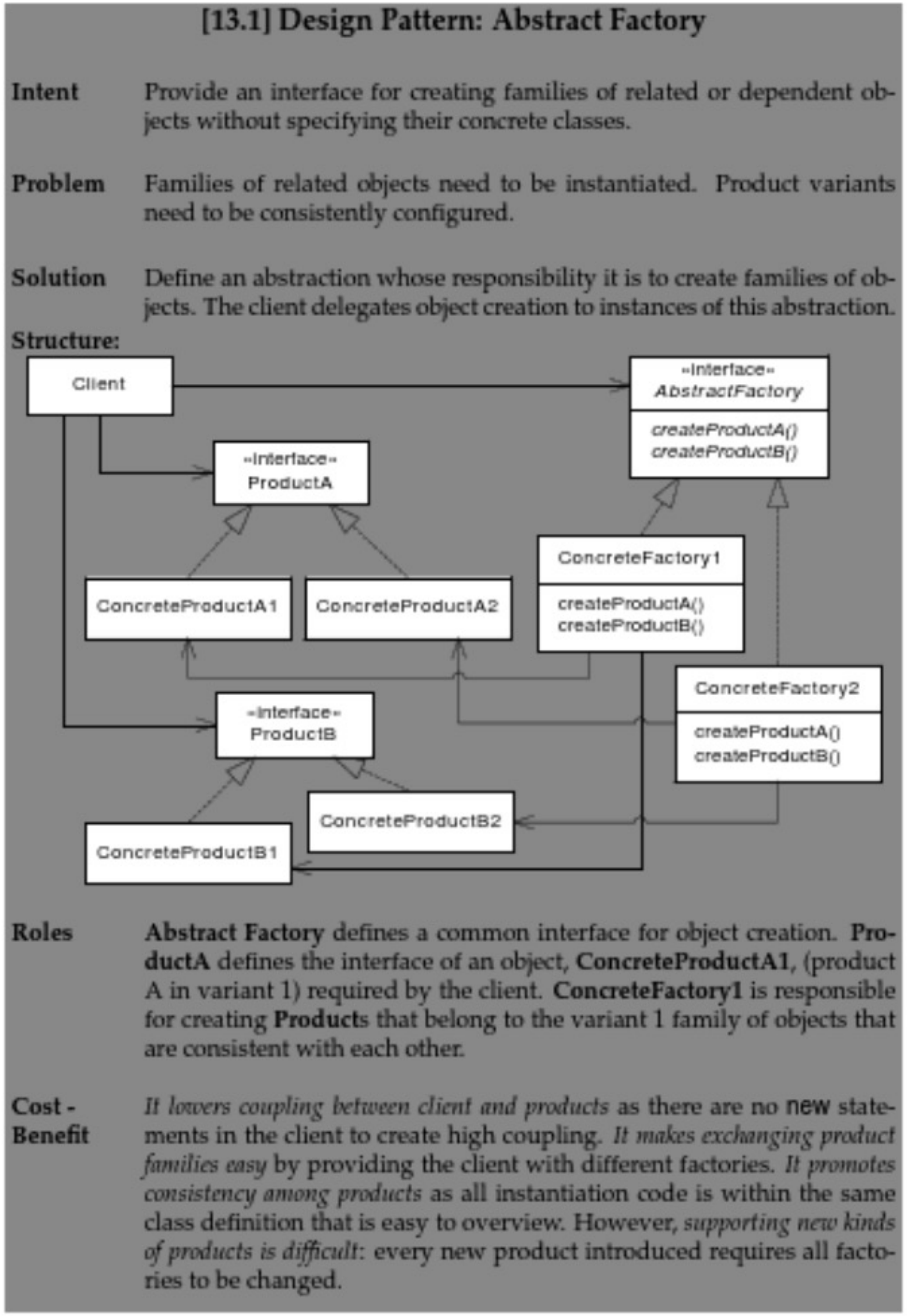
Cost - Benefit *It lowers coupling between client and products* as there are no new statements in the client to create high coupling. *It makes exchanging product families easy* by providing the client with different factories. *It promotes consistency among products* as all instantiation code is within the same class definition that is easy to overview. However, *supporting new kinds of products is difficult*: every new product introduced requires all factories to be changed.

Abstract Factory

Pros:

- Low coupling between client and products
  - Only communicate through **interfaces**
- Configuring clients is easy
  - Provide client with the proper **concrete factory**
- Promotes consistency among products
  - Factories **encapsulate configuration**, easier to avoid or track defects
- Change by addition, not modification
  - Easy to introduce **new concrete factories** for **new pay station variants**
- Client constructor parameter list stays intact
  - Client's constructor **only takes the factory object** as its parameter

☺



[13.1] Design Pattern: Abstract Factory

Intent Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

Problem Families of related objects need to be instantiated. Product variants need to be consistently configured.

Solution Define an abstraction whose responsibility it is to create families of objects. The client delegates object creation to instances of this abstraction.

Structure:

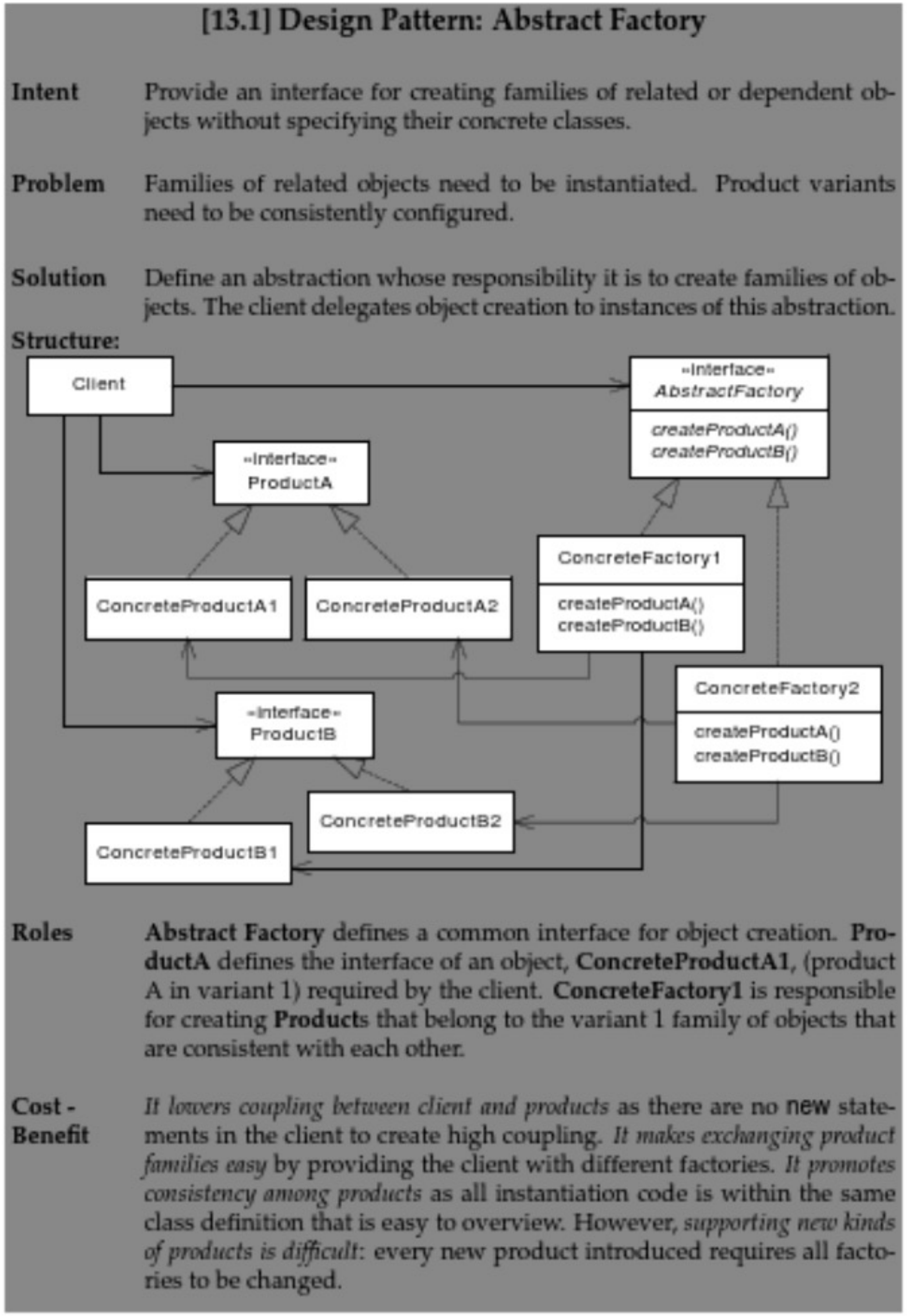
Roles **Abstract Factory** defines a common interface for object creation. **ProductA** defines the interface of an object, **ConcreteProductA1**, (product A in variant 1) required by the client. **ConcreteFactory1** is responsible for creating **Products** that belong to the variant 1 family of objects that are consistent with each other.

Cost - Benefit *It lowers coupling between client and products* as there are no new statements in the client to create high coupling. *It makes exchanging product families easy* by providing the client with different factories. *It promotes consistency among products* as all instantiation code is within the same class definition that is easy to overview. However, *supporting new kinds of products is difficult*: every new product introduced requires all factories to be changed.

Abstract Factory

Cons:

- Introduces extra classes and objects
  - Complex, unnecessary for a single variant
- Introducing new **aspects** of variation is problematic
  - E.g., logging to different databases, accepting other types of payment
  - Need to **modify** the abstract factory interface and **all concrete factory implementations**



[13.1] Design Pattern: Abstract Factory

Intent Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

Problem Families of related objects need to be instantiated. Product variants need to be consistently configured.

Solution Define an abstraction whose responsibility it is to create families of objects. The client delegates object creation to instances of this abstraction.

Structure:

Roles **Abstract Factory** defines a common interface for object creation. **ProductA** defines the interface of an object, **ConcreteProductA1**, (product A in variant 1) required by the client. **ConcreteFactory1** is responsible for creating **Products** that belong to the variant 1 family of objects that are consistent with each other.

Cost - Benefit *It lowers coupling between client and products* as there are no new statements in the client to create high coupling. *It makes exchanging product families easy* by providing the client with different factories. *It promotes consistency among products* as all instantiation code is within the same class definition that is easy to overview. However, *supporting new kinds of products is difficult*: every new product introduced requires all factories to be changed.

Pattern Fragility

Recall: The advantage of patterns is that they have been proven through experience to solve certain problems, with known benefits and liabilities

But, for patterns to be effective, It is important to have the correct implementation of the pattern **structure** and **interactions**!

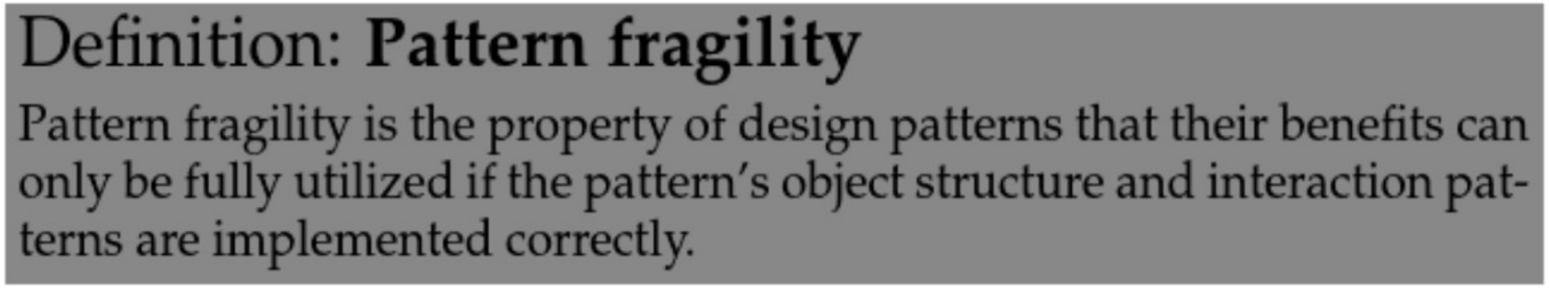
→ Patterns are means to solve problems

Pattern Fragility

Recall: The advantage of patterns is that they have been proven through experience to solve certain problems, with known benefits and liabilities

But, for patterns to be effective, It is important to have the correct implementation of the pattern **structure** and **interactions**!

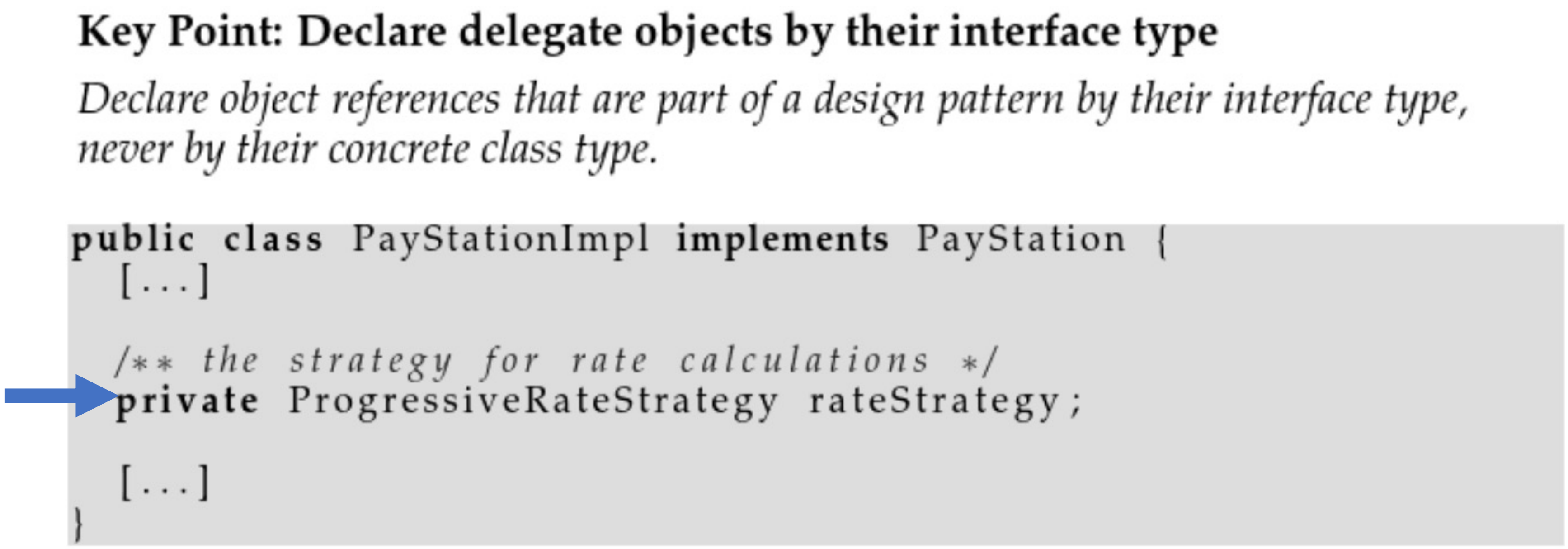
→ Patterns are means to solve problems



Definition: **Pattern fragility**

Pattern fragility is the property of design patterns that their benefits can only be fully utilized if the pattern's object structure and interaction patterns are implemented correctly.

Pattern Fragility



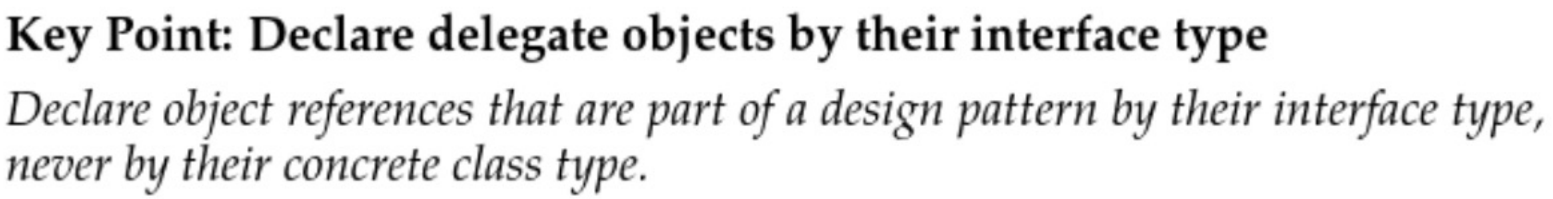
Key Point: Declare delegate objects by their interface type

Declare object references that are part of a design pattern by their interface type, never by their concrete class type.

```
public class PayStationImpl implements PayStation {
  [...]

  /** the strategy for rate calculations */
  private ProgressiveRateStrategy rateStrategy;

  [...]
}
```

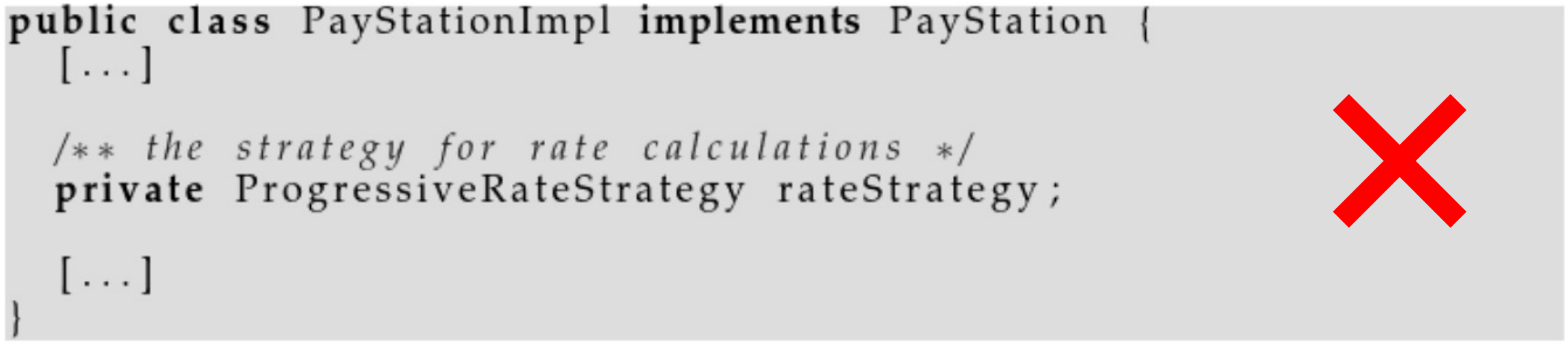
?

Pattern Fragility



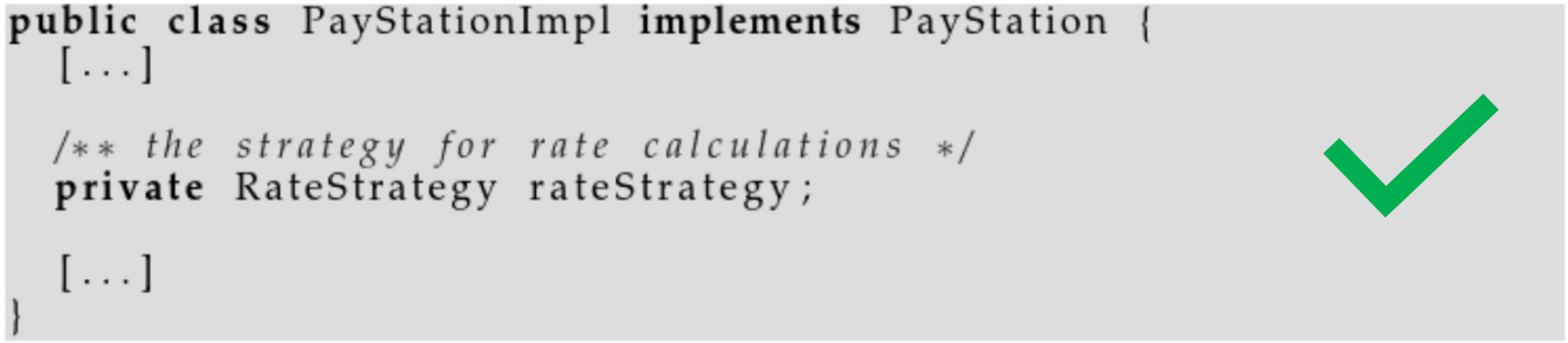
Key Point: Declare delegate objects by their interface type

Declare object references that are part of a design pattern by their interface type, never by their concrete class type.



```
public class PayStationImpl implements PayStation {
  [...]

  /** the strategy for rate calculations */
  private ProgressiveRateStrategy rateStrategy;

  [...]
}
```

✗

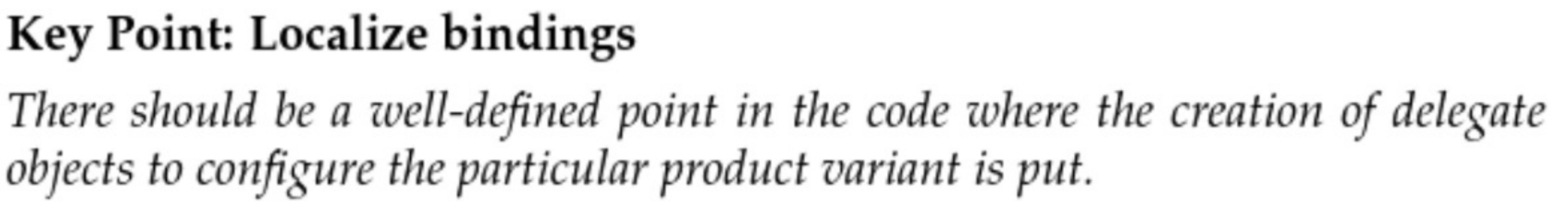


```
public class PayStationImpl implements PayStation {
  [...]

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

  [...]
}
```

✓

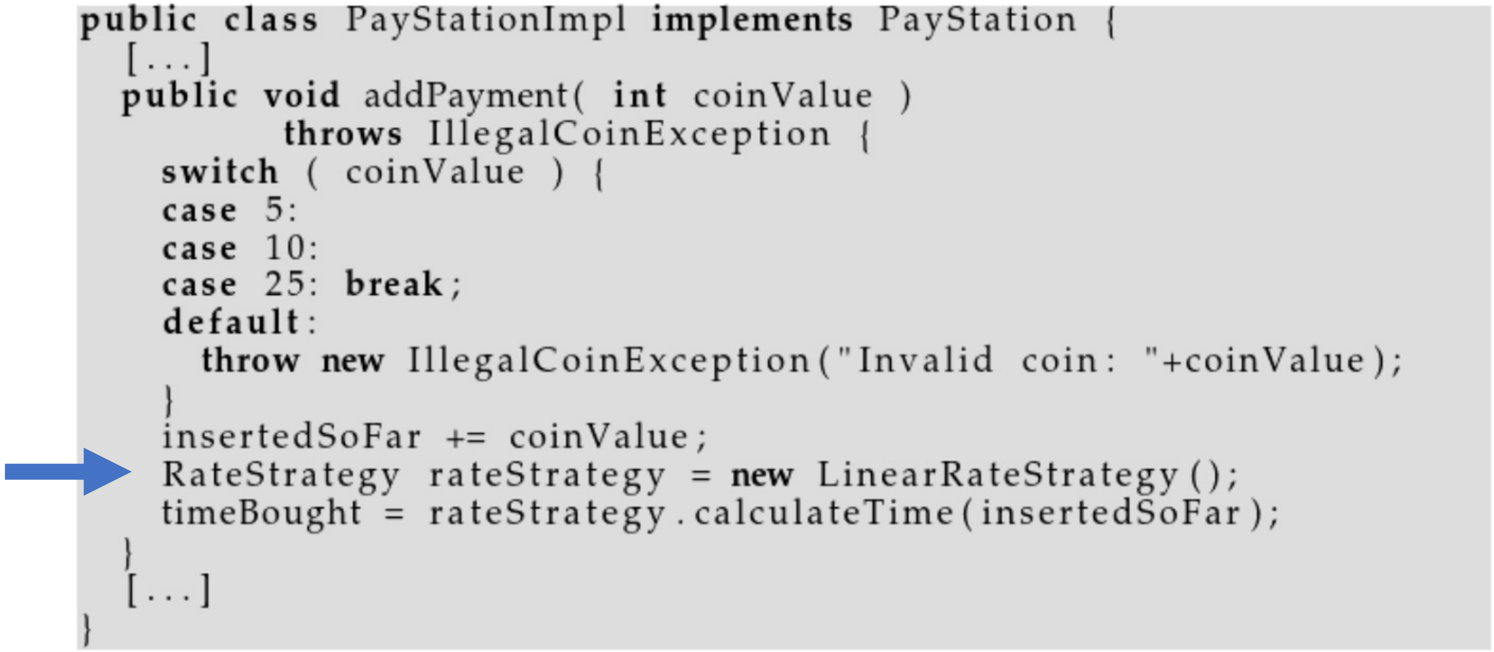
Maintain loose coupling

Pattern Fragility



Key Point: Localize bindings

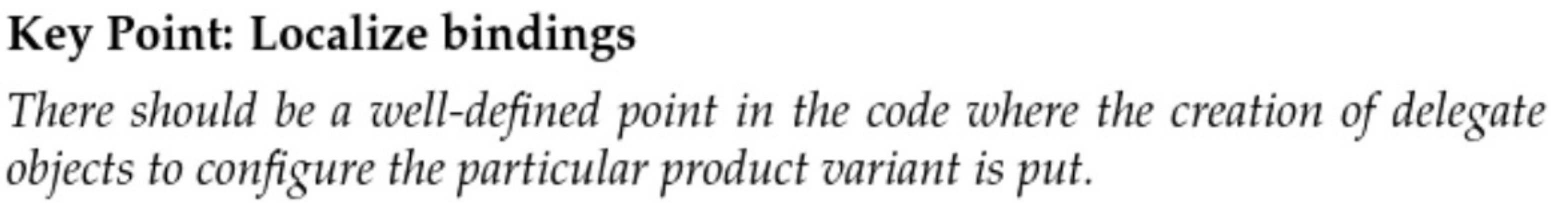
There should be a well-defined point in the code where the creation of delegate objects to configure the particular product variant is put.



```
public class PayStationImpl implements PayStation {
  [...]
  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;
    RateStrategy rateStrategy = new LinearRateStrategy();
    timeBought = rateStrategy.calculateTime(insertedSoFar);
  }
  [...]
}
```

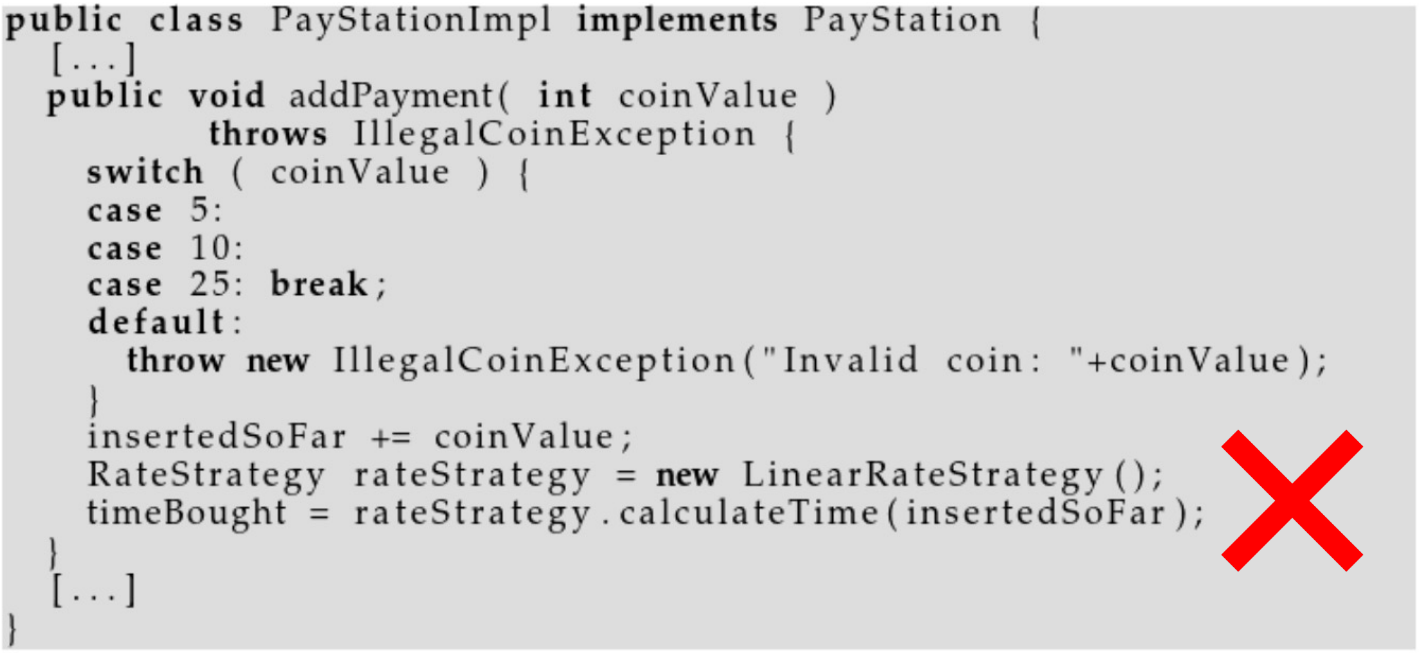
?

Pattern Fragility



Key Point: Localize bindings

There should be a well-defined point in the code where the creation of delegate objects to configure the particular product variant is put.



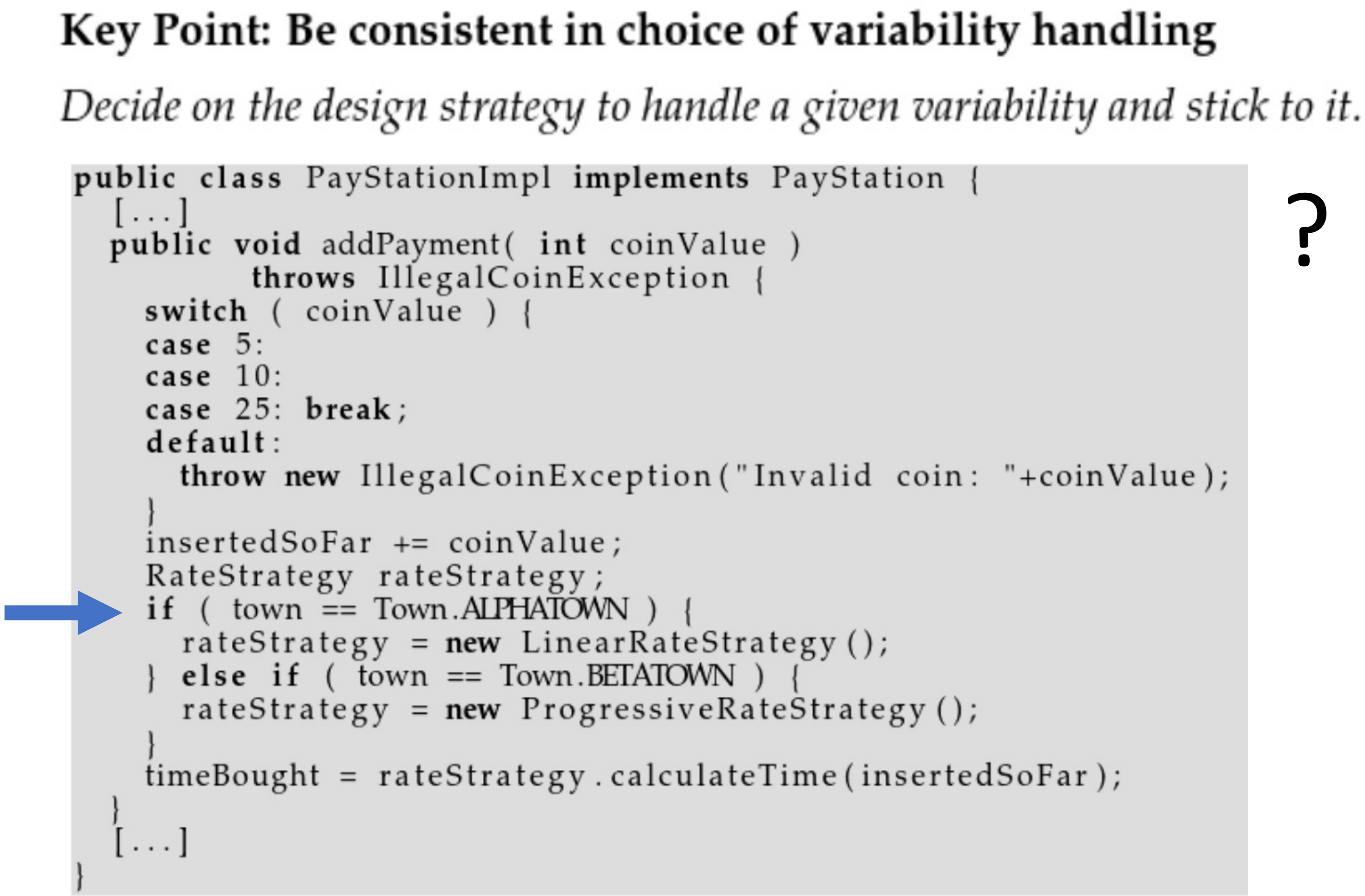
```
public class PayStationImpl implements PayStation {
  [...]
  public void addPayment( int coinValue )
          throws IllegalCoinException {
    switch ( coinValue ) {
    case 5:
    case 10:
    case 25: break;
    default:
      throw new IllegalCoinException("Invalid coin: "+coinValue);
    }
    insertedSoFar += coinValue;
    RateStrategy rateStrategy = new LinearRateStrategy();
    timeBought = rateStrategy.calculateTime(insertedSoFar);
  }
  [...]
}
```

✗

PayStationImpl is shared across variants!

Abstract Factory, or main/application startup ✓

Pattern Fragility



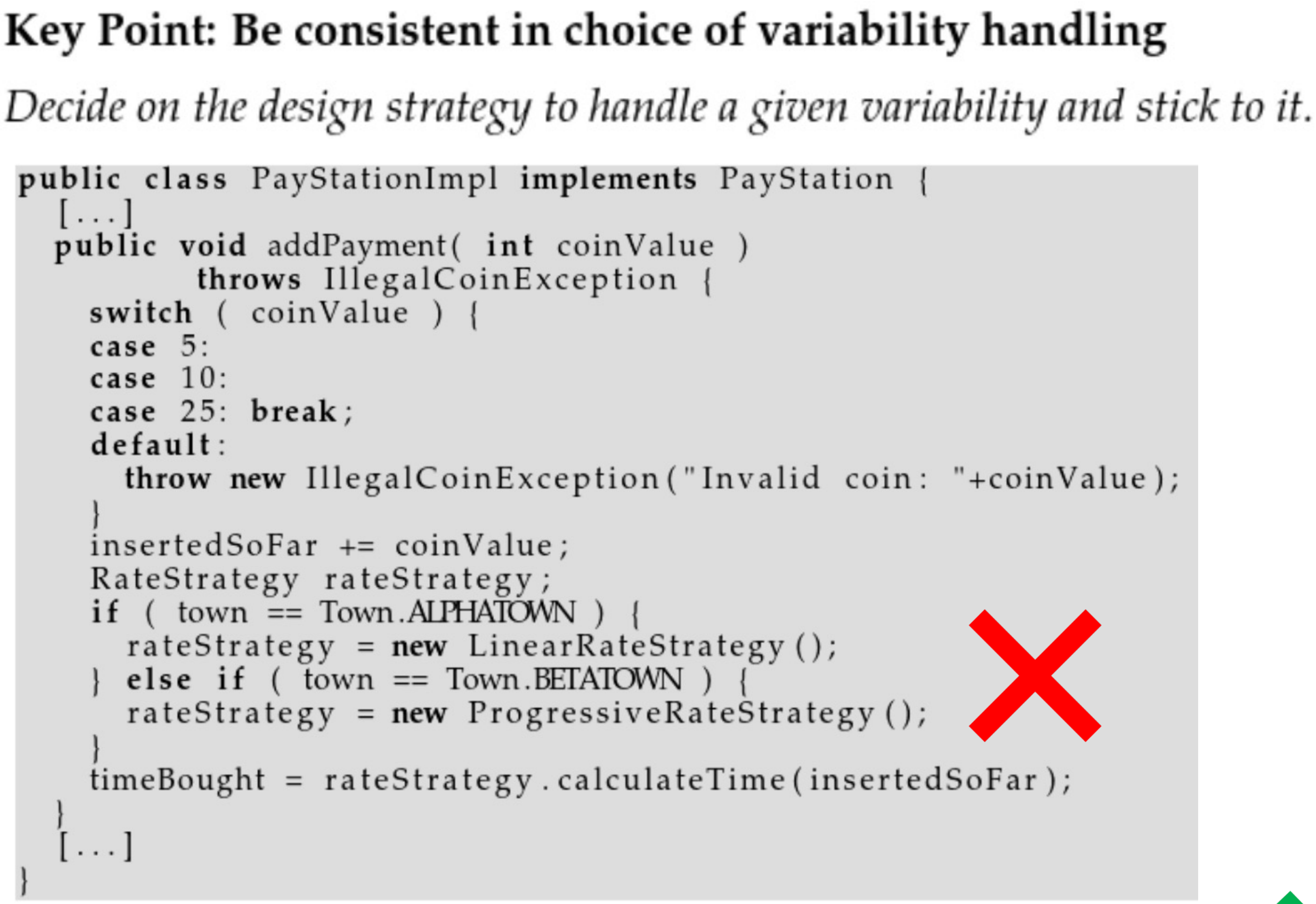
Key Point: Be consistent in choice of variability handling

Decide on the design strategy to handle a given variability and stick to it.

```
public class PayStationImpl implements PayStation {
  [...]
  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;
    RateStrategy rateStrategy;
    if ( town == Town.ALPHATOWN ) {
      rateStrategy = new LinearRateStrategy();
    } else if ( town == Town.BETATOWN ) {
      rateStrategy = new ProgressiveRateStrategy();
    }
    timeBought = rateStrategy.calculateTime(insertedSoFar);
  }
  [...]
}
```

?

Pattern Fragility



Key Point: Be consistent in choice of variability handling

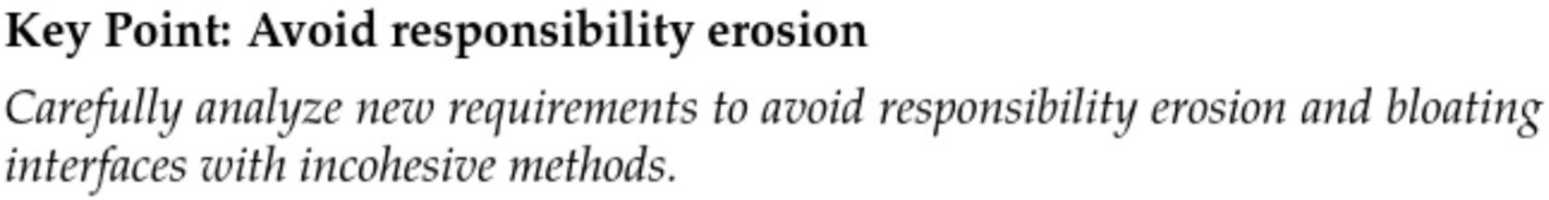
Decide on the design strategy to handle a given variability and stick to it.

```
public class PayStationImpl implements PayStation {
  [...]
  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;
    RateStrategy rateStrategy;
    if ( town == Town.ALPHATOWN ) {
      rateStrategy = new LinearRateStrategy();
    } else if ( town == Town.BETATOWN ) {
      rateStrategy = new ProgressiveRateStrategy();
    }
    timeBought = rateStrategy.calculateTime(insertedSoFar);
  }
  [...]
}
```

Maintain pattern consistency for a given variability ✓

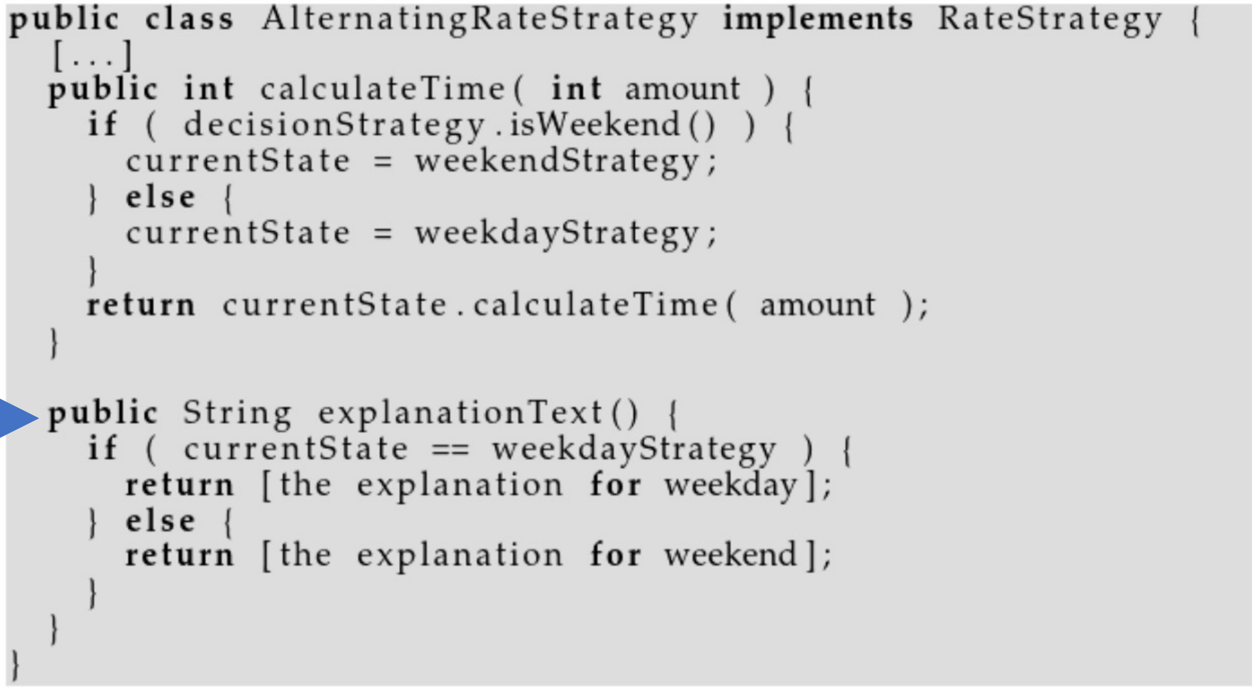
→ Keep benefits, avoid introducing liabilities of multiple patterns

Pattern Fragility



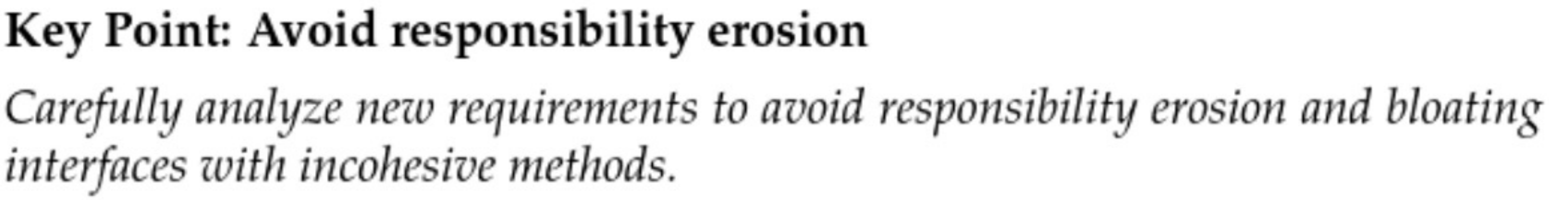
Key Point: Avoid responsibility erosion

Carefully analyze new requirements to avoid responsibility erosion and bloating interfaces with incohesive methods.



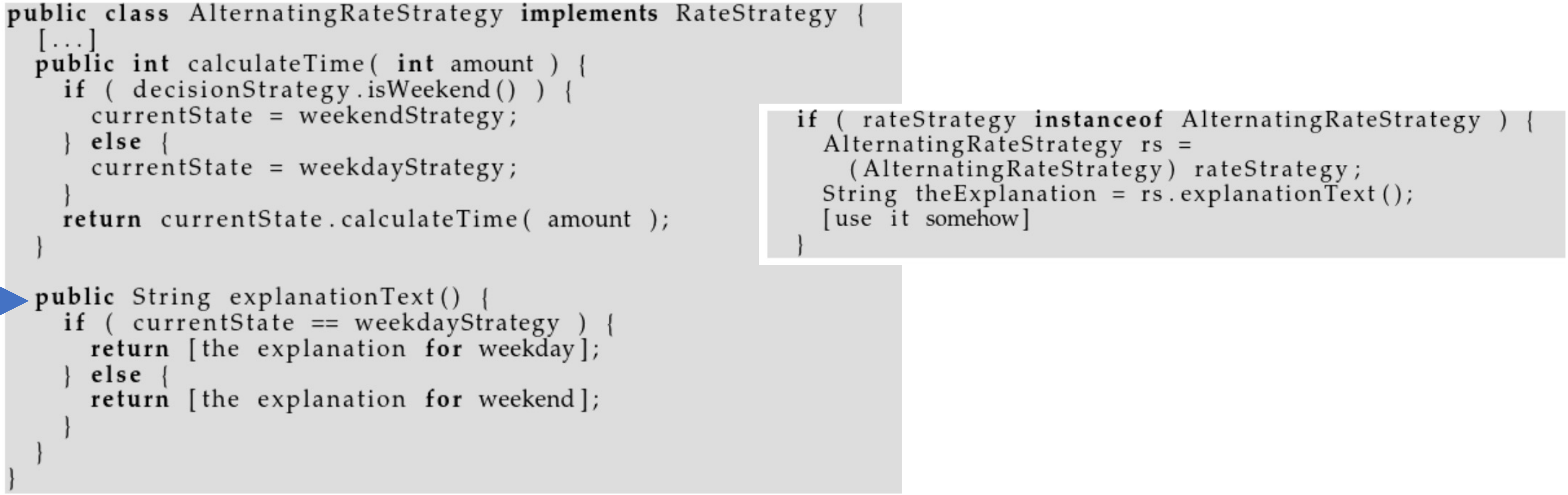
```
public class AlternatingRateStrategy implements RateStrategy {
  [...]
  public int calculateTime( int amount ) {
    if ( decisionStrategy.isWeekend() ) {
      currentState = weekendStrategy;
    } else {
      currentState = weekdayStrategy;
    }
    return currentState.calculateTime( amount );
  }

  public String explanationText() {
    if ( currentState == weekdayStrategy ) {
      return [the explanation for weekday];
    } else {
      return [the explanation for weekend];
    }
  }
}
```

Pattern Fragility



Key Point: Avoid responsibility erosion

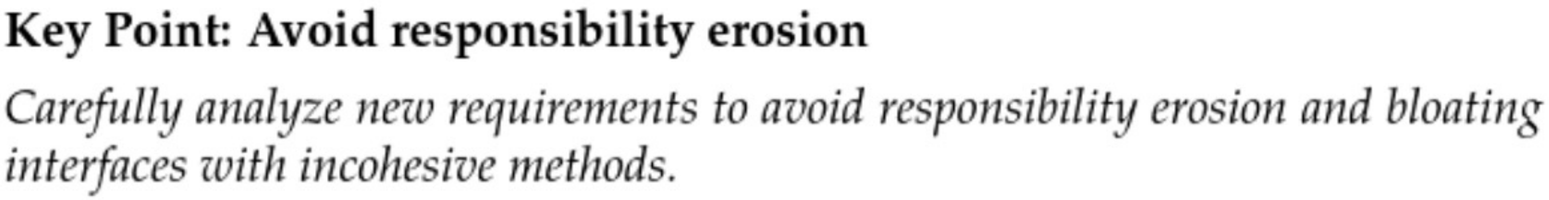
Carefully analyze new requirements to avoid responsibility erosion and bloating interfaces with incohesive methods.



```
public class AlternatingRateStrategy implements RateStrategy {
  [...]
  public int calculateTime( int amount ) {
    if ( decisionStrategy.isWeekend() ) {
      currentState = weekendStrategy;
    } else {
      currentState = weekdayStrategy;
    }
    return currentState.calculateTime( amount );
  }

  public String explanationText() {
    if ( currentState == weekdayStrategy ) {
      return [the explanation for weekday];
    } else {
      return [the explanation for weekend];
    }
  }
}
```

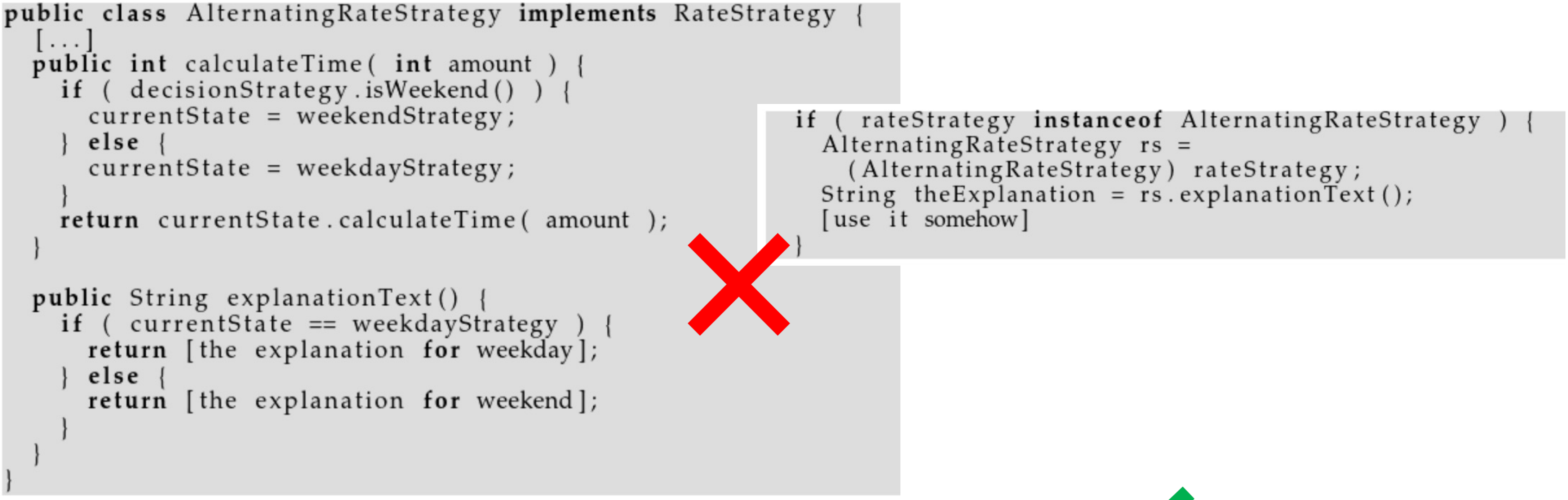
```
if ( rateStrategy instanceof AlternatingRateStrategy ) {
  AlternatingRateStrategy rs =
    (AlternatingRateStrategy) rateStrategy;
  String theExplanation = rs.explanationText();
  [use it somehow]
}
```

Pattern Fragility



Key Point: Avoid responsibility erosion

Carefully analyze new requirements to avoid responsibility erosion and bloating interfaces with incohesive methods.



```
public class AlternatingRateStrategy implements RateStrategy {
  [...]
  public int calculateTime( int amount ) {
    if ( decisionStrategy.isWeekend() ) {
      currentState = weekendStrategy;
    } else {
      currentState = weekdayStrategy;
    }
    return currentState.calculateTime( amount );
  }

  public String explanationText() {
    if ( currentState == weekdayStrategy ) {
      return [the explanation for weekday];
    } else {
      return [the explanation for weekend];
    }
  }
}
```

```
if ( rateStrategy instanceof AlternatingRateStrategy ) {
  AlternatingRateStrategy rs =
    (AlternatingRateStrategy) rateStrategy;
  String theExplanation = rs.explanationText();
  [use it somehow]
}
```

✗

Add responsibilities with intention to maintain cohesion ✓

(this would be better in RateStrategy, but be aware that it is still adding responsibilities there)

Pattern Fragility: Summary

Coding decisions or mistakes can invalidate the benefits of patterns

→ Know the patterns

→ Know the **roles** and **interactions** they involve

→ Know the implications of how they are coded

→ Understand the patterns used in a given piece of code before making changes

- Make sure everyone working on a project understands the design!

Next time: Midterm review