



AARHUS UNIVERSITET

Software Engineering and Architecture

Deriving Abstract Factory

The *dependency injection* pattern



- The Receipt class revisited
 - Add responsibility to print itself

Receipt

- know its value in minutes parking time
- print itself

- Provided by method

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

- Result:

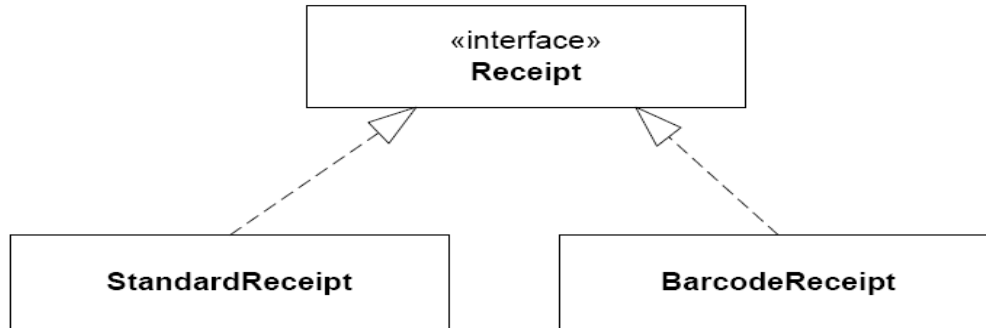

```
-----  
----- P A R K I N G   R E C E I P T -----  
          Value 049 minutes.  
          Car parked at 08:06  
-----
```

Demo: <code-demo/PayStationWithGUI>



- *Change is the only constant in software dev.*
- Req: Receipts with bar code for easy scanning

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

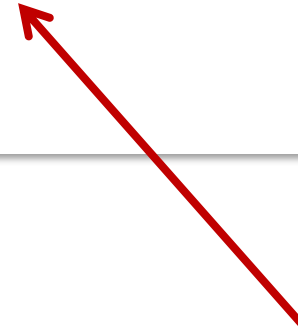




Code View

- Something must be done at:

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



A "new BarCodeReceiptImpl(tb)" in BetaTown



Yet another Variability

- **New variability point!** Resulting configurations:

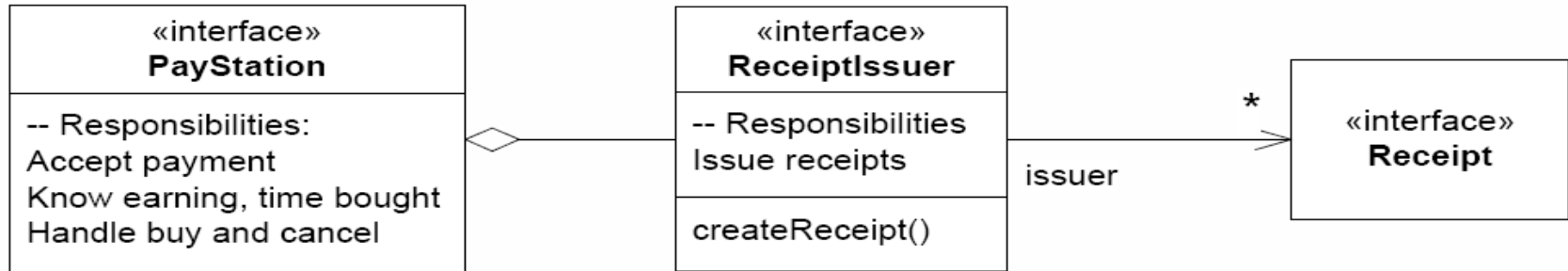
	Variability points	
Product	Rate	Receipt
Alphatown	Linear	Standard
Betatown	Progressive	Barcode
Gammatown	Alternating	Standard



AARHUS UNIVERSITET

The Compositional Design

- Cranking the 3-1-2 *blindly*
 - 3) Identify what varies: *instantiation of receipts*
 - 1) Interface express responsibility: ReceiptIssuer
 - 2) Compose behavior: delegate to ReceiptIssuer





Trying it out

- Quickly add a test:

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

- **Low cohesion:** *object creation in two different objects – why not make one cohesive object???*

TDD Principle: **Do Over**

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

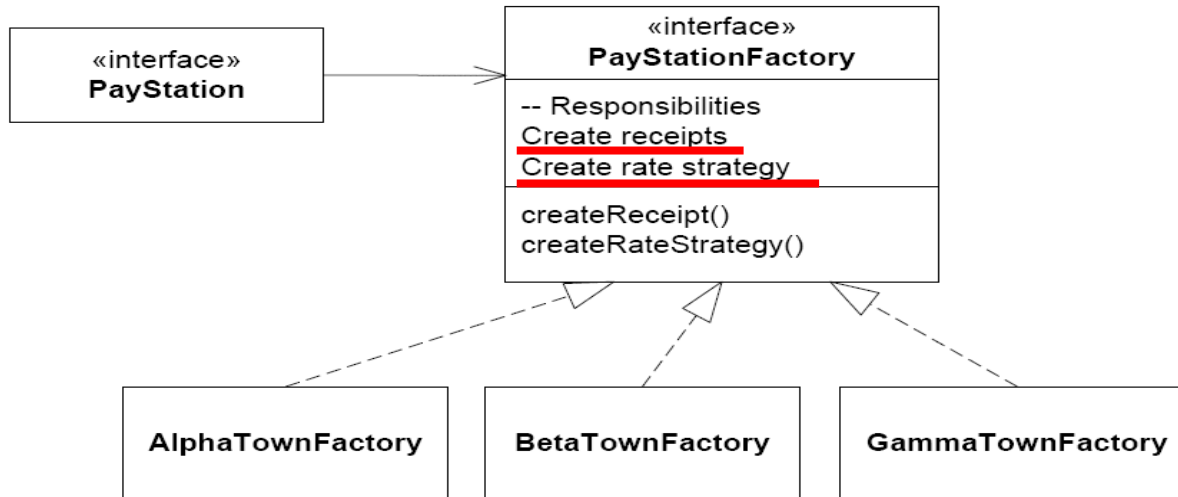


More Cohesive Design

- **One** place to “create delegates”: the **factory**

PayStationFactory

- Create receipts
- Create rate strategies





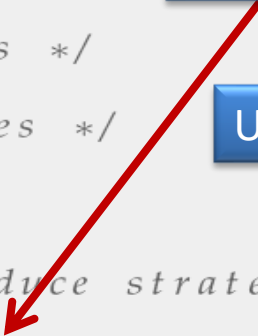
Usage

```
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
     */
    public PayStationImpl( PayStationFactory factory ) {
        this.factory = factory;
        this.rateStrategy = factory.createRateStrategy();
        reset();
    }
    [...]
    public Receipt buy() {
        Receipt r = factory.createReceipt(timeBought);
        reset();
        return r;
    }
    [...]
}
```

Only one delegate!

Uncle Bob is happy





```
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 );
}
```

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

- Why TestTownFactory()?
 - It creates a *One2OneRateStrategy* which makes my pay station core code more *evident data + evident test*



TestTownFactory

```
class TestTownFactory implements PayStationFactory {  
  
    public RateStrategy createRateStrategy() {  
        return new One2OneRateStrategy();  
    }  
    public Receipt createReceipt( int parkingTime ) {  
        return new StandardReceipt(parkingTime);  
    }  
}
```



To Configure BetaTown

- The factory for BetaTown

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

```
class BetaTownFactory implements PayStationFactory {
    public RateStrategy createRateStrategy() {
        return new ProgressiveRateStrategy();
    }
    public Receipt createReceipt( int parkingTime ) {
        return new StandardReceipt(parkingTime, true);
    }
}
```

Important benefit: *all configuration is contained in one java source file!*

Why? Cohesion – do the same thing in the same place!

- Huh? No 'barcode' receipt subclass? No composition?

```
class BetaTownFactory implements PayStationFactory {  
    public RateStrategy createRateStrategy() {  
        return new ProgressiveRateStrategy();  
    }  
    public Receipt createReceipt( int parkingTime ) {  
        return new StandardReceipt(parkingTime, true);  
    }  
}
```

But Uncle Bob again
is unhappy 😞

- Only two variants envisioned!
 - **Parametric variability much simpler = the right variability management choice!**



AARHUS UNIVERSITET

The Compositional Process



The Process Again Again

- 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 responsibility.*
- 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.*



Definition: **Dependency inversion principle**

High level modules should not depend upon low level modules. Both should depend upon abstractions. Abstractions should not depend upon details. Details should depend upon abstractions. (Martin 1996)

Who is this Martin guy?

Definition: **Dependency injection**

High-level, common, abstractions should not themselves establish dependencies to low level, implementing, classes, instead the dependencies should be established by injection, that is, by client objects. (Fowler 2004)



AARHUS UNIVERSITET

Abstract Factory



- 3-1-2 has derived yet another design pattern
 - An object (factory) whose responsibility it is to create objects (products) that the client need.

Deriving it...

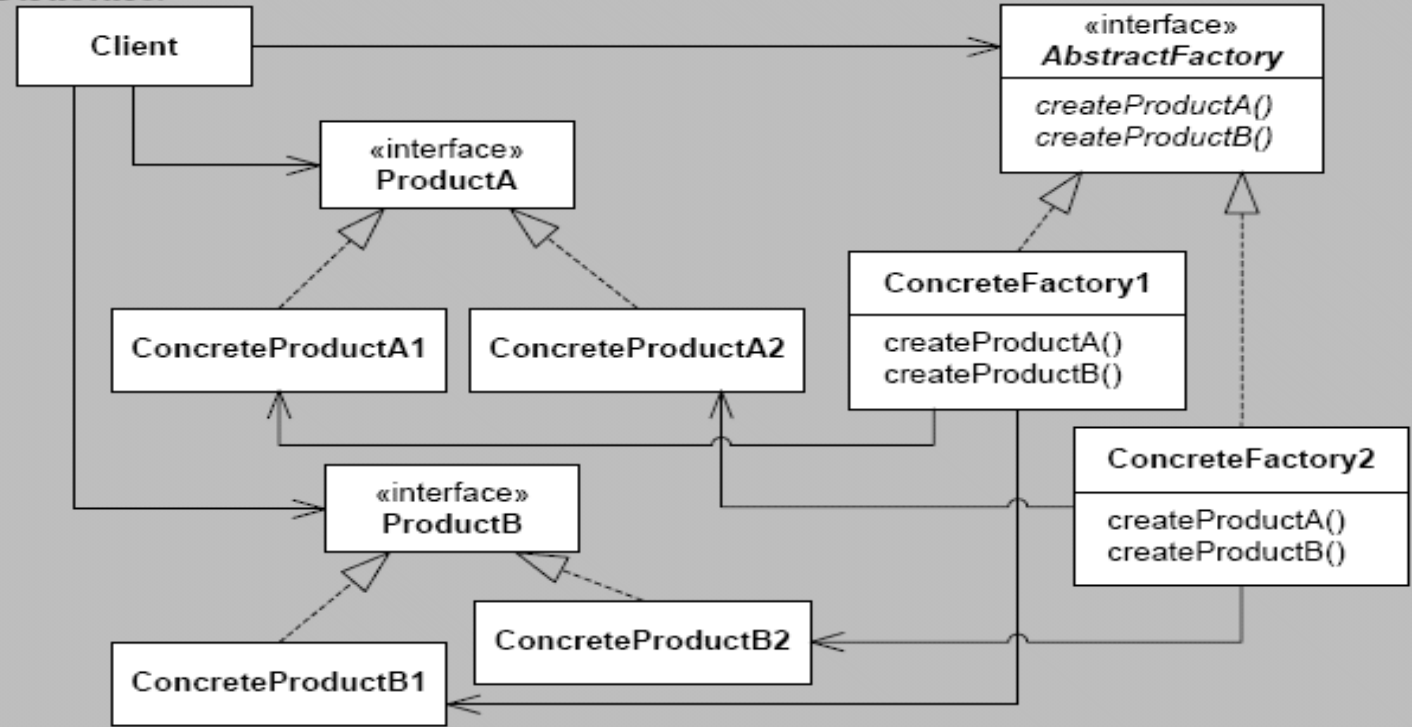
[13.1] Design Pattern: Abstract Factory

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	<i>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.</i>

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



Mandatory Note

- Abstract Factory is complex
 - Lots of relations
 - Easy to get confused or miss the whole idea
 - Easy to mis-implement a little with BIG consequences
- Morale
 - Do not underestimate it in the Mandatory...