If you look over the production code that is common to all pay station variants, you will see that it *only* refers to, and collaborates with, delegates through their interfaces. In no place does it create objects, cast references to concrete types, nor declare instance variables by a class type. Therefore, it does not depend on concrete types at any level. As argued, this leads to a low coupling and this principle is so important that it has received its own term.

> 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)

Rephrasing the definition for our case, it states that the pay station code shared by all variants (the "high level modules") should not depend on the variant's implementation (the "low level modules"), but only on interfaces (the abstractions).

Of course, some part of the code then has to make the actual binding, i.e. tell the high level modules the actual low level modules to communicate with. This binding process also has a name:

> 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)

In our case, the factory object is asked to create concrete objects to be used by the pay station, thus the factory injects the dependencies. Dependency injection is central in frameworks as discussed in Chapter 32.

13.6 Abstract Factory

The intent of the ABSTRACT FACTORY is to *Provide an interface for creating families of related or dependent objects without specifying their concrete classes.* In our pay station case, it is the family of objects that determines the variant of our pay station product. The general structure and condensed presentation of the pattern is given in design pattern box 13.1 on page 62. The central roles in abstract factory is the **client** that must create a consistent set of **products**. In our case, the client is the pay station and the products are receipts and rate strategies. Instead of creating the products itself, the role **abstract factory** does it. In the classic design pattern book, *Design Patterns: Elements of Reusable Object-Oriented Software* (Gamma et al. 1995), the two previous design patterns, STRATEGY and STATE, are classified as *behavioral* design patterns while ABSTRACT FACTORY is classified as *creational*. The intention of creational patterns is of course to *create objects*. Behavioral patterns deal with variability in the behavior of the system and the use of state and strategy in the pay station are clear examples of this.

The benefits of using an abstract factory in our designs are:

- *Low coupling between client and products.* There are no new statements in the client to make a high coupling to particular product variants. Thus the client only communicates with its products through their interface.
- *Configuring clients is easy.* All you have to do is to provide the client with the proper concrete factory. In our case, the pay station is configured simply by passing the proper pay station factory object during construction.
- *Promoting consistency among products.* Locality is important in making software easy to understand and maintain. The individual factories encapsulate the pay station's configuration: To review how a Betatown configuration looks like, you know that the place to look for it is in BetaTownFactory—and no other place. Thus any defects in configuration are traceable to a single class in the system. And as you are not required to look and change in five or ten different places in your code in order to make a configuration, the risk of introducing defects is substantially lessened.
- *Change by addition, not by modification.* I can introduce new pay stations product types easily as I can add new rate strategies and new receipt types, and provide the pay station with these by defining a new factory for it. Existing pay station code remains unchanged giving higher reliability and easier maintenance. Note, however, this argument is only true when it comes to introducing new *variants* of the existing rate strategies and receipt types! See the liability discussion below…
- *Client constructor parameter list stays intact.* Even if I need to introduce new products in the client, the client's constructor will still only take a single factory object as its parameter. Compare this to the practice of adding more and more configuration parameters to the constructor list.

Of course, ABSTRACT FACTORY has liabilities, some of which should be well known to you by now.

- *It introduces extra classes and objects.* Compared to a parametric solution where decision making is embodied in the client code I instead get several new interfaces and classes and thus objects: the factory interface as well as implementation classes. Therefore the pattern requires quite a lot of coding—you should certainly not use it if you only have a single type of product.
- *Introducing new aspects of variation is problematic.* The problem with abstract factory is that if I want to introduce new aspects that must be varied (logging to different vendor's databases, accepting other types of payment, etc.) then the abstract factory interface must be augmented with additional create-methods, and all its subclasses changed to provide behavior for them. This is certainly *change by modification*.

A classical example of the use of abstract factory is the Java Swing graphical user interface toolkit. A major problem for Java is that it runs on a variety of platforms including Windows, Macintosh, and Unix. However, Swing allows you to write a graphical user interface form (a dialog with text fields, radio buttons, drop-down list boxes, etc.) that will run on all the supported platforms. Thus to instantiate an

Swing button the Java run-time library has to make a decision whether to create a Win32 button, a Mac button, or a Unix windows manager button; to instantiate a list box is has to make a similar decision; etc. Thus, the source code could be thick with zillions of if-statements—but it is all handled elegantly by an ABSTRACT FACTORY. The Swing code delegates any instantiation request for a graphical user component to its associated factory. The factory has methods for creating buttons, list boxes, radio buttons, text fields, etc. Each concrete factory, one for Win32, one for Mac, etc., then creates the proper graphical component from each platform's graphical toolbox.

In this light, it is also fair to say that the use of a factory to create the rate strategy in the pay station is sort of a special case because it only happens initially in contrast to receipts that are created continuously. Usually, abstract factory is considered the solution to the "continuous creation" problem. Still, it makes sense to group all object creation responsibility into a single abstraction as I have done in the pay station.

13.7 Summary of Key Concepts

Object-oriented systems need to create objects. However, when you strive to make a loosely coupled design you face the problem that the new statement expresses the tightest coupling possible: you once and for all bind an object reference to a particular concrete class.

```
Receipt r = new StandardReceipt(30);
```

While the first part of this statement `Receipt r` is loosely coupled as you only rely on the interface type Receipt the exact opposite is true for the right hand side of the assignment. The creational pattern ABSTRACT FACTORY provides a compositional solution to the problem of loosening the coupling between a client and the concrete products it needs to create, by delegating the creation responsibility to an instance of a factory. Furthermore the factory becomes a localized class responsible for configuration leading to a design that promotes consistency among products. A liability of the pattern is that it is somewhat complex and requires quite a lot of coding.

This chapter also showed that the ③-①-② process should not be used mechanically without considering the resulting design's cohesion and coupling. Always consider how the existing design could be refactored in such a way that the overall compositional design has high cohesion and low coupling. In our case, the creation of rate strategies had to be included in the analysis.

13.8 Selected Solutions

Discussion of Exercise 13.1:

```
/** Test that the receipt's show method prints proper info */
@Test public void shouldPrintReceiptsCorrectly() {
  Receipt receipt = new ReceiptImpl(30);
  // Prepare a PrintStream instance that lets me inspect the
  // data written to it.
  ByteArrayOutputStream baos = new ByteArrayOutputStream();
```

```
    PrintStream ps = new PrintStream(baos);
    // let the 30 minute print itself
    receipt.print(ps);

    // get the string printed to the stream
    String output = baos.toString();
    // split the string into individual lines
    String[] lines = output.split("\n");
    // test to see that the receipt consist of five lines
    assertEquals( 5, lines.length );
    // test parts of the contents
    assertEquals( "---", lines[0].substring(0,3) );
    assertEquals( "---", lines[4].substring(0,3) );
    assertEquals( "P A R K I N G", lines[1].substring(9,22) );
    // test the receipt's value
    assertEquals( "030", lines[2].substring(22,25) );
    // test that the format of the "parking starts at" time
    // is plausible
    String parkedAtString = lines[3].substring(28,33);
    assertEquals( ':', parkedAtString.charAt(2) );
    // if the substring below is not an integer a
    // NumberFormatException is thrown which will
    // make JUnit fail this test
    Integer.parseInt( parkedAtString.substring(0,2) );
    Integer.parseInt( parkedAtString.substring(3,5) );
  }
```

This is a plausible test case for print, but a major problem with this test code is that it is longer than the corresponding production code thus there is a high probability that the defects are in the testing code.

Discussion of Exercise 13.2:

The testing code includes also the use of the One2OneRateStrategy, and the testing of Gammatown actually includes an additional dimension of variance, namely the choice of strategy to determine if it is weekend or not. Therefore a complete table will include these aspects.

	Variability points		
Product	**Rate**	**Receipt**	**Weekend**
Alphatown	Linear	Standard	–
Betatown	Progressive	Barcode	–
Gammatown	Alternating	Standard	Clock
PayStation unit test	One2One	Standard	–
Gammatown rate unit test	Alternating	–	Fixed

Here "–" means that the selection is not applicable.

Discussion of Exercise 13.3:

The only difference in the two receipt types is in one additional line printed. A compositional approach would encapsulate the responsibility of outputting this line in an interface and use delegation to print it. Something along of the following line embedded in the receipt's print method:

```
[...]
stream.println("              Car parked at "+nowstring);
additionalInfoPrinter.print(stream);
stream.println("-----------------------------------------------");
```

One concrete implementation would then print nothing and the other print the bar code. The problem I see with this solution is that I do not see viable future variations: what is the obvious new requirement for something new to print? And with only two variations, the parametric solution:

```
[...]
stream.println("              Car parked at "+nowstring);
if ( withBarCode ) {
 stream.println("||  ||||| | || ||| || ||  ||| | || |||| | || ||||");
}
stream.println("-----------------------------------------------");
```

is much shorter and does not require additional interfaces and implementation classes. If, at a later time, new requirements pop up with regards to this variability point, I would consider refactoring this design.

13.9 Review Questions

Why can't the pay station simply take a receipt object as parameter in the constructor and use this; like it did with the rate strategy?

What is the ABSTRACT FACTORY pattern? What problem does it address and what solution does it suggest? What are the roles involved? What are the benefits and liabilities?

Argue why ABSTRACT FACTORY is an example of using a compositional design approach.

Define the *dependency inversion principle* and *dependency injection* and why they are considered important principles. Argue how it relates to the use of the factory object in the pay station case.

13.10 Further Exercises

Exercise 13.4:

Walk in my footsteps. Develop the new product variant that has a new receipt type with a (fake it) bar code.

Exercise 13.5:

In the present design, each new combination requires at least one new class, namely a new instance of the PayStationFactory. Thus, if 24 product combinations are required, there will be 24 factories.

Outline a number of different techniques to avoid this multitude of factories. For each describe benefits and liabilities.

Exercise 13.6:

The first graphical user interface for Java was the Abstract Window Toolkit or AWT. AWT allowed graphical applications to be written once and then allowed to run on multiple computing platforms: Macintosh, Windows, and UNIX Motif. A central challenge for AWT was thus to couple the AWT abstractions to the concrete graphical elements of the underlying platform. That is, the statement:

```
java.awt.Button b = new java.awt.Button(``A Button'');
```

must create a button object that in turn creates a Win32 button when running on the windows platform; or creates a Mac toolkit button on the Mac OS; etc.

Sketch how the implementation of java.awt.Button may use an ABSTRACT FACTORY to create the proper buttons depending on the platform.

Note: The complete solution to this problem is complex and requires several of the design patterns mentioned in part , so focus on the ABSTRACT FACTORY aspect only.

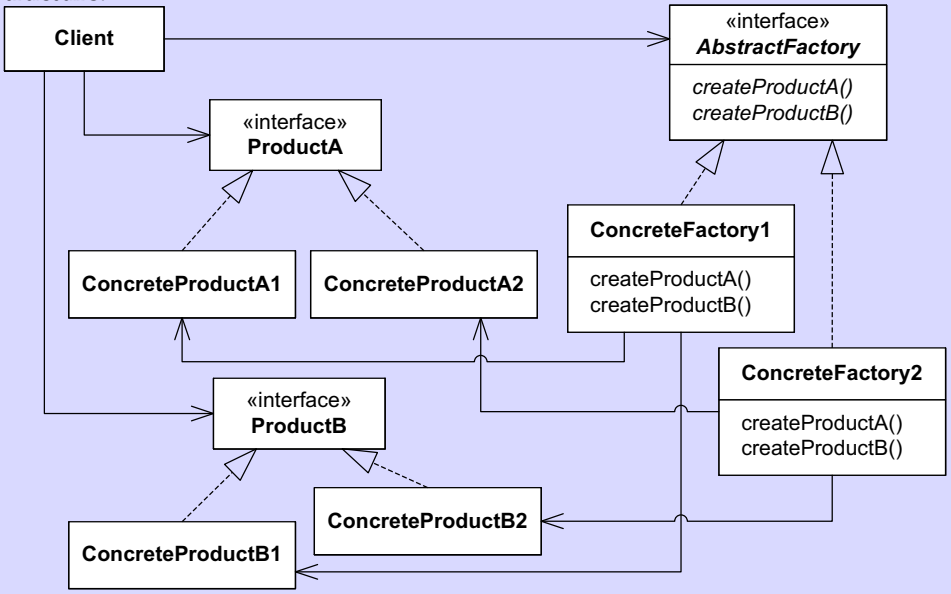
[13.1] Design Pattern: Abstract Factory

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 **Product**s 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.