CHAPTER 10

The Abstract Factory Pattern

Overview

In this chapter

I will continue our study of patterns with the Abstract Factory pattern, which is used to create families of objects.

In this chapter,

- I derive the pattern by working through an example.
- I present the key features of the Abstract Factory pattern.
- I relate the Abstract Factory pattern to the CAD/CAM problem.

Introducing the Abstract Factory Pattern

Intent: coordinate the instantiation of objects

According to the Gang of Four, the intent of the Abstract Factory pattern is to "provide an interface for creating families of related or dependent objects without specifying their concrete classes."[1]

Sometimes, several objects need to be instantiated in a coordinated fashion. For example, when dealing with user interfaces, the system might need to use one set of objects to work on one operating system and another set of objects to work on a different operating system. The Abstract Factory pattern ensures that the system always gets the correct objects for the situation.

1. Gamma, E., Helm, R., Johnson, R., Vlissides, J., *Design Patterns: Elements of Reusable Object-Oriented Software*, Reading, Mass.: Addison-Wesley, 1995, p. 87.

Learning the Abstract Factory Pattern: An Example

A motivating example: select device drivers according to the machine capacity

Suppose I have been given the task of designing a computer system to display and print shapes from a database. The type of resolution to use to display and print the shapes depends on the computer that the system is currently running on: the speed of its CPU and the amount of memory that it has available. My system must be careful about how much demand it is placing on the computer.

The challenge is that my system must control the drivers that it is using: low-resolution drivers in a less-capable machine and high-resolution drivers in a high-capacity machine, as shown in Table 10-1.

Table 10-1 Different Drivers for Different Machines

For driver...	In a low-capacity machine, use...	In a high-capacity machine, use...
Display	LRDD Low-resolution display driver	HRDD High-resolution display driver
Print	LRPD Low-resolution print driver	HRPD High-resolution print driver

Define families based on a unifying concept

In this example, the families of drivers are mutually exclusive, but this is not usually the case. Sometimes, different families will contain objects from the same classes. For example, a mid-range machine might use a low-resolution display driver (LRDD) and a high-resolution print driver (HRPD).

The families to use are based on the problem domain: which sets of objects are required for a given case? In this case, the unifying concept focuses on the demands that the objects put on the system:

- *A low-resolution family*—LRDD and LRPD, those drivers that put low demands on the system

A high-resolution family—HRDD and HRPD, those drivers that put high demands on the system

My first attempt might be to use a switch to control the selection of driver, as shown in Example 10-1.

Alternative 1: use a switch to select the driver

Example 10-1 Java Code Fragments: A Switch to Control Which Driver to Use

```
// JAVA CODE FRAGMENT

class ApControl {
    • • •
    void doDraw () {

     switch (RESOLUTION){

        case LOW:
          // use lrdd
        case HIGH:
          // use hrdd

     }
  }
  void doPrint () {
      •  • •
     switch (RESOLUTION) {
     case LOW:
      // use lrpd
      case HIGH:
      // use hrpd
     }
  }
}
```

While this does work, it presents problems. The rules for determining which driver to use are intermixed with the actual use of the driver. There are problems both with coupling and with cohesion:

…but there are problems with coupling and cohesion

- *Tight coupling*—If I change the rule on the resolution (say, I need to add a MIDDLE value), I must change the code in two places that are otherwise not related.

- *Low cohesion*—I am giving *doDraw* and *doPrint* two unrelated assignments: they must both create a shape and must also worry about which driver to use.

Tight coupling and low cohesion may not be a problem right now. However, they usually increase maintenance costs. Also, in the real world, I would likely have many more places affected than just the two shown here.

Switches may indicate a need for abstraction.

Often, a switch indicates (1) the need for polymorphic behavior, or (2) the presence of misplaced responsibilities. Consider instead a more general solution such as abstraction or giving the responsibility to other objects.

Alternative 2: use inheritance

Another alternative would be to use inheritance. I could have two different ApControls: one that uses low-resolution drivers and one that uses high-resolution drivers. Both would be derived from the same abstract class, so common code could be maintained. I show this in Figure 10-1.

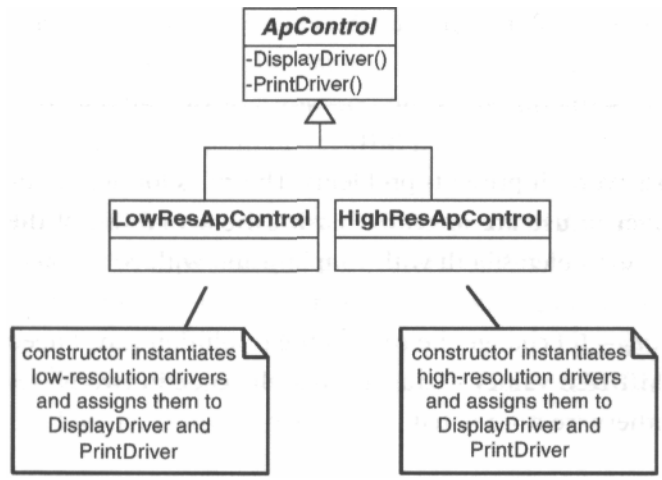

Figure 10-1 Alternative 2—handling variation with inheritance.

... but this also has problems

While inheritance could work in this simple case, it has so many disadvantages that I would rather stay with the switches. For example:

- *Combinatorial explosion*—For each different family and each new family I get in the future, I must create a new concrete class (that is, a new version of ApControl).
- *Unclear meaning*—The resultant classes do not help clarify what is going on. I have specialized each class to a particular special case. If I want my code to be easy to maintain in the future, I need to strive to make it as clear as possible what is going on. Then, I do not have to spend a lot of time trying to relearn what that section of code is trying to do.
- *Need to favor composition*—Finally, it violates the basic rule to "favor composition over inheritance."

Alternative 3: replace switches with abstraction

In my experience, I have found that switches often indicate an opportunity for abstraction. In this example, LRDD and HRDD are both display drivers and LRPD and HRPD are both print drivers. The abstractions would therefore be *display drivers* and *print drivers.* Figure 10-2 shows this conceptually. I say "conceptually" because LRDD and HRDD do not really derive from the same abstract class.

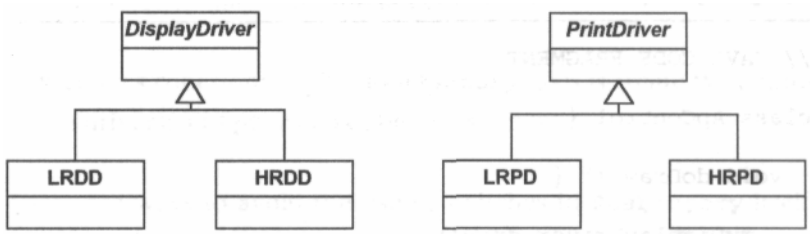

Figure 10-2 Drivers and their abstractions.

Note: At this point, I do not have to be concerned that they derive from different classes because I know I can use the Adapter pattern to adapt the drivers, making it appear they belong to the appropriate abstract class.

The code is simpler to understand

Defining the objects this way would allow for ApControl to use a DisplayDriver and a PrintDriver without using switches. ApControl is much simpler to understand because it does not have to worry about the type of drivers it has. In other words, ApControl would use a DisplayDriver object or a PrintDriver object without having to worry about the driver's resolution.

See Figure 10-3 and the code in Example 10-2.

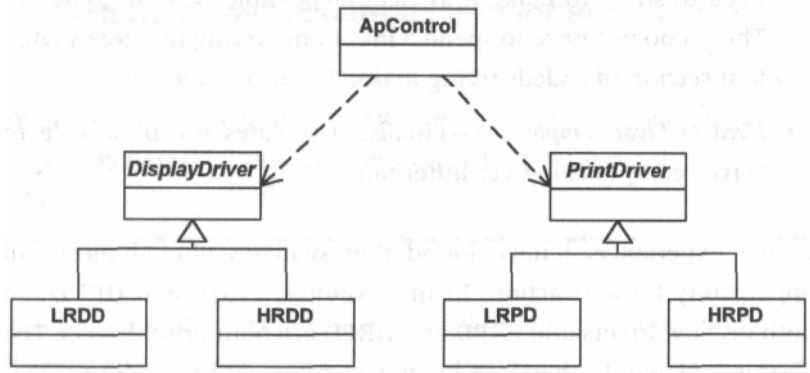

Figure 10-3 ApControl using drivers in the ideal situation.

Example 10-2 Java Code Fragments: Using Polymorphism to Solve the Problem

```
// JAVA CODE FRAGMENT

class ApControl {
   . . .
  void doDraw () {
     . . .
    myDisplayDriver.draw();
  }
  void doPrint () {
     . . .
    myPrintDriver.print();
  }
}
```

Factory objects

One question remains: How do I create the appropriate objects?

I could have ApControl do it, but this can cause maintenance problems in the future. If I have to work with a new set of objects, I will have to change ApControl. Instead, if I use a "factory" object to instantiate the objects I need, I will have prepared myself for new families of objects.

In this example, I will use a factory object to control the creation of the appropriate family of drivers. The ApControl object will use another object—the factory object—to get the appropriate type of display driver and the appropriate type of print driver for the current computer being used. The interaction would look something like the one shown in Figure 10-4.

The factory is responsible... and *cohesive*

From ApControl's point of view, things are now pretty simple. It lets ResFactory worry about keeping track of which drivers to use. Although I am still faced with writing code to do this tracking, I have decomposed the problem according to responsibility. ApControl has the responsibility for knowing how to work with the appropriate objects. ResFactory has the responsibility for deciding which objects are appropriate. I can use different factory objects or even just one object (that might use switches). In any case, it is better than what I had before.

This creates cohesion: all that ResFactory does is create the appropriate drivers; all ApControl does is use them.

. . . and it encapsulates variation in a class

There are ways to avoid the use of switches in ResFactory itself. This would allow me to make future changes without affecting any existing factory objects. I can encapsulate a variation in a class by defining an abstract class that represents the factory concept. In the case of ResFactory, I have two different behaviors (methods):

- Give me the display driver I should use.
- Give me the print driver I should use.

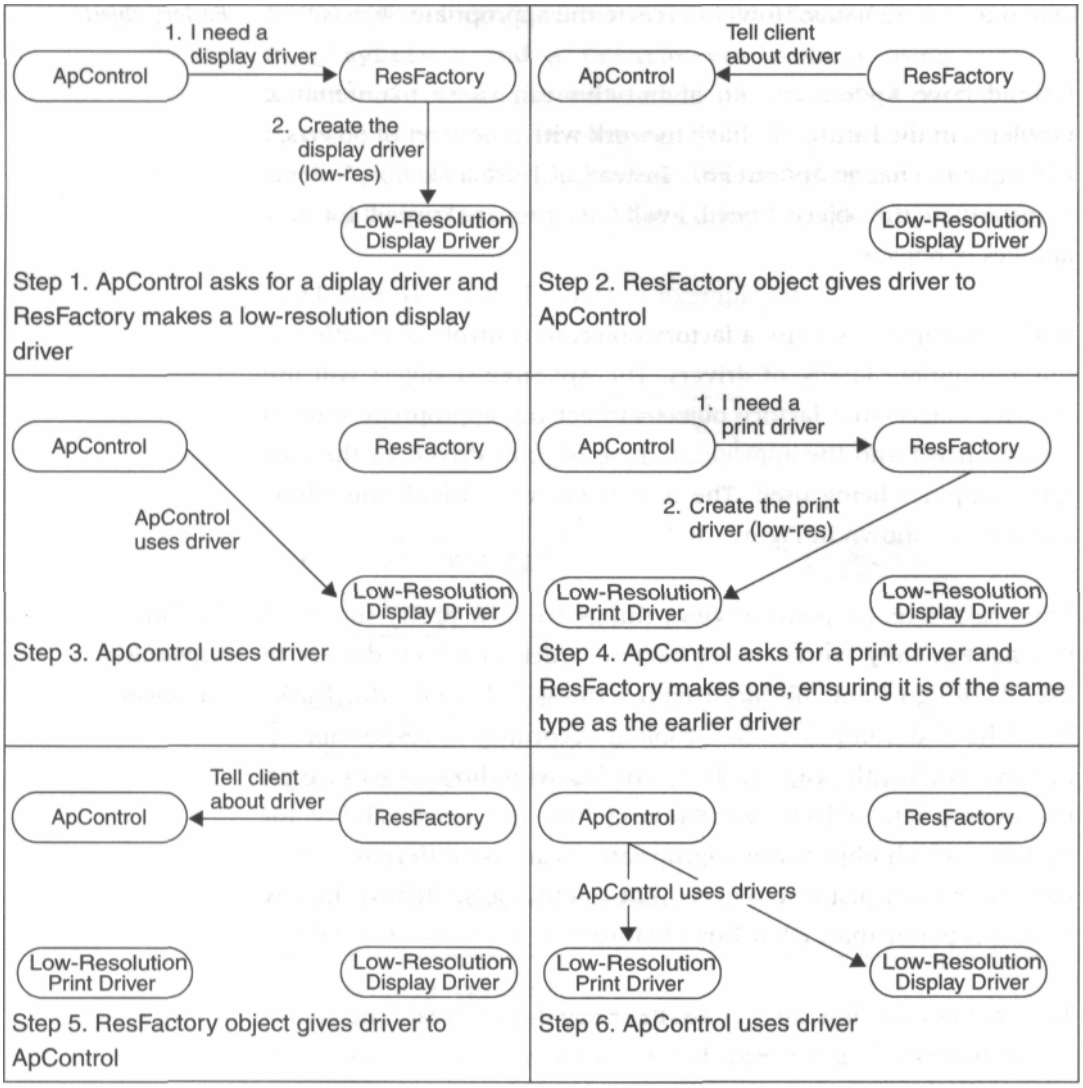

Figure 1O-4 ApControl gets its drivers from a factory object.

ResFactory can be instantiated from one of two concrete classes and derived from an abstract class that has these public methods, as shown in Figure 10-5.

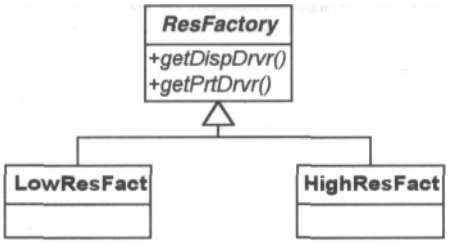

Figure 10-5 The ResFactory encapsulates the variations.

Strategies for bridging analysis and design.

Below are three key strategies involved in the Abstract Factory.

Strategy	Shown in the Design
Find what varies and encapsulate it.	The choice of which driver object to use was varying. So, I encapsulated it in ResFactory.
Favor composition over inheritance.	Put this variation in a separate object—ResFactory—and have ApControl use it as opposed to having two different ApControl objects.
Design to interfaces, not to implementations.	ApControl knows how to ask ResFactory to instantiate drivers—it does not know (or care) how ResFactory is actually doing it.

Learning the Abstract Factory Pattern: Implementing It

Example 10-3 shows how to implement the Abstract Factory objects for this design.

Implementation of the design

Example 10-3 Java Code Fragments: Implementation of ResFactory

```
class LowResFact extends ResFactory {

  DisplayDriver public
    getDispDrvr() {
      return new
   lrdd(); }

  PrintDriver public
    getPrtDrvr() {
      return new lrpd();
  }}
class HighResFact extends ResFactory {

  DisplayDriver public
    getDispDrvr() {
      return new hrdd();}

  PrintDriver public
    getPrtDrvr() {
    return new hrpd();} }
```

Putting it together: the Abstract Factory

To finish the solution, I have the ApControl talk with the appropriate factory object (either LowResFact or HighResFact); this is shown in Figure 10-6. Note that ResFactory is abstract, and that this hiding of ResFactory's implementation is what makes the pattern work. Hence, the name Abstract Factory for the pattern.

How this works

ApControl is given either a LowResFact object or a HighResFact object. It asks this object for the appropriate drivers when it needs them. The factory object instantiates the particular driver (low or high resolution) that it knows about. ApControl does not need to worry about whether a low-resolution or a high-resolution driver is returned since it uses both in the same manner.

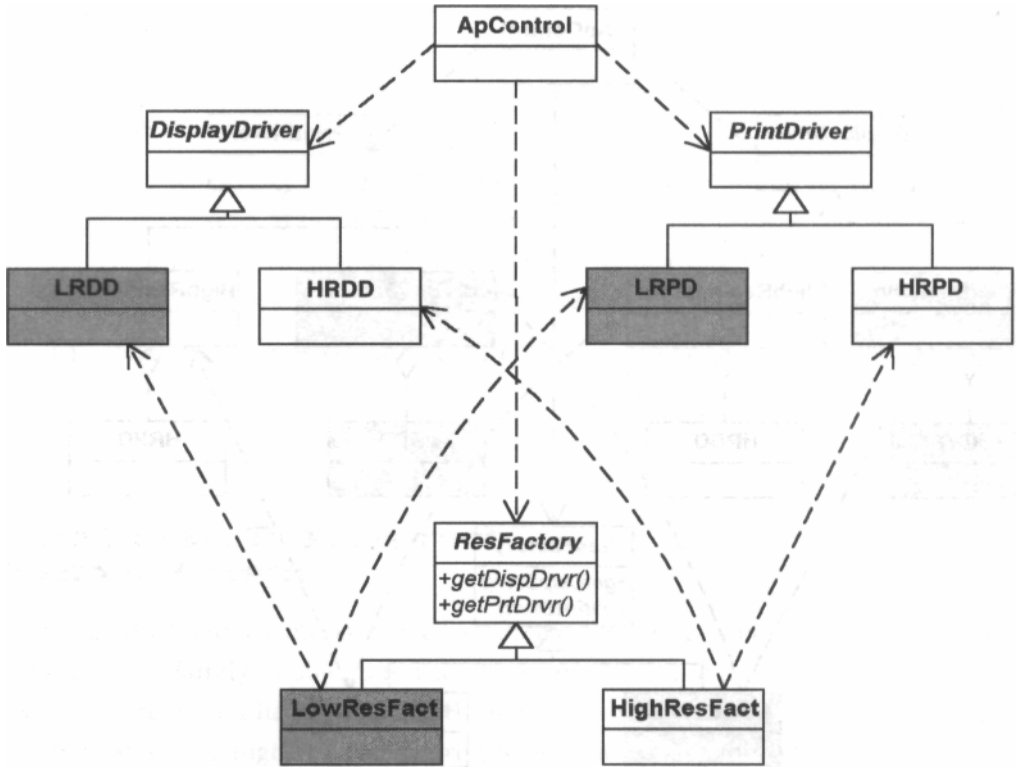

Figure 10-6 Intermediate solution using the Abstract Factory.

I have ignored one issue: LRDD and HRDD may not have been derived from the same abstract class (as may be true of LRPD and HRPD). Knowing the Adapter pattern, this does not present much of a problem. I can simply use the structure I have in Figure 10-6, but adapt the drivers as shown in Figure 10-7.

The LRDD/HRDD and LRPD/HRPD pairs do not necessarily derive from the same classes

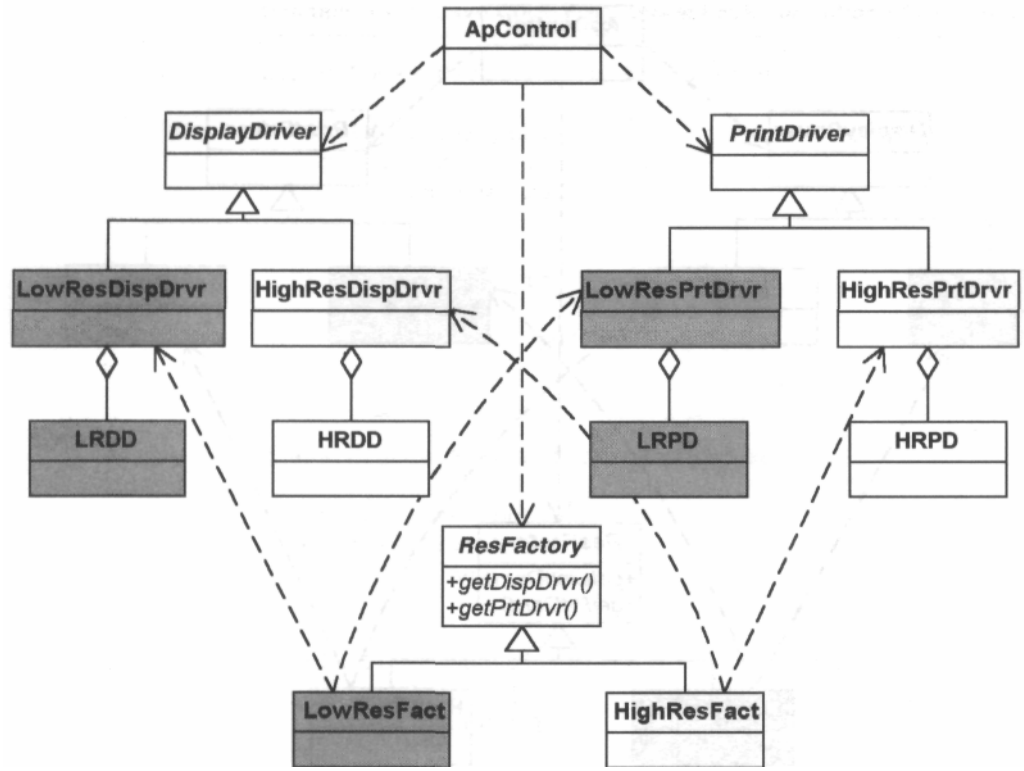

Figure 1O-7 Solving the problem with the Abstract Factory and Adapter.

How this works

The implementation of this design is essentially the same as the one before it. The only difference is that now the factory objects instantiate objects from classes I have created that adapt the objects I started with. This is an important modeling method. By combining the Adapter pattern with the Abstract Factory pattern in this way, I can treat these conceptually similar objects as if they were siblings even if they are not. This enables the Abstract Factory to be used in more situations.

In this pattern,

The roles of the objects in the Abstract Factory

- The client object just knows who to ask for the objects it needs and how to use them.
- The Abstract Factory class specifies which objects can be instantiated by defining a method for each of these different types of objects. Typically, an Abstract Factory object will have a method for each type of object that must be instantiated.
- The concrete factories specify which objects are to be instantiated.

Field Notes: The Abstract Factory Pattern

How to get the right factory object

Deciding which factory object is needed is really the same as determining which family of objects to use. For example, in the preceding driver problem, I had one family for low-resolution drivers and another family for high-resolution drivers. How do I know which set I want? In a case like this, it is most likely that a configuration file will tell me. I can then write a few lines of code that instantiate the proper factory object based on this configuration information.

I can also use an Abstract Factory so I can use a subsystem for different applications. In this case, the factory object will be passed to the subsystem, telling the subsystem which objects it is to use. In this case, it is usually known by the main system which family of objects the subsystem will need. Before the subsystem is called, the correct factory object would be instantiated.

The Abstract Factory Pattern: Key Features

Intent	You want to have families or sets of objects for particular clients (or cases).
Problem	Families of related objects need to be instantiated.
Solution	Coordinates the creation of families of objects. Gives a way to take the rules of how to perform the instantiation out of the client object that is using these created objects.
Participants and Collaborators	The AbstractFactory defines the interface for how to create each member of the family of objects required. Typically, each family is created by having its own unique ConcreteFactory.
Consequences	The pattern isolates the rules of which objects to use from the logic of how to use these objects.
Implementation	Define an abstract class that specifies which objects are to be made. Then implement one concrete class for each family. Tables or files can also be used to accomplish the same thing.
GoF Reference	Pages 87-96.

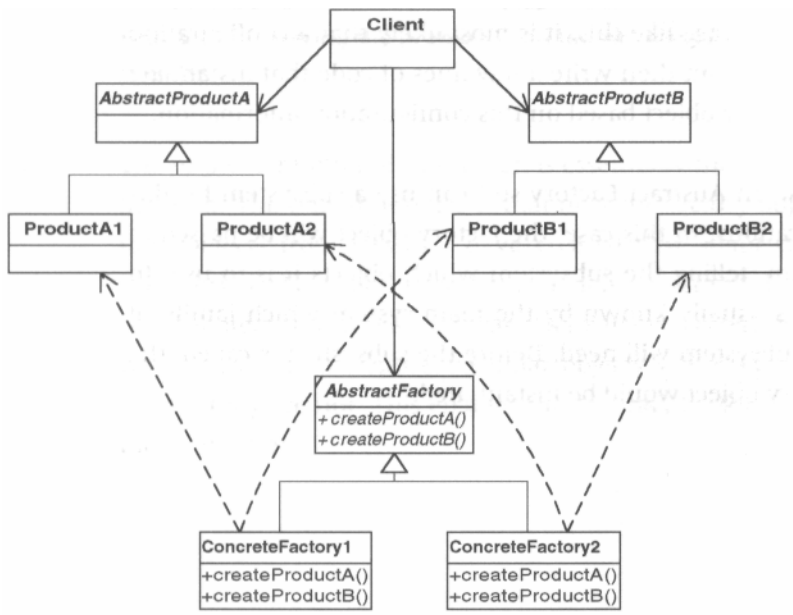

Figure 10-8 Standard, simplified view of the Abstract Factory pattern.

How Abstract Factory works and what its benefits are

Figure 10-8 shows a Client using objects derived from two different server classes (AbstractProductA and AbstractProductB). It is a design that simplifies, hides implementations, and makes a system more maintainable.

- The client object does not know which particular concrete implementations of the server objects it has because the factory object has the responsibility to create them.
- The client object does not even know which particular factory it uses since it only knows that it has an Abstract Factory object. It has a ConcreteFactoryl or a ConcreteFactory2 object, but it doesn't know which one.

I have hidden (encapsulated) from the Client the choice about which server objects are being used. This will make it easier in the future to make changes in the algorithm for making this choice because the Client is unaffected.

The Abstract Factory pattern affords us a new kind of decomposition—decomposition by responsibility. Using it decomposes our problem into

- Who is using our particular objects (ApControl)
- Who is deciding upon which particular objects to use (AbstractFactory)

Abstract Factory applies when there are families of objects,

Using the Abstract Factory is indicated when the problem domain has different families of objects present and each family is used under different circumstances.

You may define families according to any number of reasons. Examples include:

- Different operating systems (when writing cross-platform applications)

- Different performance guidelines
- Different versions of applications
- Different traits for users of the application

Once you have identified the families and the members for each family, you must decide how you are going to implement each case (that is, each family). In my example, I did this by defining an abstract class that specified which family member types could be instantiated. For each family, I then derived a class from this abstract class that would instantiate these family members.

A variation of the Abstract Factory pattern: configuration files

Sometimes you will have families of objects but do not want to control their instantiation with a different derived class for each family. Perhaps you want something more dynamic.

Examples might be

- You want to have a configuration file that specifies which objects to use. You can use a switch based on the information in the configuration file that instantiates the correct object.
- Each family can have a record in a database that contains infor mation about which objects it is to use. Each column (field) in the database indicates which specific class type to use for each make method in the Abstract Factory.

A further variation: using the Class class in Java

If you are working in Java, you can take the configuration file concept one step further. Have the information in the field names represent the class name to use. It does not need to be the full class name as long as you have a set convention. For example, you could have a set prefix or suffix to add to the name in the file. Using Java's Class class you can instantiate the correct object based on these names.[2]

2. For a good description of Java's Class class see Eckel, B., *Thinking in Java*, Upper Saddle River, N.J.: Prentice Hall, 2000.

Adapters and the Abstract Factory

In real-world projects, members in different families do not always have a common parent. For example, in the earlier driver example, it is likely that the LRDD and HRDD driver classes are not derived from the same class. In cases like this, it is necessary to adapt them so an Abstract Factory pattern can work.

Relating the Abstract Factory Pattern to the CAD/CAM Problem

In the CAD/CAM problem, the system will have to deal with many sets of features, depending upon which CAD/CAM version it is working with. In the V1 system, all of the features will be implemented for V1. Similarly, in the V2 system, all of the features will be implemented for V2.

The families that I will use for the Abstract Factory pattern will be V1 Features and V2 Features.

Summary

In this chapter

The Abstract Factory is used when you must coordinate the creation of families of objects. It gives a way to take the rules regarding how to perform the instantiation out of the client object that is using these created objects.

- First, identify the rules for instantiation and define an abstract class with an interface that has a method for each object that needs to be instantiated.
- Then, implement concrete classes from this class for each family.
- The client object uses this factory object to create the server objects that it needs.

Supplement: C++ Code Examples

Example 10-4 C++ Code Fragments: A Switch to Control Which Driver to Use

```
// C++ CODE FRAGMENT

// class ApControl

void ApControl::doDraw (} {


    .   .   .
  switch (RESOLUTION)
    { case LOW:
    // use Irdd
    case HIGH: //
    use hrdd

  }
}
void ApControl::doPrint () {


    .  .  .
  switch (RESOLUTION)
    { case LOW:
      // use lrpd
    case HIGH:
    // use hrpd } }
```

Example 10-5 C++ Code Fragments: Using Polymorphism to Solve the Problem

```
// C++ CODE FRAGMENT

// class ApControl

void ApControl::doDraw () {

  myDisplayDriver->draw();
}
void ApControl::doPrint () {

  myPrintDriver->print();
 }
```

Example 1O-6 C++ Code Fragments: Implementation of ResFactory

```
class LowResFact : public ResFactory;

DisplayDriver *
   LowResFact::getDispDrvr() {
   return new Irdd; }

PrintDriver *
   LowResFact::getPrtDrvr() {
   return new Irpd; }

class HighResFact : public ResFactory;

DisplayDriver *
   HighResFact::getDispDrvr()
   { return new hrdd;
}

PrintDriver *
   HighResFact::getPrtDrvr() {
   return new hrpd; }
```