ΕΠΛ 603 – Topics in Software Engineering

Abstract Factory

Abstract Factory

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

Example

Suppose we were 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 amount of memory that it has available. The system must be careful about how much demand it is placing on the computer.

The challenge is that the 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

Families

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

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

Solution 1: Switch

Although this does work, it has 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:

- **Tight coupling.** If we change the rule on the resolution (say, I need to add a MIDDLE value), we must change the code in two places that are otherwise not related.
- **Weak cohesion.** We are giving *doDraw* and *doPrint* two unrelated assignments: They must both create a shape and must also worry about which driver to use.

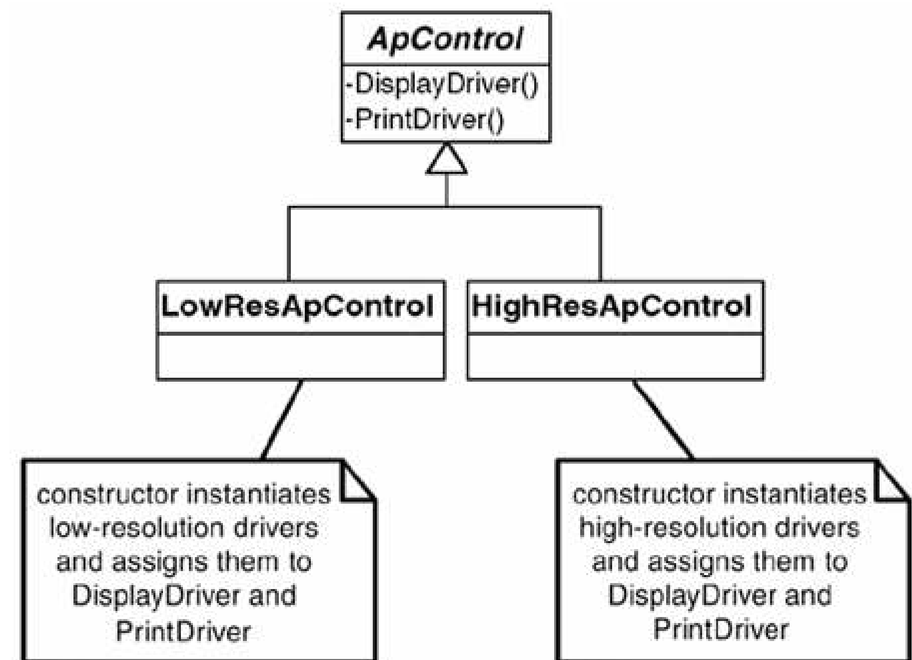
```
// JAVA CODE FRAGMENT

class ApControl {
   . . .
   public void doDraw() {
      . . .
      switch (RESOLUTION) {
         case LOW:
            // use lrdd
         case HIGH:
            // use hrdd
      }
   }
   public void doPrint() {
      . . .
      switch (RESOLUTION) {
         case LOW:
            // use lrpd
         case HIGH:
            // use hrpd
      }
   }
}
```

Solution 2: Inheritance

We 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.

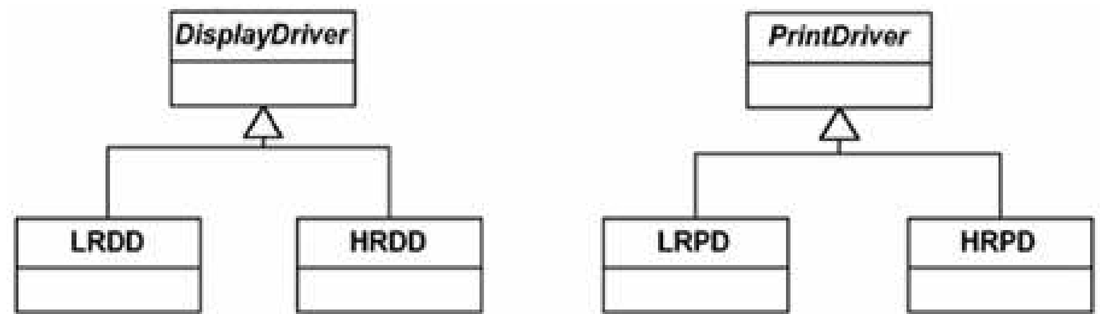
- Combinatorial explosion. For each new family we get in the future, we must create a new concrete class (that is, a new version of ApControl). The diagram would have many derivations under ApControl.

- Need to favor aggregation. Not following this rule means when other variations happen my classes will degrade even further.



Solution 3: Abstraction

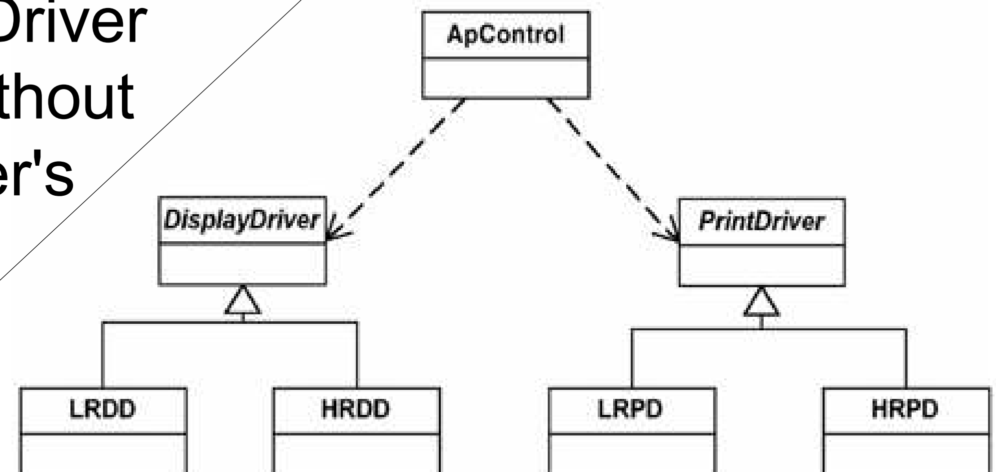
- LRDD and HRDD are both display drivers, and LRPD and HRPD are both print drivers. The abstractions would therefore be display drivers and print drivers. Figure shows this "conceptually" because LRDD and HRDD may not actually derive from the same abstract class.
- At this point, we do not have to be concerned that they may derive from different classes because we know we can use the Adapter pattern to wrap the drivers, making it appear they belong to the appropriate abstract class



Using Polymorphism

- Defining the objects this way would allow for **ApControl** to use ***DisplayDriver*** and ***PrintDriver*** without using switches.
- ApControl is much simpler to understand because it is not concerned about the type of drivers it uses. In other words, ApControl would use a DisplayDriver object or a PrintDriver object without concerning itself about the driver's resolution.

```
// JAVA CODE FRAGMENT

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



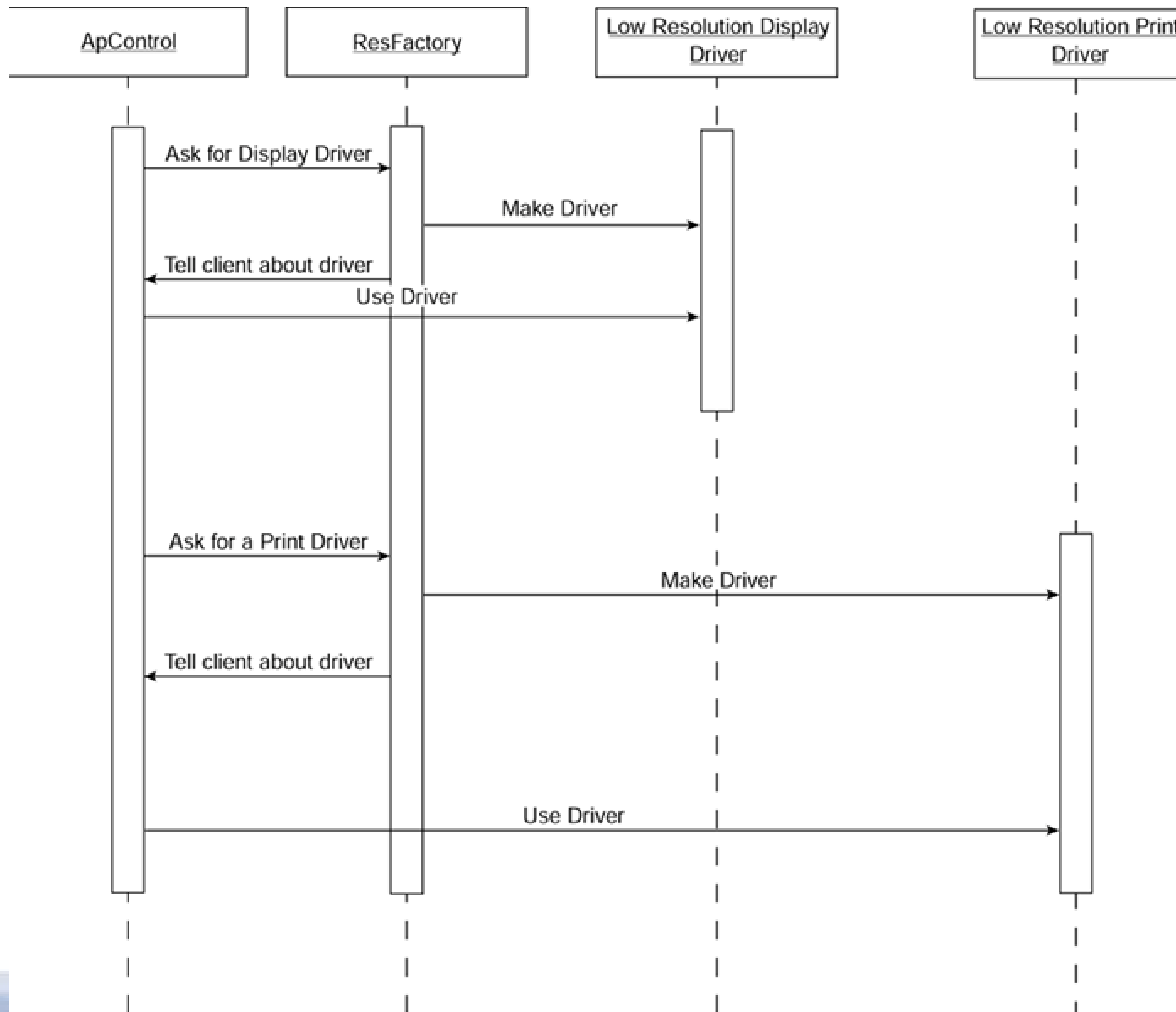
One question remains...

How do we create the appropriate objects?

- Have ApControl do it, but this can cause maintenance problems in the future. If we have a new set of objects, we will have to change ApControl.
- Instead, if we use a "factory" object to instantiate the objects we need, we will be prepared for new families of objects.

In this example, we will use a factory object (of type ResFactory, also known as Resolution Factory) 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.

ApControl gets its drivers from a factory object



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 we are still faced with writing code to do this tracking, we 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. We can use different factory objects or even just one object (that might use switches). In any case, it is better than what we had before.

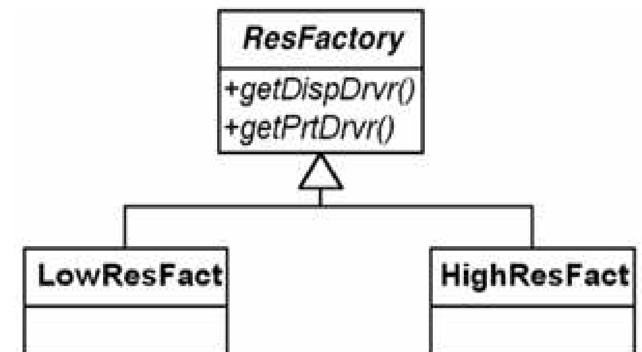
This strengthens **cohesion**: All that ResFactory does is create the appropriate drivers; all ApControl does is use them.

Avoid Switches

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

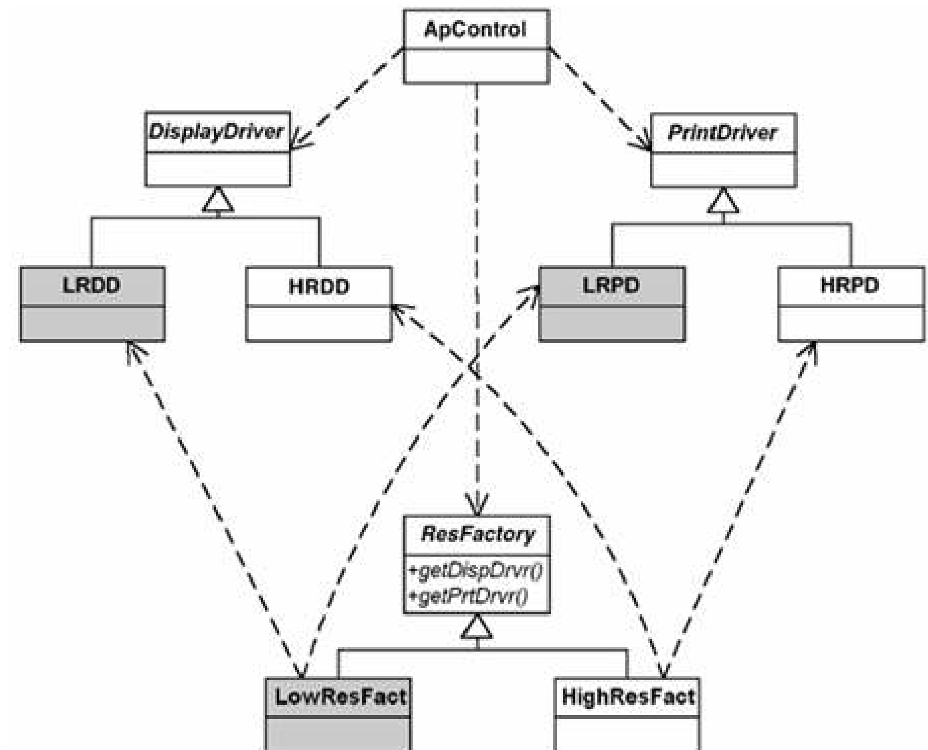
- Give me the display driver I should use.
- Give me the print driver I should use.

ResFactory can be instantiated from one of two concrete classes and derived from an abstract class that has these public methods



Putting it together: the Abstract Factory

We have ApControl talk with the appropriate factory object (either LowResFact or HighResFact). 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



Issues in the pattern

Which case do I have?

- Original solution: ApControl knew this.
- Abstract Factory solution: A configuration file saying which concrete factory object to create.

How do I manage this information?

- Original solution: ApControl managed this.
- Abstract Factory solution: Each concrete object knew which objects it was to create.

Where do I put the construction logic?

- Original solution: ApControl managed this.
- Abstract Factory solution. In the factory objects.

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, we had one family for low-resolution drivers and another family for high-resolution drivers.

How do we know which set we want?

- In a case like this, it is most likely that a configuration file will tell us. We can then write a few lines of code that instantiate the proper factory object based on this configuration information.

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.

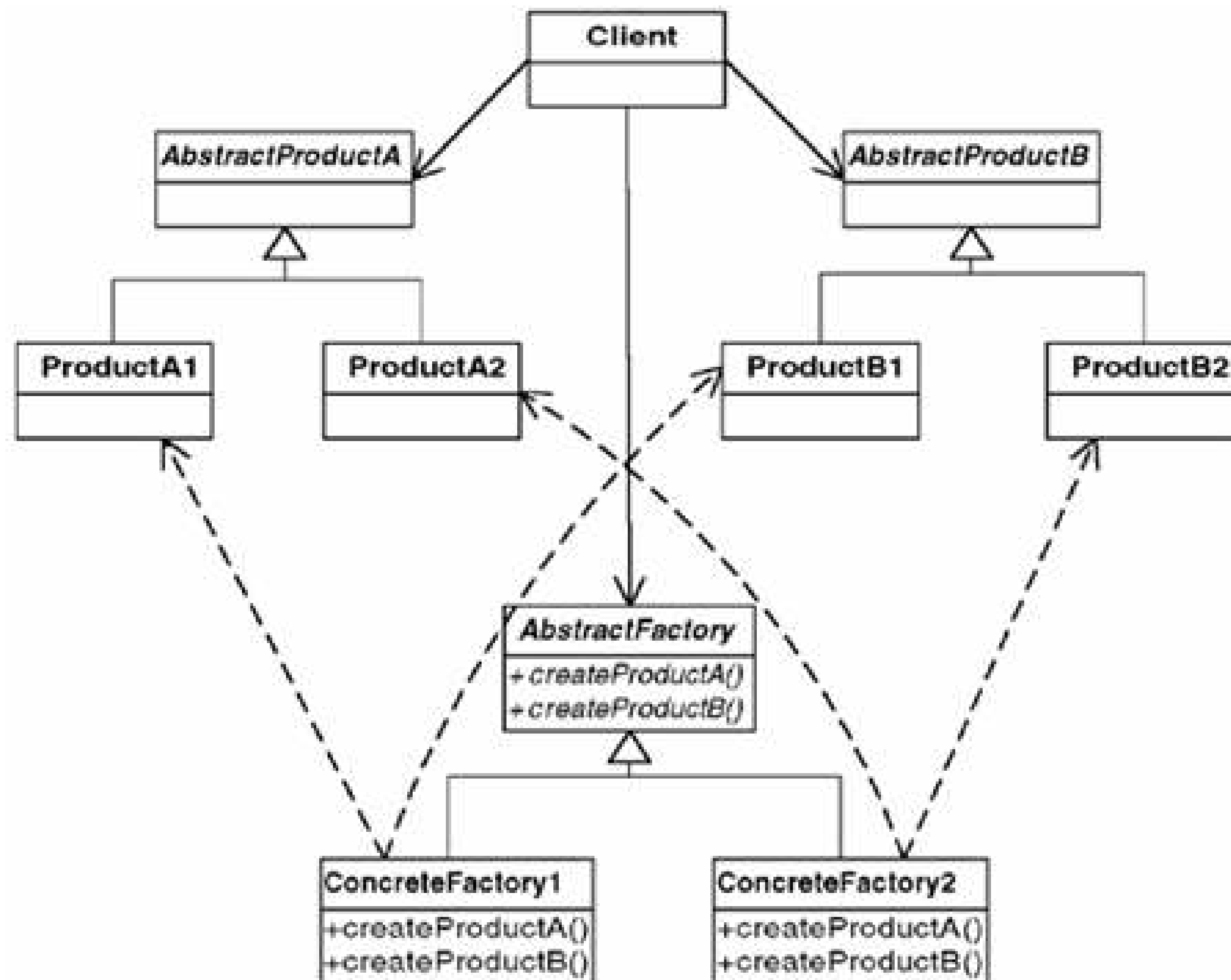
Solution: Coordinates the creation of families of objects. Gives a way to take the rules of how to perform the instantiation out of the client object that is using these created objects.

Participants - 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 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 a concrete class for each family.

Generic structure of the Abstract Factory pattern



Decomposition by responsibility

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).

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

- Different Levels in Computer Games
- Different operating systems (writing cross-platform apps)
- Different versions of applications

Relating to the CAD/CAM problem

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

The families that we will use for the Abstract Factory pattern will be V1 features and V2 features.

Summary

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 objects that it needs.