- Strategy Pattern
- Factory Method Pattern
- Abstract Factory Pattern

Strategy Pattern

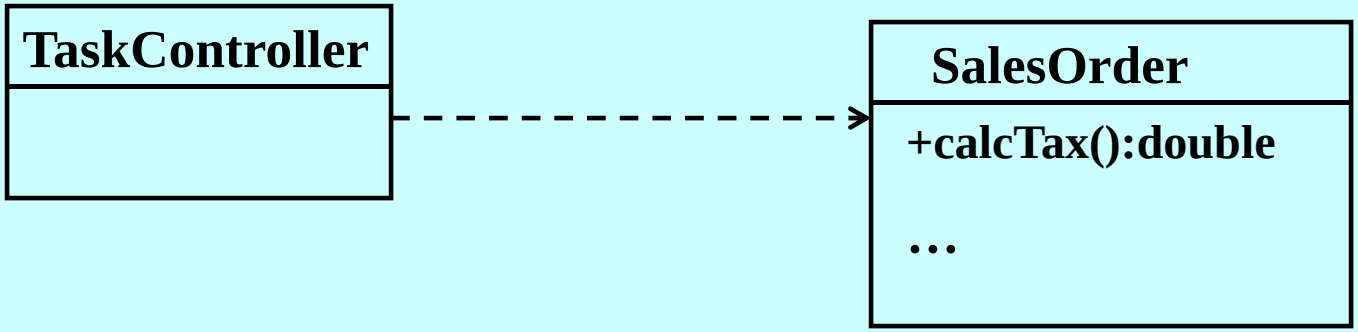
Strategy Pattern (Behavioral Pattern):

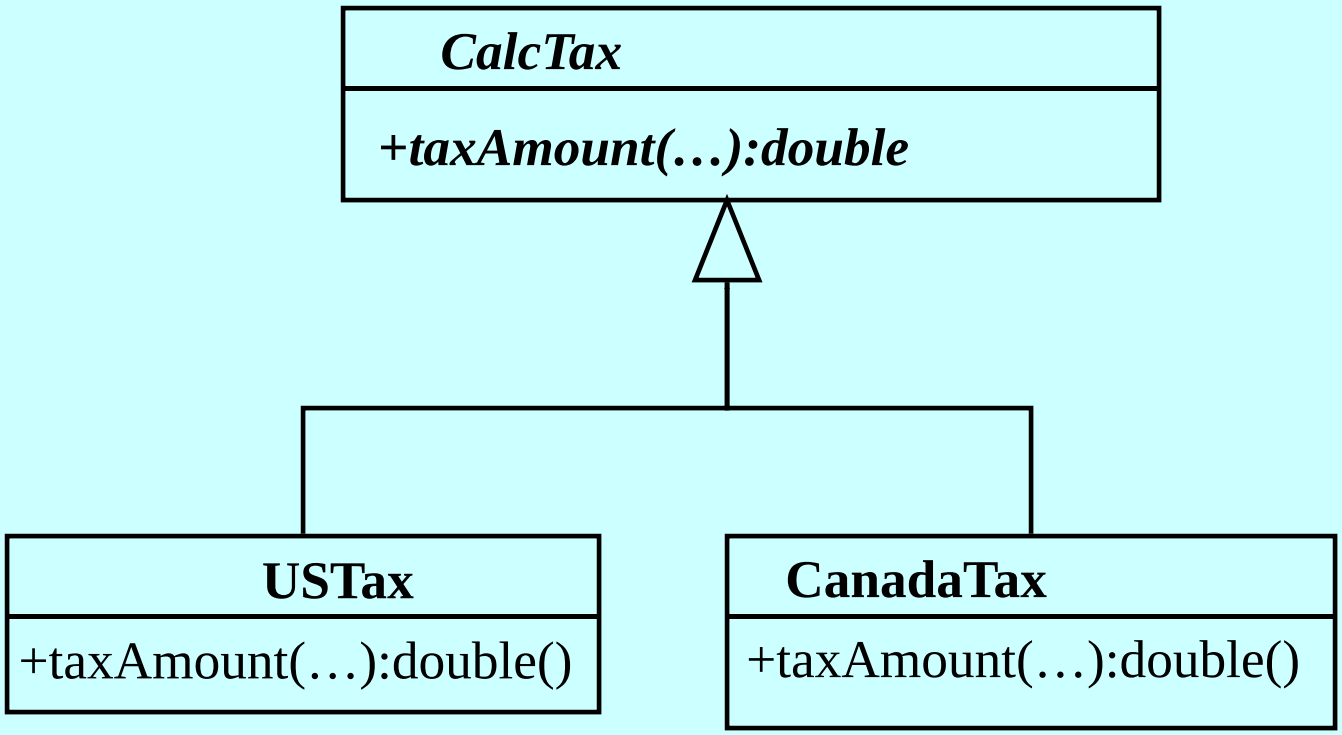
- Intent (GOF): Define a family of algorithms, encapsulate each one and make them interchangeable. Strategy let the algorithm vary independently from clients that use it.

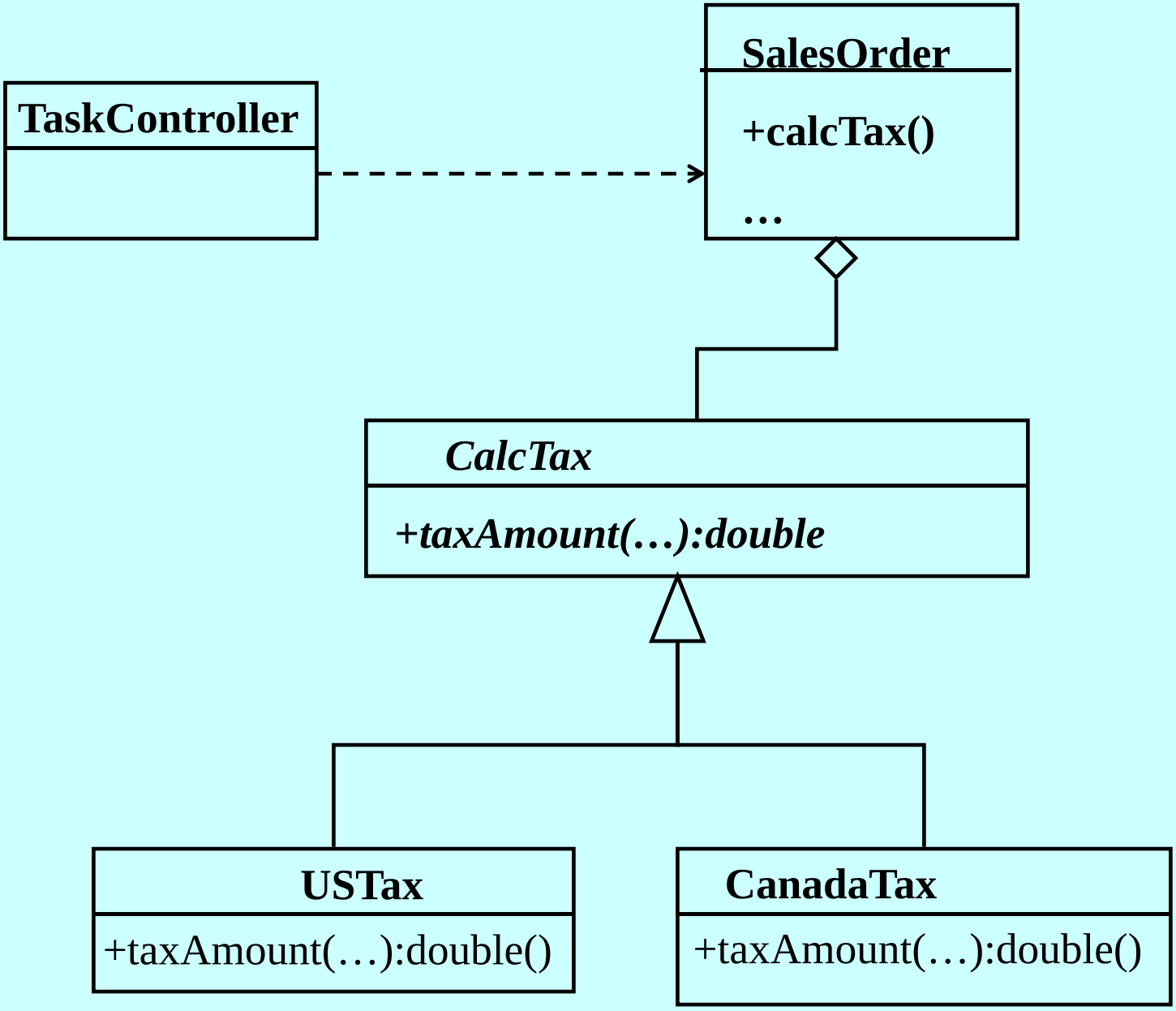
- Motivation: lets see an example..

Example:

Consider an order processing system that should be able to process sales orders in many different countries..




CalcTax
+taxAmount(…):double
USTax
+taxAmount(…):double()
CanadaTax
+taxAmount(…):double()


SalesOrder
+calcTax()
…
TaskController
CalcTax
+taxAmount(…):double
USTax
+taxAmount(…):double()
CanadaTax
+taxAmount(…):double()

```
public abstract class CalcTax {
    public abstract double taxAmount(…);
}
public class CanTax extends CalcTax {
    public double taxAmount(…) {
        // calculate tax according to rules in Canada
    }
}
public class USTax extends CalcTax {
    public double taxAmount(…) {
        // calculate tax according to rules in USA
    }
}
```

```
public class SalesOrder {
   private CalcTax ct;
   public SalesOrder( CalcTax c ) { ct = c; }
   public double calcTax() {
      return ct.taxAmount();
   }
}
public class TaxController {
   public static void main(String[] args) {
      SalesOrder so = new SalesOrder( new USTax() );
      System.out.println( so.calculateTax() );
      ……
   }
}
```

Strategy Pattern (contd.):

- **Applicability:** Use it in following cases:

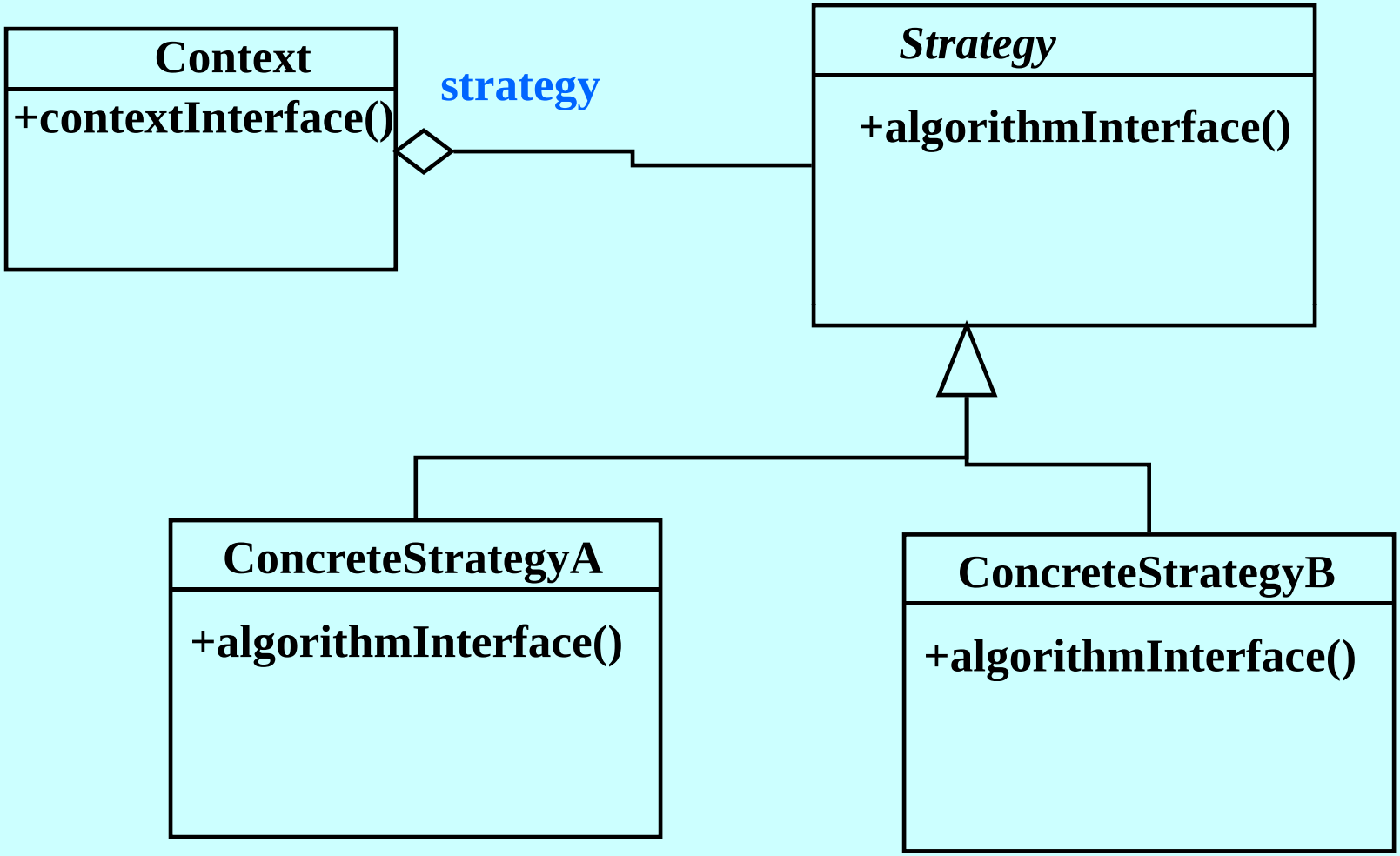
  - many related classes differ only in their behavior

  - you need different variants of an algorithm

  - a class defines many behaviors and these appear as multiple conditional statements in its methods. Instead of many conditionals, move related conditional branches into their own Strategy class.

Strategy Pattern (contd.):

- Generic Structure:



Strategy Pattern (contd.):

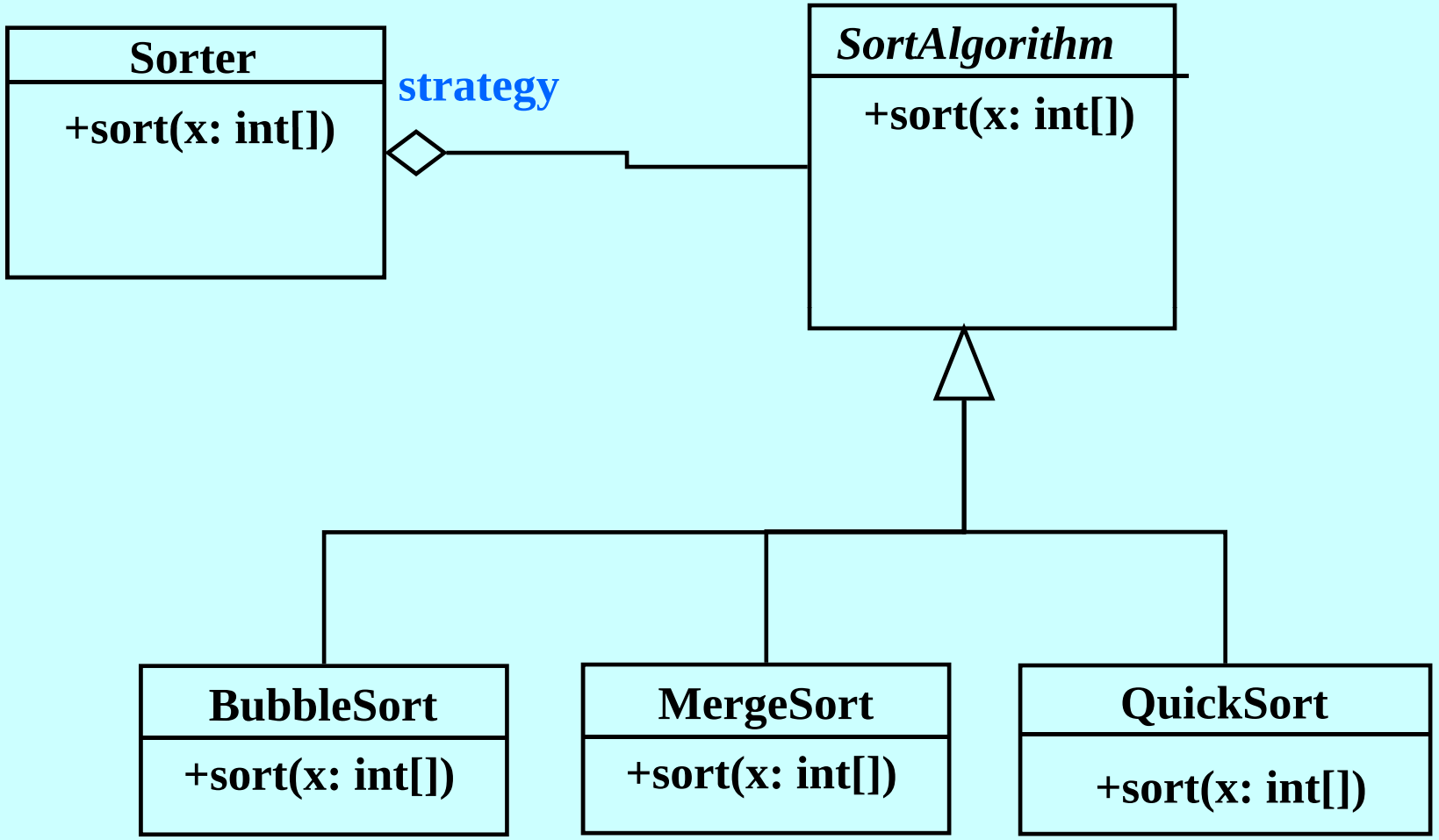
- Participants and Collaborations: Strategy, ConcreteStrategy, Context.

- A Context forwards requests from its clients to the strategy. Clients usually create and pass a ConcreteStrategy object to the Context. There is often a family of ConcreteStrategy classes for a client to choose from.

Strategy Pattern (contd.):

- **Consequences:**
  - provides an alternative to subclassing the Context to get a variety of algorithms or behaviors.
  - eliminates large conditional statements
  - provides a choice of implementations for the same behavior.
  - increased number of objects (disadvantage)

Example: A client wants to decide at run-time what algorithm it should use to sort an array of integers. Many different sort algorithms are available..



```
public abstract class SortAlgorithm {
   public abstract void sort(int[] x);
}
public class MergeSort extends SortAlgorithm {
   public void sort(int[] x) {
      // implements merge sort algo
   }
}
public class BubbleSort extends SortAlgorithm {
   public void sort(int[] x) {
      // implements merge sort algo
   }
}
```

```
public class QuickSort extends SortAlgorithm {
   public void sort(int[] x) {
      // implements quick sort algo
   }
}
```

```
public class Sorter {
   private SortAlgorithm st;
   public Sorter( SortAlgorithm s ) { st = s; }
   public void sort( int[] x ) {
      st.sort( x );
   }
}
```

```
public class Client {
   public static void main(String[] args) {
      int[] ia = { 31, 25, 67, 56, 90 };

      Sorter sorter = new Sorter( new QuickSort() );
      sorter.sort( ia );
      ……
   }
}
```

Factory Method Pattern

Factory Method Pattern (Creational Pattern):

- **Intent:** Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

- **Motivation:** lets see an example.

Example: Consider a framework for applications that can present multiple documents to the user. Two key abstractions are the classes: *Application* and *Document* (both are abstract).

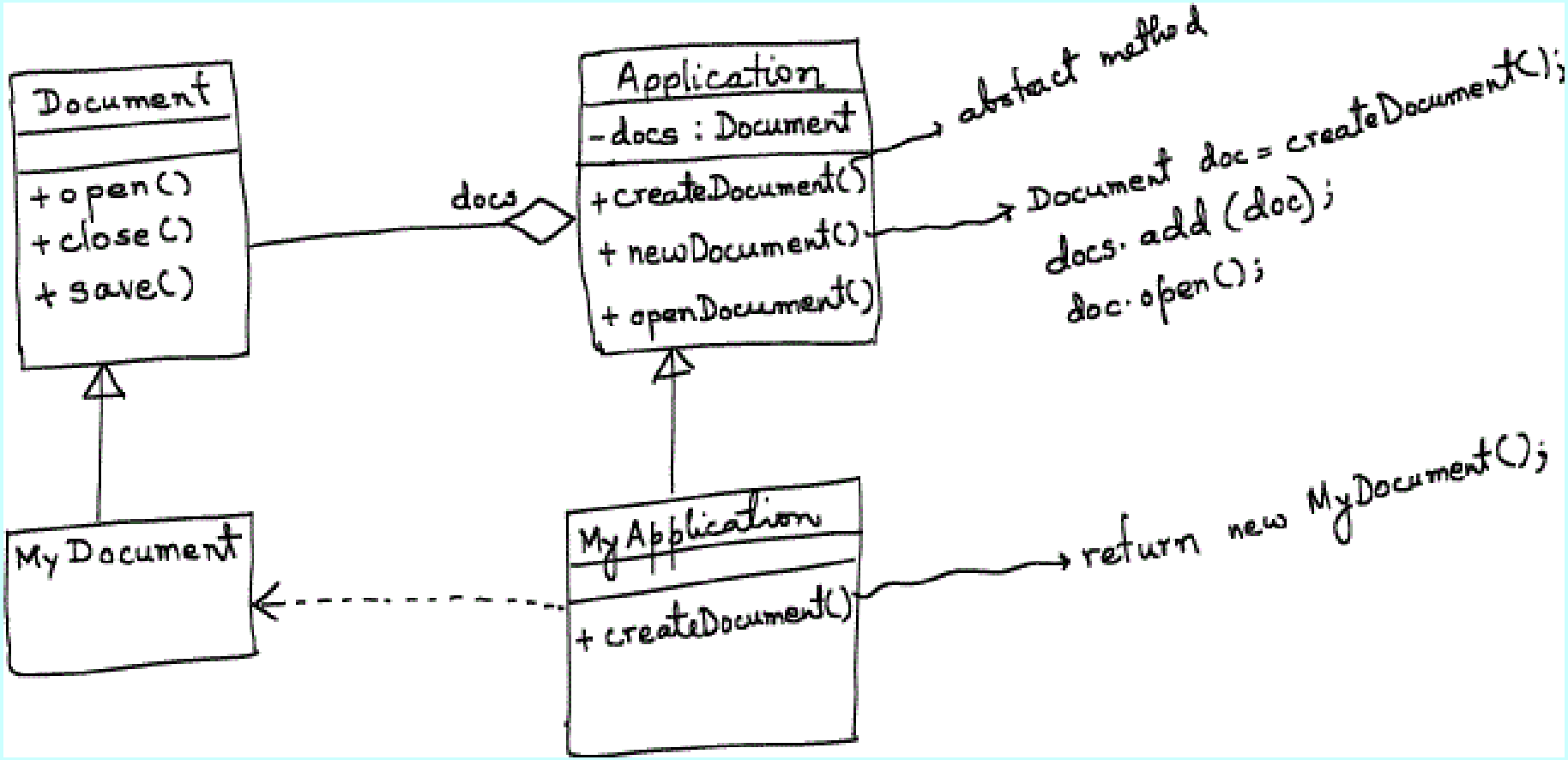
The clients have to subclass them to realize application-specific implementations. For e.g. to create a drawing application, we define classes DrawingApplication and DrawingDocument.

The *Application* class is responsible for managing documents and create them as required.

Particular Document subclass to instantiate is application-specific and *Application* class can't predict the subclass to instantiate…it only knows *when* a new document should be created, not what kind of document to create..

the framework must instantiate classes but it only knows about abstract classes, which it cannot instantiate..

Solution: The Factory method pattern offers a solution. It encapsulates the knowledge of which document subclass to create and moves this knowledge out of the framework.


Document
+ open()
+ close()
+ save()
docs
Application
- docs : Document
+ createDocument()
+ newDocument()
+ openDocument()
abstract method
Document doc = createDocument();
docs.add(doc);
doc.open();
MyDocument
MyApplication
+ createDocument()
return new MyDocument();

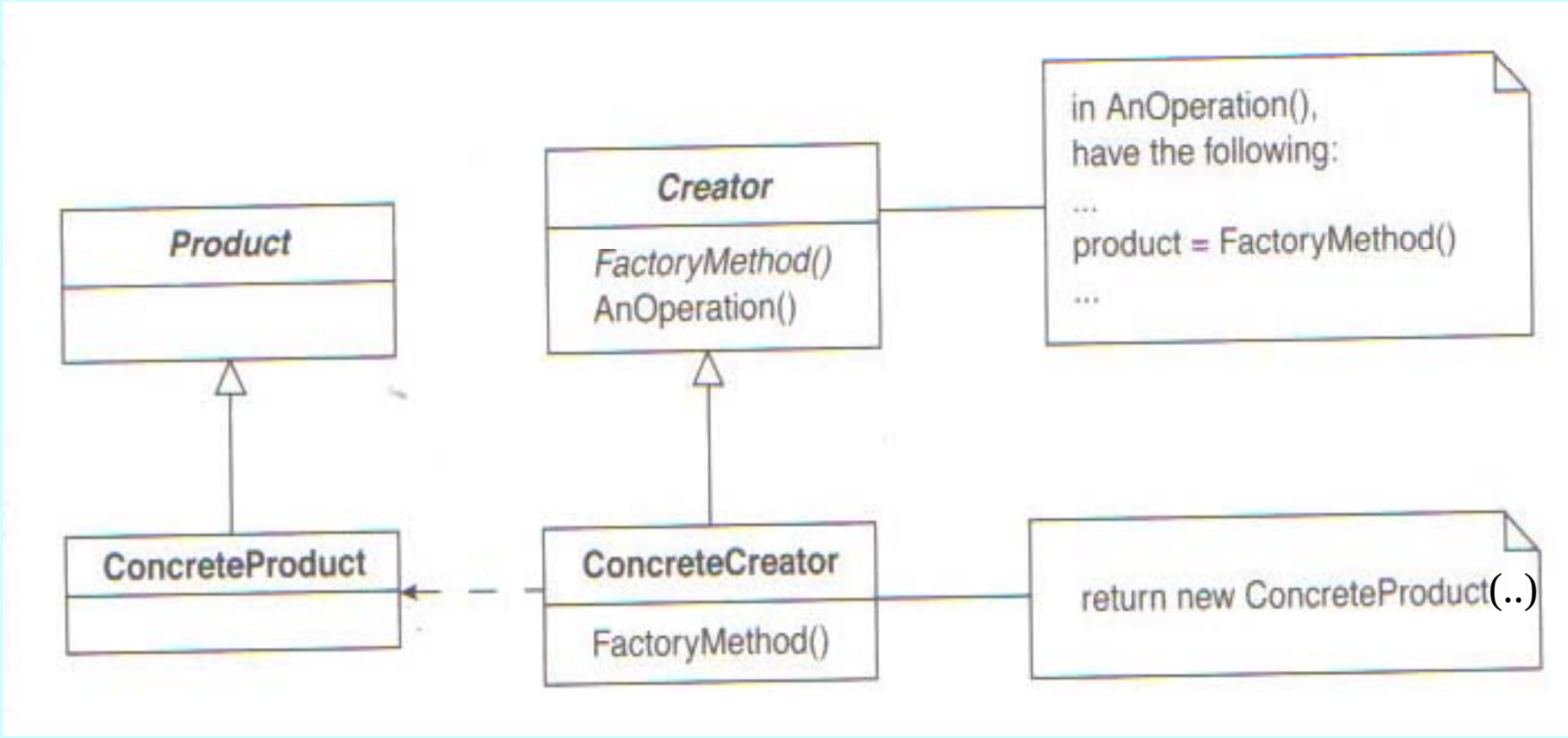
Factory Method Pattern (contd.):

Applicability: Use it when:

- **a class cannot anticipate the class of objects it must create**
- **a class wants its subclass to specify the objects it creates**

Factory Method Pattern (contd.):

- **Generic Structure:**



Factory Method Pattern (contd.):

- Participants and Collaborations: *Product* (*interface of objects the factory method creates*), ConcreteProduct (implements the Product interface), *Creator* (*declares the factory method*), ConcreteCreator (overrides the factory method).

Creator relies on its subclasses to define the factory method so that it returns an instance of the appropriate ConcreteProduct.

Factory Method Pattern (contd.):

- Consequences: several benefits and liabilities:
  - it eliminates the need to to bind application-specific classes into your code.
  - disadvantage: clients might have to subclass the Creator class just to create a particular ConcreteProduct object.
  - gives subclasses a hook for providing an extended version of an object.

Factory Method Pattern (contd.):

- **Implementation:**
  - Use an abstract method (i.e. the factory method) in the abstract class (i.e. in the Creator).
  - The code in the abstract Creator class refers to the abstract factory method when it needs to instantiate an application-specific product object but does not know which particular object it needs.

Abstract Factory Pattern

Abstract Factory Pattern (Creational Pattern):

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

- **Motivation:** sometimes several objects need to be instantiated in a coordinated fashion.

Example: design a software 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.

(from the book of Shalloway and Trott)

Alternative-1: use a switch statement

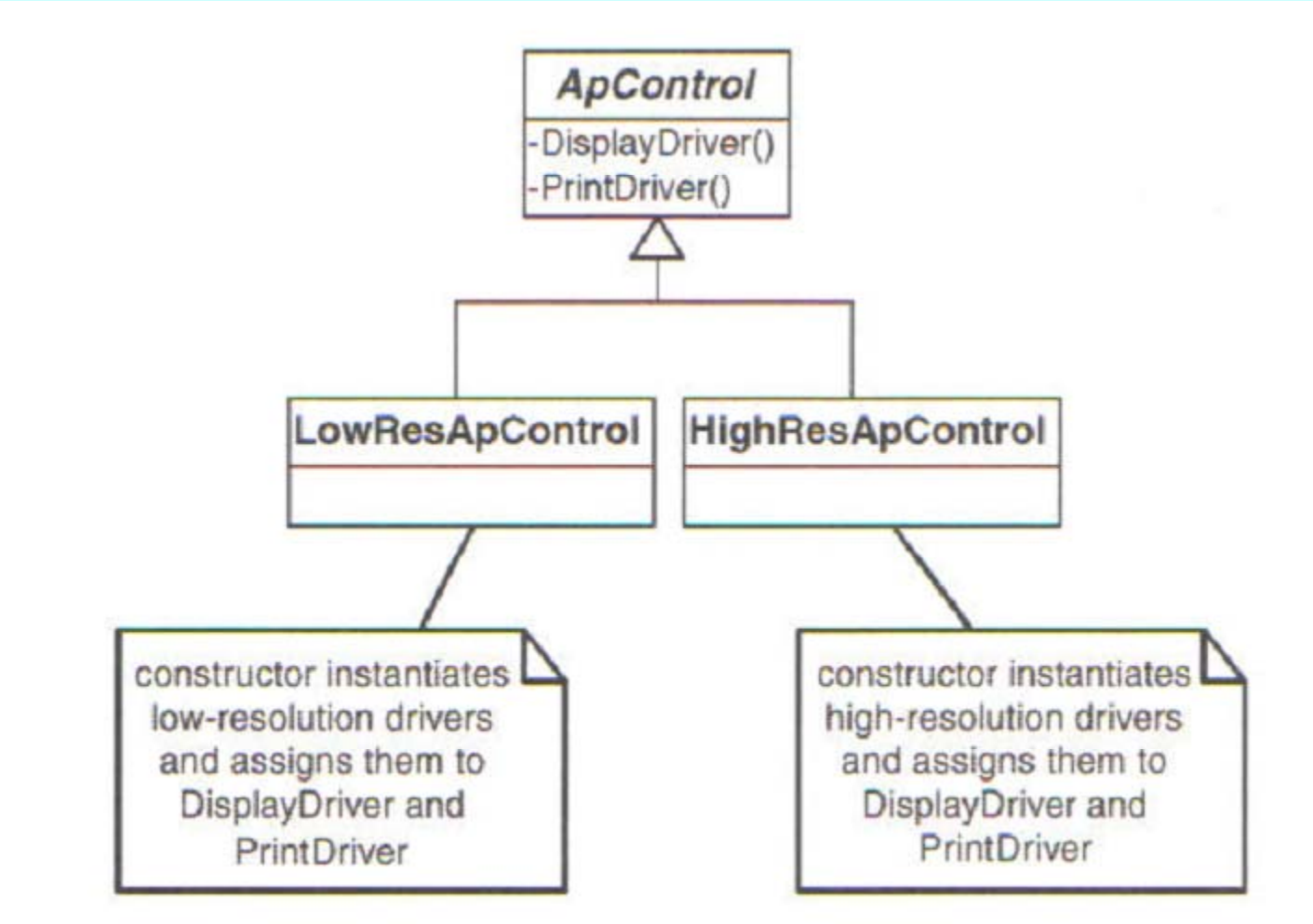
```
public class ApControl  {
    ……
   public void doDraw() {
      ……
      switch(RESOLUTION) {
         case LOW: // use LRDD
         case HIGH: // use HRDD
      }
   }
```

```
    public void doPrint() {
        ……
        switch(RESOLUTION) {
            case LOW: // use LRPD
            case HIGH: // use HRPD
        }
    }
}
```

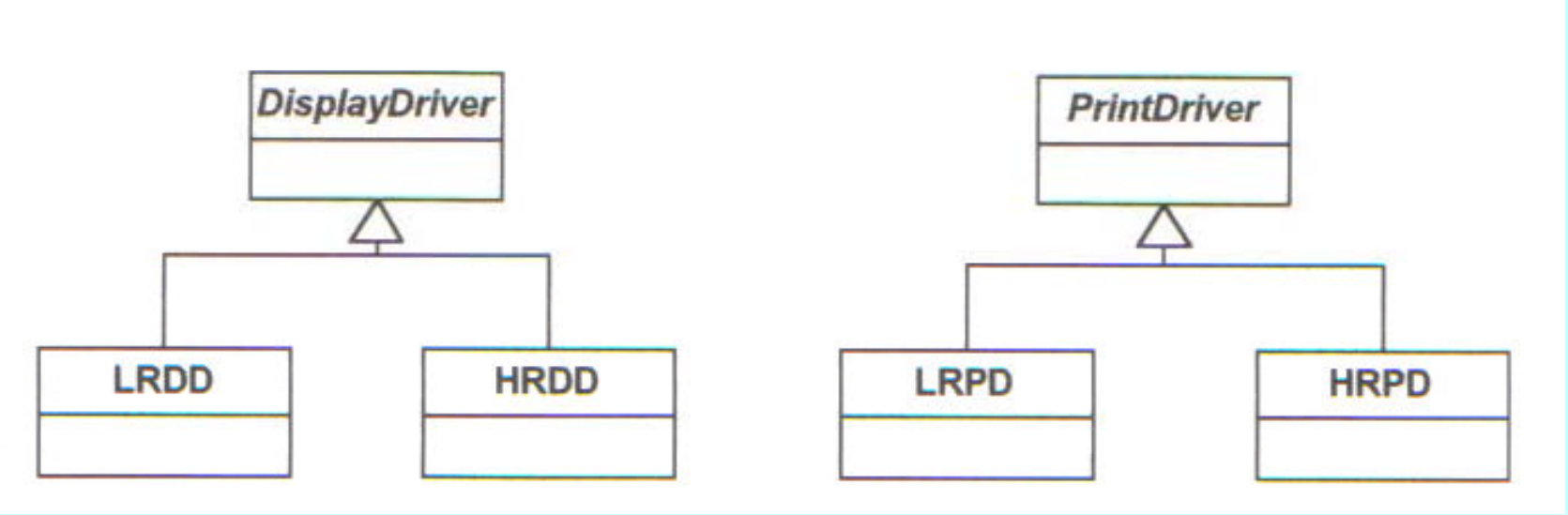
This solution does work, however it has some problems..

Alternative-2: use Inheritance

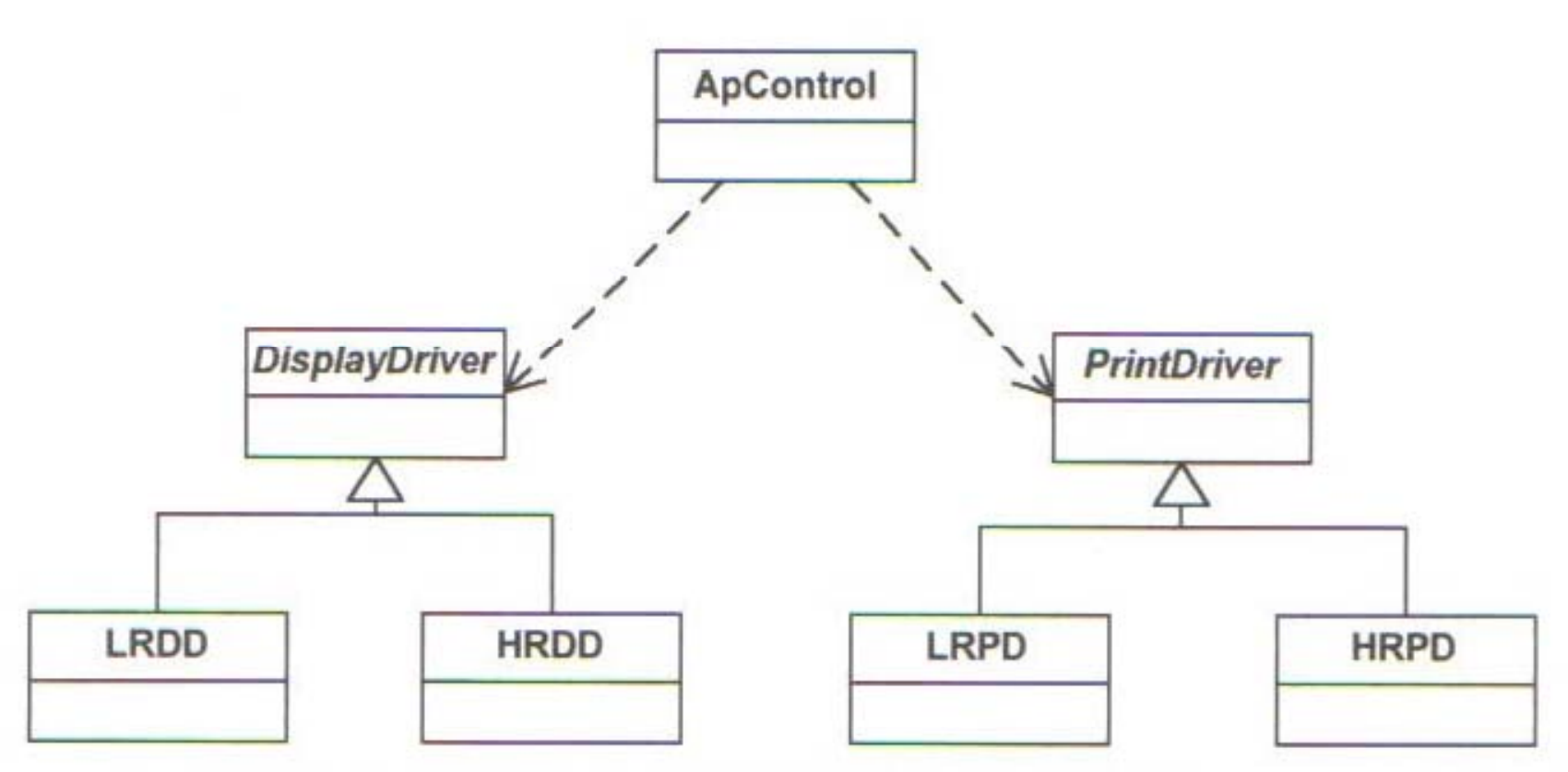
This solution also has some problems..



Alternative-3: Replace switches with abstraction



Alternative-3: (contd.)

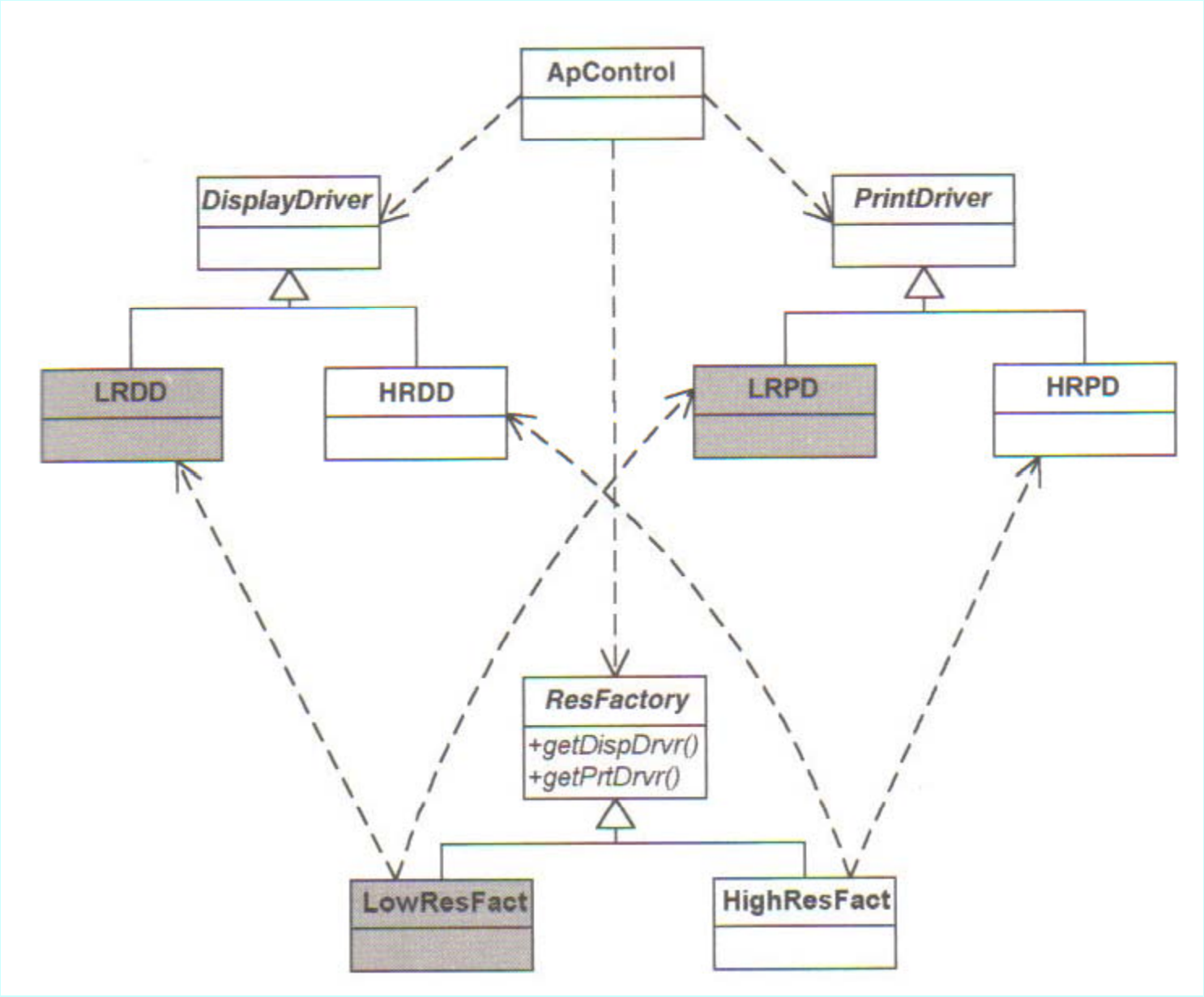


Alternative-3: (contd.)

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

How do I create appropriate objects ?

using Factory objects…


ApControl
DisplayDriver
PrintDriver
LRDD
HRDD
LRPD
HRPD
ResFactory
+getDispDrvr()
+getPrtDrvr()
LowResFact
HighResFact

Alternative-3: Implementation of Factories

```
public abstract class ResFactory  {
   abstract public DisplayDriver getDispDrvr();
   abstract public PrintDriver getPrtDrvr();
}
```

Alternative-3: Implementation of Factories (contd.)

```
public class LowResFact  {
   public DisplayDriver getDispDrvr(){
      return new LRDD();
   }
   public PrintDriver getPrtDrvr()(){
      return new LRPD();
   }
}
```

Alternative-3: Implementation of Factories (contd.)

```
public class HighResFact  {
   public DisplayDriver getDispDrvr(){
      return new HRDD();
   }
   public PrintDriver getPrtDrvr()(){
      return new HRPD();
   }
}
```

• **Another example:** user interfaces, the system might need to use one set of objects to work on one OS and another set of objects to work on a different OS. This pattern ensures that system always gets the correct objects for the situation.

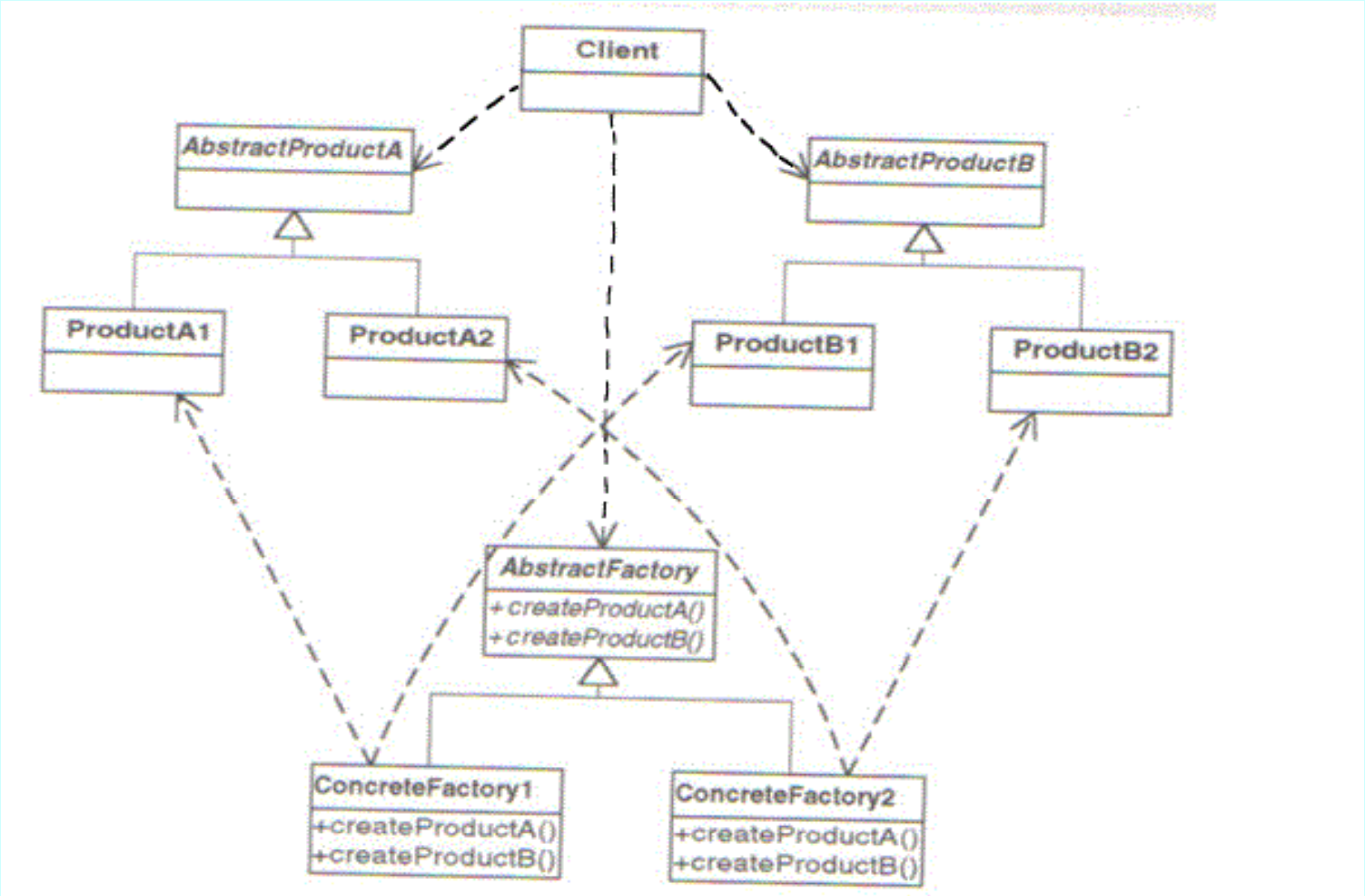
Abstract Factory Pattern (contd.):

Applicability: Use it when:

- a system should be indpt. of how its products are created
- a system should be configured with one of multiple families of objects
- a family of related product objects is designed to be used together
- you want to provide a class library of products, and you want to reveal just their interfaces, not their implementations.

Abstract Factory Pattern (contd.):

- **Generic Structure:**



Abstract Factory Pattern (contd.):

- Participants and Collaborations: AbstractFactory, ConcreteFactory, AbstractProduct, ConcreteProduct, Client.

The AbstractFactory defines the interface for how to create each member of the family of objects. Each family is created by having its own unique ConcreteFactory.

Abstract Factory Pattern (contd.):

- **Consequences:** several benefits and liabilities:
  - it isolates concrete classes
  - it makes exchanging product families easily
  - supporting new kinds of products is difficult

Abstract Factory Pattern (contd.):

- **Implementation (from Shalloway and Trot book):**
  - **define an abstract factory class that specifies which objects are to be created.**
  - **implement one concrete class for each family.**