CHAPTER 2 – DESIGN PATTERNS

2.2 Design Patterns

- ❒ The design patterns are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context.
- ❒ Design pattern is a general reusable solution to commonly occurring problem within a given context in software design.
- ❒ The design pattern identifies the participating classes and instances, their roles and collaborations, and the distribution of responsibilities.

Each design pattern has four essential elements:

1. The **pattern name** is a handle we can use to describe a design problem, its solutions, and consequences in a word or two.
2. The **problem** describes when to apply the pattern. It explains the problem and its context.
3. The **solution** describes the elements that make up the design, their relationships, responsibilities, and collaborations.
4. The **consequences** are the results and trade-offs of applying the pattern.

Gang-Of-Four:

In 1994, four authors Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides published a book titled **Design Patterns - Elements of Reusable Object-Oriented Software** which initiated the concept of Design Pattern in Software development.

These authors are collectively known as **Gang of Four (GOF)**. According to these authors, design patterns are primarily based on the following principles of object orientated design.

- Program to an interface not an implementation
- Favour object composition over inheritance

Describing design patterns:

We describe design patterns using a consistent format. Each pattern is divided into sections according to the following template.

- ✓ **Pattern Name and Classification -** The pattern's name conveys the essence of the pattern succinctly.

- ✓ **Intent -** A short statement that answers the following questions: What does the design pattern do? What is its rationale and intent? What particular design issue or problem does it address?
- ✓ **Also Known As -** Other well-known names for the pattern, if any.
- ✓ **Motivation -** A scenario that illustrates a design problem and how the class and object structures in the pattern solve the problem. The scenario will help you understand the more abstract description of the pattern that follows.
- ✓ **Applicability -** What are the situations in which the design pattern can be applied?
- ✓ **Structure -** A graphical representation of the classes in the pattern using a notation based on the Object Modeling Technique. We also use interaction diagrams to illustrate sequences of requests and collaborations between objects.
- ✓ **Participants -** The classes and/or objects participating in the design pattern and their responsibilities.
- ✓ **Collaborations -** How the participants collaborate to carry out their responsibilities.
- ✓ **Consequences**
- ✓ **Implementation**

Classification of design patterns:

We classify design patterns based on purpose, reflects what a pattern does. Patterns can have **creational**, **structural**, or **behavioral** purpose.

- ★ **Creational patterns** concern the process of object creation.
  - These design patterns provides way to create objects while hiding the creation logic, rather than instantiating objects directly using new operator.
  - This gives program more flexibility in deciding which objects need to be created for a given use case.
- ★ **Structural patterns** deal with the composition of classes or objects.
  - These design patterns concern class and object composition.
  - Concept of inheritance is used to compose interfaces and define ways to compose objects to obtain new functionalities.
- ★ **Behavioral patterns** characterize the ways in which classes or objects interact and distribute responsibility.

- These design patterns are specifically concerned with communication between objects.

		Purpose		
		Creational	Structural	Behavioral
Scope	Class	Factory Method (121)	Adapter (157)	Interpreter (274) Template Method (360)
	Object	Abstract Factory (99) Builder (110) Prototype (133) Singleton (144)	Adapter (157) Bridge (171) Composite (183) Decorator (196) Facade (208) Flyweight (218) Proxy (233)	Chain of Responsibility (251) Command (263) Iterator (289) Mediator (305) Memento (316) Observer (326) State (338) Strategy (349) Visitor (366)

Creational Design Patterns:

- Creational design patterns abstract the instantiation process.
- They help make a system independent of how its objects are created, composed, and represented.
- There are two recurring themes in these patterns.
  - First, they all encapsulate knowledge about which concrete classes the system uses.
  - Second, they hide how instances of these classes are created and put together.

The various Creational patterns are:

- **Abstract Factory:** Provide an interface for creating families of related or dependent objects without specifying their concrete classes.
- **Builder**: Separate the construction of a complex object from its representation so that the same construction process can create different representations.
- **Factory Method:** define an interface for creating an object, but let subclass decide which class to instantiate. Factory method les a class defer instantiation to subclasses.
- **Prototype:** specify the kinds of objects to create using a prototypical object, and create new objects y copying this prototype.

- **Singleton:** Ensure a class only has one object, and provide a global point of access to it.

FACTORY METHOD

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.

Also Known As

Virtual Constructor

Motivation

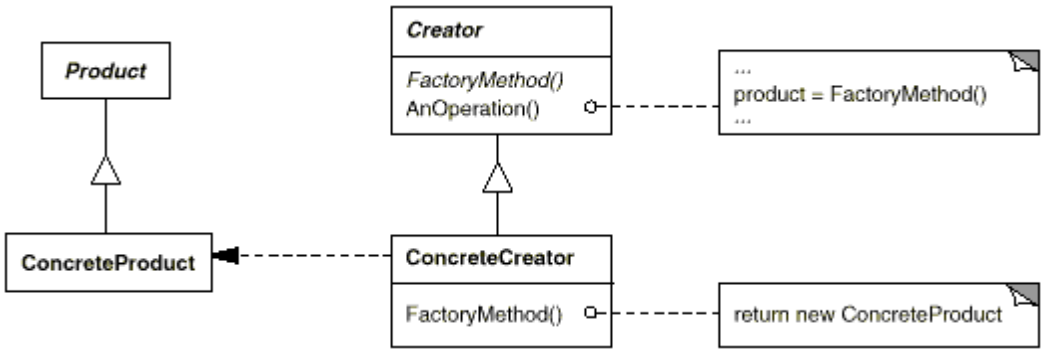
Frameworks use abstract classes to define and maintain relationships between objects. A framework is often responsible for creating these objects as well.

Applicability

Use the Factory Method pattern when

- A class can't anticipate the class of objects it must create.
- A class wants its subclasses to specify the objects it creates.
- Classes delegate responsibility to one of several helper subclasses, and you want to localize the knowledge of which helper subclass is the delegate.

Structure



Participants

· **Product** -> defines the interface of objects the factory method creates.

· **ConcreteProduct->**implements the Product interface.

· **Creator->** declares the factory method, which returns an object of type Product.

Creator may also define a default implementation of the factory method that returns a default ConcreteProduct object.

May call the factory method to create a Product object.

· **ConcreteCreator->**overrides the factory method to return an instance of a ConcreteProduct.

Collaborations

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

Consequences

Factory methods eliminate the need to bind application-specific classes into your code. The code only deals with the Product interface; therefore it can work with any user-defined ConcreteProduct classes.

A potential disadvantage of factory methods is that clients might have to subclass the Creator class just to create a particular ConcreteProduct object. Subclassing is fine when the client has to subclass the Creator class anyway, but otherwise the client now must deal with another point of evolution.

Here are two additional consequences of the Factory Method pattern:

1. Provides hooks for subclasses. Creating objects inside a class with a factory method is always more flexible than creating an object directly. Factory Method gives subclasses a hook for providing an extended version of an object.
2. Connects parallel class hierarchies.

Implementation

Consider the following issues when applying the Factory Method pattern:

1. **Two major varieties.** The two main variations of the Factory Method pattern are (1) the case when the Creator class is an abstract class and does not provide an implementation for the factory method it declares, and (2) the case when the Creator is a concrete class and provides a default implementation for the factory method.

    - ► The first case requires subclasses to define an implementation, because there's no reasonable default. It gets around the dilemma of having to instantiate unforeseeable classes.
    - ► In the second case, the concrete Creator uses the factory method primarily for flexibility.

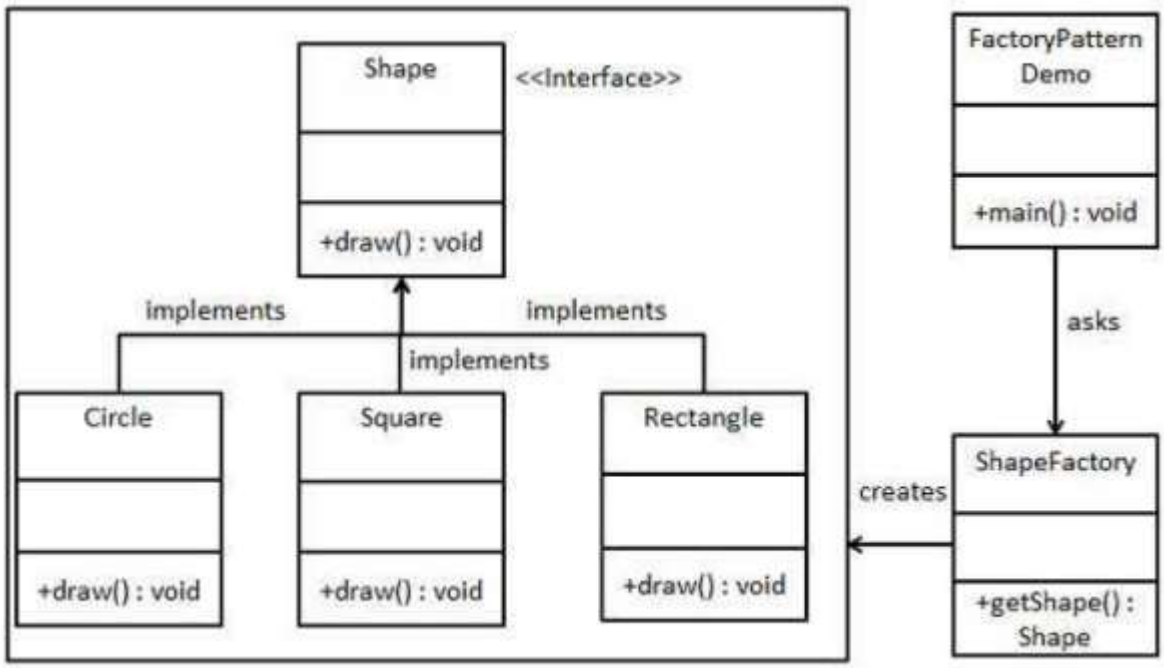
2. **Parameterized factory methods**: Another variation on the pattern lets the factory method create multiple kinds of products. The factory method takes a parameter that identifies the kind of object to create. All objects the factory method creates will share the Product interface.
3. **Using templates to avoid subclassing:** Another potential problem with factory methods is that they might force you to subclass just to create the appropriate Product objects. Another way to get around this in C++ is to

provide a template subclass of Creator that's parameterized by the Product class.

Example:

We're going to create a Shape interface and concrete classes implementing the Shape interface. A factory class ShapeFactory is defined as a next step.

FactoryPatternDemo, our demo class will use ShapeFactory to get a Shape object. It will pass information (CIRCLE / RECTANGLE / SQUARE) to ShapeFactory to get the type of object it needs.



STEP 1

Create an interface.

Shape.java

```
public interface Shape {
   void draw();
}
```

STEP 2

Create concrete classes implementing the same interface.

Rectangle.java

```
public class Rectangle implements Shape {

   @Override
   public void draw() {
      System.out.println("Inside Rectangle::draw() method.");
   }
}
```

Square.java

```
public class Square implements Shape {

   @Override
   public void draw() {
      System.out.println("Inside Square::draw() method.");
   }
}
```

Circle.java

```
public class Circle implements Shape {

   @Override
   public void draw() {
      System.out.println("Inside Circle::draw() method.");
   }
}
```

STEP 3

Create a Factory to generate object of concrete class based on given information.

ShapeFactory.java

```
public class ShapeFactory {

   //use getShape method to get object of type shape
   public Shape getShape(String shapeType){
      if(shapeType == null){
         return null;
```

```
    }
    if(shapeType.equalsIgnoreCase("CIRCLE")){
      return new Circle();
    } else if(shapeType.equalsIgnoreCase("RECTANGLE")){
      return new Rectangle();
    } else if(shapeType.equalsIgnoreCase("SQUARE")){
      return new Square();
    }
    return null;
  }
}
```

STEP 4

Use the Factory to get object of concrete class by passing an information such as type.

FactoryPatternDemo.java

```
public class FactoryPatternDemo {

  public static void main(String[] args) {
    ShapeFactory shapeFactory = new ShapeFactory();

    //get an object of Circle and call its draw method.
    Shape shape1 = shapeFactory.getShape("CIRCLE");

    //call draw method of Circle
    shape1.draw();

    //get an object of Rectangle and call its draw method.
    Shape shape2 = shapeFactory.getShape("RECTANGLE");

    //call draw method of Rectangle
    shape2.draw();

    //get an object of Square and call its draw method.
    Shape shape3 = shapeFactory.getShape("SQUARE");

    //call draw method of circle
    shape3.draw();
  }
```

}

STEP 5

Verify the output.

Inside Circle::draw() method.
Inside Rectangle::draw() method.
Inside Square::draw() method.

Factory objects have several advantages:
• Separate the responsibility of complex creation into cohesive helper objects.
• Hide potentially complex creation logic.
• Allow introduction of performance-enhancing memory management strategies, such as object caching or recycling.

2.2.2 STRUCTURAL PATTERNS

Structural patterns are concerned with how classes and objects are composed to form larger structures. This pattern is particularly useful for making independently developed class libraries work together. Structural object patterns describe ways to compose objects to realize new functionality.
Consider how multiple inheritance mixes two or more classes into one. The result is a class that combines the properties of its parent classes.
Some structural design patterns and their intents are as follows:

- **Adapter:** Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
- **Bridge:** Decouple an abstraction from its implementation so that the two can vary independently.
- **Façade:** Provide a unified interface to a set of interfaces in a subsystem.
- **Decorator:** Attach additional responsibilities to an object dynamically.
- **Proxy:** Provide a surrogate or placeholder for another object to control access to it.

Adapter

Context / Problem How to resolve incompatible interfaces, or provide a stable interface to similar components with different interfaces? Solution Convert the original interface of a component into another interface, through an intermediate adapter object.

Intent

Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

Also Known As - Wrapper

Example:

To review: The NextGEN POS system needs to support several kinds of external third-party services, including tax calculators, credit authorization services, inventory systems, and accounting systems, among others. Each has a different API, which can't be changed.

A solution is to add a level of indirection with objects that adapt the varying external interfaces to a consistent interface used within the application.

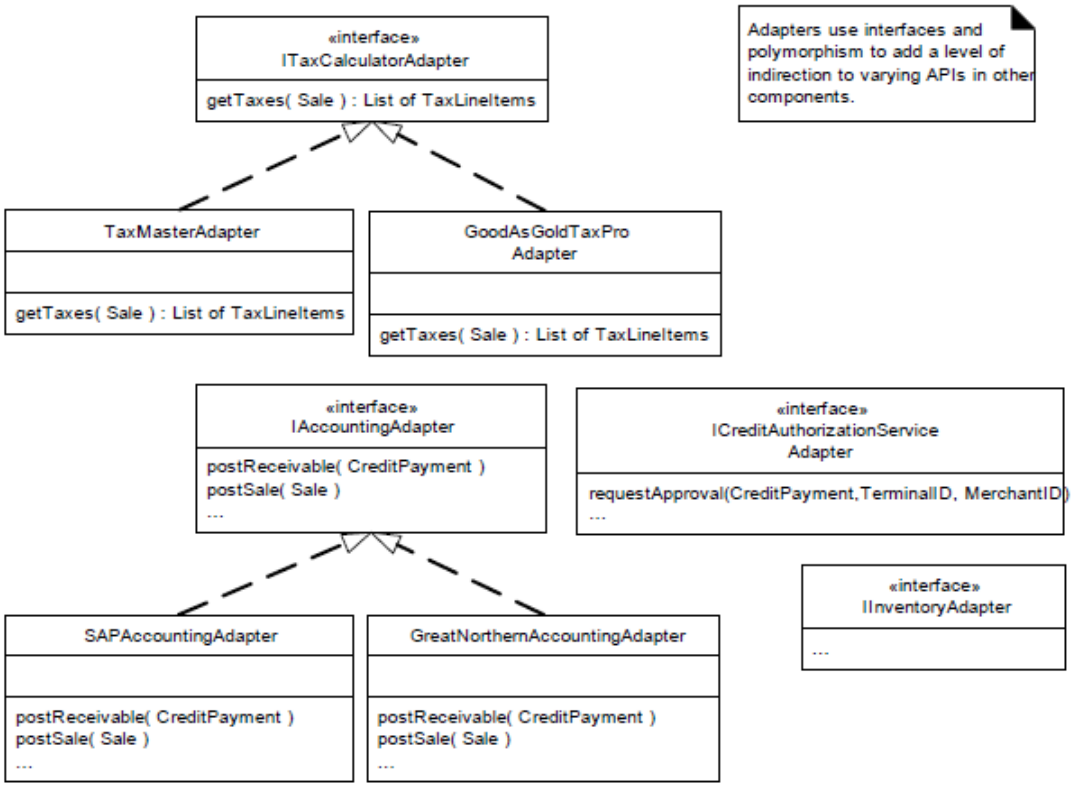


Figure 23.1 The Adapter pattern.

A particular adapter instance will be instantiated. For the chosen external service3, such as SAP for accounting, and will adapt the postSale request to the external interface, such as a SOAP XML interface over HTTPS for an intranet Web service offered by SAP.

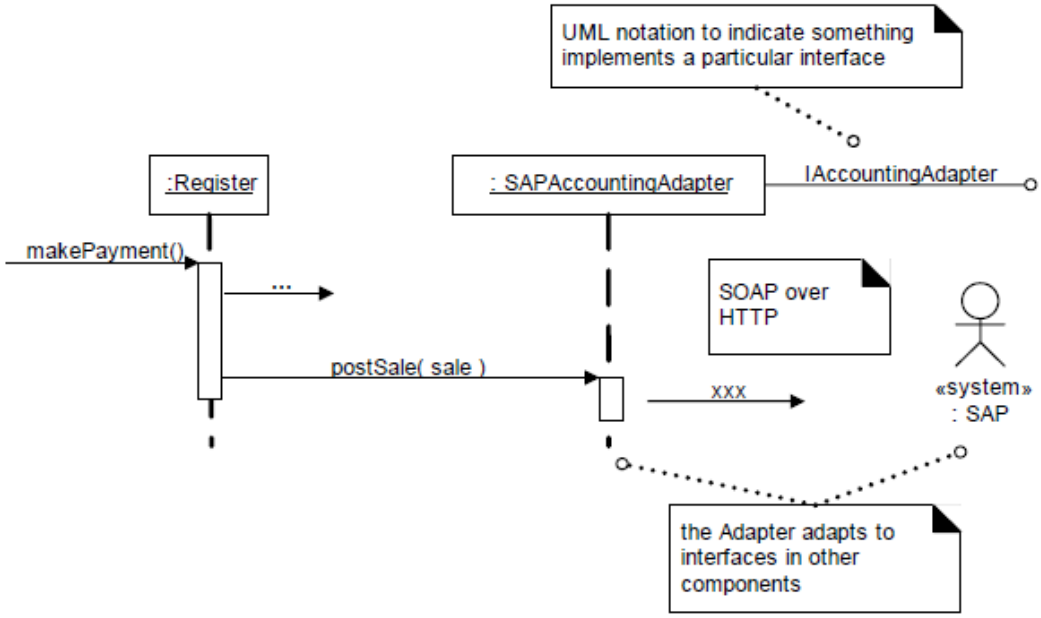


Figure 23.2 Using an Adapter.

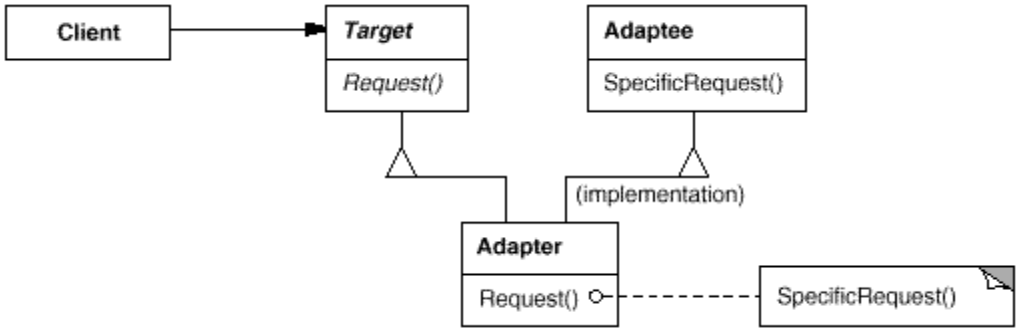
Applicability

Use the Adapter pattern when

- You want to use an existing class, and its interface does not match the one you need.
- You want to create a reusable class that cooperates with unrelated or unforeseen classes, that is, classes that don't necessarily have compatible interfaces.(object adapter only)
- You need to use several existing subclasses, but it's impractical to adapt their interface by subclassing every one. An object adapter can adapt the interface of its parent class.

Structure

A class adapter uses multiple inheritance to adapt one interface to another:



Participants

- **Target -** defines the domain-specific interface that Client uses.
- **Client** - collaborates with objects conforming to the Target interface.
- **Adaptee -** defines an existing interface that needs adapting.
- **Adapter -** Adapts the interface of Adaptee to the Target interface.

Collaborations

Clients call operations on an Adapter instance. In turn, the adapter calls adaptee operations that carry out the request.

Consequences

Class and object adapters have different trade-offs. A class adapter

- Adapts Adaptee to Target by committing to a concrete Adapter class. As a consequence, a class adapter won't work when we want to adapt a class and all its subclasses.
- Lets Adapter override some of Adaptee's behavior, since Adapter is a subclass of Adaptee.
- Introduces only one object and no additional pointer indirection is needed to get to the adaptee.

Here are other issues to consider when using the Adapter pattern:

1. **How much adapting does Adapter do?** Adapters vary in the amount of work they do to adapt Adaptee to the Target interface. The amount of work Adapter does depends on how similar the Target interface is to Adaptee's.

2**. Pluggable adapters**. A class is more reusable when you minimize the assumptions other classes must make to use it. By building interface adaptation into a class, you eliminate the assumption that other classes see the same interface.

Implementation

Although the implementation of Adapter is usually straightforward, here are some issues to keep in mind:

1. **Implementing class adapters in C++:** In a C++ implementation of a class adapter, Adapter would inherit publicly from Target and privately from Adaptee. Thus Adapter would be a subtype of Target but not of Adaptee.

2.2.2.2 BRIDGE:

Intent:

- Decouple an abstraction from its implementation so that the two can vary independently

Also known as:

Handle/Body

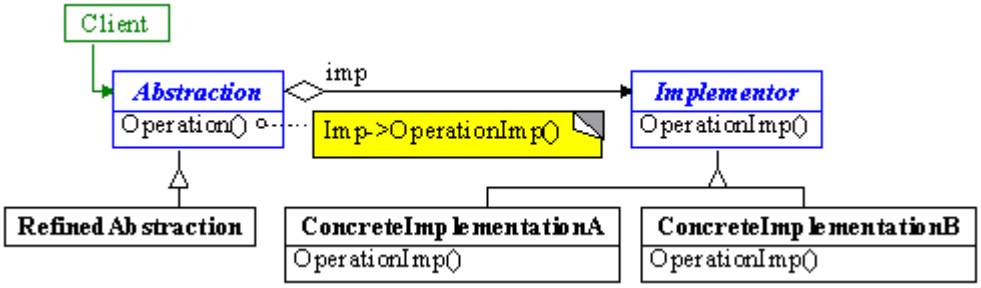
Motivation:

- **Problem:** For some classes, we want to adapt (reuse) either their abstractions, their implementations or both. How can we structure a class so that abstractions and/or implementations can easily be modified?
- Inheritance binds an implementation to the abstraction permanently, which makes it difficult to modify, extend, and reuse abstraction and implementations independently.

Applicability:

- Need to avoid a permanent binding between an abstraction and implementation.
- When abstractions and implementations should be extensible through subclassing.
- When implementation changes should not impact clients.
- When the implementation should be completely hidden from the client. (C++)
- When you have a proliferation of classes.
- When, unknown to the client, implementations are shared among objects

Structure:



Participants and Collaborations:

- **Abstraction** - defines the abstraction's interface
  - maintains a reference to the Implementor
  - forwards requests to the Implementor (collaboration)
- **RefinedAbstraction** - extends abstraction interface
- **Implementor** - defines interface for implementations
- **ConcreteImplementor** - implements Implementor interface, i.e. defines an implementation

Consequences:

- **Decouples interface and implementation:** Decoupling Abstraction and Implementor also eliminates compile-time dependencies on implementation. Changing implementation class does not require recompile of abstraction classes.
- **Improves extensibility :** Both abstraction and implementations can be extended independently

- Hides implementation details from clients
- More of a design-time pattern

Example:

Task assigned is to draw a rectangle given the end points. We can use either drawing program 1 or drawing program 2.

The rectangles are defined as two pairs of points, as represented in the following figures and the differences between the drawing programs are summarized in the following table.

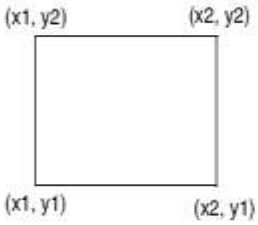


Figure 9-1 Positioning the rectangle.

Table 9-1 Different Drawing Programs

	DP1	DP2
Used to draw a line	`draw_a_line( x1, y1, x2, y2)`	`drawline( x1, x2, y1, y2)`
Used to draw a circle	`draw_a_circle( x, y, r)`	`drawcircle( x, y, r)`

By having an abstract class Rectangle, I take advantage of the fact that the only difference between the different types of Rectangle is how they implement the drawLine method. The V1Rectangle is implemented by having a reference to a DP1 object and using that object's draw_a_line method. The V2Rectangle is implemented by having a reference to a DP2 object and using that object's drawline method.

So the design is as follows:

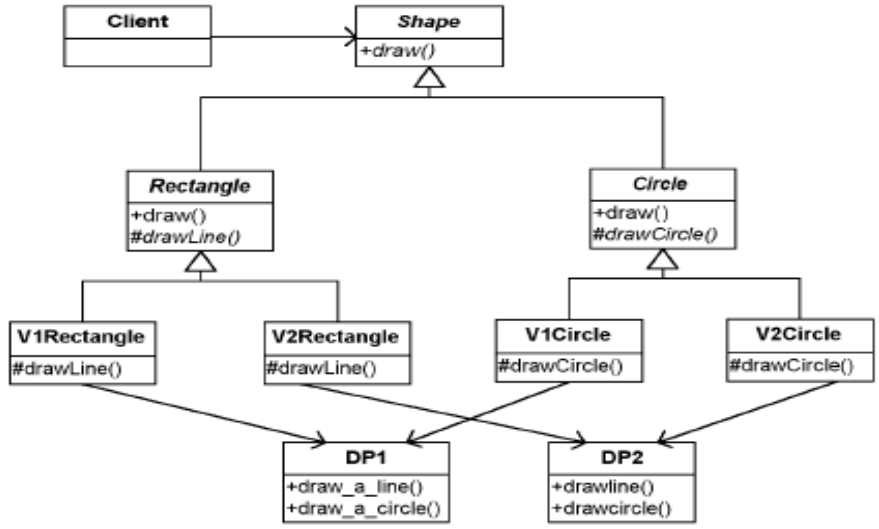


Figure 9-3 A straightforward approach: implementing two shapes and two drawing programs.

The class explosion problem arises because in this design, the coupling abstraction (the kinds of Shapes) and the implementation (the drawing programs) are tightly coupled.
Each type of shape must know what type of drawing program it is using. We need a way to separate the variations in abstraction from the variations in implementation so that the number of classes only grows linearly.
There comes the need for bridge pattern.

Figure 9-6 The Bridge pattern separates variations in abstraction and implementation.

The Bridge pattern is useful when you have an abstraction that has different implementations. It allows the abstraction and the implementation to vary independently of each other.

- You have to do two things first.

1- Commonality analysis
2-Variablity analysis
The common concepts will be represented by abstract classes; variations will be implemented by concrete classes.
In this case we different types of shapes and different type of drawing programs. So common concepts are **shape** and **Drawing.** So we encapsulate them

Figure 9-9 What is varying.

The next step is to represent the specific variations that are present. For Shape, rectangles and circles. For drawing programs,DP1 and DP2 respectively.

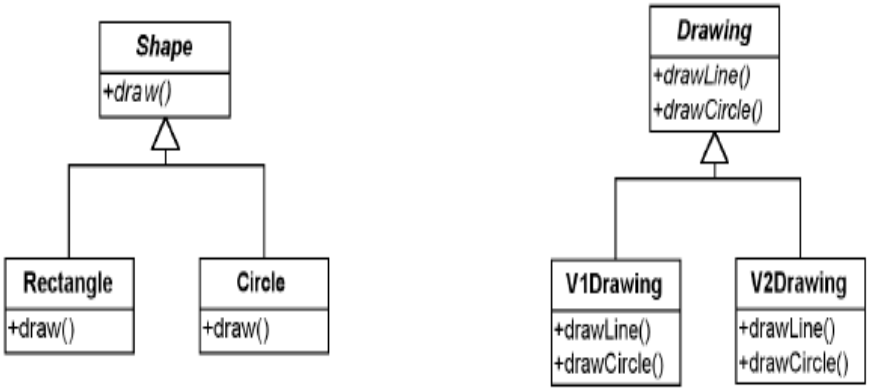


Figure 9-10 Represent the variations.

So the design is as follows:

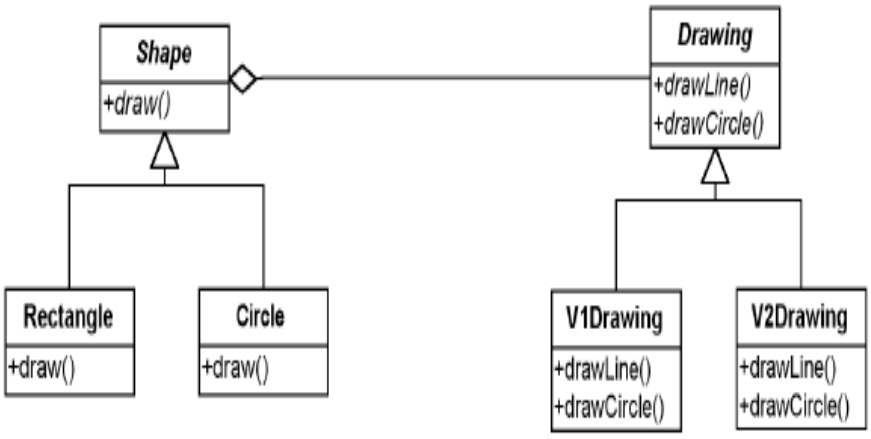


Figure 9-11 Tie the classes together.

Class diagram for the above design is as follows:

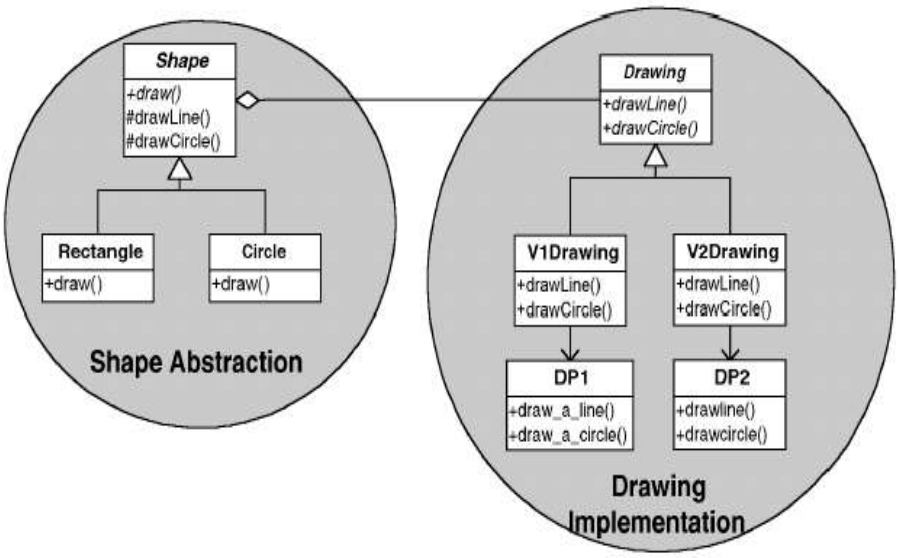


Figure 9-13 Class diagram illustrating separation of abstraction and implementation.

```
class Client {
  public static void main
    (String argv[]) {
      Shape r1, r2;
      Drawing dp;

      dp= new V1Drawing();
      r1= new Rectangle(dp,1,1,2,2);

      dp= new V2Drawing ();
      r2= new Circle(dp,2,2,3);

      r1.draw();
      r2.draw();
   }
}
```

```
 abstract class Shape {
   abstract public draw() ;
   private Drawing _dp;

   Shape (Drawing dp) {
     _dp= dp;
   }
   public void drawLine (
     double x1,double y1,
     double x2,double y2) {
       _dp.drawLine(x1,y1,x2,y2);
   }

   public void drawCircle (
     double x,double y,double r) {
       _dp.drawCircle(x,y,r);
   }
 }
```

```
abstract class Drawing {
  abstract public void drawLine (
    double x1, double y1,
    double x2, double y2);

 abstract public void drawCircle (
   double x,double y,double r);
}
```

```
class Rectangle extends Shape {
  public Rectangle (
    Drawing dp,
    double x1,double y1,
    double x2,double y2) {
      super( dp) ;
      _x1= x1; _x2= x2 ;
      _y1= y1; _y2= y2;
  }

  public void draw () {
   drawLine(_x1,_y1,_x2,_y1);
   drawLine(_x2,_y1,_x2,_y2);


  drawLine(_x2,_y2,_x1,_y2);
  drawLine(_x1,_y2,_x1,_y1);
 }
}
```

2.2.3 BEHAVIORAL PATTERNS

Behavioral patterns are concerned with algorithms and the assignment of responsibilities between objects. Behavioral class patterns use inheritance to distribute behavior between classes. Behavioral object patterns use object composition rather than inheritance. Some describe how a group of peer objects cooperates to perform a task that no single object can carry out by itself. An important issue here is how peer objects know about each other.

Some of the behavioral patterns and their intents are:

- **Command:** Encapsulate a request as an object, thereby letting you parameterize clients with different requests.
- **Iteration:** Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
- **Observer:** Define a one-to-one dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
- **Strategy:** Define a family of algorithms, encapsulate each one, and make them interchangeable.

STRATEGY

Intent

Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

Also Known As - Policy

Strategy

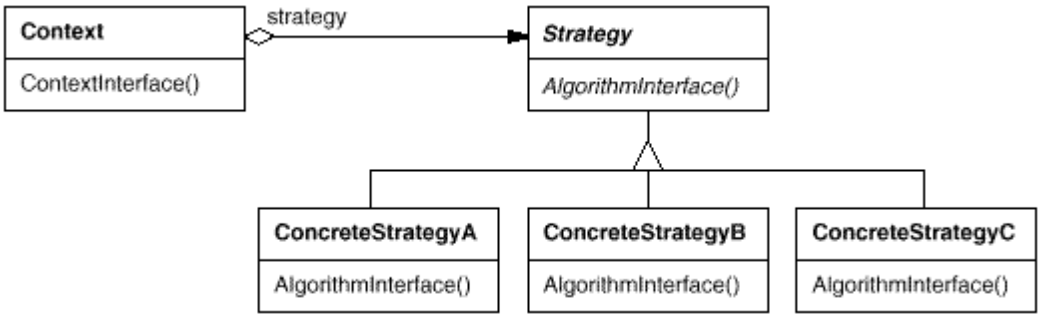
Context / Problem

How to design for varying, but related, algorithms or policies? How to design for the ability to change these algorithms or policies?

Solution

Define each algorithm/policy/strategy in a separate class, with a common interface.

Structure:



Applicability

Use the Strategy pattern when

- ❒ Many related classes differ only in their behavior. Strategies provide a way to configure a class with one of many behaviors.
- ❒ You need different variants of an algorithm. Strategies can be used when these variants are implemented as a class hierarchy of algorithms
- ❒ An algorithm uses data that clients shouldn't know about. Use the Strategy pattern to avoid exposing complex, algorithm-specific data structures.

Participants

- ★ **Strategy-** declares an interface common to all supported algorithms. Context uses this interface to call the algorithm defined by a ConcreteStrategy.
- ★ **ConcreteStrategy -** implements the algorithm using the Strategy interface.
- ★ **Context**
  - Is configured with a ConcreteStrategy object.
  - Maintains a reference to a Strategy object.
  - may define an interface that lets Strategy access its data

Consequences

The Strategy pattern has the following benefits and drawbacks:

1. **Families of related algorithms:** Hierarchies of Strategy classes define a family of algorithms or behaviors for contexts to reuse.
2. **An alternative to subclassing**: Inheritance offers another way to support a variety of algorithms or behaviors. You can subclass a Context class directly to give it different behaviors. Encapsulating the algorithm in separate Strategy classes lets you vary the algorithm independently of its context, making it easier to switch, understand, and extend.
3. **Strategies eliminate conditional statements:** The Strategy pattern offers an alternative to conditional statements for selecting desired behavior. When different behaviors are lumped into one class, it's hard to avoid using conditional statements to select the right behavior. Encapsulating the behave or in separate Strategy classes eliminates these conditional statements.

Implementation

Consider the following implementation issues:

1. **Defining the Strategy and Context interfaces:** The Strategy and Context interfaces must give a ConcreteStrategy efficient access to any data it needs from a context, and vice versa.
    a. One approach is to have Context pass data in parameters to Strategyoperations—in other words, take the data to the strategy. This keeps Strategy and Context decoupled. On the other hand, Context might pass data the Strategy doesn't need.
2. **Strategies as template parameters.** In C++ templates can be used to configure a class with a strategy. This technique is only applicable if
    a. the Strategy can be selected at compile-time
    b. it does not have to be changed at run-time.

Example

Since the behavior of pricing varies by the strategy (or algorithm), we create multiple SalePricingStrategy classes, each with a polymorphic getTotal method. Each getTotal method takes the Sale object as a parameter, so that the pricing strategy object can find the pre-discount price from the Sale, and then apply the discounting rule. The implementation of each getTotal method will be different: PercentDiscountPricingStrategy will discount by a percentage, and so on.

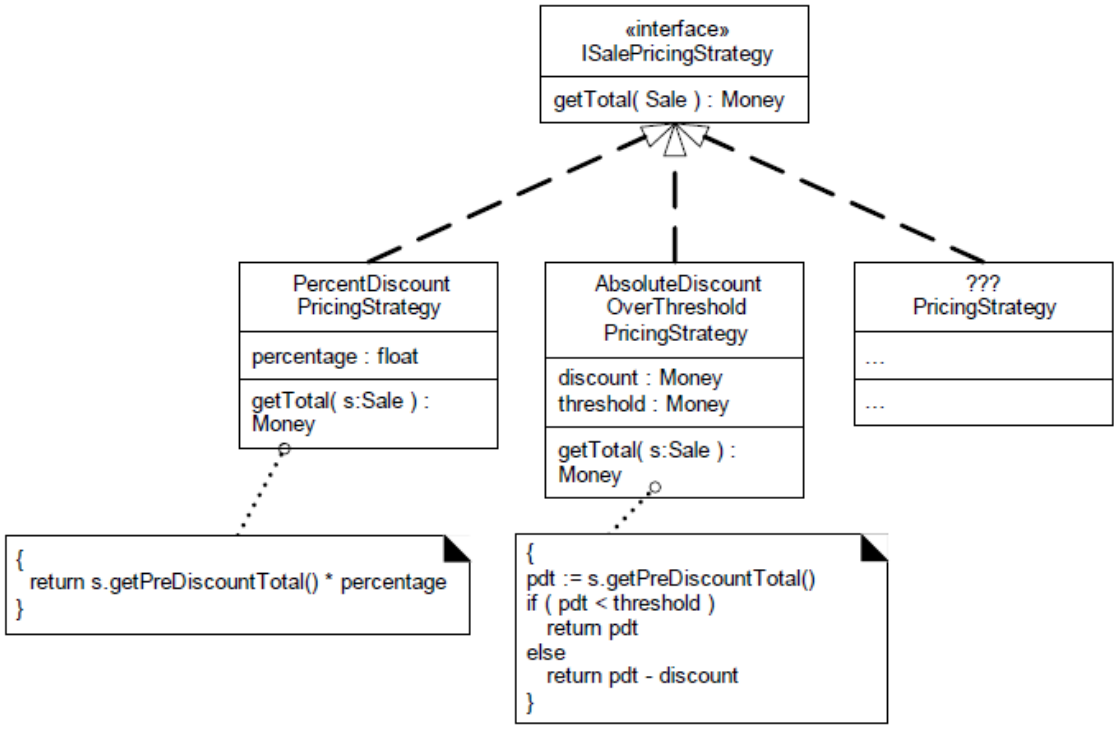


Figure 23.8 Pricing Strategy classes.

A strategy object is attached to a **context object**—the object to which it applies the algorithm. In this example, the context object is a Sale. When a getTotal message is sent to a Sale, it delegates some of the work to its strategy object, as illustrated in Figure 23.9. It is not required that the message to the context object and the strategy object have the same name, as in this example (for example, getTotal and getTotal), but it is common. However, it is common—indeed, usually required—that the context object pass a reference to itself (this) on to the strategy object, so that the strategy has parameter visibility to the context object,for further collaboration.

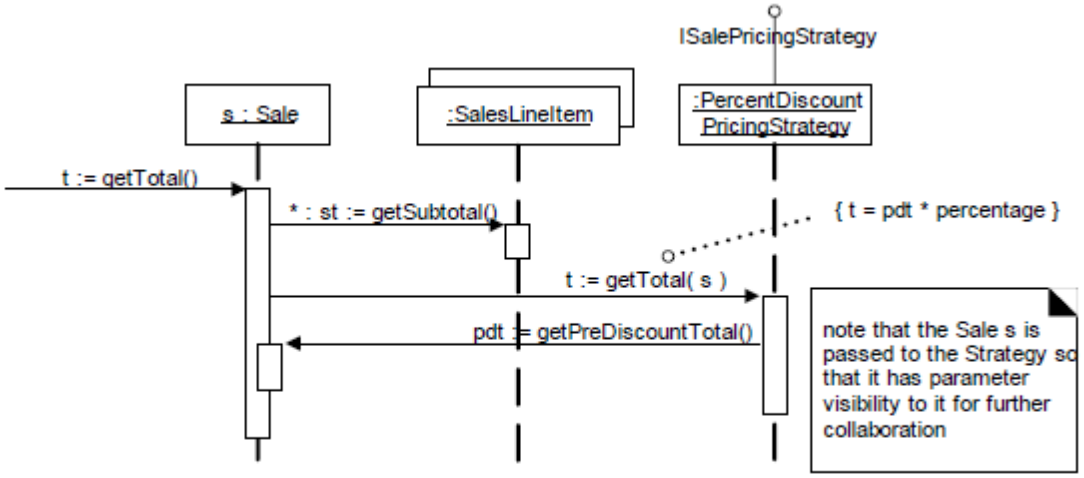


Figure 23.9 Strategy in collaboration.

Observer : refer to book