Chain of Responsibility Pattern

Intent

Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

Motivation

a. From the client's perspective, a request needs to be handled.

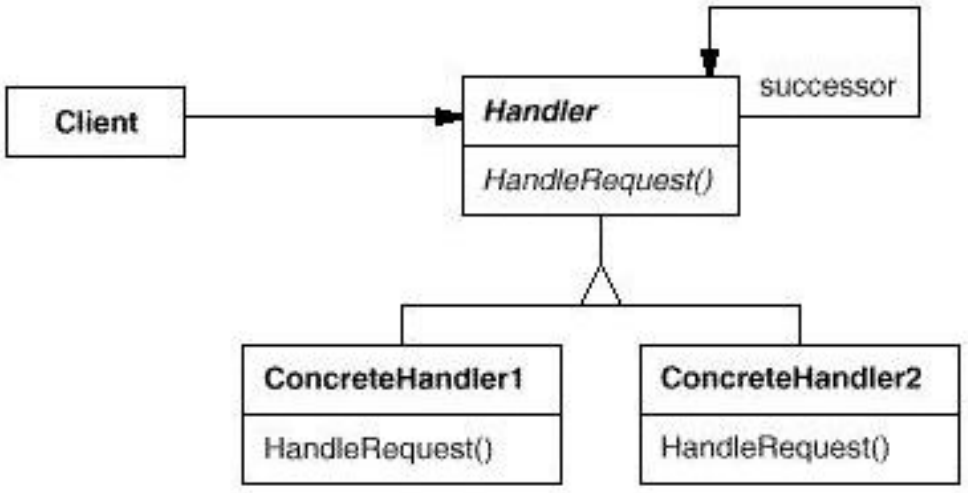
b. From the handlers‟ perspective, there are multiple resources with different skills, different roles, or different responsibilities organized in a chain structure.

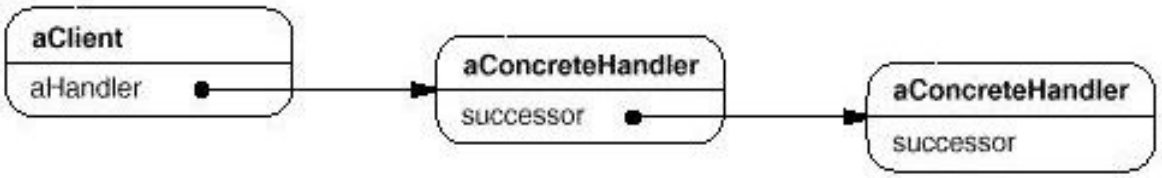
c. These resources only know their successors (whom to pass the request to if they are not able to handle it). And each resource has only one successor.

d. Request processing always starts from the first resource in the chain. If request gets processed, the work for processing the request terminates right away. If not, the request is passed to its successor.

e. The client always expects all requests get handled.

Structure





Participants

a. **Handler**

- defines an interface for handling requests.

- (optional) implements the successor link.

b. **ConcreteHandler**

- Handles requests it is responsible for.

- Holds a reference to its successor.

- If the ConcreteHandler can handle the request, it does so; otherwise it forwards the request to its successor.

c. **Client**

- Initiates the request to a ConcreteHandler object in the chain.

Applicability

a. When more than one object may handle a request and the actual handler is not known in advance.

b. You want to issue a request to one of several objects without specifying the receiver explicitly.

c. The set of objects that can handle a request should be specified dynamically.

d. When you process requests with a “handle or forward” model.

Consequences

a. Reduced coupling between the sender of a request and the receiver - the sender and receiver have no explicit knowledge of each other (except that the sender knows where to send the request).

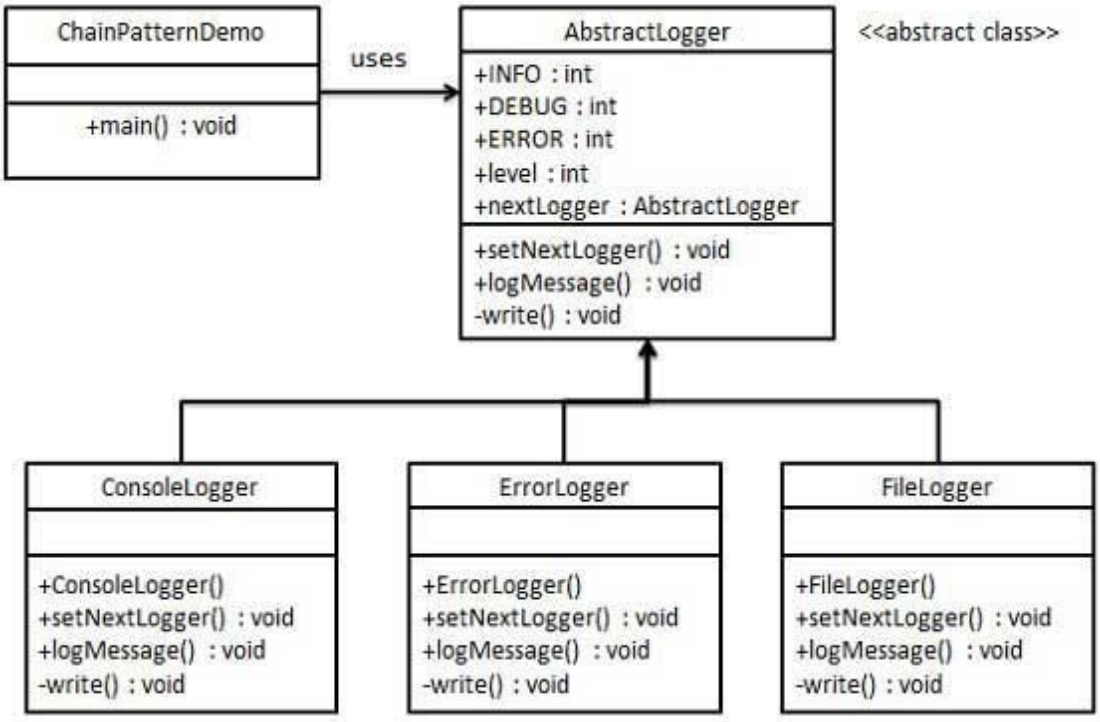
b. Added flexibility in assigning responsibilities to objects. Chain of Responsibility gives

you added flexibility in distributing responsibilities among objects. You can add or change responsibilities for handling a request by adding to or otherwise changing the chain at run-time.

c. Technically, the chain does not consider how receipt is guaranteed - a request could fall off the end of the chain without being handled (within a distributed environment, for example)

Implementation

We have created an abstract class *AbstractLogger* with a level of logging. Then we have created three types of loggers extending the *AbstractLogger*. Each logger checks the level of message to its level and print accordingly otherwise does not print and pass the message to its next logger.



Step 1

Create an abstract logger class.

AbstractLogger.java

```
public abstract class AbstractLogger {
  public static int INFO = 1;
  public static int DEBUG = 2;
  public static int ERROR = 3;
```

```
  protected int level;
  //next element in chain or responsibility
  protected AbstractLogger nextLogger;
  public void setNextLogger(AbstractLogger nextLogger){
    this.nextLogger = nextLogger;
  }
  public void logMessage(int level, String message){
    if(this.level <= level){
      write(message);
    }
    if(nextLogger !=null){
      nextLogger.logMessage(level, message);
    }
  }
  abstract protected void write(String message);
}
```

Step 2

Create concrete classes extending the logger.

ConsoleLogger.java

```
public class ConsoleLogger extends AbstractLogger {
  public ConsoleLogger(int level){
    this.level = level;
  }
  @Override
  protected void write(String message) {
    System.out.println("Standard Console::Logger: " + message);
  }
}
```

ErrorLogger.java

```
public class ErrorLogger extends AbstractLogger {
  public ErrorLogger(int level){
    this.level = level;
  }
  @Override
  protected void write(String message) {
    System.out.println("Error Console::Logger: " + message);
  }
}
```

FileLogger.java

```
public class FileLogger extends AbstractLogger {
  public FileLogger(int level){
    this.level = level;
  }
  @Override
  protected void write(String message) {
    System.out.println("File::Logger: " + message);
  }
}
```

Step 3

Create different types of loggers. Assign them error levels and set next logger in each logger. Next logger in each logger represents the part of the chain.

ChainPatternDemo.java

```
public class ChainPatternDemo {
  private static AbstractLogger getChainOfLoggers(){
    AbstractLogger errorLogger = new ErrorLogger(AbstractLogger.ERROR);
```

```
    AbstractLogger fileLogger = new FileLogger(AbstractLogger.DEBUG);
    AbstractLogger consoleLogger = new ConsoleLogger(AbstractLogger.INFO);
    errorLogger.setNextLogger(fileLogger);
    fileLogger.setNextLogger(consoleLogger);
    return errorLogger;
  }

  public static void main(String[] args) {
    AbstractLogger loggerChain = getChainOfLoggers();
    loggerChain.logMessage(AbstractLogger.INFO,
      "This is an information.");
    loggerChain.logMessage(AbstractLogger.DEBUG,
      "This is an debug level information.");
    loggerChain.logMessage(AbstractLogger.ERROR,
      "This is an error information.");
  }
}
```

Step 4

Verify the output.

```
Standard Console::Logger: This is an information.
File::Logger: This is an debug level information.
Standard Console::Logger: This is an debug level information.
Error Console::Logger: This is an error information.
File::Logger: This is an error information.
Standard Console::Logger: This is an error information.
```

Command

Intent

Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

Motivation

f. Sometimes it's necessary to issue requests without knowing anything about the operation being requested or the receiver of the request.

g. For some applications, you have to store and pass requests around like other objects.

h. For unsuccessfully executed requests, roll-back is required.

Applicability

Use the Command pattern when you want to:

a. Pass an object as an action to perform. The object has a callback method to be called by the invoker. Commands are an object-oriented replacement for procedural language callback functions.

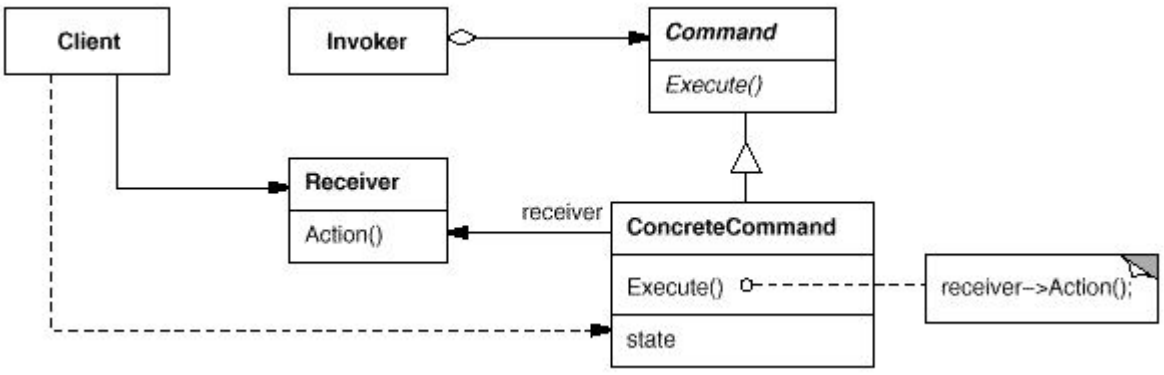
b. Specify, queue, and execute requests at different times.

c. Support transactions.

d. Support undo.

e. Support logging and system recovery.

Structure



Participants

a. **Command**

- declares an interface for executing an operation.

b. **ConcreteCommand**

- defines a binding between a Receiver object and an action.

- implements Execute by invoking the corresponding operation(s) on Receiver.

c. **Invoker**

- asks the command to carry out the request.

d. **Receiver**

- knows how to perform the operations associated with carrying out a request. Any class may serve as a Receiver.

e. **Client**

- creates a ConcreteCommand object and sets its receiver.

Consequences

a. Command decouples the object that invokes the operation from the one that knows how to perform it.

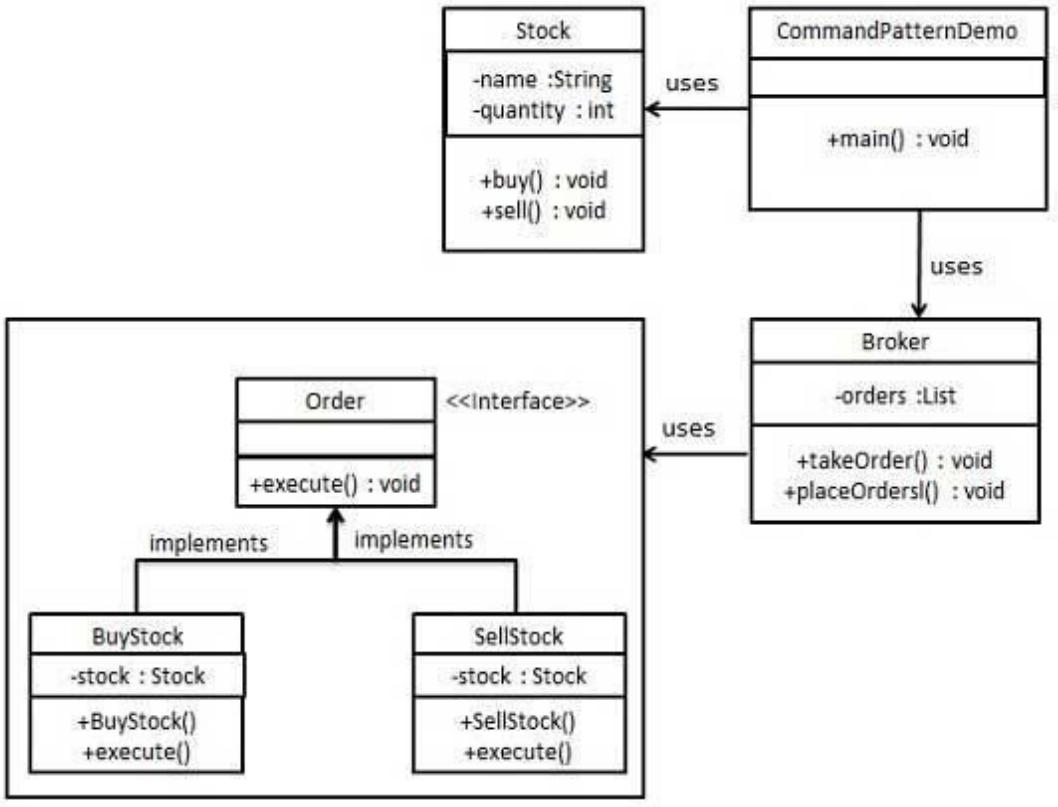
b. Commands are regular objects. They can be manipulated and extended like any other object.

c. It's easy to add new Commands, because you don't have to change existing classes.

Implementation

We have created an interface *Order* which is acting as a command. We have created a *Stock* class which acts as a request. We have concrete command classes *BuyStock* and *SellStock* implementing *Order* interface which will do actual command processing. A class *Broker* is created which acts as an invoker object. It can take and place orders.

Broker object uses command pattern to identify which object will execute which command based on the type of command. *CommandPatternDemo*, our demo class, will use *Broker* class to demonstrate command pattern.



Step 1

Create a command interface.

Order.java

```
public interface Order {
   void execute();
}
```

Step 2

Create a request class.

Stock.java

```
public class Stock {
   private String name = "ABC";
   private int quantity = 10;
   public void buy(){
      System.out.println("Stock [ Name: "+name+",
         Quantity: " + quantity +" ] bought");
   }
   public void sell(){
      System.out.println("Stock [ Name: "+name+",
         Quantity: " + quantity +" ] sold");
   }
}
```

Step 3

Create concrete classes implementing the *Order* interface.

BuyStock.java

```
public class BuyStock implements Order {
   private Stock abcStock;
   public BuyStock(Stock abcStock){
      this.abcStock = abcStock;
   }
   public void execute() {
      abcStock.buy();
   }
}
```

SellStock.java

```
public class SellStock implements Order {
  private Stock abcStock;
  public SellStock(Stock abcStock){
    this.abcStock = abcStock;
  }
  public void execute() {
    abcStock.sell();
  }
}
```

Step 4

Create command invoker class.

Broker.java

```
  public class Broker {
  private List<Order> orderList = new ArrayList<Order>();
  public void takeOrder(Order order){
    orderList.add(order);
  }
  public void placeOrders(){
    for (Order order : orderList) {
      order.execute();
    }
    orderList.clear();
  }
}
```

Step 5

Use the Broker class to take and execute commands.

CommandPatternDemo.java

```
public class CommandPatternDemo {
  public static void main(String[] args) {
    Stock abcStock = new Stock();
    BuyStock buyStockOrder = new BuyStock(abcStock);
    SellStock sellStockOrder = new SellStock(abcStock);
    Broker broker = new Broker();
    broker.takeOrder(buyStockOrder);
    broker.takeOrder(sellStockOrder);
    broker.placeOrders();
  }
}
```

Step 6

Verify the output.

```
Stock [ Name: ABC, Quantity: 10 ] bought
Stock [ Name: ABC, Quantity: 10 ] sold
```

Façade

Intent

Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

Motivation

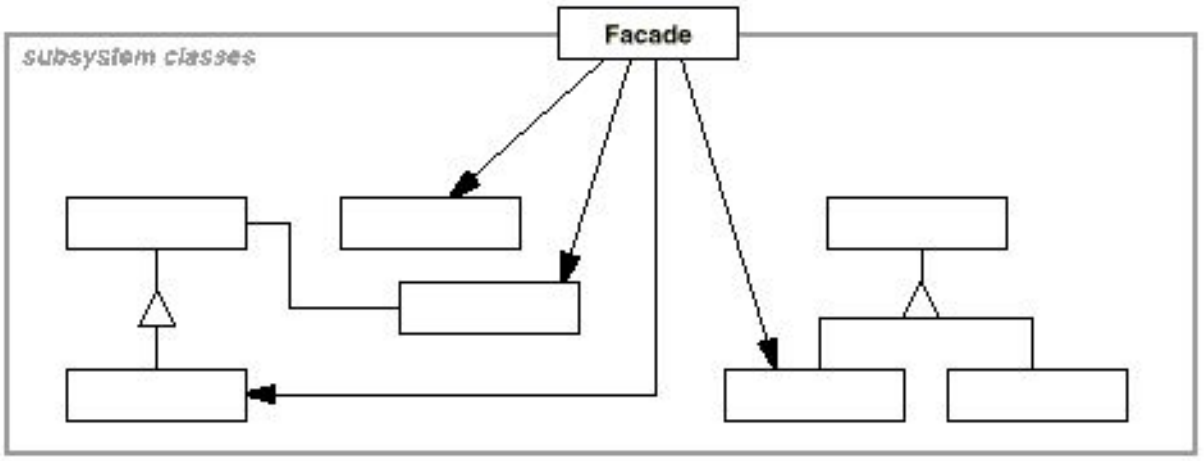
a. Structuring a system into subsystems helps reduce complexity.

b. Subsystems are groups of classes, and/or other subsystems.

c. The interface exposed by the classes in a subsystem or set of subsystems can become quite complex.

d. One way to reduce this complexity is to introduce a facade object that provides a single, simplified interface to the more general facilities of a subsystem.

Structure



Participants

a. Façade

- knows which subsystem classes are responsible for a request.
- delegates client requests to appropriate subsystem objects.

b. Subsystem classes

- implement subsystem functionality.
- handle work assigned by the Facade object.
- have no knowledge of the facade; that is, they keep no references to it.

Consequences

a. It hides the implementation of the subsystem from clients, making the subsystem easier to use.

b. It promotes weak coupling between the subsystem and its clients. This allows you to change the classes that comprise the subsystem without affecting the clients.

c. It reduces compilation dependencies in large software systems.

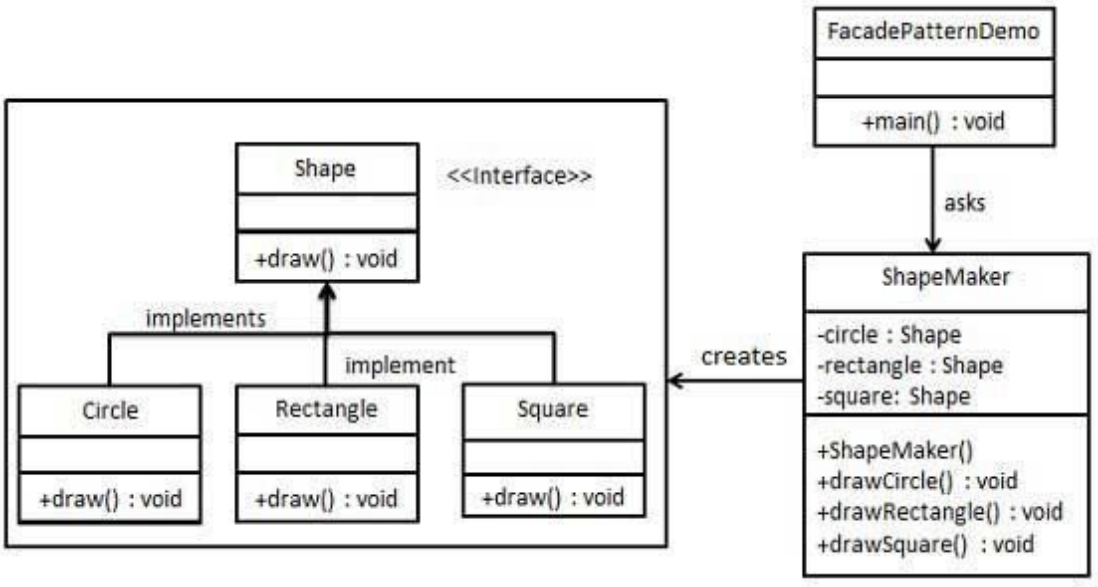
d. However, it does not prevent sophisticated clients from accessing the underlying classes.

e. And Facade does not add any functionality, it just simplifies interfaces.

Implementation

We are going to create a *Shape* interface and concrete classes implementing the *Shape* interface. A facade class *ShapeMaker* is defined as a next step.

ShapeMaker class uses the concrete classes to delegate user calls to these classes. *FacadePatternDemo*, our demo class, will use *ShapeMaker* class to show the results.



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("Rectangle::draw()");
   }
}
```

Square.java

```
public class Square implements Shape {
   @Override
   public void draw() {
      System.out.println("Square::draw()");
   }
}
```

Circle.java

```
public class Circle implements Shape {
   @Override
   public void draw() {
      System.out.println("Circle::draw()");
   }
}
```

Step 3

Create a facade class.

ShapeMaker.java

```
public class ShapeMaker {
   private Shape circle;
   private Shape rectangle;
   private Shape square;
   public ShapeMaker() {
      circle = new Circle();
      rectangle = new Rectangle();
      square = new Square();
   }
   public void drawCircle(){
      circle.draw();
   }
   public void drawRectangle(){
      rectangle.draw();
   }
   public void drawSquare(){
      square.draw();
   }
}
```

Step 4

Use the facade to draw various types of shapes.

FacadePatternDemo.java

```
public class FacadePatternDemo {
   public static void main(String[] args) {
      ShapeMaker shapeMaker = new ShapeMaker();
```

```
    shapeMaker.drawCircle();
    shapeMaker.drawRectangle();
    shapeMaker.drawSquare();
  }
}
```

Step 5

Verify the output.

```
Circle::draw()
Rectangle::draw()
Square::draw()
```

Mediator

Intent

Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.

Motivation

a. In a system where multiple objects communicate to each other by directly sending messages to others, communication between these objects may become complex.

b. This makes the program harder to maintain since any change may affect code in other objects or classes.

c. Use the Mediator Pattern to centralize complex communications and control between related objects.

d. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently

Applicability

Use the Mediator pattern when

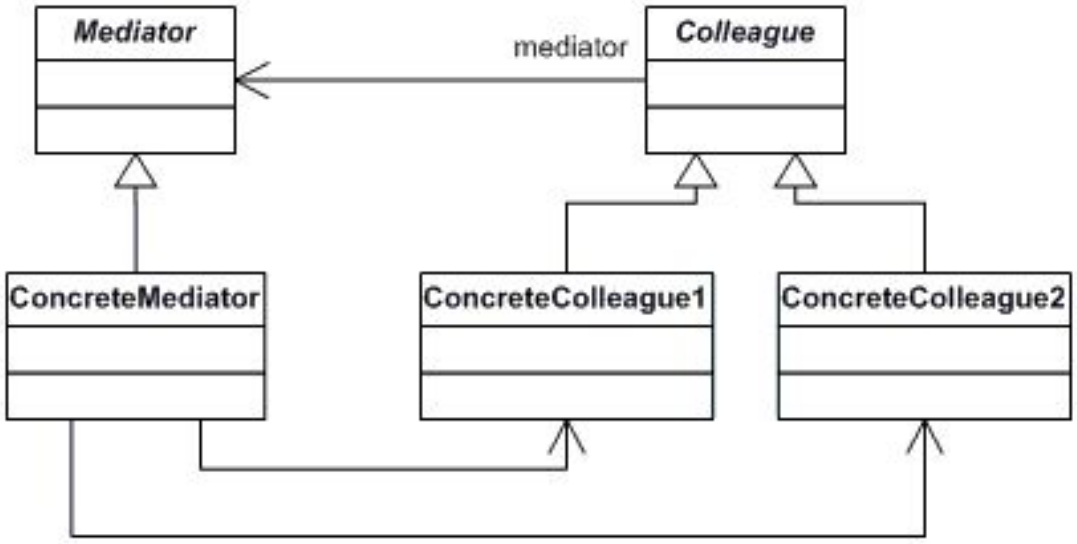
a. A set of objects communicate in well-defined but complex ways. The resulting interdependencies are unstructured and difficult to understand.

b. Reusing an object is difficult because it refers to and communicates with many other

objects.

c. A behavior that's distributed between several classes should be customizable without a lot of subclassing.

Structure



Participants

a. Mediator

- defines an interface for communicating with Colleague objects.

b. ConcreteMediator

- implements cooperative behavior by coordinating Colleague objects.

- knows and maintains its colleagues.

c. Colleague classes

- each Colleague class knows its Mediator object.

- each colleague communicates with its mediator whenever it would have otherwise communicated with another colleague.

Consequences

The Mediator pattern has the following benefits and drawbacks:

a. A mediator localizes behavior that otherwise would be distributed among several objects. Changing this behavior requires subclassing Mediator only; Colleague classes

can be reused as is.

b. It decouples colleagues. A mediator promotes loose coupling between colleagues.

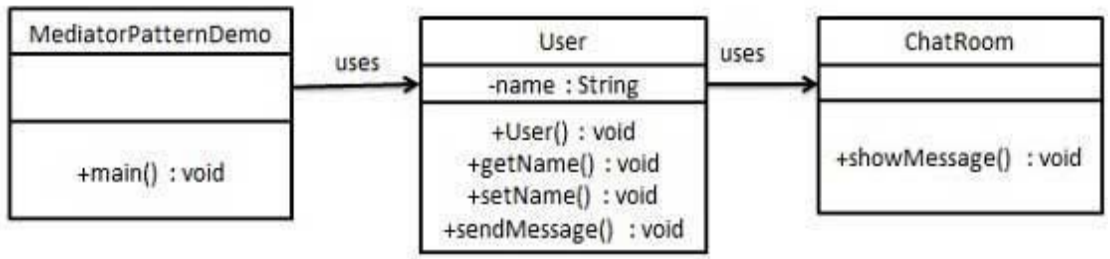
c. It simplifies object protocols. A mediator replaces many-to-many interactions with one-to-many interactions between the mediator and its colleagues.

d. It centralizes control which can make the mediator itself a monolith that's hard to maintain.

Implementation

We are demonstrating mediator pattern by example of a chat room where multiple users can send message to chat room and it is the responsibility of chat room to show the messages to all users. We have created two classes *ChatRoom* and *User*. *User* objects will use *ChatRoom* method to share their messages.

MediatorPatternDemo, our demo class, will use *User* objects to show communication between them.



Step 1

Create mediator class.

ChatRoom.java

```
public class ChatRoom {
  public static void showMessage(User user, String message){
    System.out.println(new Date().toString() + " [" + user.getName() + "] : " + message);
  }
}
```

Step 2

Create user class

User.java

```
public class User {
  private String name;
  public String getName() {
    return name;
  }
  public void setName(String name) {
    this.name = name;
  }
  public User(String name){
    this.name  = name;
  }
  public void sendMessage(String message){
    ChatRoom.showMessage(this,message);
  }
}
```

Step 3

Use the *User* object to show communications between them.

MediatorPatternDemo.java

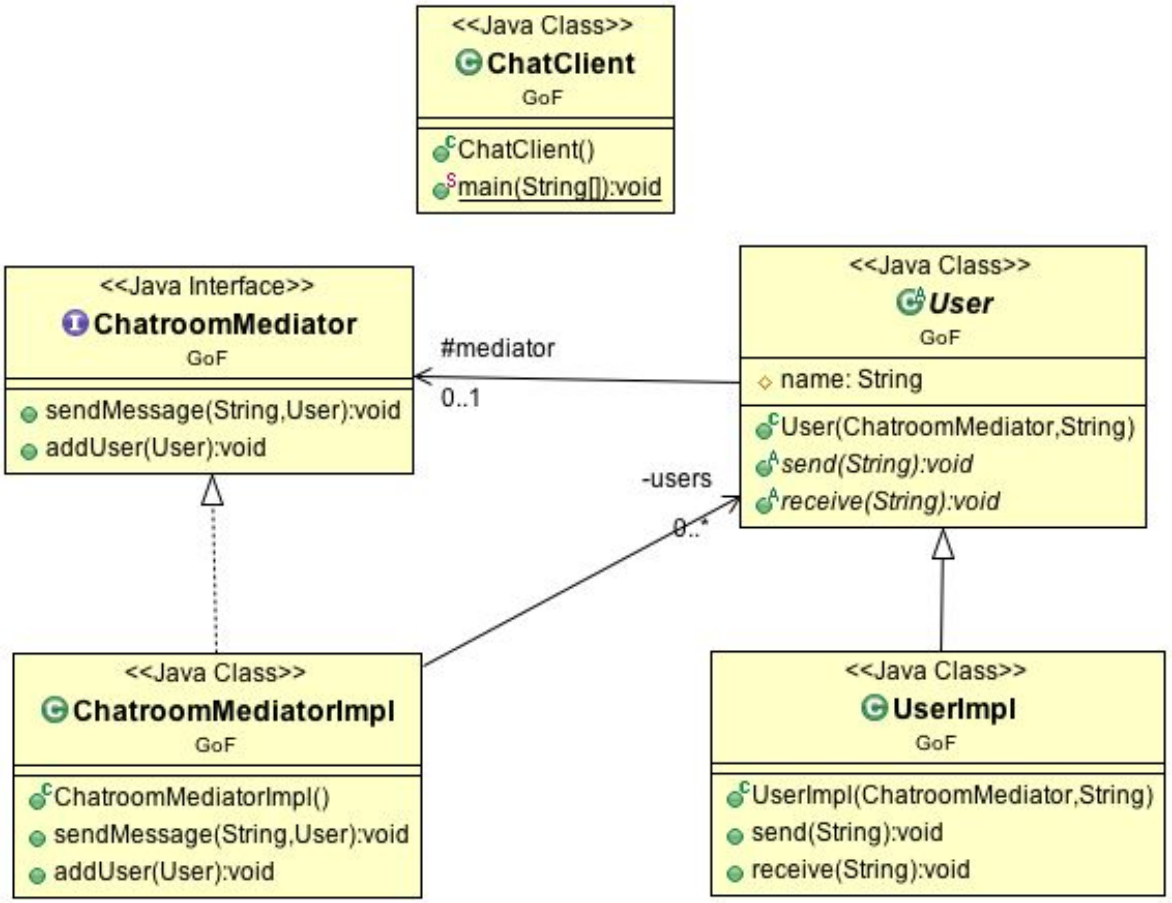
```
public class MediatorPatternDemo {
  public static void main(String[] args) {
    User robert = new User("Robert");
    User john = new User("John");
    robert.sendMessage("Hi! John!");
    john.sendMessage("Hello! Robert!");
  }
}
```

Step 4

Verify the output.

```
Thu Jan 31 16:05:46 IST 2013 [Robert] : Hi! John!
Thu Jan 31 16:05:46 IST 2013 [John] : Hello! Robert!
```

Another Mediator example



ChatroomMediator.java

```
public interface ChatroomMediator {

	public void sendMessage(String msg, User user);

	public void addUser(User user);
}
```

User.java

```
public abstract class User {

	protected ChatroomMediator mediator;
	protected String name;

	public User(ChatroomMediator med, String name) {
```

```
        this.mediator = med;
        this.name = name;
    }

    public abstract void send(String msg);
    public abstract void receive(String msg);
}
```

ChatroomMediatorImpl.java

```
public class ChatroomMediatorImpl implements ChatroomMediator {

    private List<User> users;

    public ChatroomMediatorImpl() {
        this.users = new ArrayList<User>();
    }

    @Override
    public void sendMessage(String msg, User user) {
        for (User u : this.users) {
            if (u!=user) {
                u.receive(msg);
            }
        }
    }

    @Override
    public void addUser(User user) {
        this.users.add(user);
    }
}
```

UserImpl.java

```
public class UserImpl extends User {

    public UserImpl(ChatroomMediator med, String name) {
        super(med, name);
    }

    @Override
```

```
	public void send(String msg) {

		System.out.println(this.name + ": Sending message = " + msg);
		mediator.sendMessage(msg, this);
	}

	@Override
	public void receive(String msg) {

		System.out.println(this.name + ": Received Message = " + msg);
	}

}
```

ChatClient.java

```
public class ChatClient {

	public static void main(String[] args) {

		ChatroomMediator mediator = new ChatroomMediatorImpl();
		User user1 = new UserImpl(mediator, "Pan");
		User user2 = new UserImpl(mediator, "Lis");
		User user3 = new UserImpl(mediator, "Sar");
		mediator.addUser(user1);
		mediator.addUser(user2);
		mediator.addUser(user3);

		user1.send("Hi all");
	}

}
```

Observer

Intent

Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

Motivation

a. Sometimes we want to model a 1-to-many publisher-subscriber relationship in our applications.

b. The multiple subscribers are dependent on the state of the publisher; therefore they need to be notified of any change of state with the publisher.

c. In order to reuse the publisher and subscriber classes, their relationship has to be decoupled.

Applicability

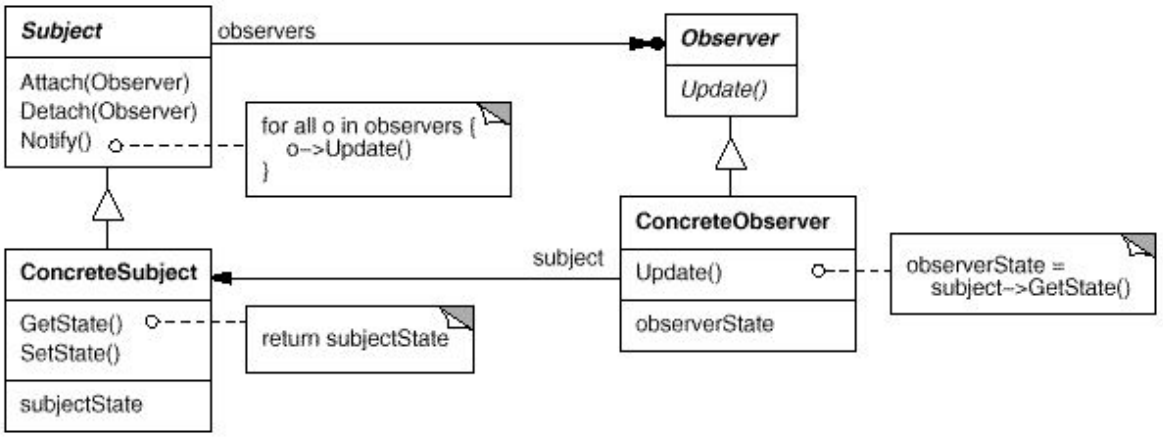
Use the Observer pattern when

a. When an abstraction has two aspects, one dependent on the other. Encapsulating these aspects in separate objects lets you vary and reuse them independently.

b. When a change to one object requires changing others, and you don't know how many objects need to be changed.

c. When an object should be able to notify other objects without making assumptions about who these objects are. In other words, you don't want these objects tightly coupled.

Structure



Participants

a. **Subject**

- knows its observers. Any number of Observer objects may observe a subject.

- provides an interface for attaching and detaching Observer objects.

b. **Observer**

- defines an updating interface for objects that should be notified of changes in a subject.

c. **ConcreteSubject**

- stores state of interest to ConcreteObserver objects.

- sends a notification to its observers when its state changes.

d. **ConcreteObserver**

- maintains a reference to a ConcreteSubject object.

- stores state that should stay consistent with the subject's.

- implements the Observer updating interface to keep its state consistent with the subject's.

Consequences

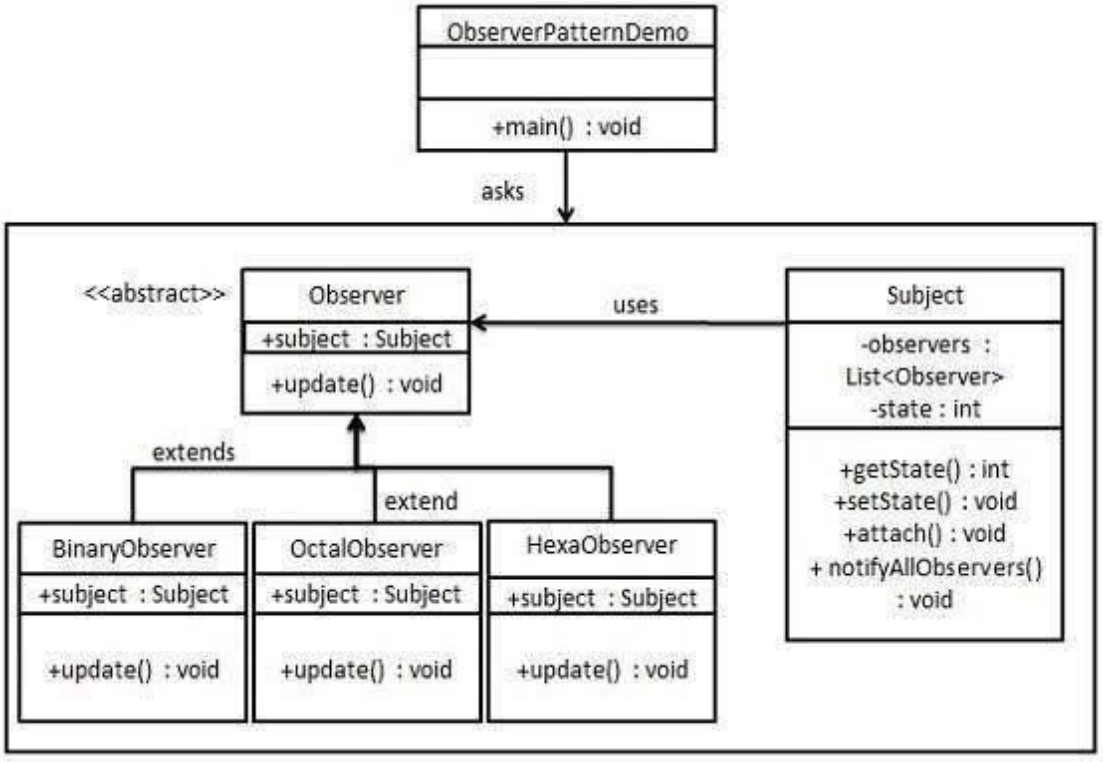
- it supports layering of software applications.

- it supports broadcast communication without having to know all observers. The subject only knows that it has to update a list of observers whenever there is a change of state.

Implementation

Observer pattern uses three actor classes. Subject, Observer and Client. Subject is an object having methods to attach and detach observers to a client object. We have created an abstract class *Observer* and a concrete class *Subject* that is extending class *Observer*.

ObserverPatternDemo, our demo class, will use *Subject* and concrete class object to show observer pattern in action.



Step 1

Create Subject class.

Subject.java

```
public class Subject {
  private List<Observer> observers = new ArrayList<Observer>();
  private int state;
  public int getState() {
    return state;
  }
  public void setState(int state) {
    this.state = state;
    notifyAllObservers();
  }
  public void attach(Observer observer){
    observers.add(observer);
  }
```

```
  public void notifyAllObservers(){
    for (Observer observer : observers) {
      observer.update();
    }
  }
}
```

Step 2

Create Observer class.

Observer.java

```
public abstract class Observer {
  protected Subject subject;
  public abstract void update();
}
```

Step 3

Create concrete observer classes

BinaryObserver.java

```
public class BinaryObserver extends Observer{
  public BinaryObserver(Subject subject){
    this.subject = subject;
    this.subject.attach(this);
  }
  @Override
  public void update() {
    System.out.println( "Binary String: " + Integer.toBinaryString( subject.getState() ) );
  }
}
```

OctalObserver.java

```
public class OctalObserver extends Observer{
  public OctalObserver(Subject subject){
    this.subject = subject;
    this.subject.attach(this);
  }
  @Override
  public void update() {
   System.out.println( "Octal String: " + Integer.toOctalString( subject.getState() ) );
  }
}
```

HexaObserver.java

```
public class HexaObserver extends Observer{
  public HexaObserver(Subject subject){
    this.subject = subject;
    this.subject.attach(this);
  }
  @Override
  public void update() {
    System.out.println( "Hex String: " + Integer.toHexString( subject.getState()
).toUpperCase() );
  }
}
```

Step 4

Use *Subject* and concrete observer objects.

ObserverPatternDemo.java

```
public class ObserverPatternDemo {
  public static void main(String[] args) {
    Subject subject = new Subject();
```

```
    new HexaObserver(subject);
    new OctalObserver(subject);
    new BinaryObserver(subject);
    System.out.println("First state change: 15");
    subject.setState(15);
    System.out.println("Second state change: 10");
    subject.setState(10);
  }
}
```

Step 5

Verify the output.

```
First state change: 15
Hex String: F
Octal String: 17
Binary String: 1111
Second state change: 10
Hex String: A
Octal String: 12
Binary String: 1010
```

Visitor

Intent

Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

Motivation

a. Polymorphism works well for objects of the same hierarchy when you iterate through a collection of them and call a common method, also known as „Single Dispatch".

b. But sometimes „Single dispatch" does not support certain operations well (for example, to accumulate state information in the object structure, or apply business rules depending on the amount of the accumulated information).

c. Visitor also allows us to externalize operations of an object structure into a separate class yet still supports polymorphism by "Double Dispatch".

Applicability

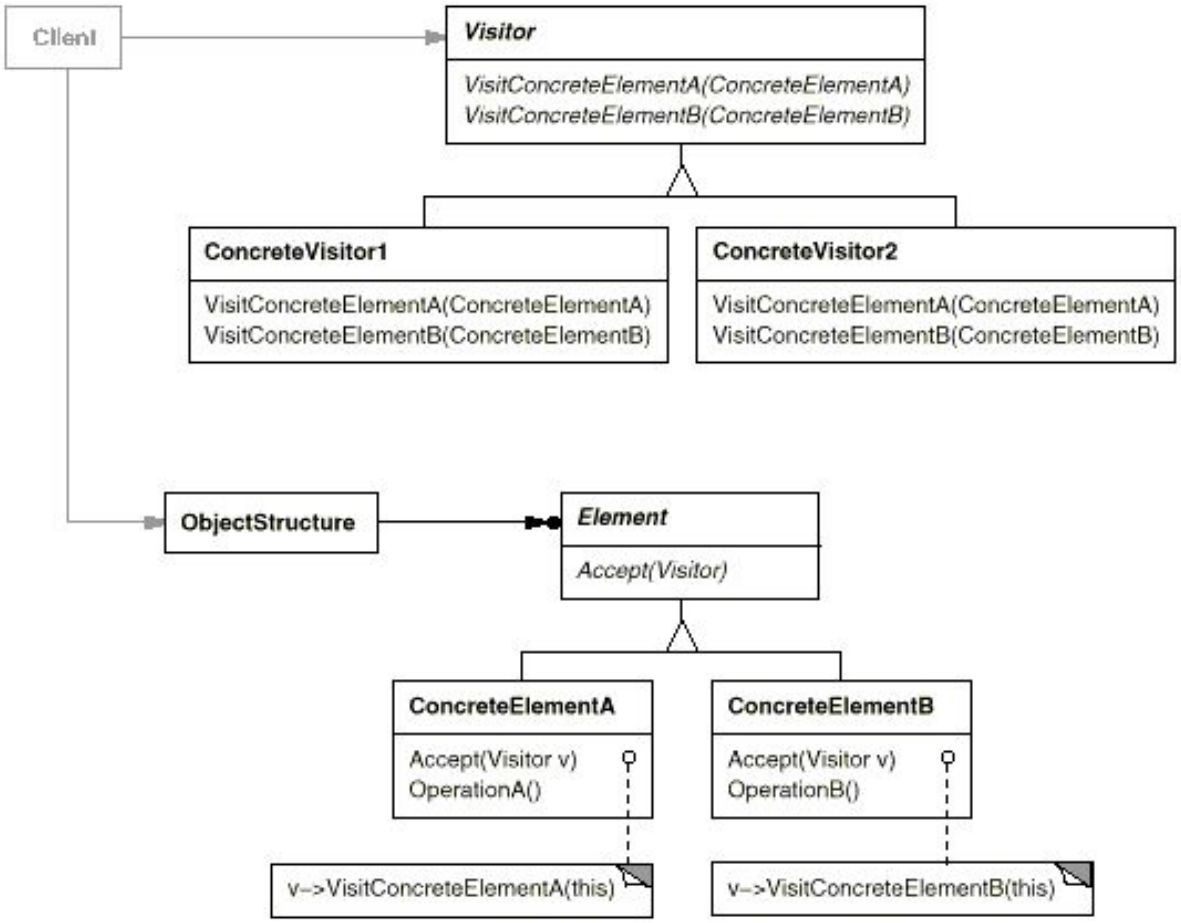
Use the Visitor pattern when

a. An object structure contains objects from different class hierarchies, and you want to perform operations on these objects that depend on their concrete classes.

b. Many distinct and unrelated operations need to be performed on objects in an object structure, and you want to avoid "polluting" the classes with these operations. Visitor lets you keep related operations together by defining them in one class. For example, the shoppingCart"s calculateTotal() and calculateShippingFee() methods are considered "unrelated" operations.

c. The class defining the object structure rarely changes, but you often want to define new operations over the structure. Changing the object structure class requires redefining the interface to all visitors, which is potentially costly. If the object structure class changes often, then it's probably better to define the operations in those classes.

Structure



Participants

a. **Visitor**

- declares a Visit operation for each class of ConcreteElement in the object structure.

b. **ConcreteVisitor**

- implements each operation declared by Visitor. Each operation implements a fragment of the algorithm defined for the corresponding class of object in the structure. ConcreteVisitor provides the context for the algorithm and stores its local state. This state often accumulates results during the traversal of the structure.

c. **Element**

- defines an „accept" operation that takes a visitor as an argument.

d. **ConcreteElement**

- implements the „accept‟ operation that takes a visitor as an argument.

e. **ObjectStructure**

- can enumerate its elements, as a composite structure or a collection such as a list or a set.

Consequences

Some of the benefits and liabilities of the Visitor pattern are as follows:

a. Visitor makes adding new operations easy.

b. A visitor gathers related operations and separates unrelated ones. (for example, separate getShippingFee from OrderItem)

c. Adding a new Concrete Element class is hard.

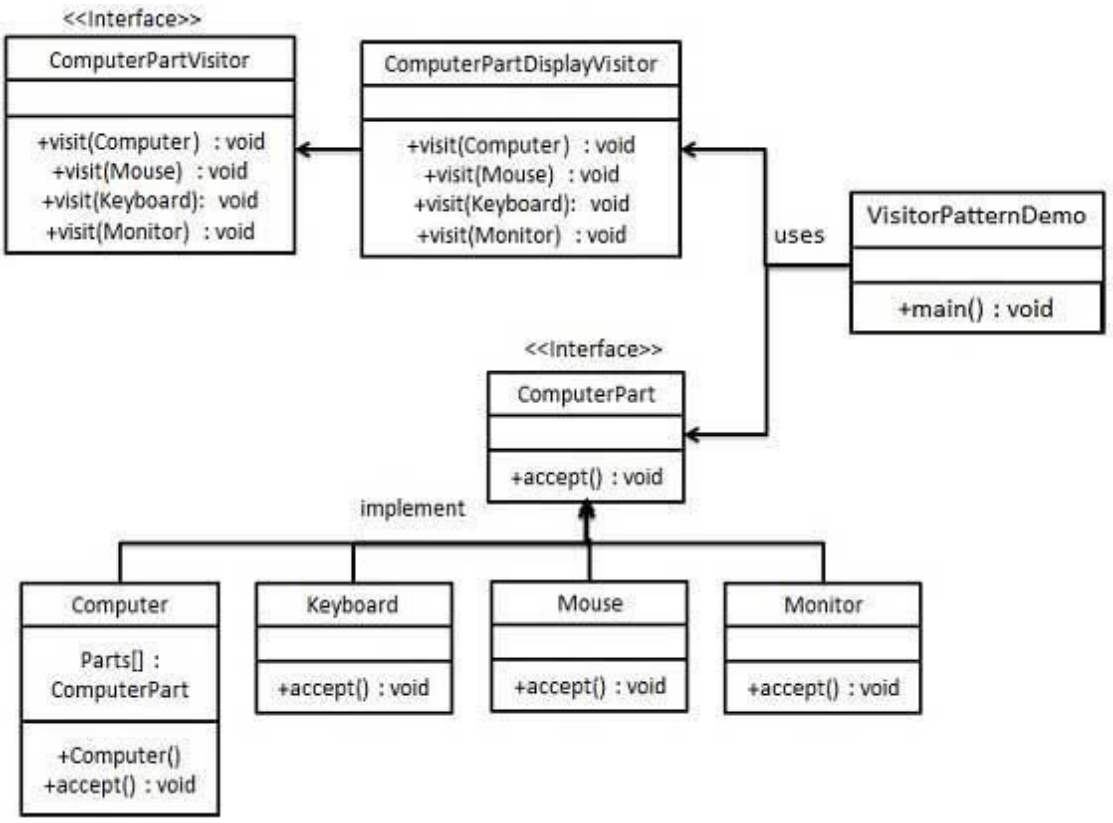
d. Can accumulate state across different objects in an object structure.

e. It may compromise encapsulation.

Implementation

We are going to create a *ComputerPart* interface defining accept opearation.
Keyboard, *Mouse*, *Monitor* and *Computer* are concrete classes implementing *ComputerPart* interface. We will define another interface *ComputerPartVisitor* which will define a visitor class operations. *Computer* uses concrete visitor to do corresponding action.

VisitorPatternDemo, our demo class, will use *Computer* and *ComputerPartVisitor* classes to demonstrate use of visitor pattern.



Step 1

Define an interface to represent element.

ComputerPart.java

```
public interface ComputerPart {
  public void accept(ComputerPartVisitor computerPartVisitor);
}
```

Step 2

Create concrete classes extending the above class.

Keyboard.java

```
public class Keyboard implements ComputerPart {
  @Override
  public void accept(ComputerPartVisitor computerPartVisitor) {
    computerPartVisitor.visit(this);
  }
```

```
}
```

Monitor.java

```
public class Monitor implements ComputerPart {
 @Override
  public void accept(ComputerPartVisitor computerPartVisitor) {
    computerPartVisitor.visit(this);
  }
}
```

Mouse.java

```
public class Mouse implements ComputerPart {
  @Override
  public void accept(ComputerPartVisitor computerPartVisitor) {
    computerPartVisitor.visit(this);
  }
}
```

Computer.java

```
public class Computer implements ComputerPart {
  ComputerPart[] parts;
  Computer(){
    parts = new ComputerPart[] {new Mouse(), new Keyboard(), new Monitor()};
  }
  @Override
  public void accept(ComputerPartVisitor computerPartVisitor) {
    for (int i = 0; i < parts.length; i++) {
      parts[i].accept(computerPartVisitor);
    }
    computerPartVisitor.visit(this);
```

```
  }
}
```

Step 3

Define an interface to represent visitor.

ComputerPartVisitor.java

```
public interface ComputerPartVisitor {
        public void visit(Computer computer);
        public void visit(Mouse mouse);
        public void visit(Keyboard keyboard);
        public void visit(Monitor monitor);
}
```

Step 4

Create concrete visitor implementing the above class.

ComputerPartDisplayVisitor.java

```
public class ComputerPartDisplayVisitor implements ComputerPartVisitor {
  @Override
  public void visit(Computer computer) {
    System.out.println("Displaying Computer.");
  }
  @Override
  public void visit(Mouse mouse) {
    System.out.println("Displaying Mouse.");
  }
  @Override
  public void visit(Keyboard keyboard) {
    System.out.println("Displaying Keyboard.");
  }
```

```
  @Override
  public void visit(Monitor monitor) {
    System.out.println("Displaying Monitor.");
  }
}
```

Step 5

Use the *ComputerPartDisplayVisitor* to display parts of *Computer*.

VisitorPatternDemo.java

```
public class VisitorPatternDemo {
  public static void main(String[] args) {
    ComputerPart computer = new Computer();
    computer.accept(new ComputerPartDisplayVisitor());
  }
}
```

Step 6

Verify the output.

```
Displaying Mouse.
Displaying Keyboard.
Displaying Monitor.
Displaying Computer.
```

Proxy

Intent

Provide a **surrogate** or **placeholder** for another object to control access to it.

Motivation

a. To defer the full cost of its creation and initialization until we actually need to use it.

b. There are situations in which a client does not or cannot reference an object directly, but wants to still interact with the object.

c. A proxy object can act as the intermediary between the client and the target object.

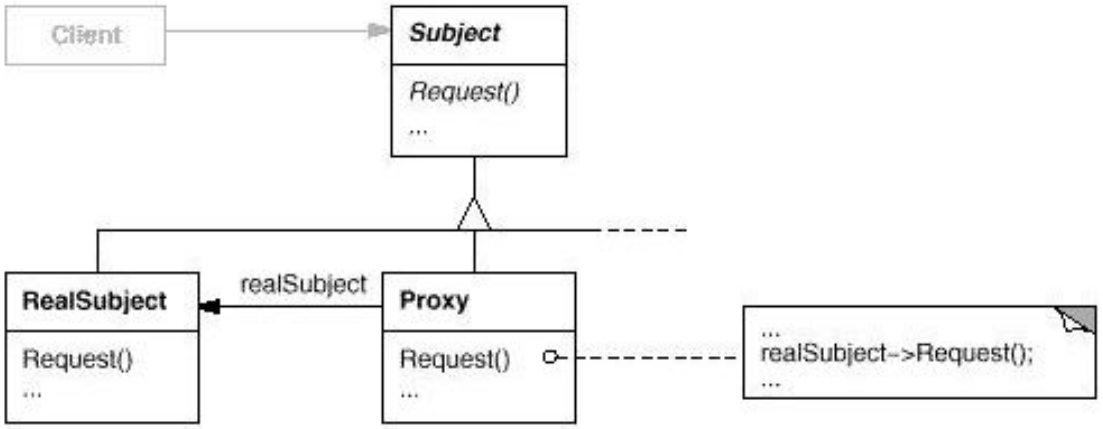
Applicability

a. Remote Proxy - Provides a reference to an object located in a different address space on the same or different machine.

b. Protection (Access) Proxy - Provides different clients with different levels of access to a target object.

c. Cache Proxy - Provides temporary storage of the results of expensive target operations so that multiple clients can share the results.

Structure



Participants

a. **Proxy**

- maintains a reference that allows the proxy to access the real subject.

- provides an interface identical to Subject's so that a proxy can be substituted for the real subject.

- controls access to the real subject and may be responsible for creating and deleting it

b. **Subject**
- defines the common interface for RealSubject and Proxy so that a Proxy can be used anywhere a RealSubject is expected.

c. **RealSubject**
- defines the real object that the proxy represents.

Factory

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

a. Sometimes an object may only know that it needs an object of a certain type but does not know exactly which one from the set of subclasses of the parent class is to be selected.

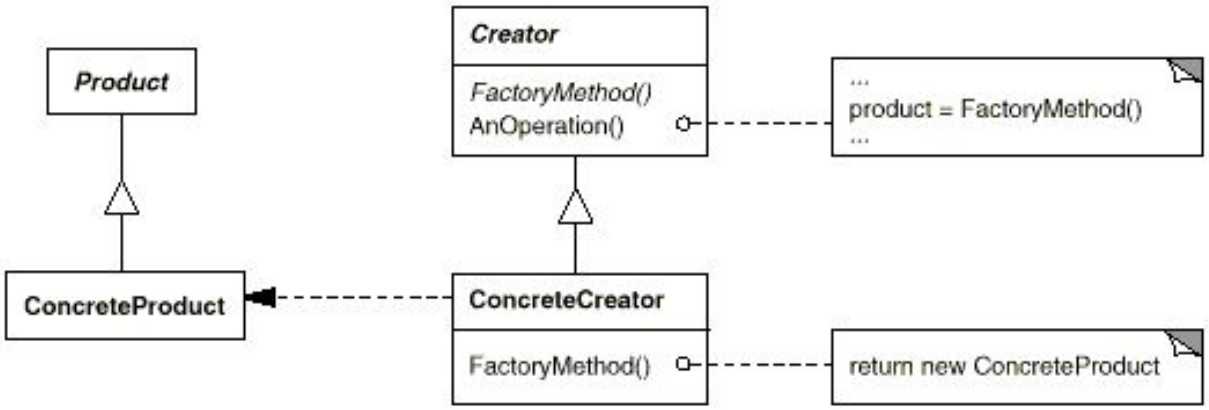
b. The choice of an appropriate class may depend on factors such as:

– The state of the running application.
– Application configuration settings.
– Expansion of requirements or enhancements.
c. Factory Method recommends encapsulating the functionality required, to select and instantiate an appropriate class, inside a designated method referred to as a factory method.

d. A factory method can be defined as a method in a class that:

– Selects an appropriate class from a class hierarchy based on the application context and other influencing factors.
– Instantiates the selected class and returns it as an instance of the parent class type.

Structure



Participants

a. **Product**

- defines the interface of objects the factory method creates.

b. **ConcreteProduct**

- implements the Product interface.

c. **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.

d. **ConcreteCreator**

- overrides the factory method to return an instance of a ConcreteProduct.

Applicability

Use the Factory Method pattern when

a. A class can't anticipate the class of objects it must create.

b. A class wants its subclasses to specify the objects it creates.

c. Classes delegate responsibility to one of several helper subclasses, and you want to localize the knowledge of which helper subclass is the delegate.

Consequences

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

b. Any change in a concrete product class does not have any impact on the client code.

Abstract Factory

Intent

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

Motivation

a. Your application needs to create different series of products that belong to different inheritance hierarchies (or suites/ families of related, dependent classes).

b. You have a common interface to create all different families of products.

Applicability

Use the Abstract Factory pattern in any of the following situations:

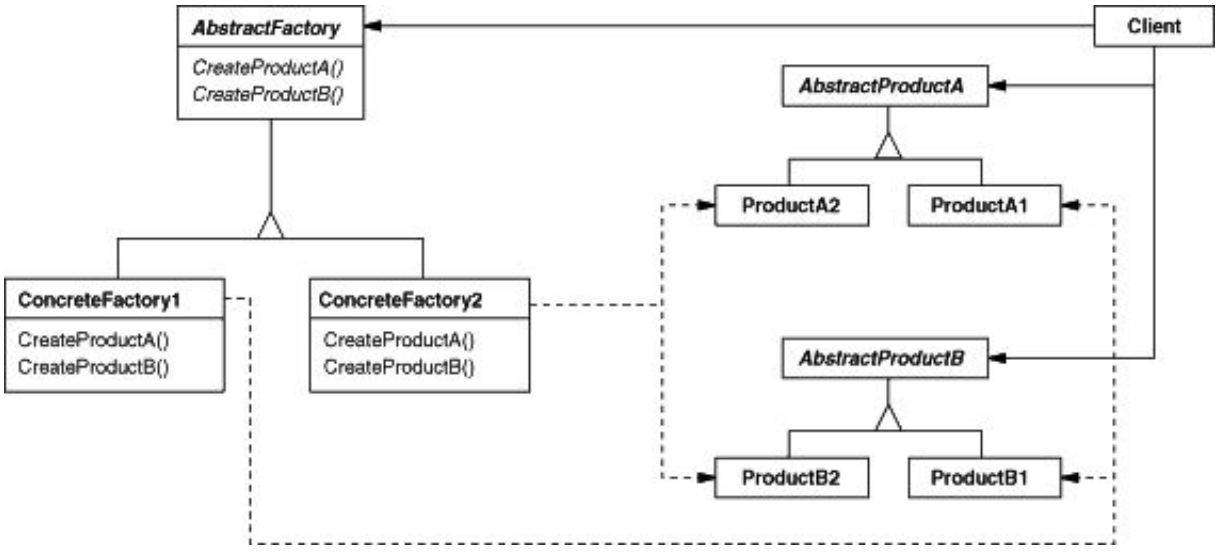
a. A system should be independent of how its products are created, composed, and represented.

b. A class can't anticipate the class of objects it must create.

c. A system must use just one of a set of families of products

d. A family of related product objects is designed to be used together, and you need to enforce this constraint.

Structure



Participants

a. **AbstractFactory**

- declares an interface for operations that create abstract product objects.

b. **ConcreteFactory**

- implements the operations to create concrete product objects.

c. **AbstractProduct**

- declares an interface for a type of product object.

d. **ConcreteProduct**

- defines a product object to be created by the corresponding concrete factory.
- implements the AbstractProduct interface.

e. **Client**

- uses only interfaces declared by AbstractFactory and AbstractProduct classes.

Consequences

c. It isolates concrete classes. Product class names are isolated in the implementation of the concrete factory; they do not appear in client code.

d. It makes exchanging product families easy.

e. It promotes consistency among products.

f. Supporting new kinds of products is difficult.

Composite

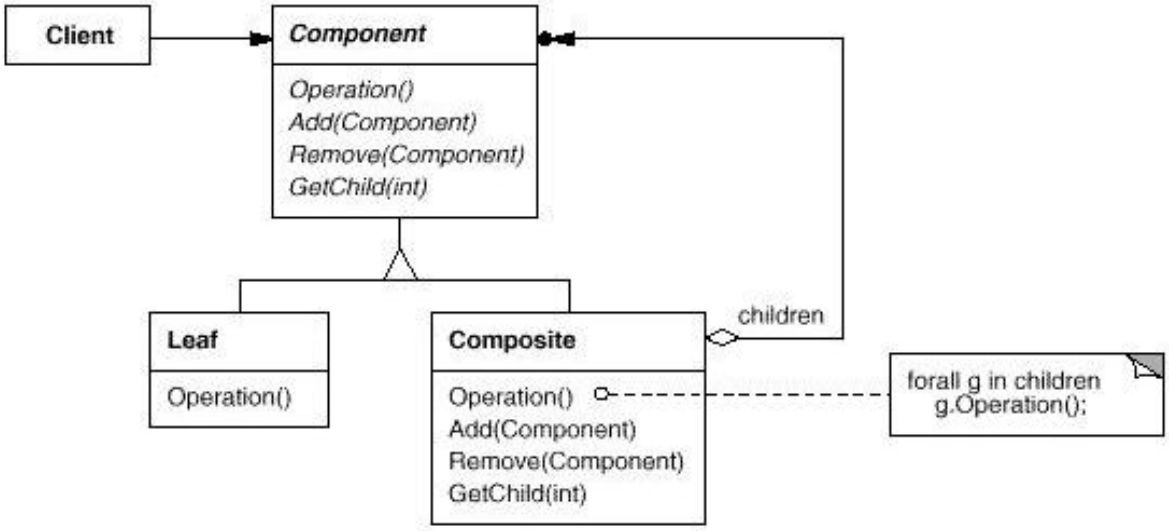
Intent

Compose objects into tree structures to represent part-whole or parent-child hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

Motivation

Many times you need to model a system that deals with relationships between objects that are of the same type but on different levels (for example in whole-part, parent-child, or supervisor-employee, relationships.) and in your system the number of levels is unknown until runtime.

Structure



Participants

a. **Component**

- declares the interface for objects in the composition.
- implements default behavior for the interface common to all classes, as appropriate.
- declares an interface for accessing and managing its child components.
- (optional) defines an interface for accessing a component's parent in the recursive structure, and implements it if that's appropriate.

b. **Leaf**

- represents leaf objects in the composition. A leaf has no children.
- defines behavior for primitive objects in the composition.

c. **Composite**

- defines behavior for components having children.
- stores child components.
- implements child-related operations in the Component interface.

d. **Client**

- manipulates objects in the composition through the Component interface.

Applicability

Use the Composite pattern when

a. You want to represent part-whole or parent-child hierarchies of objects

b. You want clients to be able to ignore the difference between compositions of objects and individual objects. Clients will treat all objects in the composite structure uniformly.

State

Intent

Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.

Motivation

a. Objects have states and behavior (behavior as functions to process different data/states). Many times, the behavior and states are independent while other times behavior varies for different states the object is in.

b. We achieve the above by factoring the states and their corresponding behavior into separate classes.

Applicability

Use the State pattern in the following cases:

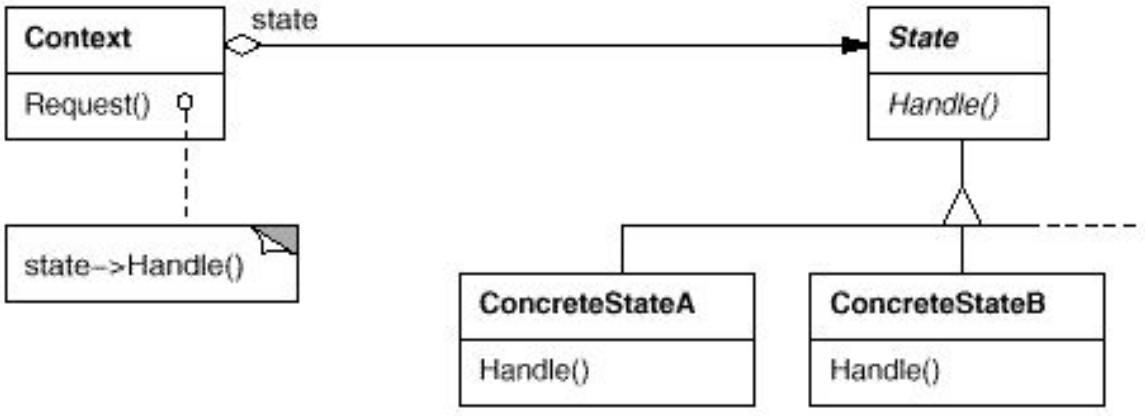
a. An object's behavior depends on its state, and it must change its behavior at run-time depending on that state.

b. Operations have large, multipart conditional statements that depend on the object's state. This state is usually represented by one or more enumerated constants. Often, several operations will contain this same conditional structure. The State pattern puts each branch of the conditional in a separate class. This lets you treat the object's state as an object in its own right that can vary independently from other objects.

c. The number of all possible states is limited.

d. When we need to design state machines or vending machines.

Structure



Participants

a. **Context**

- defines the interface of interest to clients.

- maintains an instance of a ConcreteState subclass that defines the current state.

b. **State**

- defines an interface for encapsulating the behavior associated with a particular state of the Context.

c. **ConcreteState subclasses**

- each subclass implements a behavior associated with a state of the Context.

Consequences

a. It localizes state-specific behavior in each concrete state.

b. State information can be represented by an instance variable or by the type of the concrete state.

Strategy Pattern

Intent

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

Motivation

a. The client and a strategy it uses to perform a function should not be hard-wired together

- to keep the client lean;
- or make the change of strategy easier;
- or add new strategies without affecting client code.

b. Your application may need different strategies for different situations during runtime.

Applicability

Use the Strategy pattern when

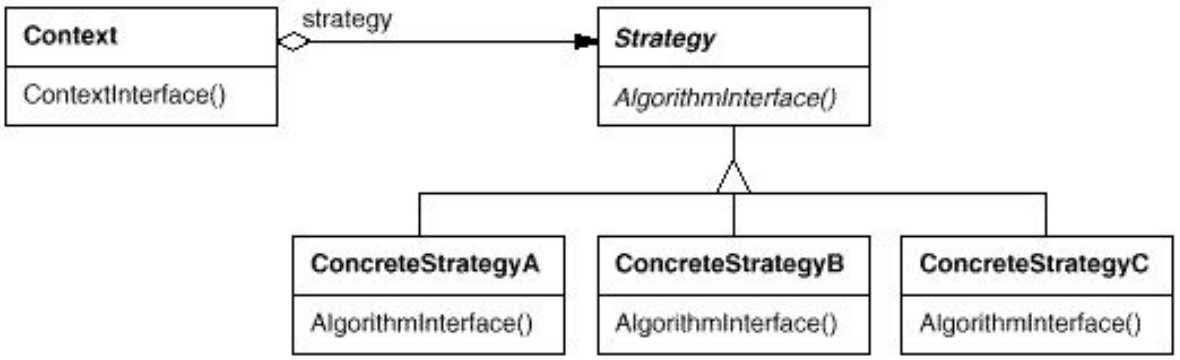
a. Many related classes differ only in their behavior. Strategies provide a way to configure a class with one of many behaviors.

b. You need different variants of an algorithm. For example, you might define algorithms reflecting different space/time trade-offs. Strategies can be used when these variants are implemented as a class hierarchy of algorithms.

c. An algorithm uses data that clients shouldn't know about. Use the Strategy pattern to avoid exposing complex, algorithm-specific data structures.

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

Structure



Participants

a. Strategy - declares an interface common to all supported algorithms.

b. Concrete Strategy - implements the algorithm using the Strategy interface.

c. Context - is configured with one or more Concrete Strategy objects. In multi-layered enterprise applications, a Context often uses a Hash Map that stores server-side information. (For example, compression strategies for transporting different files at runtime).

Consequences

a. Families of related algorithms are defined by hierarchies of Strategy classes with inheritance.

b. Encapsulating the algorithm in separate Strategy classes lets you vary the algorithm independently of its context, making it easier to switch, understand, and extend.

c. Strategies eliminate conditional statements (from client) by encapsulating the behavior in separate Strategy classes.

d. Strategies can provide different implementations of the same behavior. The client can choose among strategies with different time and space trade-offs.

e. You should use the Strategy pattern only when the variation in behavior is relevant to clients.

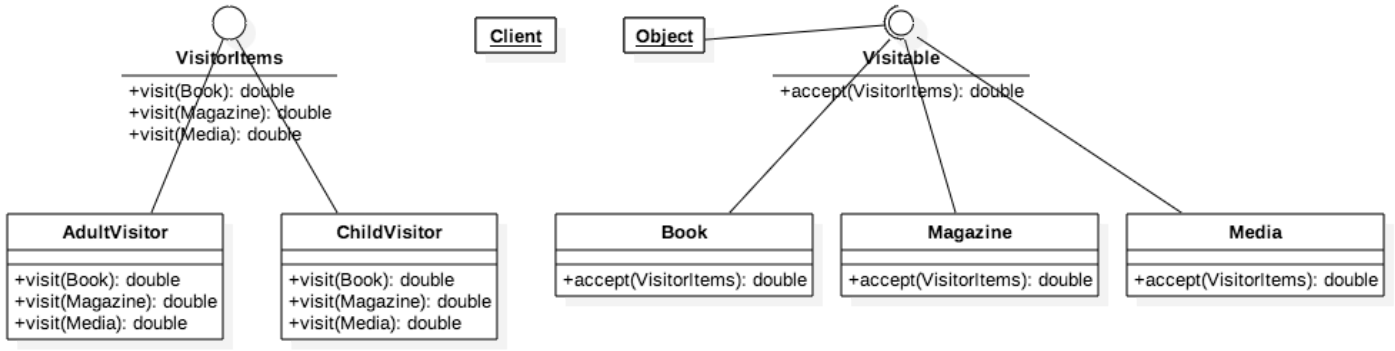
f. Strategies are stateless objects across invocations.

1. (20 points) Final May 2015

You are developing a library system for a Public Library. All items that its members (adults and children) can check out are books, magazines and media items. Loan periods for everything are 4 weeks. A fine is charged for each overdue item. Details are listed below:

Overdue Fines	**Book/Day**	**Magazine/Day**	**Media/Day**	**Maximum Fine**
for Adults	$0.10	$0.15	$0.20	$10.00
for Children	$0.05	$0.05	$0.05	$3.00

You now need to develop the FineCalculator class with the **Visitor Pattern**. You are also responsible for all other necessary classes/interfaces to make your FineCalculator work.
1) Draw the UML diagram for your solution with all necessary participants, responsibilities and relationships clearly specified.
2) Write Java code that shows all necessary attributes/methods with implementation logic for your design. (No need to provide test code).



Visitor.java

```
public interface Visitor {
        public double visit(Book book);
        public double visit(Magazine magazine);
        public double visit(Media media);
}
```

AdultVisitor.java

```
public class AdultVisitor implements Visitor {
```

```
    private double fine;
    private double maxFine = 10;

    @Override
    public double visit(Book book) {
        System.out.println("Book fine for Adult");
        fine = book.getOverdue() * 0.1;

        if (fine >= maxFine) {
            return maxFine;
        }
        return fine;
    }

    @Override
    public double visit(Magazine magazine) {
        System.out.println("Magazine fine for Adult");
        fine = magazine.getOverdue() * 0.15;

        if (fine >= maxFine) {
            return maxFine;
        }
        return fine;
    }

    @Override
    public double visit(Media media) {
        System.out.println("Media fine for Adult");
        fine = media.getOverdue() * 0.2;

        if (fine >= maxFine) {
            return maxFine;
        }
        return fine;
    }
}
```

ChildVisitor.java

```
public class ChildVisitor implements Visitor {

    private double fine;
```

```
	private double maxFine = 3;

	@Override
	public double visit(Book book) {
		System.out.println("Book fine for Child");
		fine = book.getOverdue() * 0.05;

		if (fine >= maxFine) {
			return maxFine;
		}
		return fine;
	}

	@Override
	public double visit(Magazine magazine) {
		System.out.println("Magazine fine for Child");
		fine = magazine.getOverdue() * 0.1;

		if (fine >= maxFine) {
			return maxFine;
		}
		return fine;
	}

	@Override
	public double visit(Media media) {
		System.out.println("Media fine for Child");
		fine = media.getOverdue() * 0.05;

		if (fine >= maxFine) {
			return maxFine;
		}
		return fine;
	}
}
```

Visitable.java

```
public interface Visitable {
	public double accept(Visitor visitor);
}
```

Book.java

```
public class Book implements Visitable {

	private int loanPeriod;
	private int loaned;

	Book(int day) {
		loaned = day;
	}

	public int getOverdue() {
		if (loaned > loanPeriod) {
			return loaned - loanPeriod;
		}
		return 0;
	}

	@Override
	public double accept(Visitor visitor) {
		return visitor.visit(this);
	}

}
```

Magazine.java

```
public class Magazine implements Visitable {

	private int loanPeriod;
	private int loaned;

	Magazine(int day) {
		loanPeriod = day;
	}

	public int getOverdue() {
		if (loaned > loanPeriod) {
			return loaned - loanPeriod;
		}
		return 0;
```

```
	}

	@Override
	public double accept(Visitor visitor) {
		return visitor.visit(this);
	}
}
```

Media.java

```
public class Media implements Visitable {

	private int loanPeriod;
	private int loaned;

	Media(int day) {
		loanPeriod = day;
	}

	public double getOverdue() {
		if (loaned > loanPeriod) {
			return loaned - loanPeriod;
		}
		return 0;
	}

	@Override
	public double accept(Visitor visitor) {
		return visitor.visit(this);
	}
}
```

VisitorTest.java

```
public class VisitorTest {

	public static void main(String[] args) {
		AdultVisitor fineCalc1 = new AdultVisitor();
		ChildVisitor fineCalc2 = new ChildVisitor();

		Book book = new Book(30);
		Magazine magazine = new Magazine(36);
```

```
            System.out.println(book.accept(fineCalc1));
            System.out.println(book.accept(fineCalc2));
            System.out.println(magazine.accept(fineCalc1));
            System.out.println(magazine.accept(fineCalc2));
        }

}
```

2. (20 points) Final May 2015

You are developing an application that deals with the following workflow –
1) LOCAL - Create a user record in the local database. (via DBFaçade).
2) REMOTE - Create a payment account on a remote server. (via BillPayFaçade).
3) REMOTE – Submit credit card info for payment method details (via BillPayFaçade).

Both the DBFaçade and BillPayFaçade are classes locally available to you. Below are 2 interfaces they implement respectively

```
public interface DBFacadeInterface {
        public void save(User user);
        public void remove(User user);
}
public interface BillPayFacadeInterface {
        public Account createPaymentAccount(User user);
        public void removePaymentAccount(Account account);
        public boolean submitCreditCard(CreditCard card);
}
```

During the workflow, in case step 3) fails for any user, you must rollback step 1) and step 2) to maintain data integrity. Please design/implement the above requirements with the **Command Pattern**. You need –
1) Draw the UML diagram for your solution with all necessary **participants, responsibilities and relationships clearly specified**.
2) Write Java code to implement your design illustrated above. (Need to write code to show how to roll back steps 1&2 if step 3 in the workflow fails).

~~3. (20 points) Midterm July 2016~~

You are working on a framework for GUI application development. The main components that you develop include GUI widgets (buttons, textboxes, checkboxes, dropdown boxes, scrollbars, etc.) and GUI containers (a special container called Window and other regular containers). Your idea is to design the model that stores the containers and widgets in a tree structure using a Composite pattern.

The requirement is for each GUI, there is one and only one special top-level container (called a Window). All widgets and regular containers can be placed on the Window. Regular containers can hold widgets as well as other regular containers. Your design should allow adding/removing widgets onto/from containers and painting them on the screen by simply calling the paint() method on each of the widgets and/or containers.

The way it works is illustrated below –
Painting always starts from the top (the Window) by calling its paint() method first. Then below the top-level Window, if widgets are found, the widgets' paint() methods are called one by one. If regular containers are also found, you should call the first container's paint() method and all its children's paint() methods before moving to the second container on the window. This process goes as deep as the tree structure until all widgets and containers are painted.

1). Design your tree structure using the Composite pattern with a structure diagram showing
participants and their interfaces or responsibilities. 2). Write Java code for your implementation. (For all different widgets, you can use one Widget superclass to represent all of them.)

// YOUR CODE FOR PROB 25 GOES HERE

4. (20 points) Midterm July 2016

You are developing a model for displaying different contents on a client device. Suppose there are only 5 different content types (from most specific to most generic) - image, post, category, archive, and front-page. And correspondingly there are 5 different templates that can be used to display the content types. Below is a table that shows which template is good for which content types.

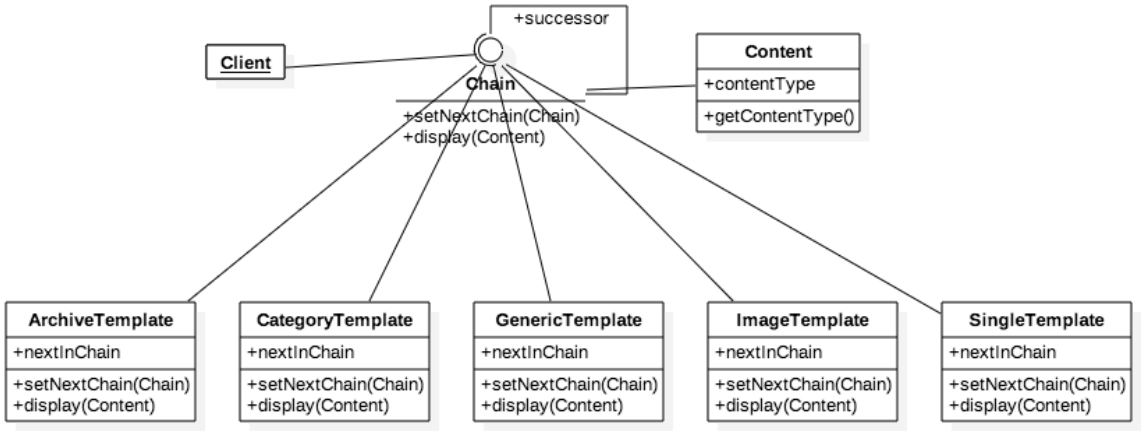
	Image	Post	Category	Archive	Front-page
ImageTemplate	Yes	No	No	No	No
SingleTemplate	Yes	Yes	No	No	No
CategoryTemplate	Yes	Yes	Yes	No	No
ArchiveTemplate	Yes	Yes	Yes	Yes	No
GenericTemplate	Yes	Yes	Yes	Yes	Yes

But the rule is you must always use the most specific template to display any content type. For example, for the 'Category' content type, the 'CategoryTemplate', the 'ArchiveTemplate' and the 'GenericTemplate' are all possible templates, according to the above table. But since the 'CategoryTemplate' is the most specific one among the 3, you must use this one.

1). Use the **Chain of Responsibility pattern** to design the model with a structure diagram, showing all the necessary participants and their relationships and key behavior.
2). Build the chain of templates to make sure the rule is always followed.
3). Implement 1) and 2) with Java code.

You can use the skeleton code for Content for your implementation

```
public class Content {
        private String contentType; //image, post, category, archive, front-page
        //...
        public Content(String type){
                this.contentType = type;
        }
        //...
}
// YOUR CODE FOR PROB 26 GOES HERE
```



Content.java

```
public class Content {
   private String contentType;

   public Content(String type) {
      this.contentType = type;
   }

   public String getContentType() {
      return contentType;
   }
}
```

Chain.java

```
public interface Chain {
        public void setNextChain(Chain nextChain);

        public void display(Content content);
}
```

ArchiveTemplate.java

```
public class ArchiveTemplate implements Chain {

   private Chain nextInChain;

   @Override
```

```
    public void setNextChain(Chain nextChain) {
        nextInChain = nextChain;
    }

    @Override
    public void display(Content content) {
        if (content.getContentType() == "Image" || content.getContentType() == "Post" ||
content.getContentType() == "Category" || content.getContentType() == "Archive") {
            System.out.println("Content type: " + content.getContentType());
        } else {
            nextInChain.display(content);
        }
    }
}
```

CategoryTemplate.java

```
public class CategoryTemplate implements Chain {

    private Chain nextInChain;

    @Override
    public void setNextChain(Chain nextChain) {
        nextInChain = nextChain;
    }

    @Override
    public void display(Content content) {
        if (content.getContentType() == "Image" || content.getContentType() == "Post" ||
content.getContentType() == "Category") {
            System.out.println("Category template: " + content.getContentType());
        } else {
            nextInChain.display(content);
        }
    }
}
```

GenericTemplate.java

```
public class GenericTemplate implements Chain {

    private Chain nextInChain;
```

```
    @Override
    public void setNextChain(Chain nextChain) {
        nextInChain = nextChain;
    }

    @Override
    public void display(Content content) {
        if (content.getContentType() == "Image" || content.getContentType() == "Post" ||
content.getContentType() == "Category" || content.getContentType() == "Archive" ||
content.getContentType() == "Front-page")
            System.out.println("Generic template: " + content.getContentType());
        else {
            System.out.println("Only works for Image, Post, Category, Archive, Front-page");
        }
    }
}
```

ImageTemplate.java

```
public class ImageTemplate implements Chain {

    private Chain nextInChain;

    @Override
    public void setNextChain(Chain nextChain) {
        this.nextInChain = nextChain;
    }

    @Override
    public void display(Content content) {
        if (content.getContentType() == "Image") {
            System.out.println("Image template: " + content.getContentType());
        } else {
            nextInChain.display(content);
        }
    }
}
```

SingleTemplate.java

```
public class SingleTemplate implements Chain {
   private Chain nextInChain;

   @Override
   public void setNextChain(Chain nextChain) {
      nextInChain = nextChain;
   }

   @Override
   public void display(Content content) {
      if (content.getContentType() == "Image" || content.getContentType() == "Post") {
         System.out.println("Single template: " + content.getContentType());
      } else {
         nextInChain.display(content);
      }
   }
}
```

TestChain.java

```
public class TestChain {
   public static void main(String[] args) {
      Chain c1 = new ImageTemplate();
      Chain c2 = new SingleTemplate();
      Chain c3 = new CategoryTemplate();
      Chain c4 = new ArchiveTemplate();
      Chain c5 = new GenericTemplate();

      c1.setNextChain(c2);
      c2.setNextChain(c3);
      c3.setNextChain(c4);
      c4.setNextChain(c5);

      Content request = new Content("Post");
      c1.display(request);
   }
}
```

5. (40 points) Midterm March 2015

You are developing a framework for building GUI applications. The idea is to use an object model to store all GUI widgets (buttons, textboxes, checkboxes, dropdown boxes, scrollbars, etc.) and containers of them (containers and windows). To hold the widgets and manage their layout, you use a container that associates a widget manager with it (which manages positions and sizes of widgets that are placed on the container). But containers can also be placed on other containers so you can manage layout easily. To make a GUI executable/displayable, all widgets and containers have to be put on one and only one window widget that is a special container. So the model that stores all widgets, containers and a window is actually a data structure that starts from the window which can hold widgets and containers which in turn can hold other widgets and other containers so on and on. One important rule though is any container cannot hold itself and any 2 containers cannot hold each other.

To manage widgets layout/size on a container, you need to develop widget managers. The idea is to have 3 managers to start with – the central (position) manager, the quadrant manager and a sequential manager. At runtime, the GUI developer can specifically set a widget manager for a container or let it to the framework to handle it. That means if no widget manager is set for a container, the framework should first try applying the central manager (good for containers with 1 or 2 widgets only), then try quadrant manager (if a container has a total of exactly 4 widgets and/or containers) if central manager is not a good choice. If the quadrant manager still cannot manage the widgets nicely, the framework will pick the sequential manager as the choice of last resort (that is to simply lay the widgets sequentially one after another).

To display a GUI developed with this framework, you need to develop a displaying module that always starts from the window widget. Then it moves down to all the widgets and containers that are placed on it until all widgets and containers are displayed. When designing/implementing the displaying module, you have to keep in mind that multiple platforms are considered (Windows, Macintosh, and Linux with possibilities of other new platforms to add later on). But one class that common to all platforms is that you need to provide a class that goes through all widgets/containers stored in the data structure and call the corresponding displaying API to display them correctly.

You are responsible to design/implement the following components with design patterns we have learned so far in the course.

1) (25 points) The data model that stores all widgets/containers for a GUI application that allows application developers to easily manage widgets on the GUI.
2) (20 points) The widget-managing module that manages positions and sizes.
3) (20 points) The displaying API/implementations for rendering on different platforms.

4) (15 points) The ‘common’ class that helps the displaying API to render all widgets/containers stored in the data model.
// YOUR CODE FOR PROB 26 GOES HERE

6. You are designing an order processing system for a web-based business. All normal domestic orders will have to be shipped by FedEx. All large bulk orders are handled separately within the warehouse and might go to different shipping carriers. Most Favored Customers (your largest customers) may have special requirements for shipping and tracking, and they’re going to get what they want. All other international orders are handled in a generic manner. Draw both the class and object structures of your design and provide some skeleton code that shows how it works.