CSE 223: Programming -2
10-Behavioral Design Patterns (I)

Prof. Dr. Khaled Nagi

Department of Computer and Systems Engineering,

Faculty of Engineering, Alexandria University, Egypt.

khaled.nagi@alexu.edu.eg

Agenda

- *They are used to organize, manage, and combine behavior*.
- *They help you define the communication between objects in your system and how the flow is controlled in a complex program.*
- List
  - Chain of responsibility
  - Command
  - Little language
  - Mediator
  - Snapshot
  - Observer

    Part I
  - State
  - Null object
  - Strategy
  - Template method
  - Visitor

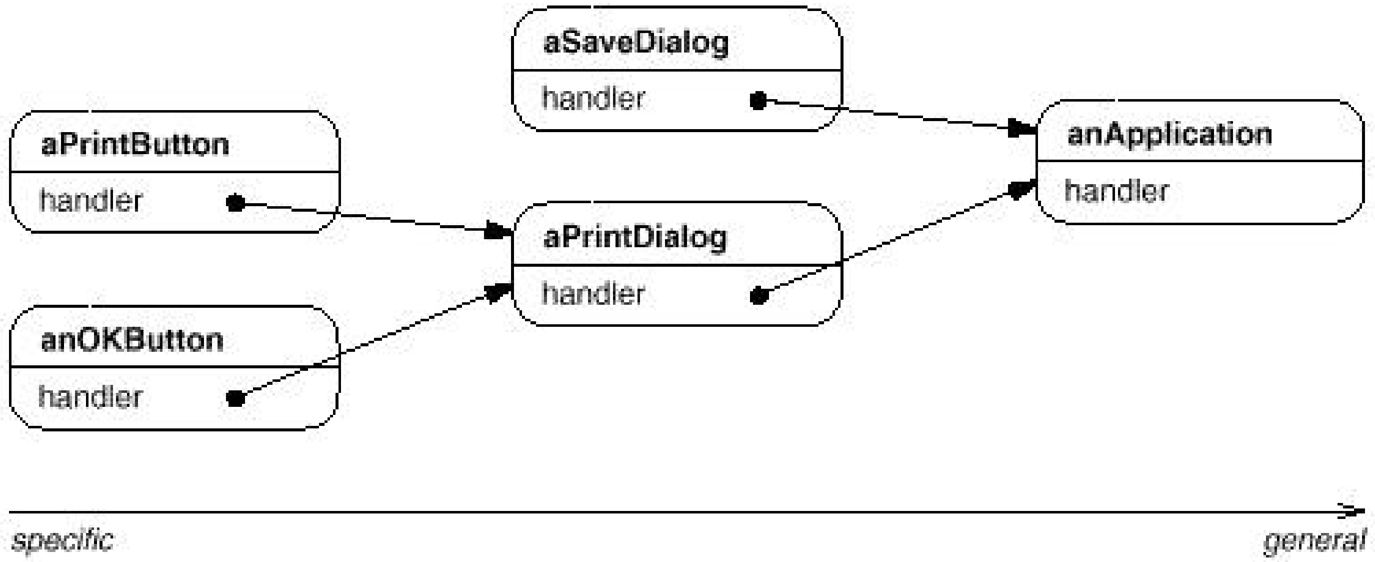
Chain of responsibility

Chain of responsibility

- 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
  - Consider a context-sensitive help system for a GUI
  - The object that ultimately provides the help isn't known explicitly to the object (e.g., a button) that initiates the help request
  - So use a chain of objects to decouple the senders from the receivers. The request gets passed along the chain until one of the objects handles it.
  - Each object on the chain shares a common interface for handling requests and for accessing its successor on the chain

Chain of responsibility

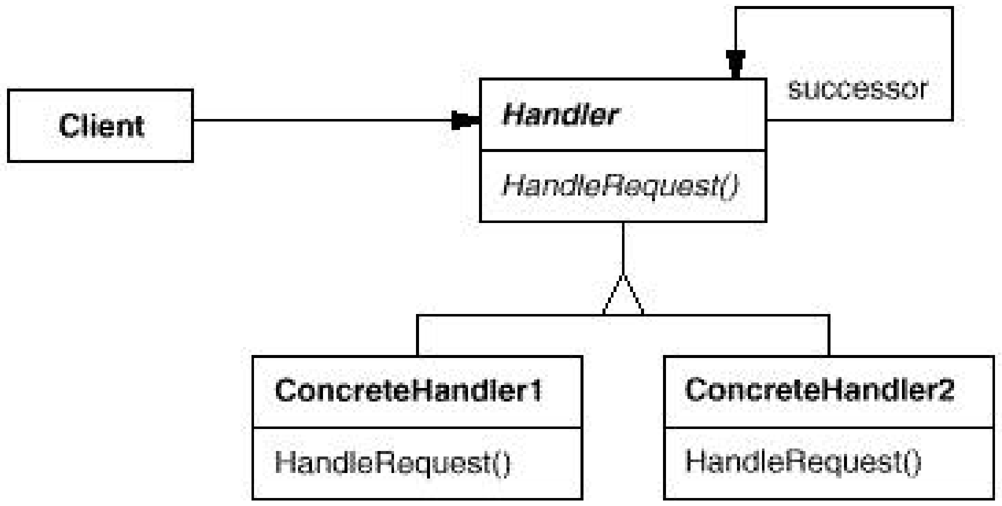
- Motivation

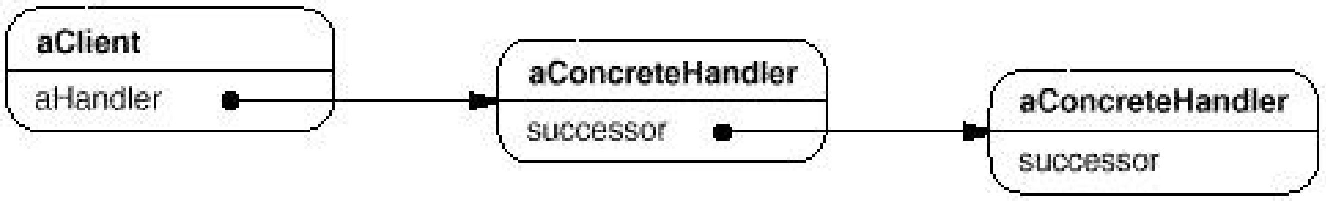


Chain of responsibility

- Applicability
  - When more than one object may handle a request and the actual handler is not know in advance
  - When requests follow a “handle or forward” model - that is, some requests can be handled where they are generated while others must be forwarded to another object to be handled
- Consequences
  - Reduced coupling between the sender of a request and the receiver – the sender and receiver have no explicit knowledge of each other
  - Receipt is not guaranteed - a request could fall off the end of the chain without being handled
  - The chain of handlers can be modified dynamically

Chain of responsibility





Chain of responsibility

- Example 1
  - The designers of a set of GUI classes need to have a way to propagate GUI events, such as MOUSE_CLICKED, to the individual component where the event occurred and then to the object or objects that are going to handle the event.
  - Solution
    - First post the event to the component where the event occurred. That component can handle the event or post the event to its container component (or both!). The next component in the chain can again either handle the event or pass it up the component containment hierarchy until the event is handled.
  - ***This technique was actually used in the Java 1.0 AWT***

```
public boolean action(Event event, Object obj) {
  if (event.target == test_button)
    doTestButtonAction();
  else if (event.target == exit_button)
    doExitButtonAction();
  else
    return super.action(event,obj);
  return true;  // Return true to indicate the event has been
                // handled and should not be propagated further.
}
```

Chain of responsibility

- Example 1
  - ***In Java 1.1 the AWT event model was changed from the Chain of Responsibility (CoR) pattern to the Observer pattern.***
- Reason
  - Efficiency
    - GUIs frequently generate many events, such as MOUSE_MOVE events.
    - In many cases, the application did not care about these events.
    - The GUI framework would propagate the event up the containment hierarchy until some component handled it.
    - ***This caused the GUI to slow noticeably.***
  - Flexibility:
    - The CoR pattern assumes a common Handler superclass or interface for all objects which can handle chained requests.
    - In the case of the Java 1.0 AWT, every object that could handle an event had to be a subclass of the Component class.
    - ***Events could not be handled be non-GUI objects, limiting the flexibility of the program.***

Chain of responsibility

- Example 2
  - The approval of purchasing requests.
  - The approval authority depends on the dollar amount of the purchase.
  - The approval authority could change at any time
- Solution
  - **PurchaseRequest** objects forward the approval request to a **PurchaseApproval** object.
  - Depending on the dollar amount, the **PurchaseApproval** object may approve the request or forward it on to the next approving authority in the chain.
  - The approval authority at any level in the chain can be easily modified without affecting the original **PurchaseRequest** object.

Chain of responsibility

- Handling requests
  - Classical

```
public interface Handler {
  public void handleRequest();
}
```

  - Handling several kinds of requests
    - Solution 1

```
public interface Handler {
  public void handleHelp();
  public void handlePrint();
  public void handleFormat();
}
```

```
public class ConcreteHandler implements Handler {
  private Handler successor;

  public ConcreteHandler(Handler successor) {
    this.successor = successor;
  }

  public void handleHelp() {
    // We handle help ourselves, so help code is here.
  }

  public void handlePrint() {
    successor.handlePrint();
  }
```

if we add a new kind of request we need to change the interface which means that all concrete handlers need to be modified!

Chain of responsibility

- Handling several kinds of requests
  - Solution 2: separate handler interfaces for each type of request

```
public interface HelpHandler {
  public void handleHelp();
}

public interface PrintHandler {
  public void handlePrint();
}

public interface FormatHandler {
  public void handleFormat();
}
```

The concrete handler must have successor references to each type of request that it deals with, in case it needs to pass the request on to its successor.

Chain of responsibility

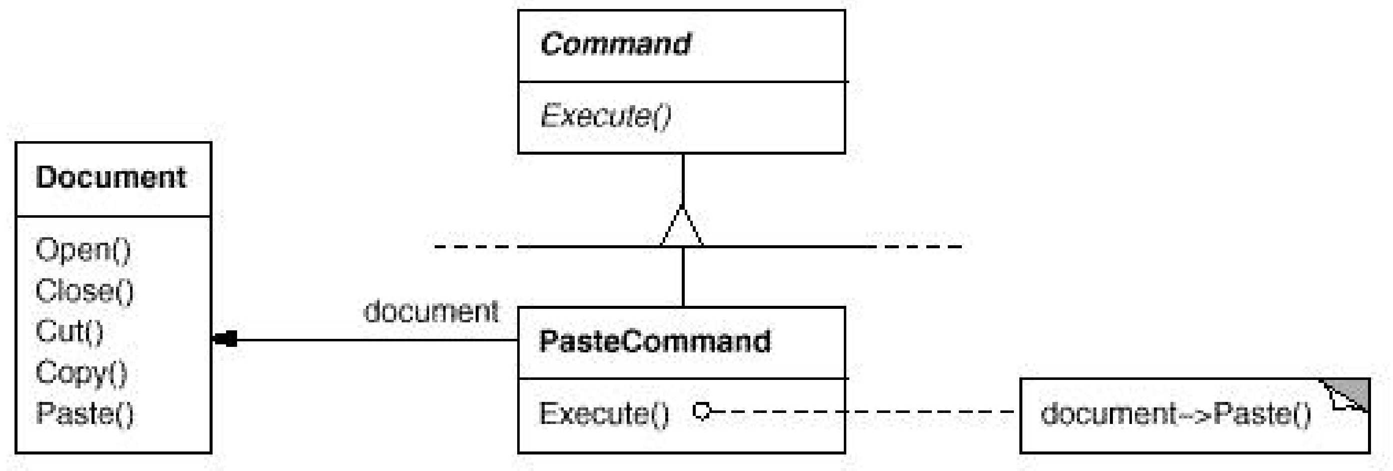
- Handling several kinds of requests
  - Solution 3: single method in the Handler interface which takes an argument describing the type of request

```
public interface Handler {
  public void handleRequest(String request);
}


public class ConcreteHandler implements Handler {
  private Handler successor;

  public ConcreteHandler(Handler successor) {
    this.successor = successor;
  }

  public void handleRequest(String request) {
    if (request.equals("Help")) {
      // We handle help ourselves, so help code is here.
    }
    else
      // Pass it on!
      successor.handle(request);
  }
}
```

Command Pattern

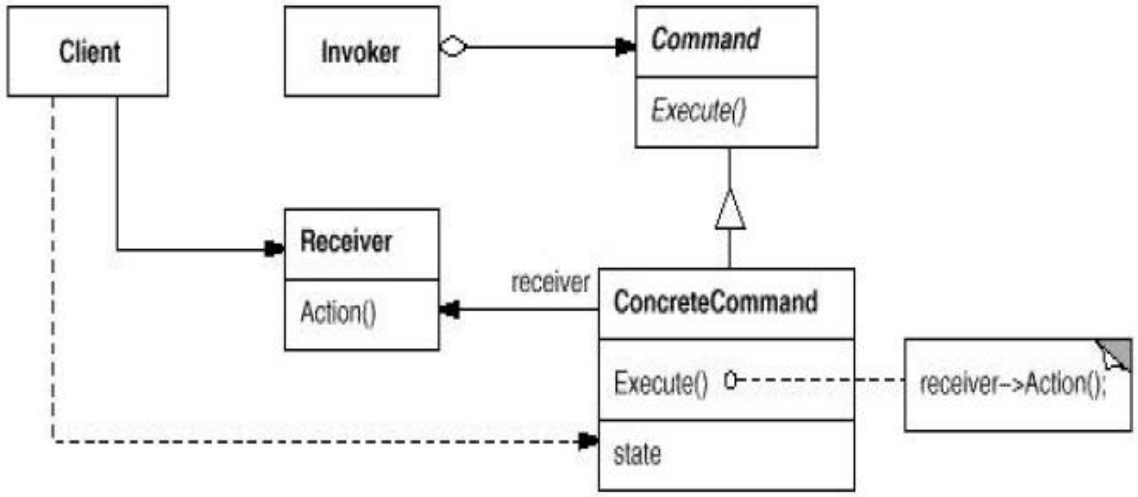
Command Pattern

- Motivation
  - You want to implement a callback function capability
  - You want to specify, queue, and execute requests at different times
  - You need to support undo and change log operations
- Solution

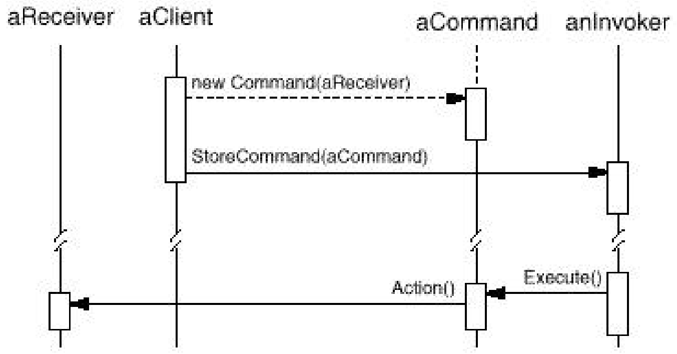


Command Pattern

- UML class Diagram

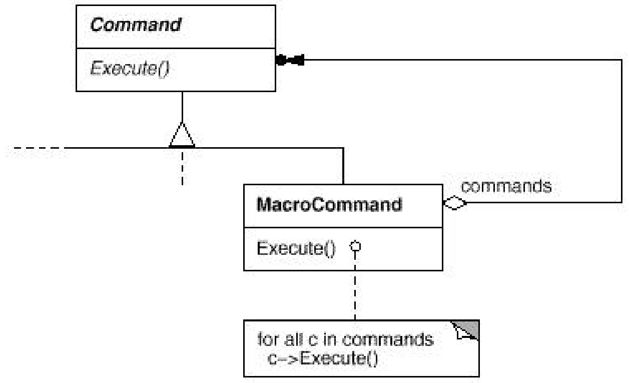


- Collaboration Diagram



Command Pattern

- Consequences
  - Command decouples the object that invokes the operation from the one that knows how to perform it
  - Commands are first-class objects. They can be manipulated and extended like any other object.
  - Commands can be made into a composite command
- How intelligent should a command object be?
  - Dumb
    - Delegates the required action to a receiver object
    - Simple functor
  - Smart:
    - Implements everything itself without delegating to a receiver object at all
    - Functor



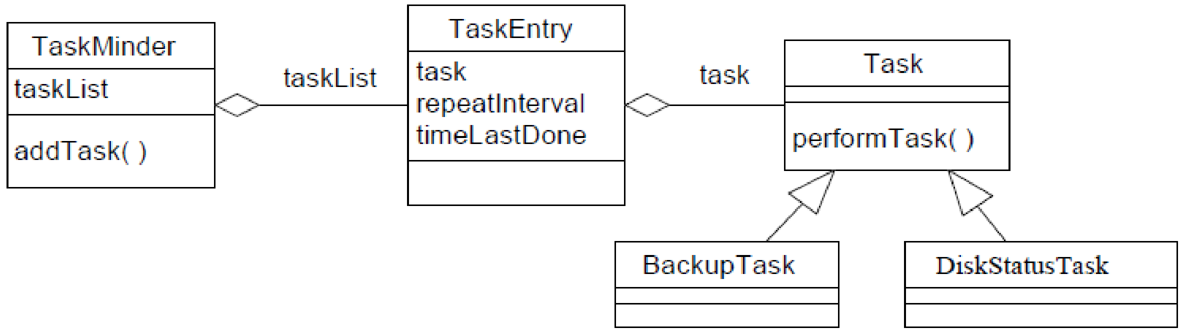
- ***A command object frequently delegates the desired behavior to another receiver object.***

Command Pattern

- Example 1
  - A GUI system has several buttons that perform various actions.
  - We want to have menu items that perform the same action as its corresponding button.
  - Have an action listener for each paired button and menu item.
  - Keep the required actions in the **actionPerformed**() method of this one action listener.
  - In Java 2, *Swing Action objects* are used for this purpose
- Example 2
  - A scheduler: write a class that can periodically execute one or more methods of various objects.
  - For example, we want to run a backup operation every hour and a disk status operation every ten minutes.
  - We want to decouple the class that schedules the execution of these methods with the classes that actually provide the behavior we want to execute.

Command Pattern

- Example 2 (Continued)



```
public interface Task {
  public void performTask();
}
```

```
public class FortuneTask implements Task {
  int nextFortune = 0;
  String[] fortunes = {"She who studies hard, gets A",
    "Seeth the pattern and knoweth the truth",
    "He who leaves state the day after final, graduates not" };
  public FortuneTask() {}
  public void performTask() {
    System.out.println("Your fortune is: " +
                       fortunes[nextFortune]);
    nextFortune = (nextFortune + 1) % fortunes.length;
  }
  public String toString() {return ("Fortune Telling Task");}
}
```

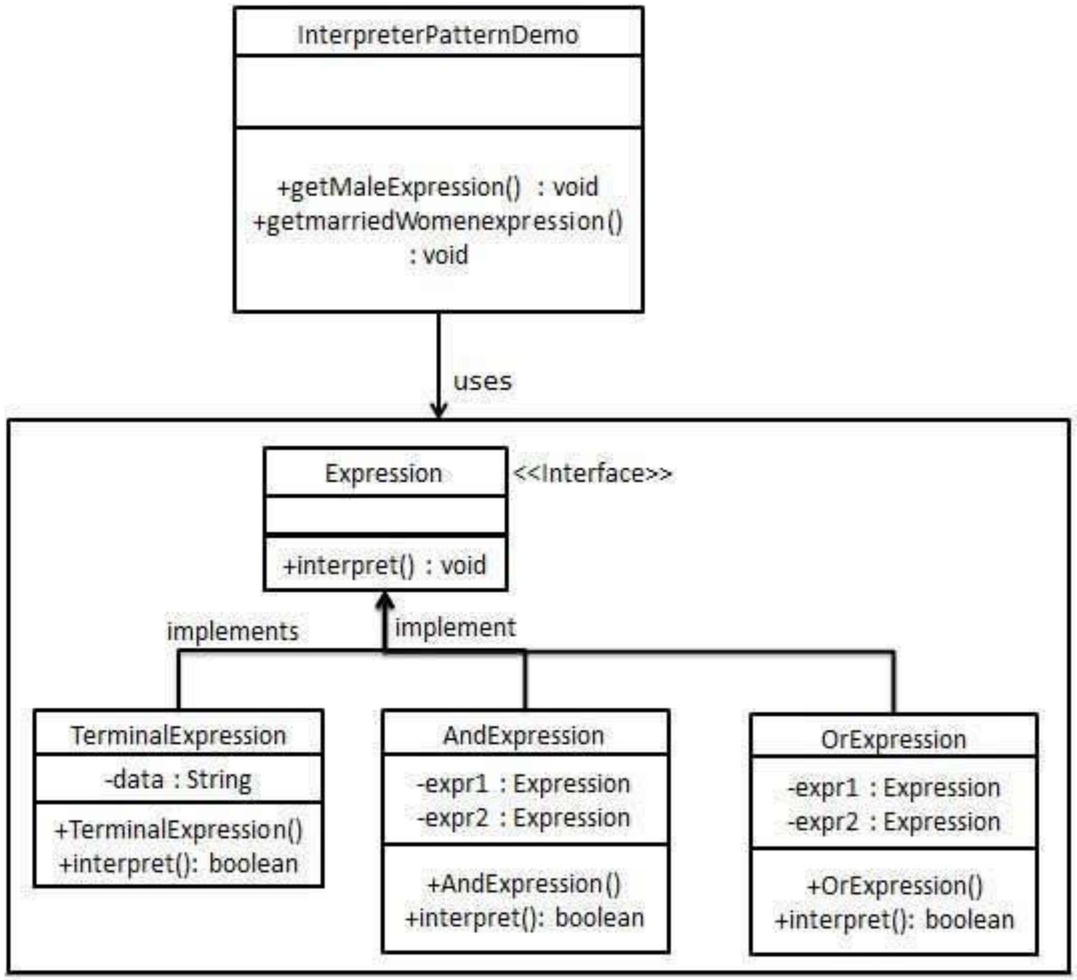
```
public class FibonacciTask implements Task {
  int n1 = 1;
  int n2 = 0;
  int num;

  public FibonacciTask() {}
  public void performTask() {
    num = n1 + n2;
    System.out.println("Next Fibonacci number is: " + num);
    n1 = n2;
    n2 = num;
  }
  public String toString() {return ("Fibonacci Sequence Task");}
}
```

Little Language

Little Language

- Objectives
  - representing code as data
  - When you need to solve a problem, instead of writing a program to solve just that one problem, build a language that can solve a range of related problems.
  - Usually called **Interpreter** pattern
  - A way to evaluate language grammar or expression.
  - It involves implementing an expression interface which tells to interpret a particular context.
  - This pattern is used in **SQL parsing**, symbol processing engine etc.

Little Language



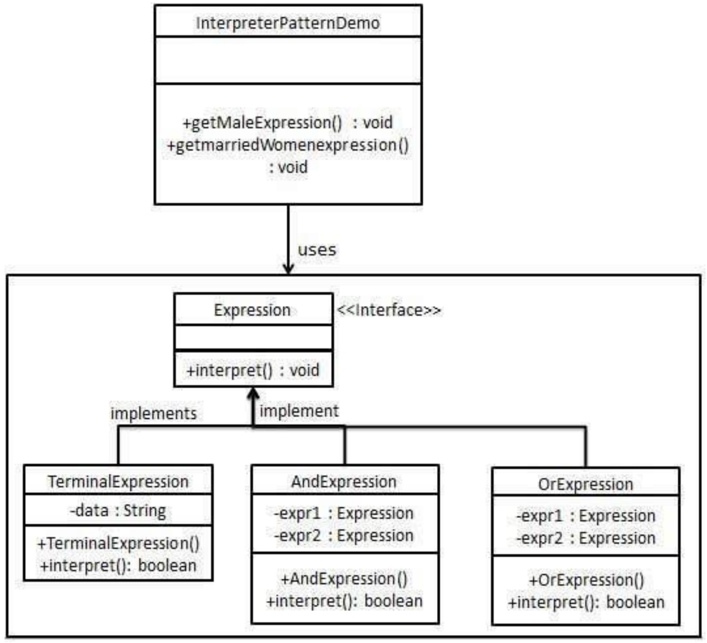
Little Language

- **Step 1:** Create an expression interface.

```
public interface Expression {
   public boolean interpret(String context);
}
```

- **Step 2:** Create concrete classes

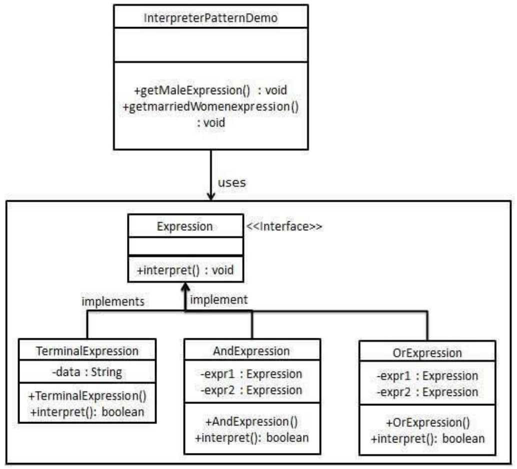
```
public class TerminalExpression implements Expression {

   private String data;

   public TerminalExpression(String data){
      this.data = data;
   }

   @Override
   public boolean interpret(String context) {

      if(context.contains(data)){
         return true;
      }
      return false;
   }
}
```



Little Language

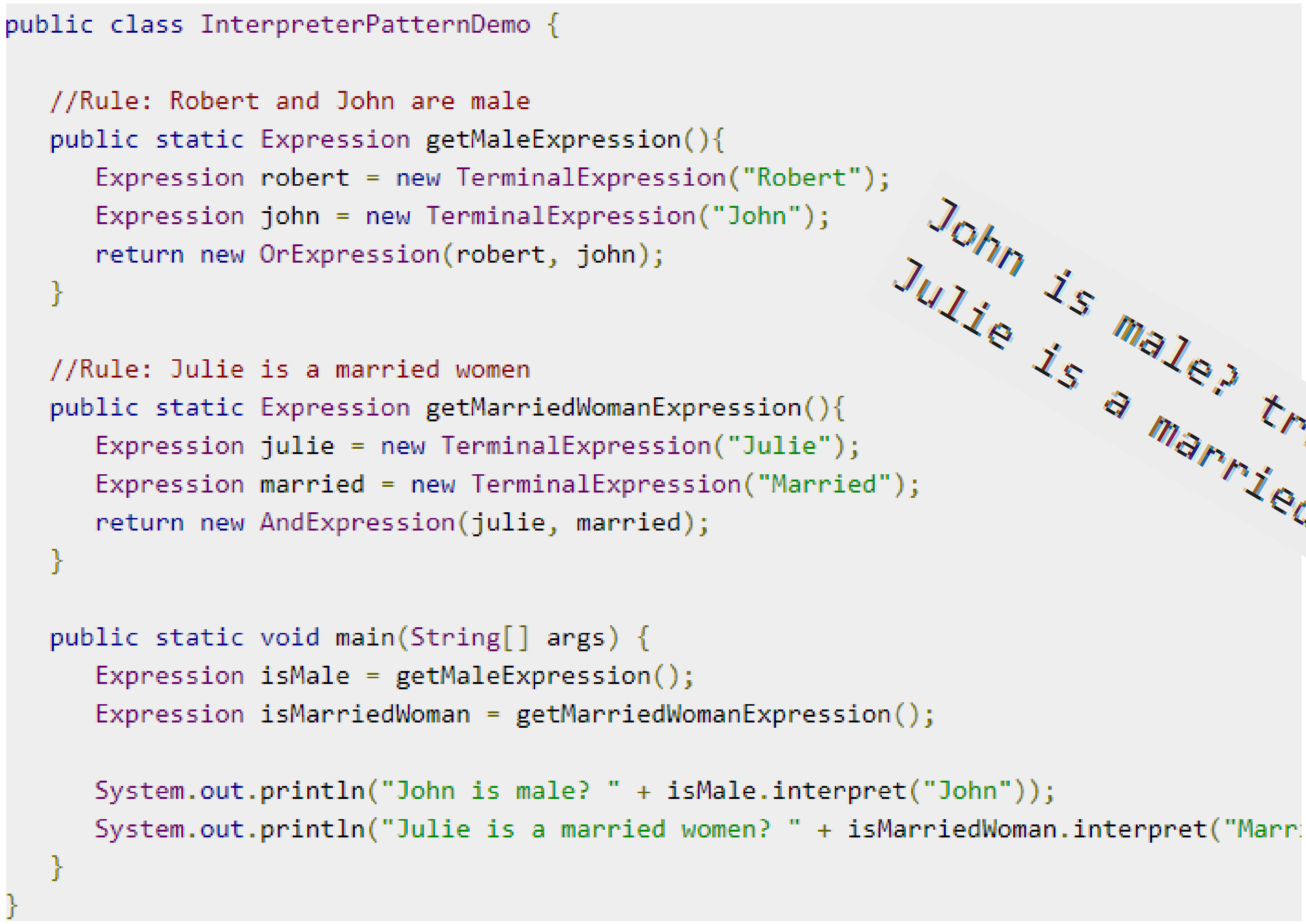
- **Step 2:** cont.

```java
public class OrExpression implements Expression {

   private Expression expr1 = null;
   private Expression expr2 = null;

   public OrExpression(Expression expr1, Expression expr2) {
      this.expr1 = expr1;
      this.expr2 = expr2;
   }

   @Override
   public boolean interpret(String context) {
      return expr1.interpret(context) || expr2.interpret(context);
   }
}
```

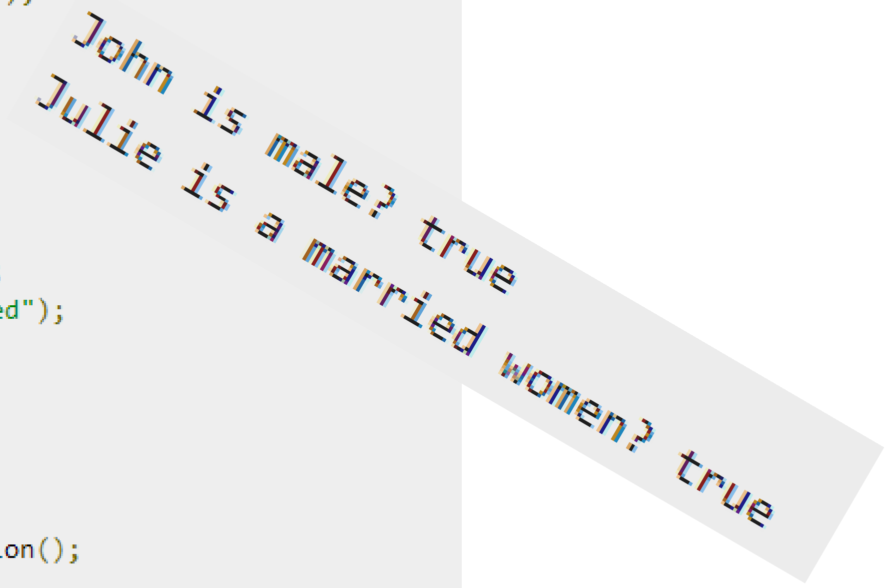


Little Language

- **Step 3:** *InterpreterPatternDemo* uses *Expression* class to create rules and then parse them.



```
public class InterpreterPatternDemo {

   //Rule: Robert and John are male
   public static Expression getMaleExpression(){
      Expression robert = new TerminalExpression("Robert");
      Expression john = new TerminalExpression("John");
      return new OrExpression(robert, john);
   }

   //Rule: Julie is a married women
   public static Expression getMarriedWomanExpression(){
      Expression julie = new TerminalExpression("Julie");
      Expression married = new TerminalExpression("Married");
      return new AndExpression(julie, married);
   }

   public static void main(String[] args) {
      Expression isMale = getMaleExpression();
      Expression isMarriedWoman = getMarriedWomanExpression();

      System.out.println("John is male? " + isMale.interpret("John"));
      System.out.println("Julie is a married women? " + isMarriedWoman.interpret("Marr
   }
}
```

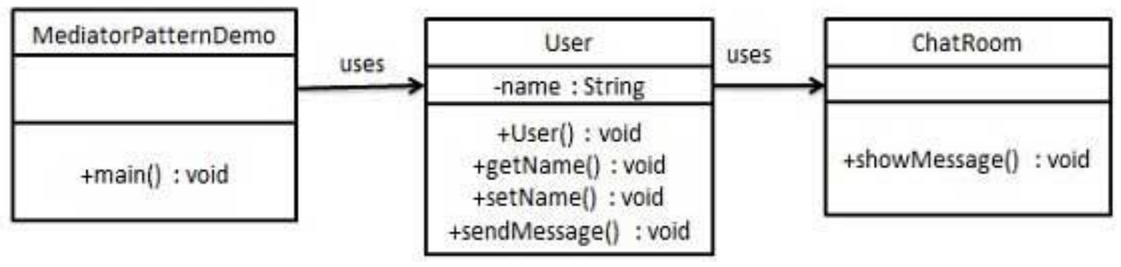


```
John is male? true
Julie is a married women? true
```

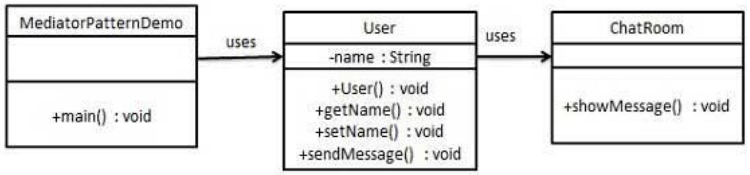
Mediator Pattern

Mediator Pattern

- It used to reduce communication complexity between multiple objects.
- It supports easy maintenance of the code by loose coupling.
- Example
  - A chat room where multiple users can send message to chat room.
  - It is the responsibility of chat room to show the messages to all users.
  - Two classes **ChatRoom** and **User**.
    - User objects will use ChatRoom method to share their messages.
  - **MediatorPatternDemo**, the demo class, will use User objects to show communication between them.



Mediator Pattern



- **Step 1:** Create mediator class

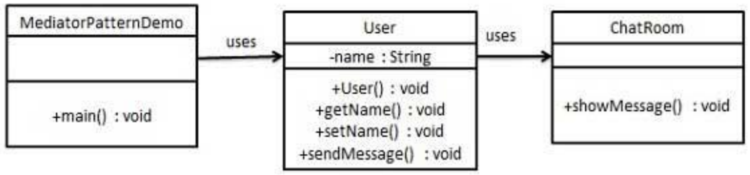
```
import java.util.Date;

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

```
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);
   }
}
```

Mediator Pattern

- **Step 3:** Use the User object to show communications between them



```
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!");
   }
}
```

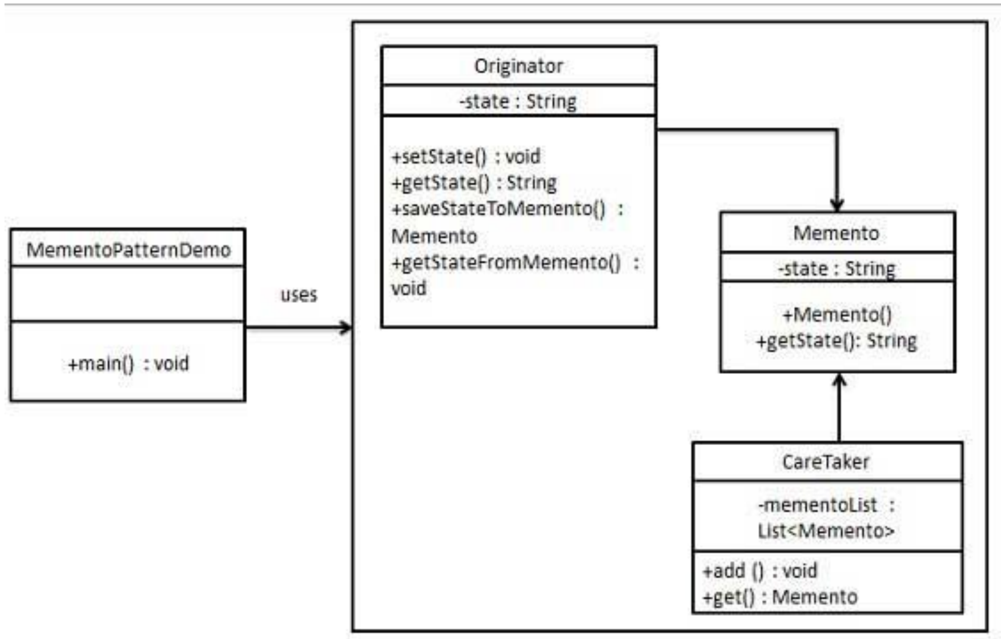
- **Step 4:** Output

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

Snapshot Pattern

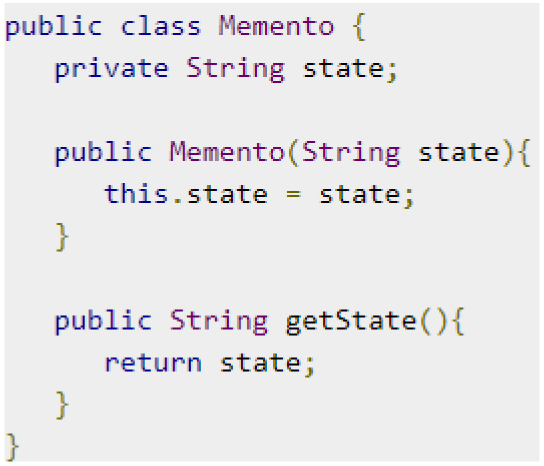
Snapshot pattern

- It is used to restore state of an object to a previous state.
- It is also often called *Memento*
- Example:
  - Autosave in word, excel, etc.
- It uses three actor classes.
  - **Memento** contains state of an object to be restored.
  - **Originator** creates and stores states in Memento objects
  - **Caretaker** object is responsible to restore object state from Memento.



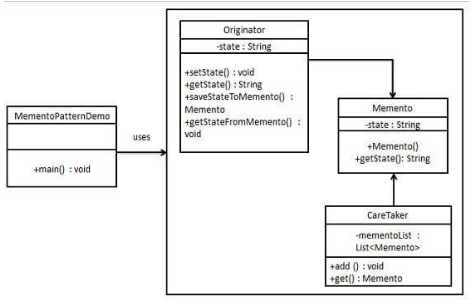
Snapshot pattern

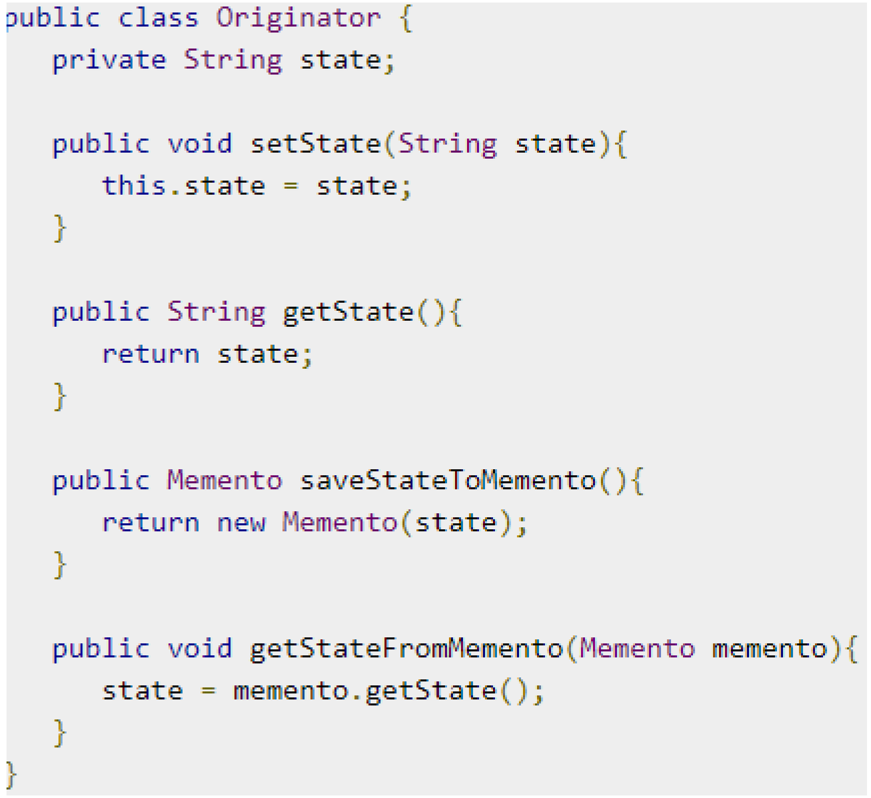
- **Step 1:** Create Memento class



```java
public class Memento {
   private String state;

   public Memento(String state){
      this.state = state;
   }

   public String getState(){
      return state;
   }
}
```

- **Step 2:** Create Originator class

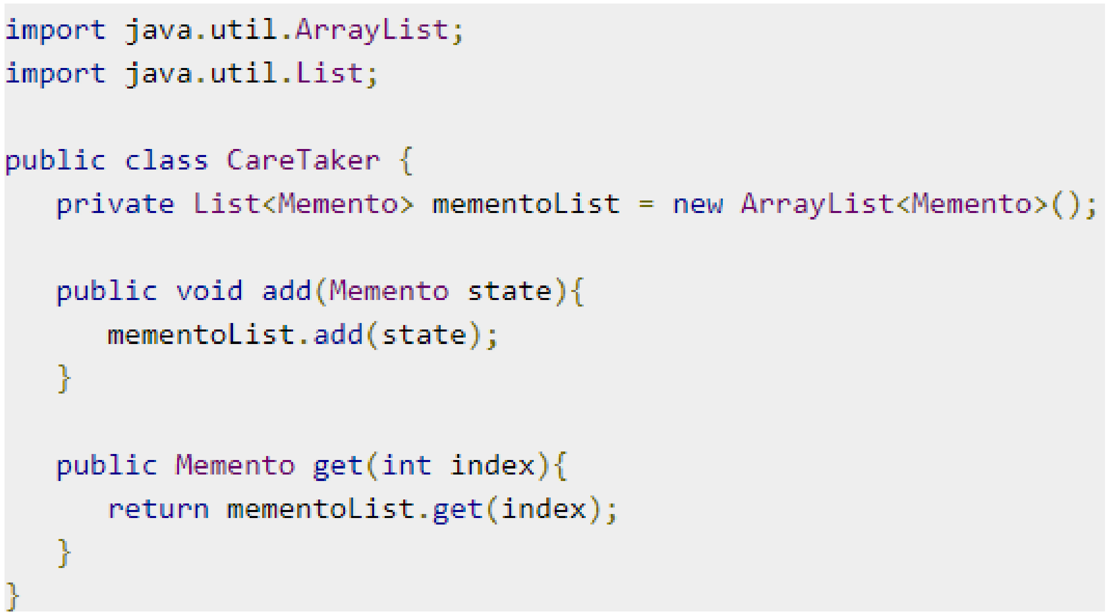




```java
public class Originator {
   private String state;

   public void setState(String state){
      this.state = state;
   }

   public String getState(){
      return state;
   }

   public Memento saveStateToMemento(){
      return new Memento(state);
   }

   public void getStateFromMemento(Memento memento){
      state = memento.getState();
   }
}
```

Snapshot pattern

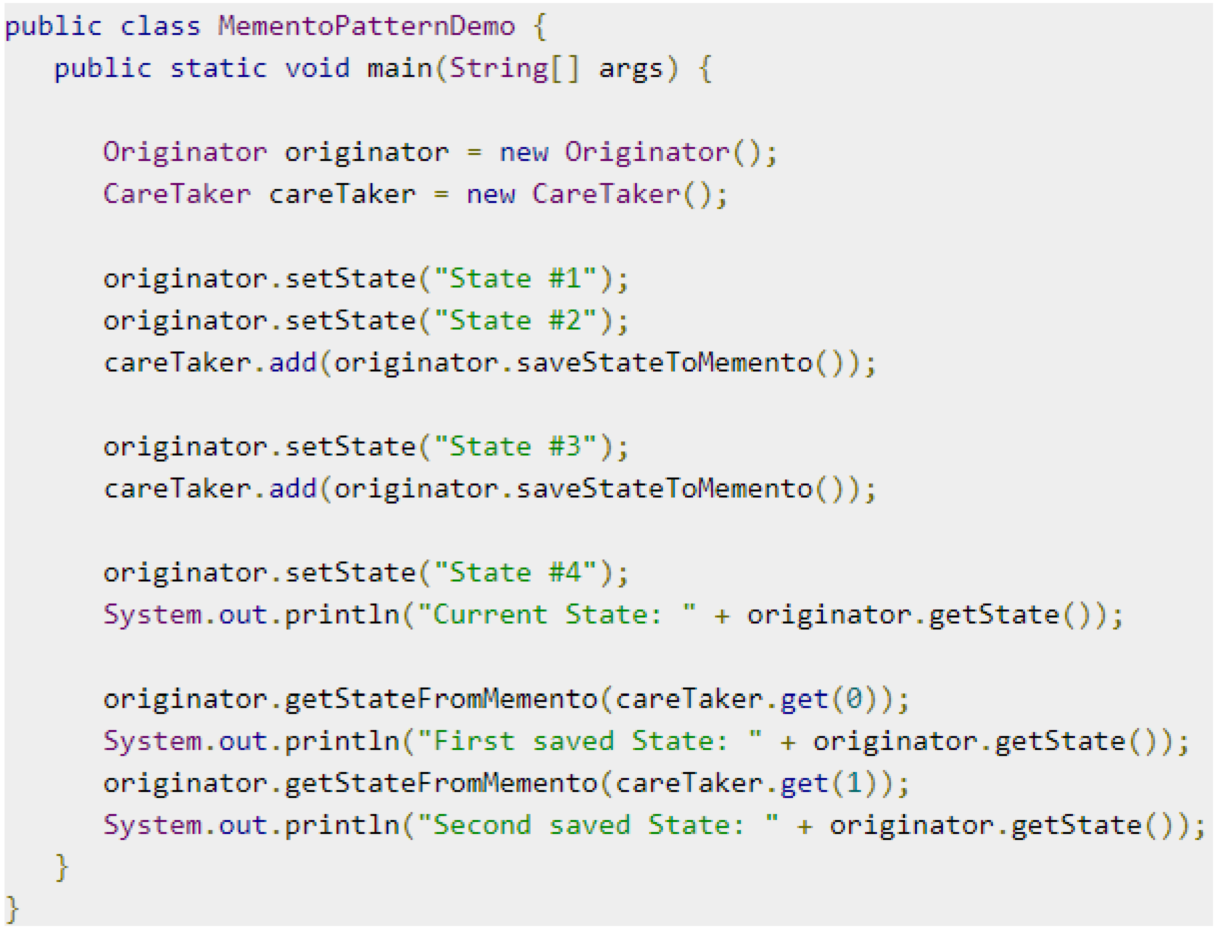
- **Step 3:** Create CareTaker class



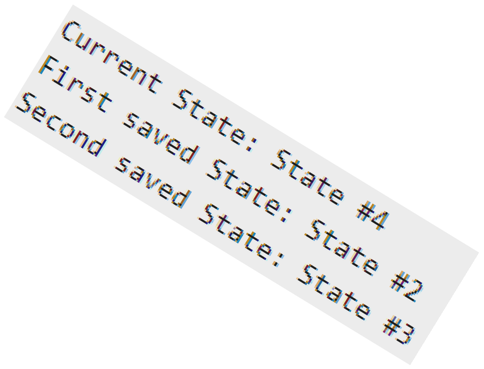
```java
import java.util.ArrayList;
import java.util.List;

public class CareTaker {
   private List<Memento> mementoList = new ArrayList<Memento>();

   public void add(Memento state){
      mementoList.add(state);
   }

   public Memento get(int index){
      return mementoList.get(index);
   }
}
```

Snapshot pattern

- **Step 4:** Use CareTaker and Originator objects



```
public class MementoPatternDemo {
   public static void main(String[] args) {

      Originator originator = new Originator();
      CareTaker careTaker = new CareTaker();

      originator.setState("State #1");
      originator.setState("State #2");
      careTaker.add(originator.saveStateToMemento());

      originator.setState("State #3");
      careTaker.add(originator.saveStateToMemento());

      originator.setState("State #4");
      System.out.println("Current State: " + originator.getState());

      originator.getStateFromMemento(careTaker.get(0));
      System.out.println("First saved State: " + originator.getState());
      originator.getStateFromMemento(careTaker.get(1));
      System.out.println("Second saved State: " + originator.getState());
   }
}
```

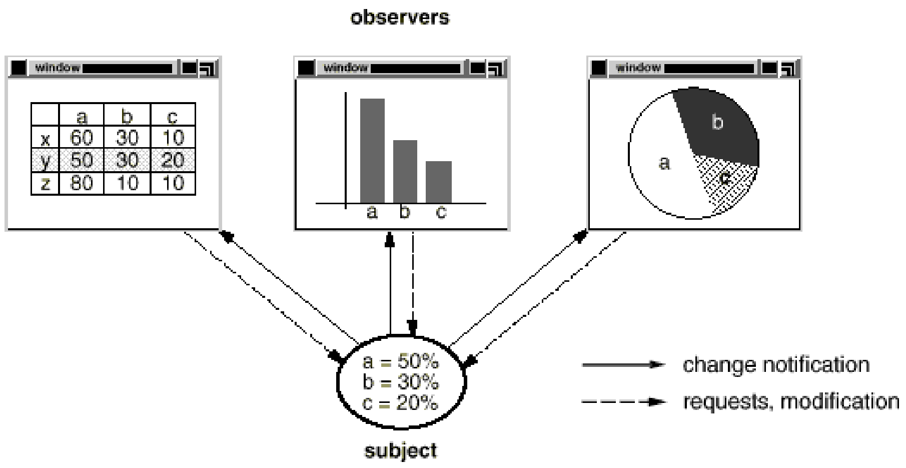


```
Current State: State #4
First saved State: State #2
Second saved State: State #3
```

Observer Pattern

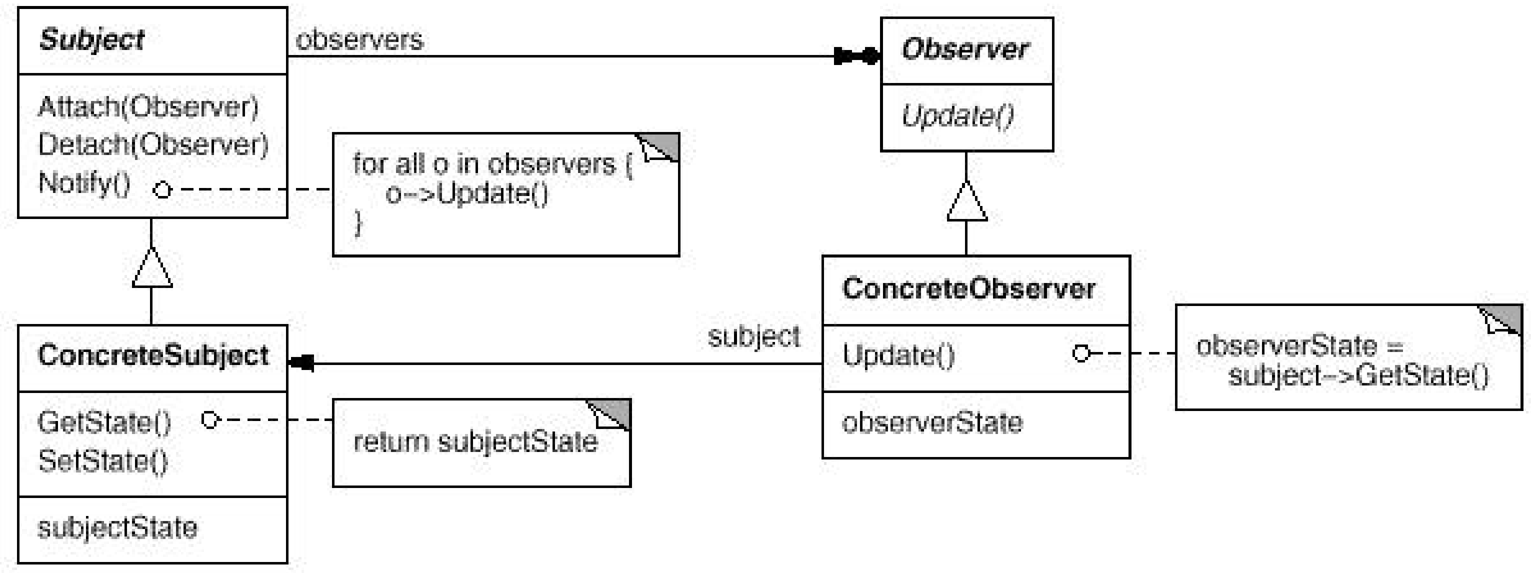
Observer Pattern

- Intent
  - Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically
- Also Known As *Dependents*, *Publish-Subscribe*
- Motivation
  - The need to maintain consistency between related objects without making classes tightly coupled

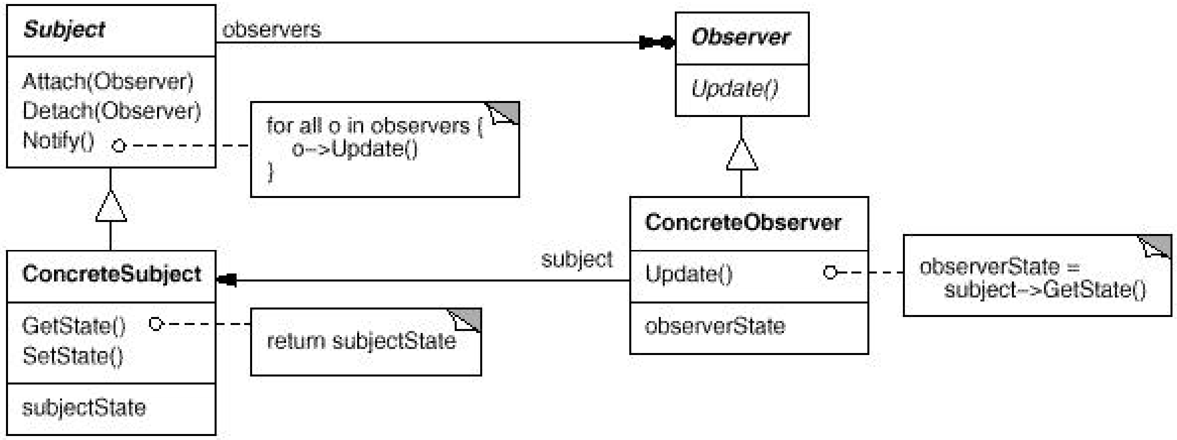


Observer Pattern

- Applicability
  - When an abstraction has two aspects, one dependent on the other.
  - When a change to one object requires changing others
  - When an object should be able to notify other objects without making assumptions about those objects
  - Encapsulating these aspects in separate objects lets you vary and reuse them independently.



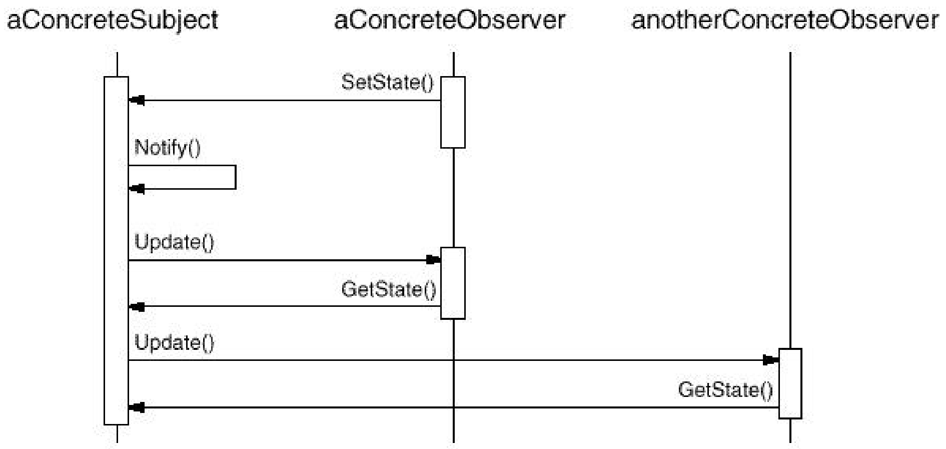
Observer Pattern



- **Subject**
  - Keeps track of its observers
  - Provides an interface for attaching and detaching Observer objects
- **Observer**
  - Defines an interface for update notification
- **ConcreteSubject**
  - The object being observed
  - Stores state of interest to **ConcreteObserver** objects
  - Sends a notification to its observers when its state changes

Observer Pattern

- **ConcreteObserver**
  - The observing object
  - Stores state that should stay consistent with the subject's
  - Implements the Observer update interface to keep its state consistent with the subject's



Observer Pattern

- Consequences
  - Minimal coupling between the Subject and the Observer
  - Can reuse subjects without reusing their observers and vice versa
  - Observers can be added without modifying the subject
  - All subject knows is its list of observers
  - Subject does not need to know the concrete class of an observer, just that each observer implements the update interface
  - Subject and observer can belong to different abstraction layers
  - Support for event broadcasting
  - Subject sends notification to all subscribed observers
  - Observers can be added/removed at any time
  - Liabilities
    - Possible cascading of notifications
      - Observers are not necessarily aware of each other and must be careful about triggering updates
    - Simple update interface requires observers to deduce changed item

Observer Pattern

- Implementation Issues
  - How does the subject keep track of its observers?
    - Array, linked list
  - What if an observer wants to observe more than one subject?
    - Have the subject tell the observer who it is via the update interface
  - Who triggers the update?
    - The subject whenever its state changes
    - The observers after they cause one or more state changes
    - Some third party object(s)
  - Make sure the subject updates its state before sending out notifications
  - How much info about the change should the subject send to the observers?
    - Push Model – Lots
    - Pull Model - Very Little
  - Can the observers subscribe to specific events of interest? *publish-subscribe*
  - What if an observer wants to be notified only after several subjects have changed state?
    - Use an intermediary object which acts as a mediator
    - Subjects send notifications to the mediator object which performs any necessary processing before notifying the observers

Observer Pattern

```
/**
 * A subject to observe!
 */
public class ConcreteSubject extends Observable {

  private String name;
  private float price;

  public ConcreteSubject(String name, float price) {
    this.name = name;
    this.price = price;
    System.out.println("ConcreteSubject created: " + name + " at "
      + price);
  }

  public String getName() {return name;}

  public float getPrice() {return price;}

  public void setName(String name) {
    this.name = name;
    setChanged();
    notifyObservers(name);
  }

  public void setPrice(float price) {
    this.price = price;
    setChanged();
    notifyObservers(new Float(price));
  }

}
```

Observer Pattern

```
// An observer of name changes.
public class NameObserver implements Observer {
  private String name;

  public NameObserver() {
    name = null;
    System.out.println("NameObserver created: Name is " + name);
  }
  public void update(Observable obj, Object arg) {
    if (arg instanceof String) {
      name = (String)arg;
      System.out.println("NameObserver: Name changed to " + name);
    } else {
      System.out.println("NameObserver: Some other change to
   subject!");
    }
  }
}
// An observer of price changes.
public class PriceObserver implements Observer {
  private float price;

  public PriceObserver() {
    price = 0;
    System.out.println("PriceObserver created: Price is " + price);
  }
  public void update(Observable obj, Object arg) {
    if (arg instanceof Float) {
      price = ((Float)arg).floatValue();
      System.out.println("PriceObserver: Price changed to " +
                         price);
    } else {
      System.out.println("PriceObserver: Some other change to
                          subject!");
    }
  }
```

Observer Pattern

```java
// Test program for ConcreteSubject, NameObserver and PriceObserver
public class TestObservers {
  public static void main(String args[]) {
    // Create the Subject and Observers.
    ConcreteSubject s = new ConcreteSubject("Corn Pops", 1.29f);
    NameObserver nameObs = new NameObserver();
    PriceObserver priceObs = new PriceObserver();
    // Add those Observers!
    s.addObserver(nameObs);
    s.addObserver(priceObs);
    // Make changes to the Subject.
    s.setName("Frosted Flakes");
    s.setPrice(4.57f);
    s.setPrice(9.22f);
    s.setName("Sugar Crispies");
  }
}
```

```
ConcreteSubject created: Corn Pops at 1.29
NameObserver created: Name is null
PriceObserver created: Price is 0.0
PriceObserver: Some other change to subject!
NameObserver: Name changed to Frosted Flakes
PriceObserver: Price changed to 4.57
NameObserver: Some other change to subject!
PriceObserver: Price changed to 9.22
NameObserver: Some other change to subject!
PriceObserver: Some other change to subject!
NameObserver: Name changed to Sugar Crispies
```