Behavioral Patterns

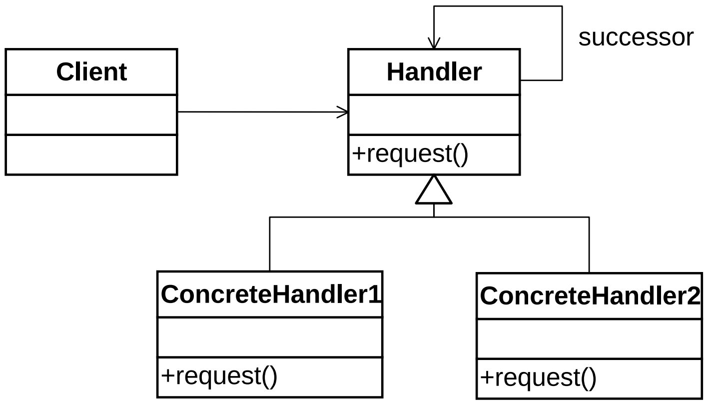
- Chain of Responsibility
- Command
- Interpreter
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Template Method
- Visitor

Behavioral patterns are concerned with algorithms and the assignment of responsibilities between objects.

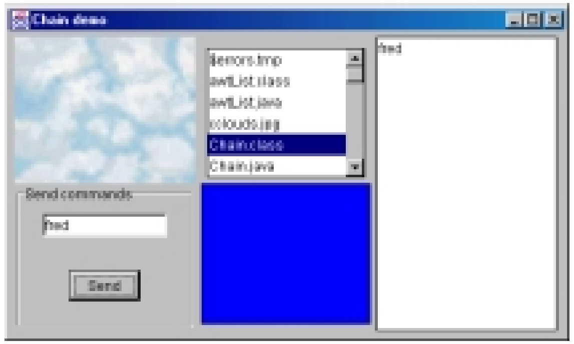
Chain of Responsibility

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.

Structure - Chain of Responsibility



Example - Chain of Responsibility



Code - Chain of Responsibility

```
public interface Chain
{
  public abstract void addChain(Chain c);
  public abstract void sendToChain(String mesg);
  public Chain getChain();
}
```

Note: Java only permits single inheritance, so we make Chain an interface and have to include a “nextChain”or successor reference in each chainable class.

Code - Chain of Responsibility

```
public class ColorImage extends JPanel
implements Chain      {

    private Chain nextChain;

//-----------------------------------
    public ColorImage() {
        super();
        setBorder(new LineBorder(Color.black));
    }

//-----------------------------------
    public void addChain(Chain c) {
        nextChain = c;
    }
```


Code - Chain of Responsibility

```
//-----------------------------------
    public void sendToChain(String mesg) {
        Color c = getColor(mesg);
        if (c != null) {
            setBackground(c);
            repaint();
        } else {
            if (nextChain != null)
                nextChain.sendToChain(mesg);
        }
    }
//-----------------------------------
    public Chain getChain() {
        return nextChain;
    }
```


Code - Chain of Responsibility

```
//-----------------------------------
    private Color getColor(String mesg) {
        String lmesg = mesg.toLowerCase();
        Color c = null;
        if (lmesg.equals("red"))
            c = Color.red;
        if (lmesg.equals("blue"))
            c = Color.blue;
        if (lmesg.equals("green"))
            c= Color.green;
        return c;
    }
}
```

Code - Chain of Responsibility

```
//set up the chain of responsibility
   sender.addChain(imager);
   imager.addChain(colorImage);
   colorImage.addChain(fileList);
   fileList.addChain(restList);
```


Applicability - Chain of Responsibility

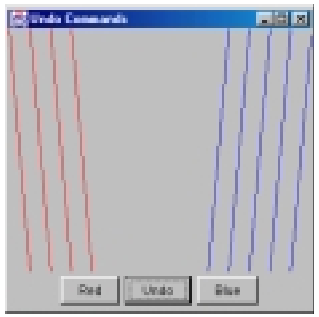
- When more than one object may handle a request, and the handler isn't known *a priori*.
- You want to issue a request to one of several objects without specifying the receiver explicitly.
- The set of objects that can handle a request should be specified dynamically.

Consequences - Chain of Responsibility

- Reduced Coupling
  - object does not know handling object
  - simplifies interconnection
- Added flexibility in assigning responsibilities to objects
- Receipt is not guaranteed.

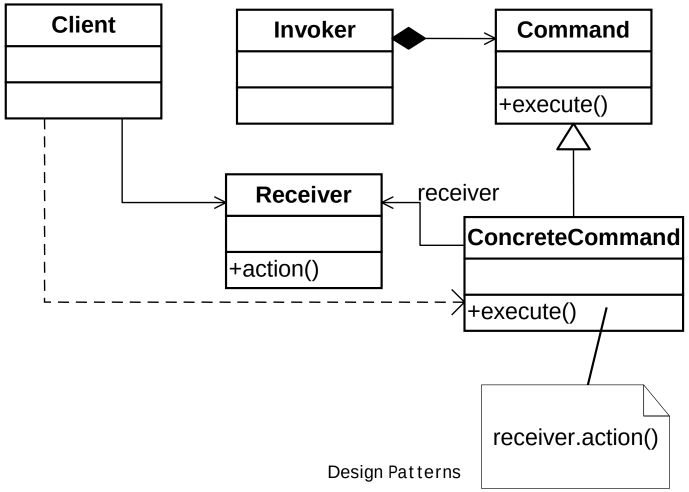
Command

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

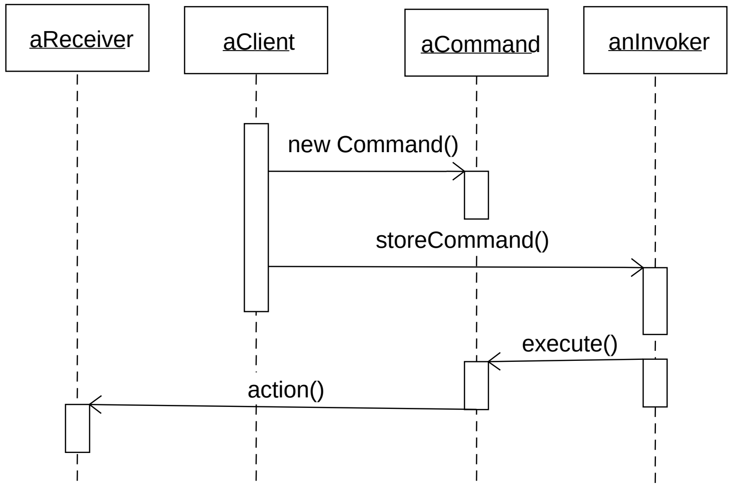


```
public interface Command
{
   public void Execute();
   public void unDo();
}
```

Structure - Command



Collaboration - Command



Applicability - Command

- when you want to parameterise objects by an action to perform.
- to specify, queue and execute requests at different times.
- to support undo.

Consequences - Command

- Decouples object that invokes operation from object that knows how to perform it
- Commands can be manipulated and extended like any other object.
- Can assemble commands into composite command (macros) using Composite pattern.
- Easy to add new commands - no change to existing classes.

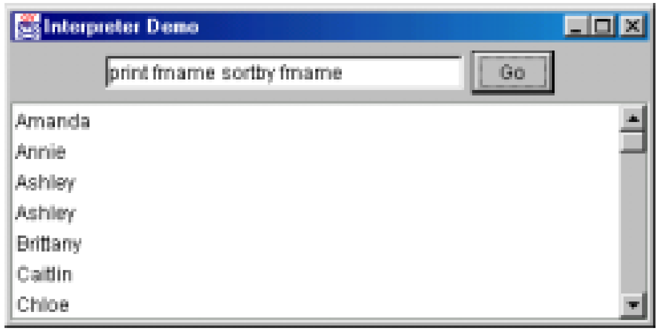
Interpreter

Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.

The pattern uses a class to represent each grammar rule. A sentence in the language is represented using these classes as an abstract syntax tree.

(see - Compiler course)

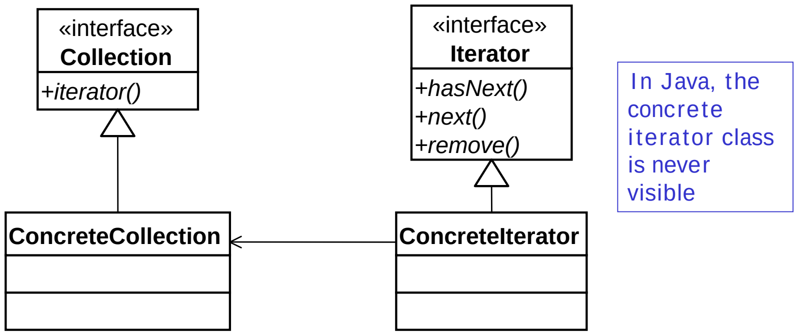
Example - Interpreter



Command line interpreter

Iterator

Provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation



Example usage - Iterator

```
public class IteratorDemo {
    Collection agg = new ArrayList();

    public IteratorDemo() {
        agg.add("one");
        agg.add("two");
        agg.add("three");
    }

    public void print() {
        Iterator I = agg.iterator();
        while (I.hasNext()) {
          System.out.println(I.next());
        }
    }
}
```

Applicability - Iterator

- to access an aggregate object's contents without exposing its internal representation.
- to support multiple traversals of aggregate objects.
- to provide a uniform interface for traversing different aggregate structures (i.e. to support polymorphic iteration.)

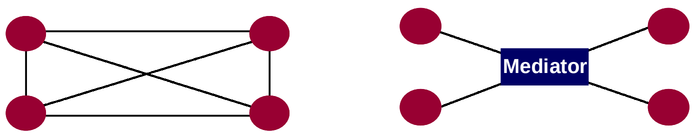
Consequences - Iterator

- supports variations in the traversal of an aggregate (e.g. ListIterator supports "previous").
- Iterators simplify the aggregate interface - no need for traversal methods in aggregate class.
- More than one traversal can be pending on an aggregate. An iterator keeps track of its own traversal state (e.g. position in list). Therefore, more than one traversal can be in progress at once.

Mediator

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.

before *after*



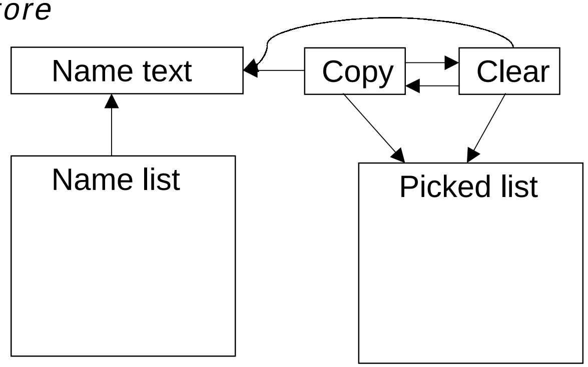
colleagues

Mediator routes requests between colleagues.

Example - Mediator

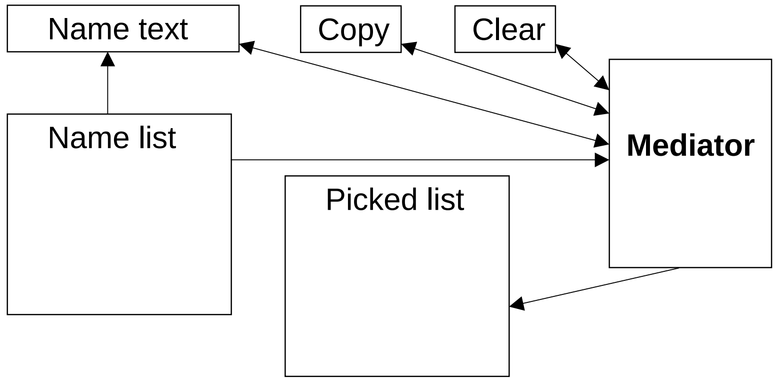
Example - Mediator

before



Example - Mediator

after



Applicability - Mediator

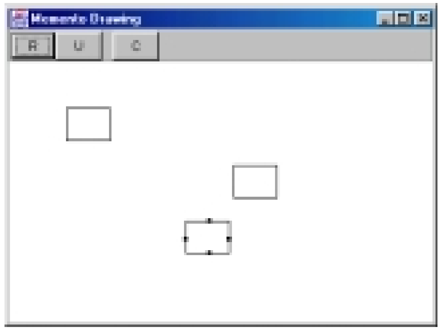
- when a set of objects communicate in well-defined but complex ways. The resulting interdependencies are unstructured and difficult to understand.
- reusing an object is difficult because it refers to and communicates with many other objects.
- a behavior that is distributed between several classes should be customizable without a lot of subclassing.

Consequences - Mediator

- It limits subclassing - only need to subclass Mediator not each Colleague.
- It decouples colleagues - can vary mediator and colleague classes independently.
- It simplifies object protocols - replaces many-to-many with one-to-many communication.
- It abstracts how objects cooperate.
- It centralizes control - can become a complex monolith that is difficult to maintain (*God class*).

Memento

Without violating encapsulation, capture and externalise an object's internal state so that the object can be restored to this state later.



Example:
remember position & size of rectangles for undo

Code - Memento

```
public class visRectangle {
    int x, y, w, h;    //package protected
    ...
}
public class Memento {
    visRectangle rect;
    private int x, y, w, h; //saved fields
    public Memento(visRectangle r) {
        rect = r; x = rect.x;  y = rect.y;
        w = rect.w;  h = rect.h;
    }
    public void restore() {
        rect.x = x;  rect.y = y;
        rect.h = h;  rect.w = w;
    }
}
```


Code - Memento

Saving state:

```
public void rememberPosition(visRectangle rect) {
    Memento m = new Memento(rect);
    undoList.addElement(m);
}
```

Restoring state:

```
private void undo() {
   Memento m = (Memento) undoList.lastElement();
   undoList.removeElement(m);
   m.restore();     //and restore old position
}
```


Applicability - Memento

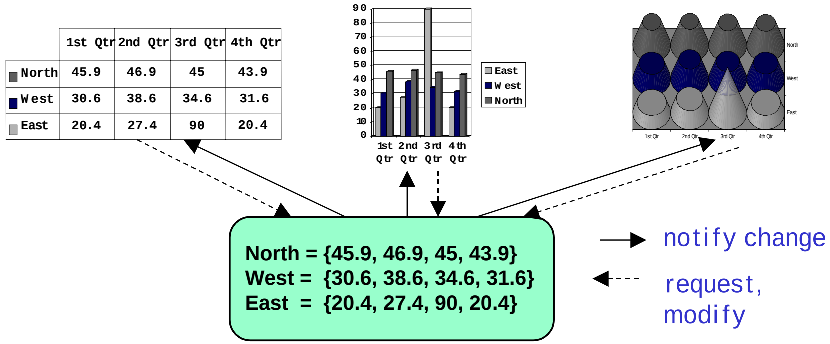
- Use when a snapshot of (some portion of) an object's state must be saved so that it can be restored to that state later, *and*
- a direct interface to obtaining the state would expose implementation details and break the object's encapsulation

Consequences - Memento

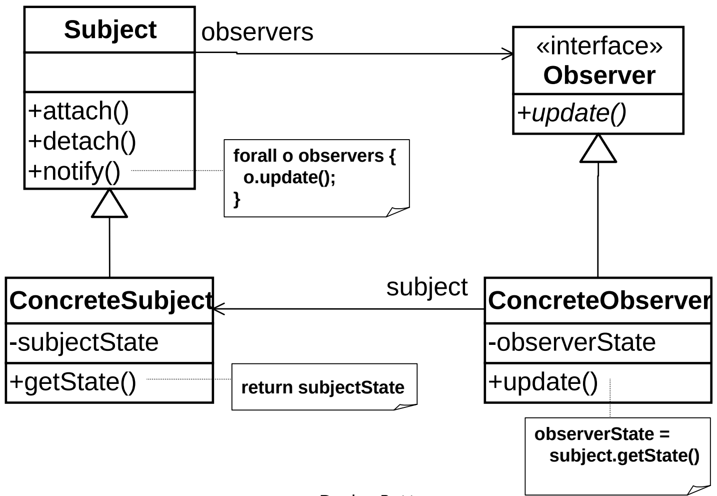
- Preserving encapsulation boundaries - avoids exposing originating object's state.
- Simplifies originating object - removes storage management burden.
- Using mementos can be expensive - creation, copying and restoring can have high overhead.
- Can be difficult in some languages to ensure that only the originating object can access memento's state.
- Hidden cost in managing mementos - caretaking.

Observer

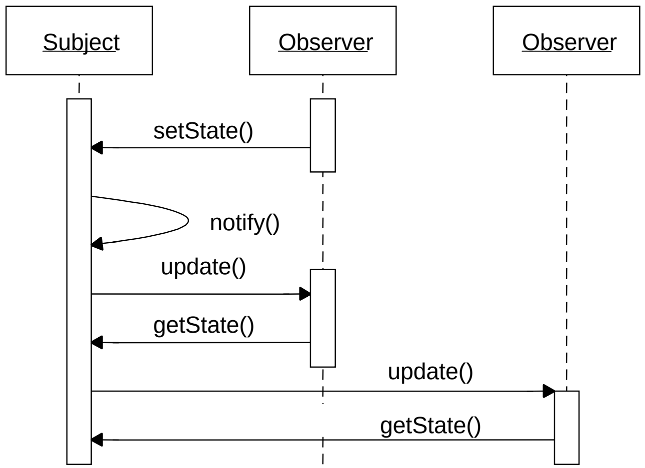
Define a one-to-many dependency between objects so that when one object changes, all its dependents are notified and updated automatically.



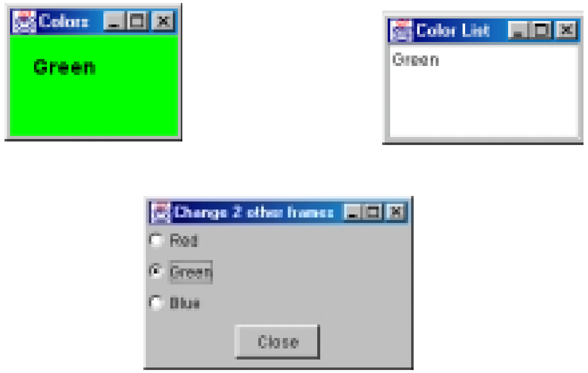
Structure - Observer



Collaborations - Observer



Example - Observer



Example - Observer

registerInterest() is the attach() pattern method

```
public interface Subject {
    public void registerInterest(Observer obs);
}
```

sendNotify() is the update() pattern method
the updated state is passed as a parameter to
this method

```
public interface Observer {
    public void sendNotify(String s);
}
```


Example - Observer

```
public class Watch2Windows extends JxFrame
implements ActionListener, ItemListener, Subject {
  private Collection observers;
  ...
  private void notifyObservers(JRadioButton rad) {
   String color = rad.getText();
   for (Iterator I=observers.iterator(); I.hasNext();)
       ((Observer)I.next()).sendNotify(color);
  }

  public void registerInterest(Observer obs) {
   observers.add(obs);
  }
  ...
}
```


Example - Observer

```
public class ListFrame extends JFrame
implements Observer {
    ...
    public void sendNotify(String s) {
        listData.addElement(s);
    }
}
```

Applicability - Observer

- When an abstraction has two or more inter-dependent aspects. Encapsulating these aspects in different objects lets you vary and reuse them independently.
- When a change to one object requires changing others, and you do not know how many objects need to be changed.
- To decouple subject from observers.

Consequences - Observer

- Abstracts coupling between Subject and Observer.
- Support for broadcast communication.
- Unexpected updates - or spurious updates to observers.

State

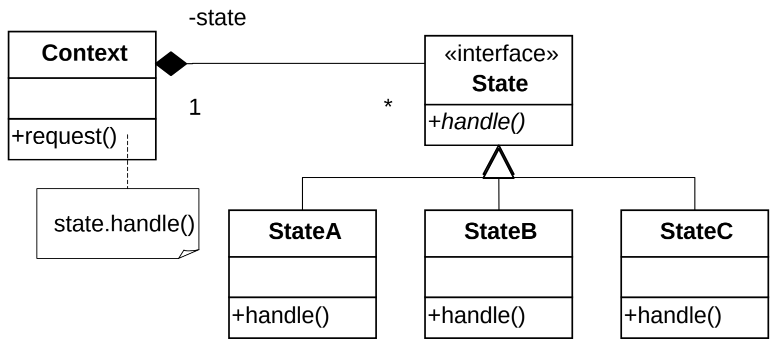
Allow an object to alter its behaviour when its internal state changes. The object will appear to change its class.

Objects often have internal modes or states with different behaviour (responses to messages) in each mode.

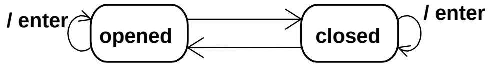
State pattern introduces explicit subclasses - a different subclass for each mode.

In this pattern, the choice between responses to methods is handled by polymorphism of state subclasses, not by the programmer.

Structure - State



Example - State




```
public class Door {
   private static final int Opened = 1;
   private static final int Closed = 2;
   int state = Opened;

   public void open()  { state = Opened; }
   public void close() { state = Closed; }

   public boolean enter() {
      if (state == Opened)
          return true;
     else if (state == Closed)
       return false;
     else
        throw new Error();
   }
}
```

before

Example - State

```
interface State { boolean enter(); }
class Opened implements State {
   public boolean enter() {return true;}}
class Closed implements State {
   public boolean enter() {return false;}}
public class Door {
   private Opened opened = new Opened();
   private Closed closed = new Closed();
   State state = opened;
   public void open()  { state = opened; }
   public void close() { state = closed; }
   public boolean enter() {
      return state.enter();
   }
}
```

after

Applicability - State

- An object’s behaviour depends on its state and it must change its behaviour at runtime depending on that state.
- Methods have large multipart conditional statements that depend on the object’s state. Often, several operations will contain this same conditional structure. The State pattern puts each branch of the conditional into a separate class. Each state of the original object is now a separate object and can vary independently from other state objects.

Consequences - State

- Localises state-specific behaviour and partitions behaviour for different states.
- It makes state transitions explicit.
- State objects can be shared - a single instance can be ensured by using the Singleton pattern.