CS 635 Advanced Object-Oriented Design & Programming
Spring Semester, 2011
Doc 16 Memento & Chain of Responsibility
April 14, 2011

References

Wikipedia, http://en.wikipedia.org/wiki/Chain_of_responsibility_pattern

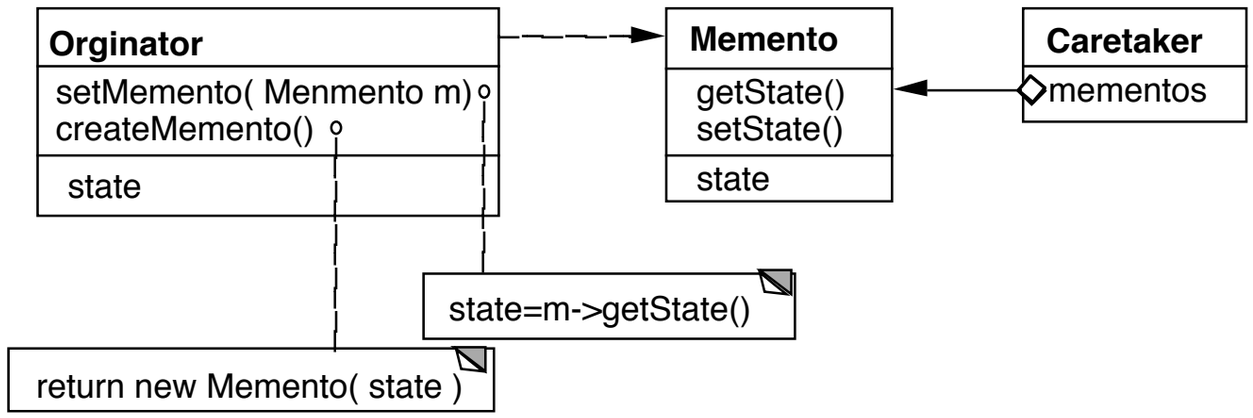
Design Patterns: Elements of Reusable Object-Oriented Software, Gamma, Helm, Johnson, Vlissides, Addison-Wesley, 1995, pp. 283-292, 223-232

The Design Patterns Smalltalk Companion, Alpert, Brown, Woolf, 1998, pp. 47-62

Memento

Store an object's internal state, so the object can be restored to this state later without violating encapsulation

undo, rollbacks



Only originator:

Can access Memento’s get/set state methods

Create Memento

Example

```
package Examples;
class Memento{
        private Hashtable savedState = new Hashtable();

        protected Memento() {}; //Give some protection

        protected void setState( String stateName, Object stateValue ) {
                savedState.put( stateName, stateValue );
        }

        protected Object getState( String stateName) {
                return savedState.get( stateName);
        }

        protected Object getState(String stateName, Object defaultValue ) {
                if ( savedState.containsKey( stateName ) )
                        return savedState.get( stateName);
                else
                        return defaultValue;
        }
}
```

Sample Originator

```
package Examples;
class ComplexObject {
        private String name;
        private int someData;
        private Vector objectAsState = new Vector();

        public Memento createMemento() {
                Memento currentState = new Memento();
                currentState.setState( "name", name );
                currentState.setState( "someData", new Integer(someData) );
                currentState.setState( "objectAsState", objectAsState.clone() );
                return currentState;
        }

        public void restoreState( Memento oldState) {
                name = (String) oldState.getState( "name", name );
                objectAsState = (Vector) oldState.getState( "objectAsState" );
                Integer data = (Integer) oldState.getState( "someData");
                someData = data.intValue();
        }
```

Why not let the Originator save its old state?

```
class ComplexObject {
        private String name;
        private int someData;
        private Vector objectAsState = new Vector();
        private Stack history;

        public createMemento() {
                Memento currentState = new Memento();
                currentState.setState( "name", name );
                currentState.setState( "someData", new Integer(someData) );
                currentState.setState( "objectAsState", objectAsState.clone() );
                history.push(currentState);
        }

        public void restoreState() {
                Memento oldState = history.pop();
                name = (String) oldState.getState( "name", name );
                objectAsState = (Vector) oldState.getState( "objectAsState" );
                Integer data = (Integer) oldState.getState( "someData");
                someData = data.intValue();
        }
```

Some Consequences

Expensive

Narrow & Wide interfaces - Keep data hidden

```
Class Memento {
public:
        virtual ~Memento();
private:
        friend class Originator;
        Memento();
        void setState(State*);
        State* GetState();
```

```
class Originator {
        private String state;

 private class Memento {
        private String state;
        public Memento(String stateToSave)
                { state = stateToSave; }
public String getState() { return state; }
        }

        public Object memento()
                { return new Memento(state);}
```

Using Clone to Save State

```
interface Memento extends Cloneable { }

class ComplexObject implements Memento {
        private String name;
        private int someData;

        public Memento createMemento() {
                Memento myState = null;
                try {
                        myState =  (Memento) this.clone();
                }
                catch (CloneNotSupportedException notReachable) {
                }
                return myState;
        }

        public void  restoreState( Memento savedState) {
                ComplexObject myNewState = (ComplexObject)savedState;
                name = myNewState.name;
                someData = myNewState.someData;
        }
}
```

What if Protocol

When there are complex validations or
performing operations that make it difficult to restore later

Make a copy of the Originator

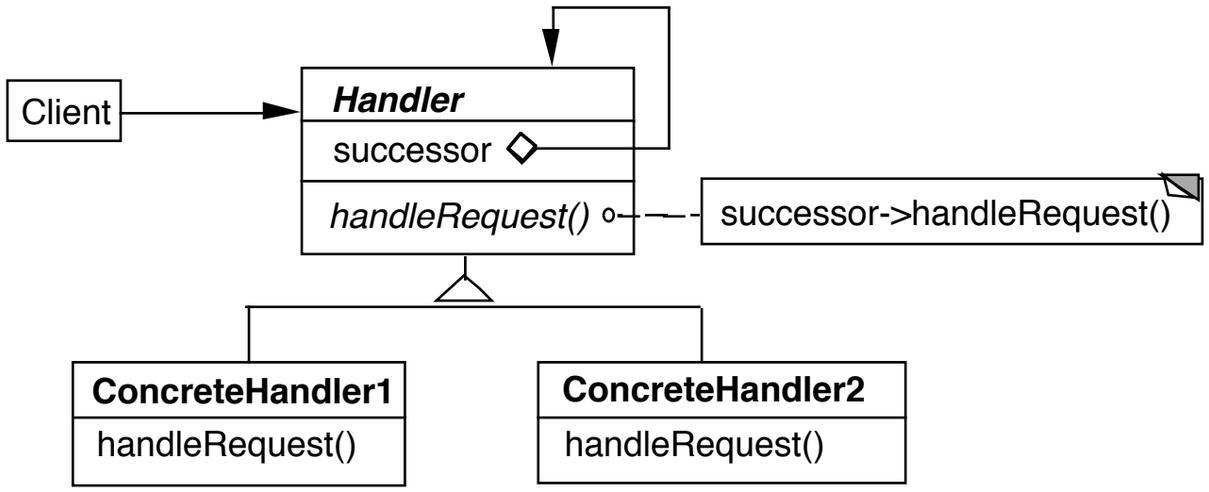
Perform operations on the copy

Check if operations invalidate the internal state of copy

If so discard the copy & raise an exception

Else perform the operations on the Originator

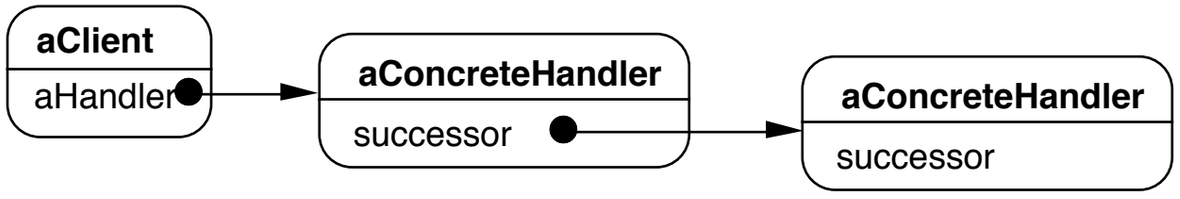
Chain of Responsibility



Dynamically create chain of handlers

Multiple handlers may be able to handle a request

Only one handler actually handles the request



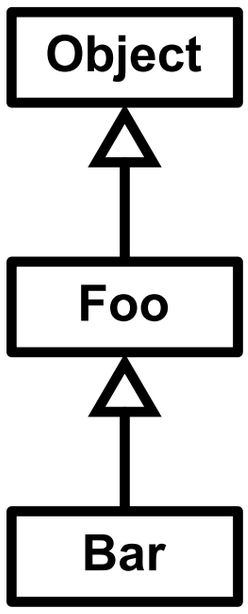
Consequences

Reduced coupling

Added flexibility in assigning responsibilities to objects

Not guaranteed that request will be handled

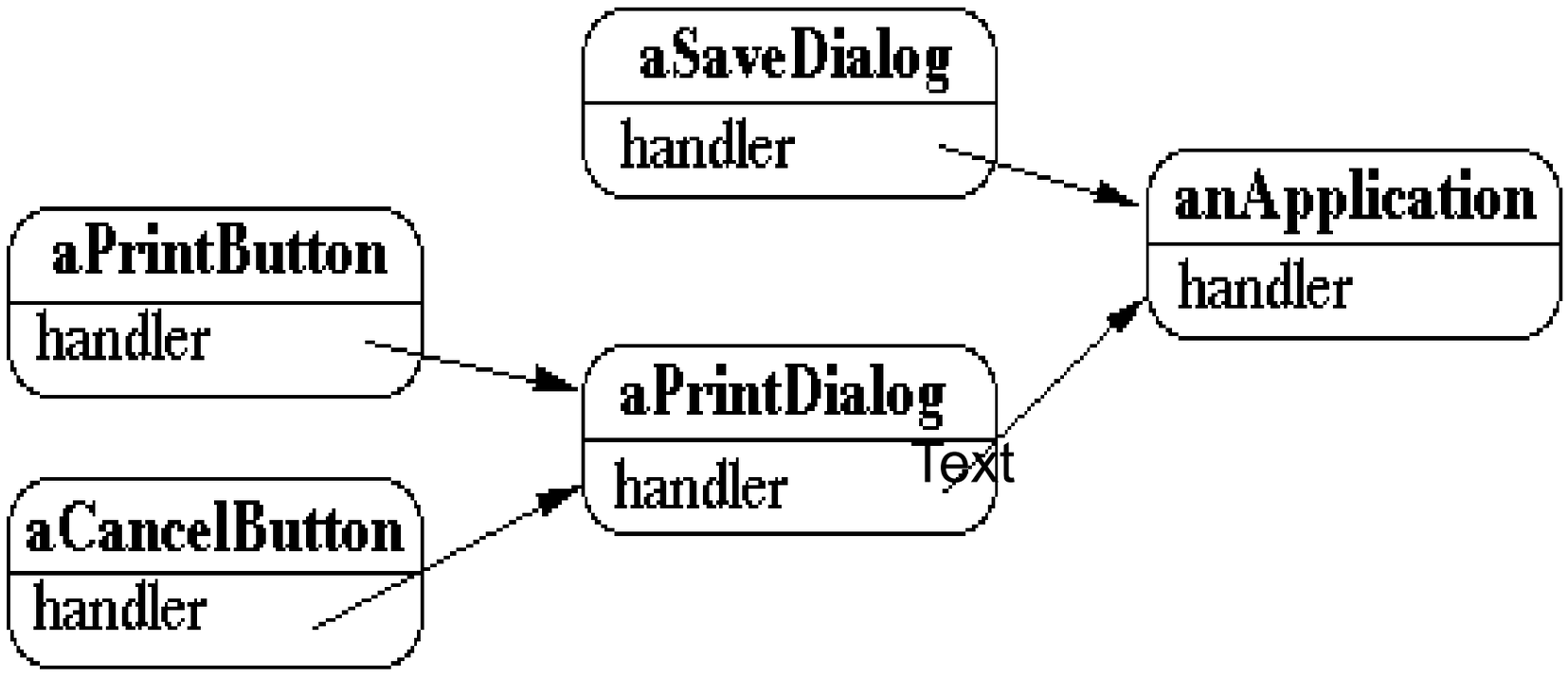
Finding Methods




```
test = new Bar();
test.toString();
```

Context Help System

User clicks on component for help



Tree of handlers

From specific to general

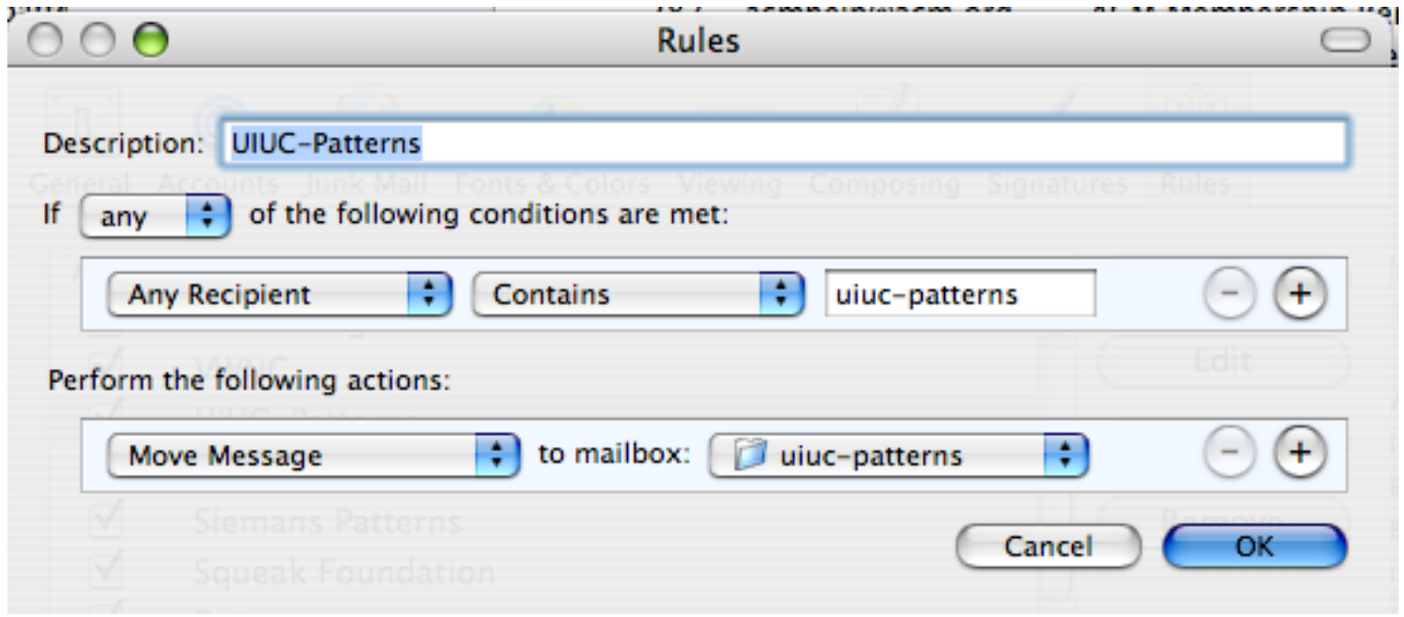
Email Filters in Mail Client

User creates a set of rules

- delete
- move
- modify

Chain the rules

First rule that applies handles the mail



Other Examples

Java 1.0 AWT action(Event)

javax.servlet.Filter

http://tomcat.apache.org/tomcat-4.1-doc/servletapi/javax/servlet/Filter.html

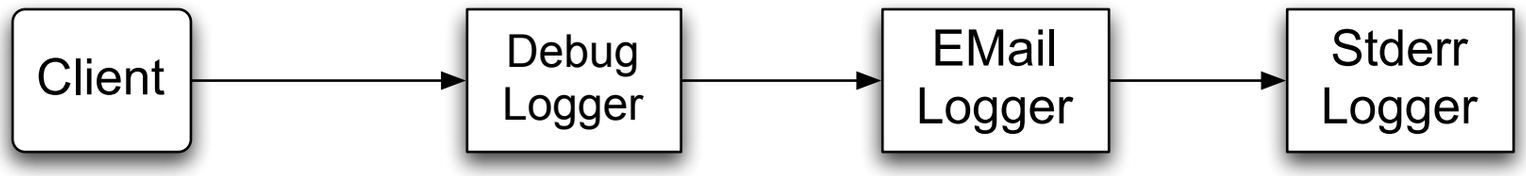
Microsoft Windows global keyboard events

http://www.javaworld.com/javaworld/jw-08-2004/jw-0816-chain.html

Apache Commons Chain

http://commons.apache.org/chain/

Logger Example




```
class ChainOfResponsibilityExample {
   public static void main(String[] args) {
      // building the chain of responsibility
      Logger l = new DebugLogger(Logger.DEBUG).setNext(
                  new EMailLogger(Logger.ERR).setNext(
                   new StderrLogger(Logger.NOTICE) ) );

      l.message("Entering function x.", Logger.DEBUG);    // handled by DebugLogger
      l.message("Step1 completed.", Logger.NOTICE);       // handled by Debug- and
StderrLogger
      l.message("An error has occurred.", Logger.ERR);    // handled by all three Logger
   }
}
```

First Attempt

```
abstract class Logger {
   public static int ERR = 3;
   public static int NOTICE = 5;
   public static int DEBUG = 7;
   protected int mask;

   protected Logger next;
   public Logger setNext(Logger l) {
      next = l;
      return this; }

   abstract public void message(String msg, int priority);
}

class DebugLogger extends Logger {
   public DebugLogger(int mask) {
      this.mask = mask; }

   public void message(String msg, int priority) {
      if (priority <= mask) debug log here
      if (next != null) next.message(msg, priority);
   }
}
```

```
class EMailLogger extends Logger {
   public EMailLogger(int mask) { this.mask = mask; }

   public void message(String msg, int priority) {
      if (priority <= mask) send email here;
      if (next != null) next.message(msg, priority);
   }
```

Improved Logger

```
abstract class Logger {
   public static int ERR = 3;
   public static int NOTICE = 5;
   public static int DEBUG = 7;
   protected int mask;

   protected Logger next;
   public Logger setNext(Logger l) {
          next = l;
         return this; }

   public void message(String msg, int priority) {
      if (priority <= mask) log(msg);
      if (next != null) next.message(msg, priority);
   }

   abstract void log(String message);
}

class StderrLogger extends Logger {
   public StderrLogger(int mask) { this.mask = mask; }

   void message(String msg, int priority) { send to err}
}
```

```
class EMailLogger extends Logger {
   public EMailLogger(int mask) { this.mask = mask; }

    void message(String msg, int priority) { email here}
   }
}

class DebugLogger extends Logger {
   public DebugLogger(int mask) { this.mask = mask; }

   void message(String msg, int priority) { debug stuff}
   }
}
```

Is this the Chain of Responsibility?

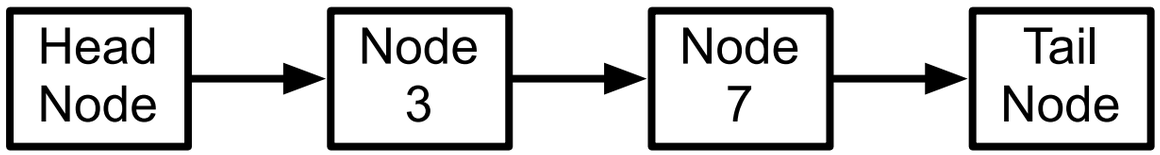
Object-Oriented Recursion

A method polymorphically sends its message to a different receiver

Eventually a method is called that performs the task

The recursion then unwinds back to the original message send

Linked List toString



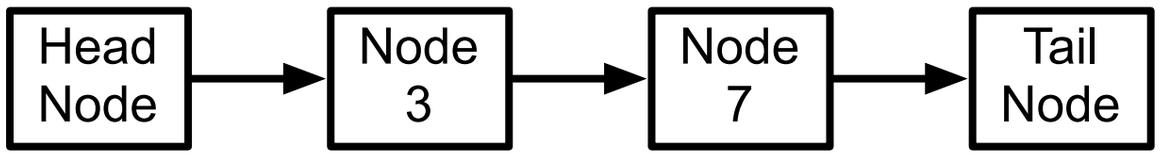
(3 7)

```
class HeadNode {
        public String toString() {
                return "(" + next.toString();
        }
}
```

```
class TailNode {
        public String toString() {
                return " )";
        }
}
```

```
class Node {
        public String toString() {
                return " " + element + next.toString();
        }
}
```

Linked List add




```
class HeadNode {
        public void add(int value) {
                next.add(value);
        }
}
```

```
class TailNode {
        public void add(int value) {
                prependNode(value);
        }
}
```

```
class Node {
        public void add(int value) {
                if (element > value)
                        prependNode(value);
                else
                        next.add(value);
        }
}
```

OO Recursion

Decorator

Chain of Responsibility