

CS 635 Advanced Object-Oriented Design & Programming
Spring Semester, 2019
Doc 7 Command Processor, Memento, Interpreter, Composite
Sep 19, 2019

Copyright ©, All rights reserved. 2019 SDSU & Roger Whitney,
5500 Campanile Drive, San Diego, CA 92182-7700 USA.
OpenContent (<http://www.opencontent.org/opl.shtml>) license
defines the copyright on this document.

Command Processor Pattern

Command Processor manages the command objects

The command processor:

- Contains all command objects

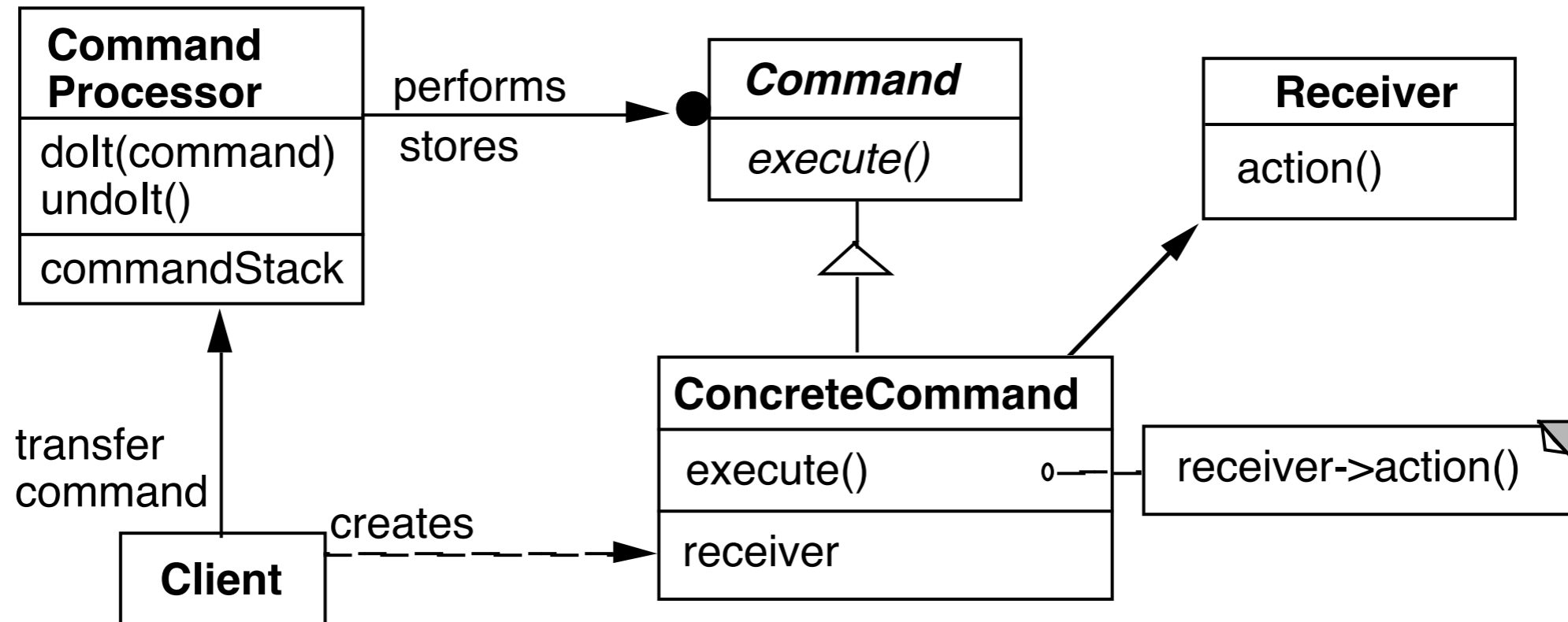
- Schedules the execution of commands

- May store the commands for later unto

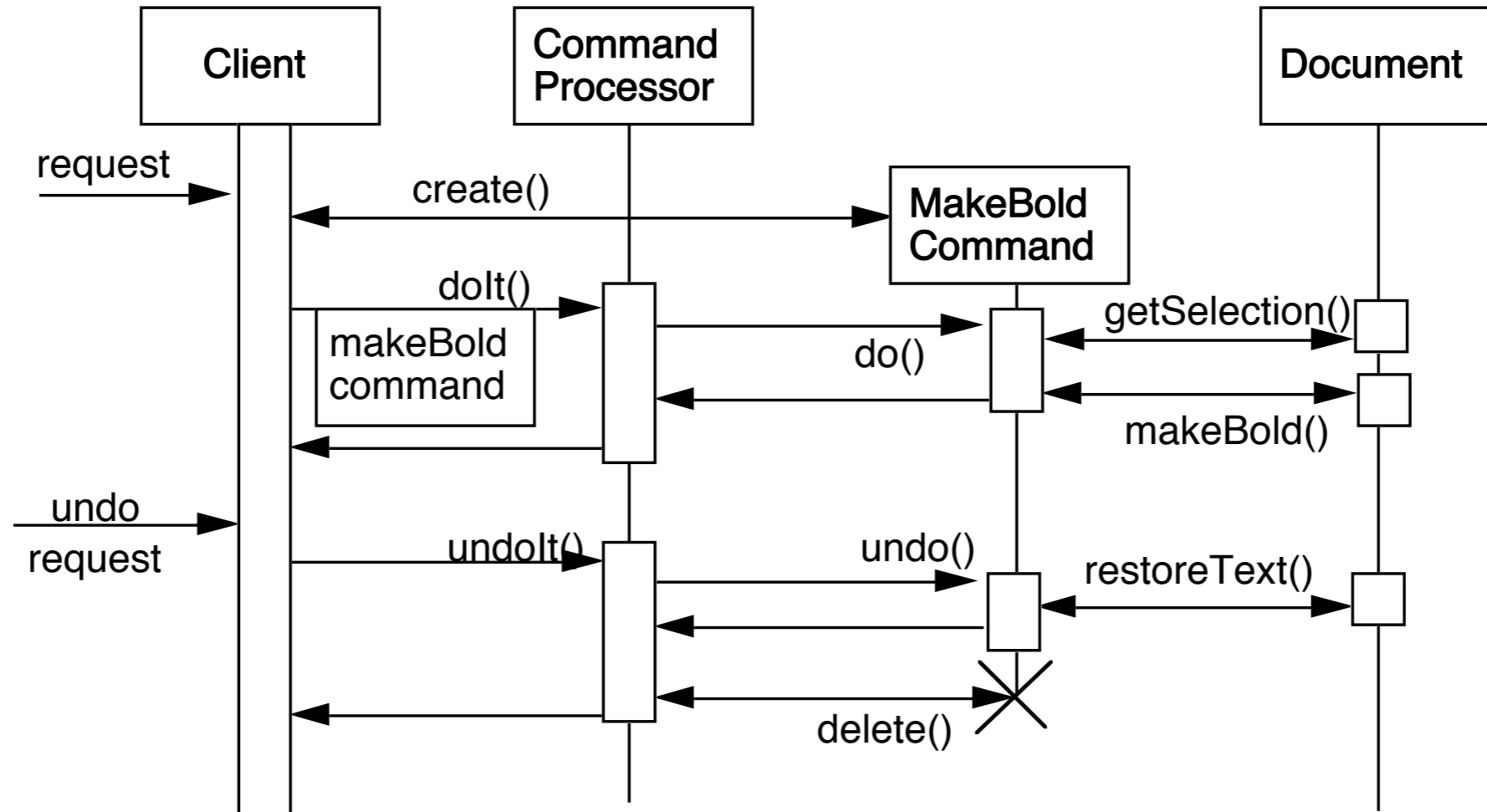
- May log the sequence of commands for testing purposes

- Uses singleton to insure only one instance

Structure



Dynamics



Benefits

Flexibility in the way requests are activated

Different user interface elements can generate the same kind of command object

Allows the user to configure commands performed by a user interface element

Flexibility in the number and functionality of requests

Adding new commands and providing for a macro language comes easy

Programming execution-related services

Commands can be stored for later replay

Commands can be logged

Commands can be rolled back

Testability at application level

Concurrency

Allows for the execution of commands in separate threads

Liabilities

Efficiency loss

Potential for an excessive number of command classes

Try reducing the number of command classes by:

- Grouping commands around abstractions

- Unifying simple commands classes by passing the receiver object as a parameter

Complexity

How do commands get additional parameters they need?

Undo

Undo

Some examples

Counter

```
counter.increase();    //increase counter by 1  
counter.decrease();   //decrease counter by 1
```


Undo

Some examples

Text editing

Replace "Should" with "Could" at start of 3rd sentence in 5 paragraph

Undo - Some Issues

Redo

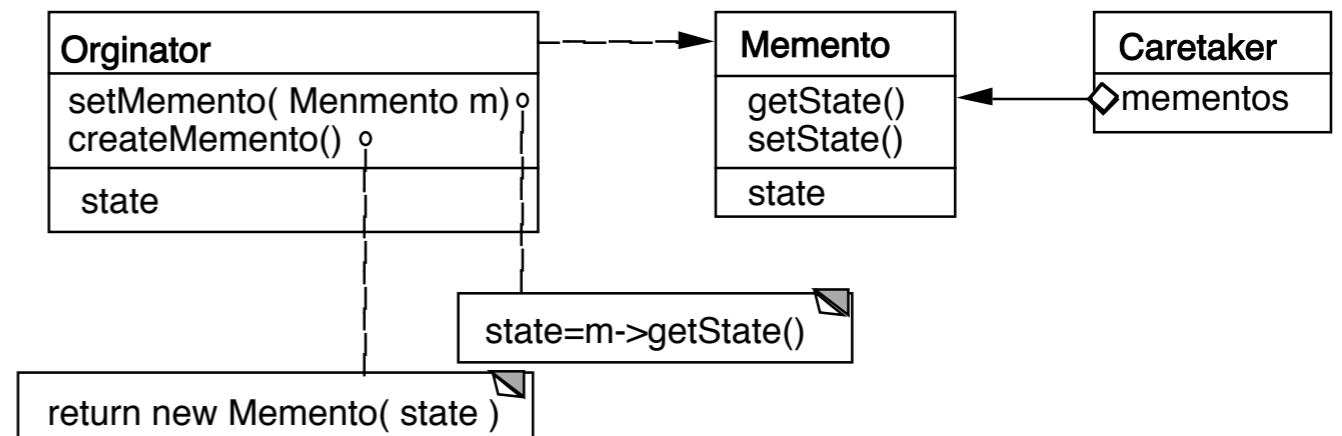
Multiple undo

Memento

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
Space

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;
```

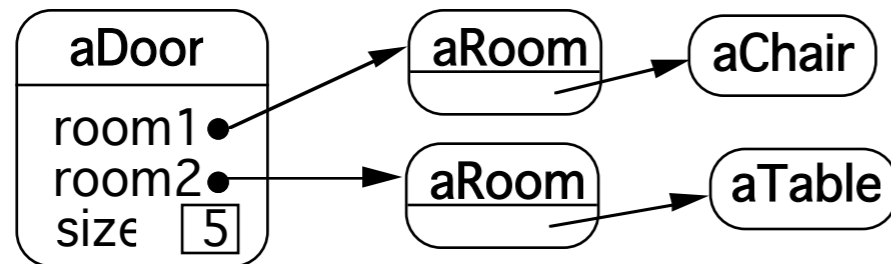
```
    }
```

```
}
```

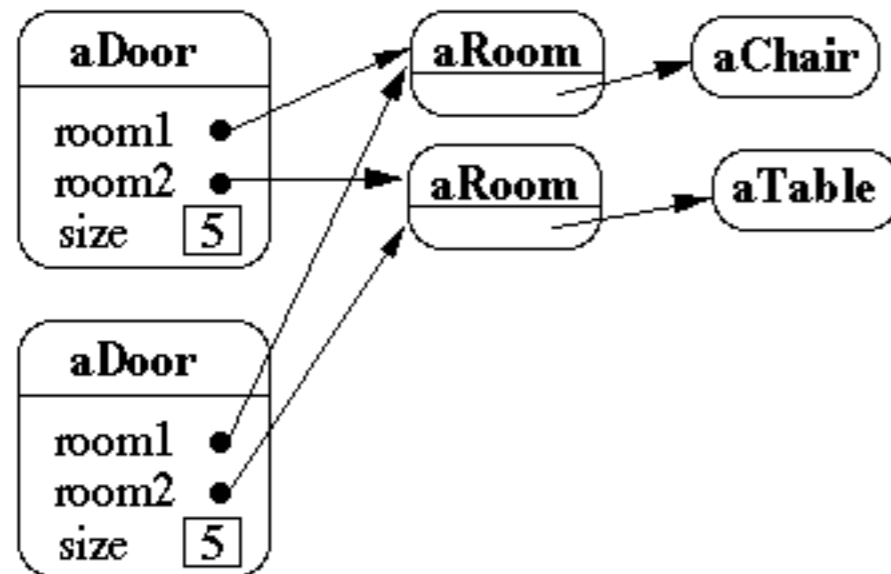
Copying Issues

Shallow Copy Verse Deep Copy

Original Objects

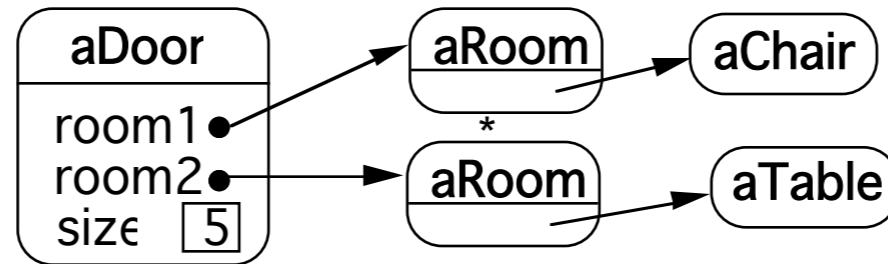


Shallow Copy

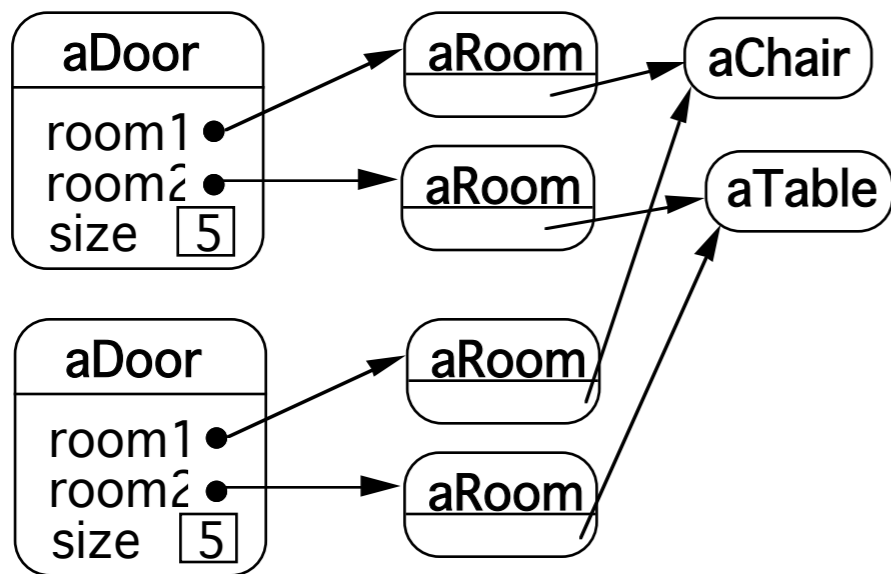


Shallow Copy Verse Deep Copy

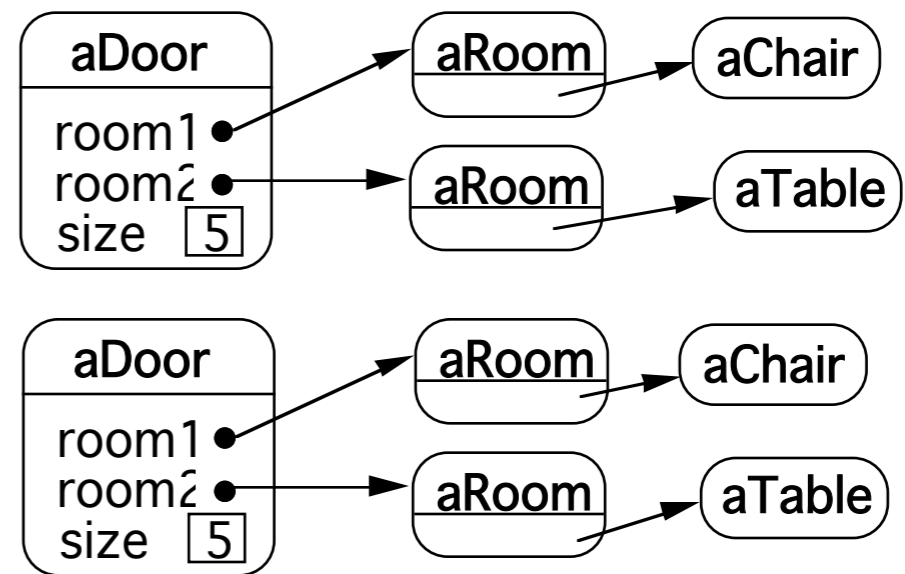
Original Objects



Deep Copy



Deeper Copy



Cloning Issues - C++ Copy Constructors

```
class Door {
public:
    Door();
    Door( const Door&);
    virtual Door* clone() const;

    virtual void Initialize( Room*, Room* );
    // stuff not shown
private:
    Room* room1;
    Room* room2;
}

Door::Door ( const Door& other ) //Copy constructor {
    room1 = other.room1;
    room2 = other.room2;
}

Door* Door::clone() const {
    return new Door( *this );
}
```

Cloning Issues - Java Clone

Shallow Copy

```
class Door implements Cloneable {  
    private Room room1;  
    private Room room2;  
  
    public Object clone() throws CloneNotSupportedException {  
        return super.clone();  
    }  
}
```

Deep Copy

```
public class Door implements Cloneable {  
    private Room room1;  
    private Room room2;  
  
    public Object clone() throws CloneNotSupportedException {  
        Door thisCloned =(Door) super.clone();  
        thisCloned.room1 = (Room)room1.clone();  
        thisCloned.room2 = (Room)room2.clone();  
        return thisCloned;  
    }  
}
```

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

Memento & Functional Programming

Immutable data

Data that can not change

Functional languages have primarily immutable data

If data can not change

Don't need memento pattern

Datomic

Database system where all data is immutable

Transactions become easy

Read and writes become independent

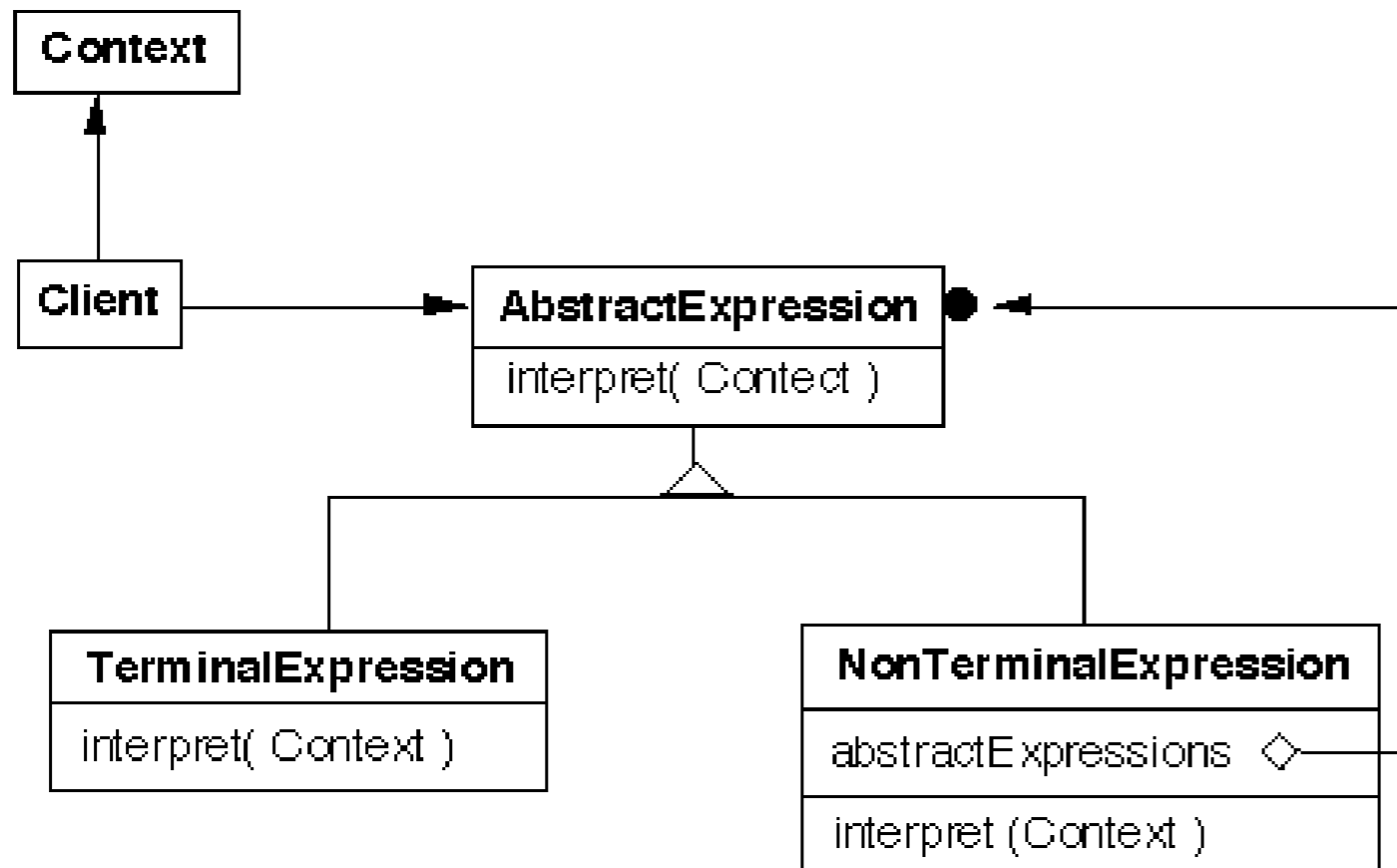
Historical data,
role backs are easy

Auditability

Interpreter

Interpreter

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



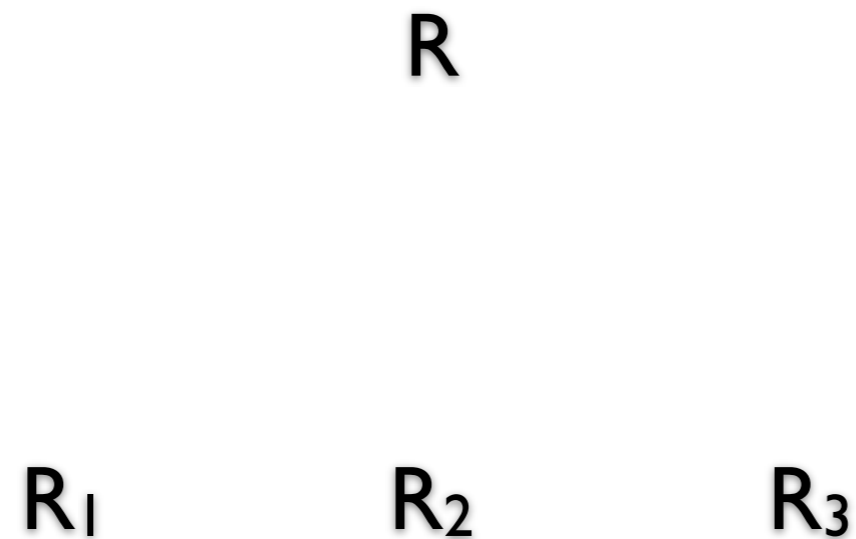
Grammar & Classes

Given a language defined by a grammar like:

$$R ::= R_1 R_2 R_3$$

you create a class for each rule

The classes can be used to construct a tree that represents elements of the language



Example - Boolean Expressions

BooleanExpression ::=

Variable	
Constant	
Or	
And	
Not	
BooleanExpression	

And ::= '(' BooleanExpression 'and' BooleanExpression ')'

Or ::= '(' BooleanExpression 'or' BooleanExpression ')'

Not ::= 'not' BooleanExpression

Constant ::= 'true' | 'false'

Variable ::= String

Sample Expression

((true or x) or (w and x))

or

Evaluate with

x = true

w = false

or

and

true

x

w

w

Sample Classes

```
public interface BooleanExpression{  
    public boolean evaluate( Context values );  
    public String toString();  
}
```

And

```
public class And implements BooleanExpression {
    private BooleanExpression leftOperand;
    private BooleanExpression rightOperand;

    public And( BooleanExpression leftOperand, BooleanExpression rightOperand) {
        this.leftOperand = leftOperand;
        this.rightOperand = rightOperand;
    }

    public boolean evaluate( Context values ) {
        return leftOperand.evaluate( values ) && rightOperand.evaluate( values );
    }

    public String toString(){
        return "(" + leftOperand.toString() + " and " + rightOperand.toString() + ")";
    }
}
```

Constant

```
public class Constant implements BooleanExpression {
    private boolean value;
    private static Constant True = new Constant( true );
    private static Constant False = new Constant( false );

    public static Constant getTrue() { return True; }

    public static Constant getFalse(){ return False; }

    private Constant( boolean value) { this.value = value; }

    public boolean evaluate( Context values ) { return value; }

    public String toString() {
        return String.valueOf( value );
    }
}
```


Variable

```
public class Variable implements BooleanExpression {  
  
    private String name;  
  
    private Variable( String name ) {  
        this.name = name;  
    }  
  
    public boolean evaluate( Context values ) {  
        return values.getValue( name );  
    }  
  
    public String toString() { return name; }  
}
```

Context

```
public class Context {  
    Hashtable<String,Boolean> values = new Hashtable<String,Boolean>();  
  
    public boolean getValue( String variableName ) {  
        return values.get( variableName );  
    }  
  
    public void setValue( String variableName, boolean value ) {  
        values.put( variableName, value );  
    }  
}
```

((true or x) or (w and x))

```
public class Test {  
    public static void main( String args[] ) throws Exception {  
        BooleanExpression left =  
            new Or( Constant.getTrue(), new Variable( "x" ) );  
        BooleanExpression right =  
            new And( new Variable( "w" ), new Variable( "x" ) );  
  
        BooleanExpression all = new Or( left, right );  
  
        System.out.println( all );  
        Context values = new Context();  
        values.setValue( "x", true );  
        values.setValue( "w", false );  
  
        System.out.println( all.evaluate( values ) );  
    }  
}
```

Consequences

It's easy to change and extend the grammar

Implementing the grammar is easy

Complex grammars are hard to maintain

Use JavaCC or SmaCC instead

Adding new ways to interpret expressions

The visitor pattern is useful here

Complicates design when a language is simple

Supports combinations of elements better than implicit language

Implementation

The pattern does not talk about parsing!

Flyweight

If terminal symbols are repeated many times using the Flyweight pattern can reduce space usage

Composite

Abstract syntax tree is an instance of the composite

Iterator

Can be used to traverse the structure

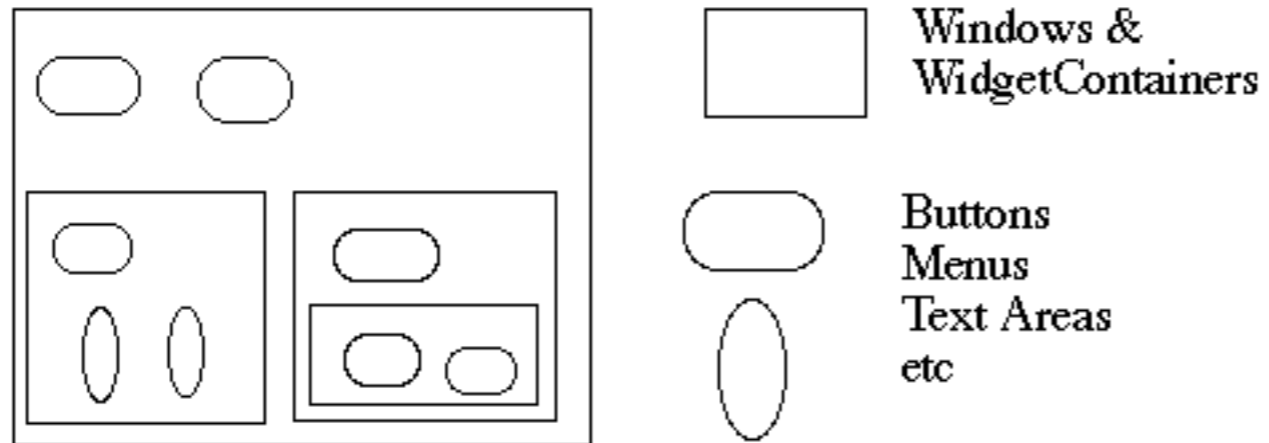
Visitor

Can be used to place behavior in one class

Composite

Composite Motivation

Application Window



How does the window hold and deal with the different items it has to manage?

Widgets are different that WidgetContainers

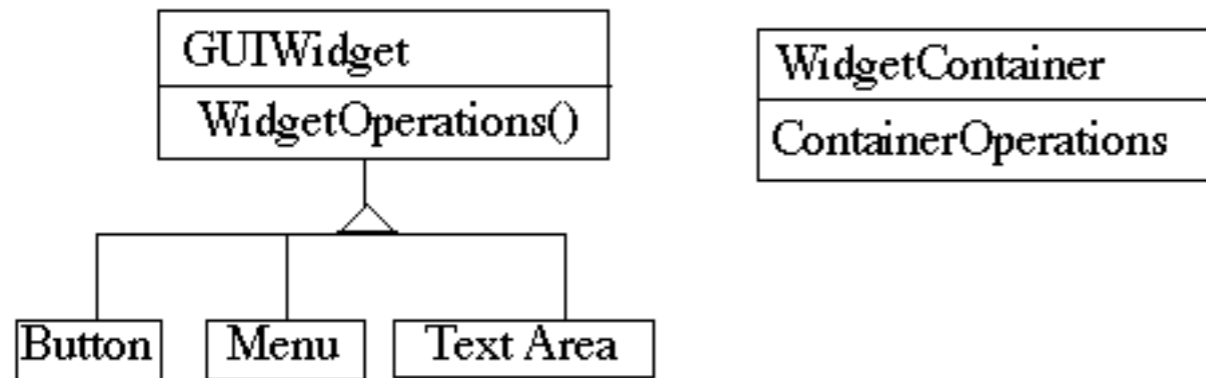
Bad News

```
class Window {
    Buttons[] myButtons;
    Menus[] myMenus;
    TextAreas[] myTextAreas;
    WidgetContainer[] myContainers;

    public void update() {
        if ( myButtons != null )
            for ( int k = 0; k < myButtons.length(); k++ )
                myButtons[k].refresh();
        if ( myMenus != null )
            for ( int k = 0; k < myMenus.length(); k++ )
                myMenus[k].display();
        if ( myTextAreas != null )
            for ( int k = 0; k < myButtons.length(); k++ )
                myTextAreas[k].refresh();
        if ( myContainers != null )
            for ( int k = 0; k < myContainers.length(); k++ )
                myContainers[k].updateElements();
        etc.
    }

    public void fooOperation(){
        if (myButtons != null)
            etc.
    }
}
```

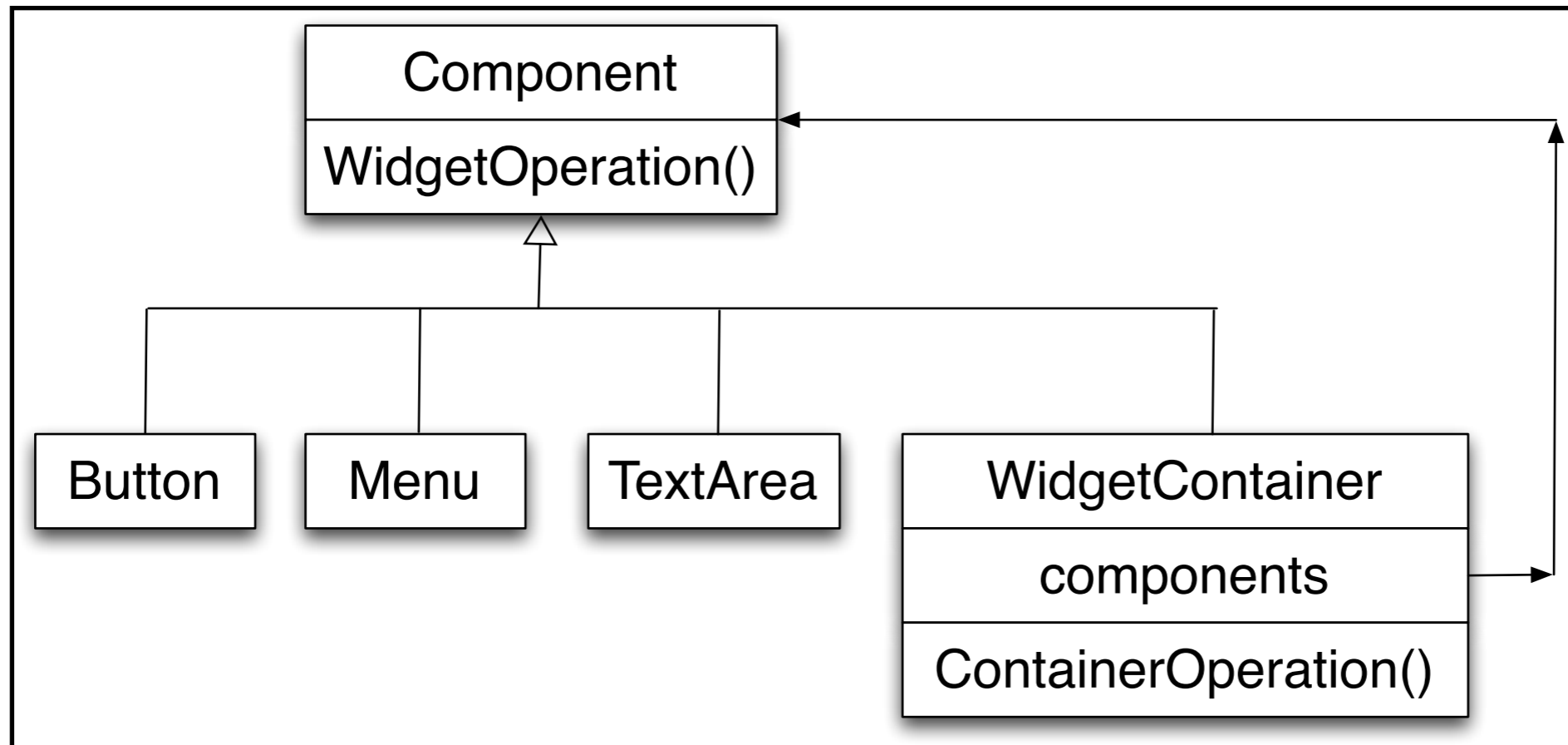

An Improvement



```
class Window {
    GUIWidgets[] myWidgets;
    WidgetContainer[] myContainers;

    public void update(){
        if ( myWidgets != null )
            for ( int k = 0; k < myWidgets.length(); k++ )
                myWidgets[k].update();
        if ( myContainers != null )
            for ( int k = 0; k < myContainers.length(); k++ )
                myContainers[k].updateElements();
        etc.
    }
}
```

Composite Pattern



Composite Pattern

Component implements default behavior for widgets when possible

Button, Menu, etc overrides Component methods when needed

WidgetContainer will have to overrides all widgetOperations

```
class WidgetContainer {
    Component[] myComponents;

    public void update() {
        if ( myComponents != null )
            for ( int k = 0; k < myComponents.length(); k++ )
                myComponents[k].update();
    }
}
```

Issue - WidgetContainer Operations

Should the WidgetContainer operations be declared in Component?

Pro - Transparency

Declaring them in the Component gives all subclasses the same interface

All subclasses can be treated alike. (?)

Con - Safety

Declaring them in WidgetContainer is safer

Adding or removing widgets to non-WidgetContainers is an error

One out is to check the type of the object before using a WidgetContainer operation

Issue - Parent References

```
class WidgetContainer
{
    Component[] myComponents;

    public void update() {
        if ( myComponents != null )
            for ( int k = 0; k < myComponents.length(); k++ )
                myComponents[k].update();
    }

    public add( Component aComponent ) {
        myComponents.append( aComponent );
        aComponent.setParent( this );
    }
}
```

```
class Button extends Component {
    private Component parent;
    public void setParent( Component myParent) {
        parent = myParent;
    }
}
```

etc.

More Issues

Should Component implement a list of Components?

The button etc. will have a useless data member

Child ordering is important in some cases

Who should delete components?

Applicability

Use Composite pattern when you want

To represent part-whole hierarchies of objects

Clients to be able to ignore the difference between compositions of objects and individual objects