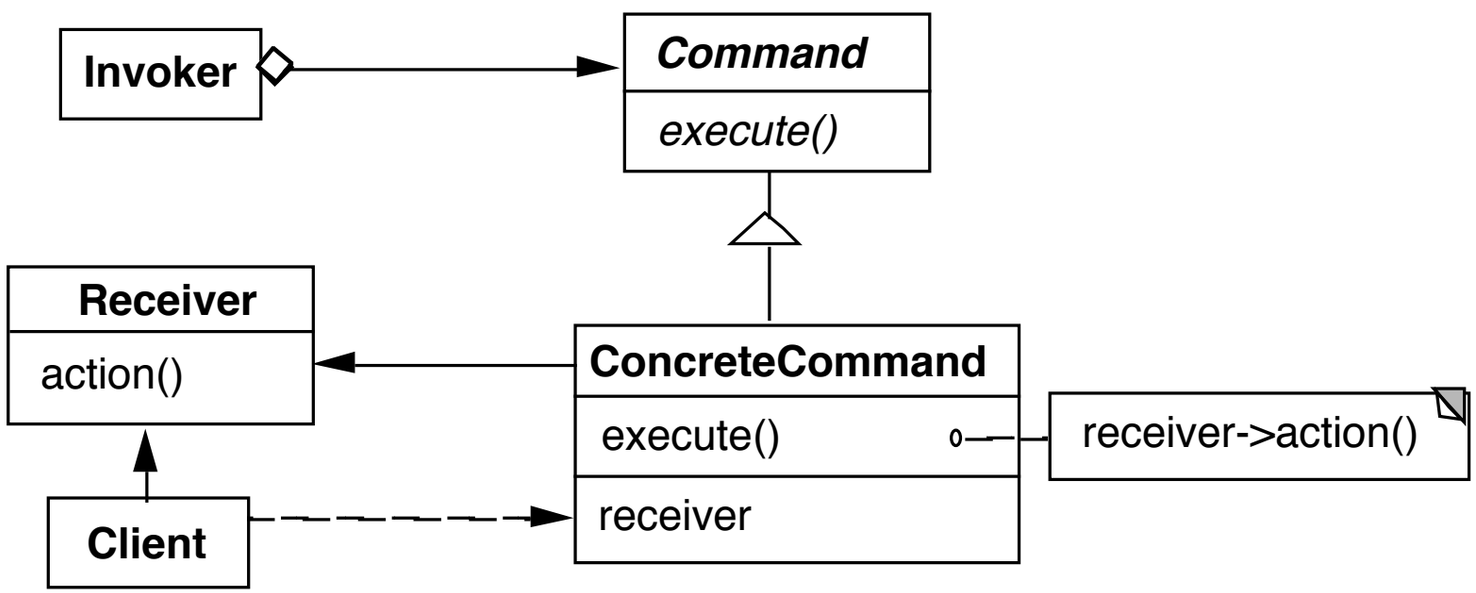
CS 635 Advanced Object-Oriented Design & Programming
Fall Semester, 2022
Doc 11 Command, Memento
Oct 4, 2022

Command

Command

Encapsulates a request as an object



Example

Invoker be a menu

Client be a word processing program

Receiver a document

Action be save

Sample Command

```
public abstract class Command {
    public abstract void execute();
    public abstract void undo();
}

public class IncreaseCommand extends Command {
    private Counter subject;

    public IncreaseCommand(Counter toIncrease) {
       subject = toIncrease;
    }

    public abstract void execute() { subject.increase() };

    public abstract void undo() { subject.decrease() };
}
```

Sample Command - Text Editing

Requires more details

- Text that is being edited
- Location in text to changed
- Replacement text

Undo requires

- Text that is being edited
- Location in text that was changed
- Text that was replaced

When to Use the Command Pattern

Need action as a parameter (replaces callback functions)
Lambda's replace this use

Specify, queue, and execute requests at different times

Undo

Logging changes

High-level operations built on primitive operations

A transaction encapsulates a set of changes to data

Systems that use transaction often can use the command pattern

Macro language

Consequences

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

It is easy to add new commands, because you do not have to change existing classes

You can assemble commands into a composite object

Pluggable Commands

Can create one general Command using reflection

Don’t hard code the method called in the command

Pass the method to call an argument

Java Example of Pluggable Command

```
import java.util.*;
import java.lang.reflect.*;

public class Command
    {
    private Object receiver;
    private Method command;
    private Object[] arguments;

    public Command(Object receiver, Method command,
                   Object[] arguments )
    {
    this.receiver = receiver;
    this.command = command;
    this.arguments = arguments;
    }

    public void execute() throws InvocationTargetException,
                                 IllegalAccessException
    {
    command.invoke( receiver, arguments );
    }
}
```

Using the Pluggable Command

```
public class Test {
    public static void main(String[] args) throws Exception
        {
        Vector sample = new Vector();
        Class[] argumentTypes = { Object.class };
        Method add =
            Vector.class.getMethod( "addElement", argumentTypes);
        Object[] arguments = { "cat" };

        Command test = new Command(sample, add, arguments );
        test.execute();
        System.out.println( sample.elementAt( 0));
        }
    }
```

Output

cat

Pluggable Commands using Lambdas

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


public class PluggableCommand {
    Command do;
    Command undo;

    public PluggableCommand(Command do, Command undo) {
        this.do = do;
        this.undo = undo;
    }

    public void execute() { do.execute(); }

    public void undo() { undo.execute(); }
```

Pluggable Commands using Lambdas

```
final Counter example = new Counter();
PluggableCommand increase;

increase = new PluggableCommand(
    () -> example.increase(),
    () -> example.decrease());
increase.execute();
```

Note
Java's lambdas put restrictions on the variable example

Command Pattern & Lambda

Lambda’s can replace command objects for

Callbacks

Batch processing

Logging

Macro language

Functional Programming & Command

Simple cases - can just use function

But what if function needs
State
Receiver

Closures

```
function counter()
  n = 0
  return () -> n += 1
end
```

```
counter_a = counter()
counter_b = counter()
counter_a()          # 1
counter_a()          # 2
counter_a()          # 3
counter_b()          # 1
```

So functions can maintain state

With Multiple Functions

```
function counter(start = 0)
  n = start
  return () -> n += 1, () -> n = start
end
```

```
(plus_a, reset_a) = counter(10)
(plus_b, reset_b) = counter()

plus_a()                    # 11
plus_a()                    # 12
reset_a()                # 10
plus_b()                    # 1
```

General Command

```
type Command
  execute::Function
  undo::Function
end

function execute(command::Command)
  command.execute()
end

function undo(command::Command)
  command.undo()
end
```

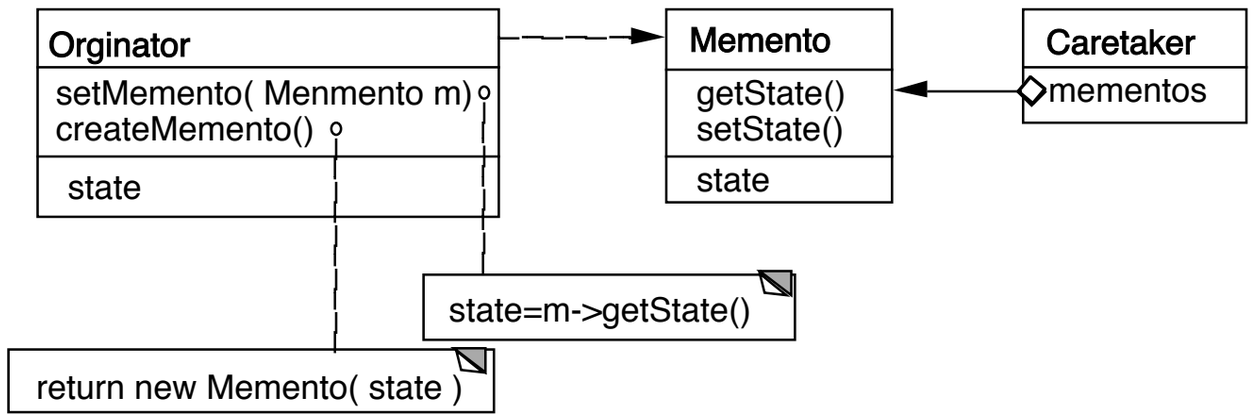
```
function counter(start)
  n = start
  return Command(()-> n += 1, ()-> n -= 1)
end


  count = counter(5)
execute(count)          # 6
undo(count)             # 5
```

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);}
```

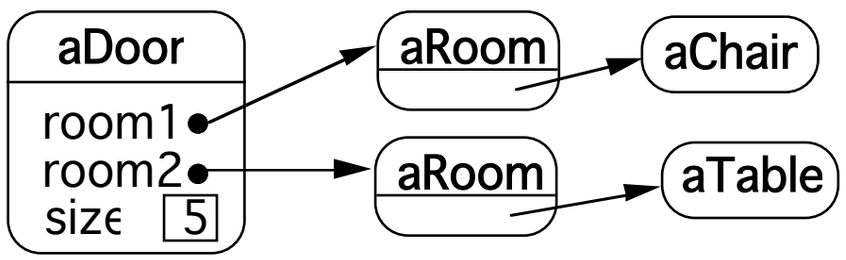
Using Clone to Save State

```
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;
    }
}
```

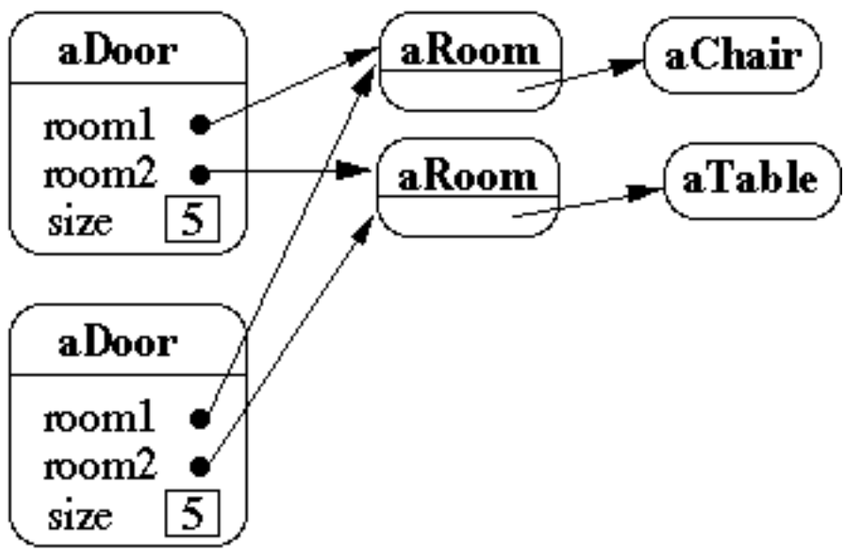
Copying Issues

Shallow Copy Verse Deep Copy

Original Objects

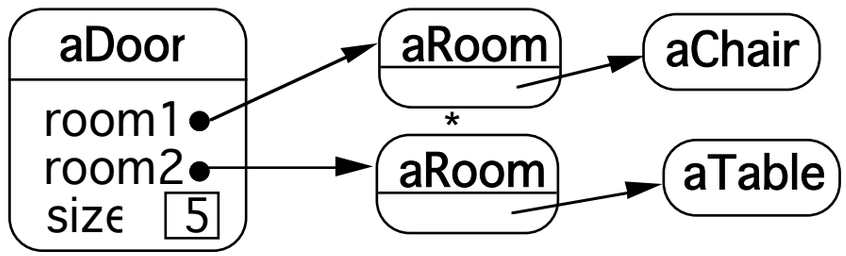


Shallow Copy

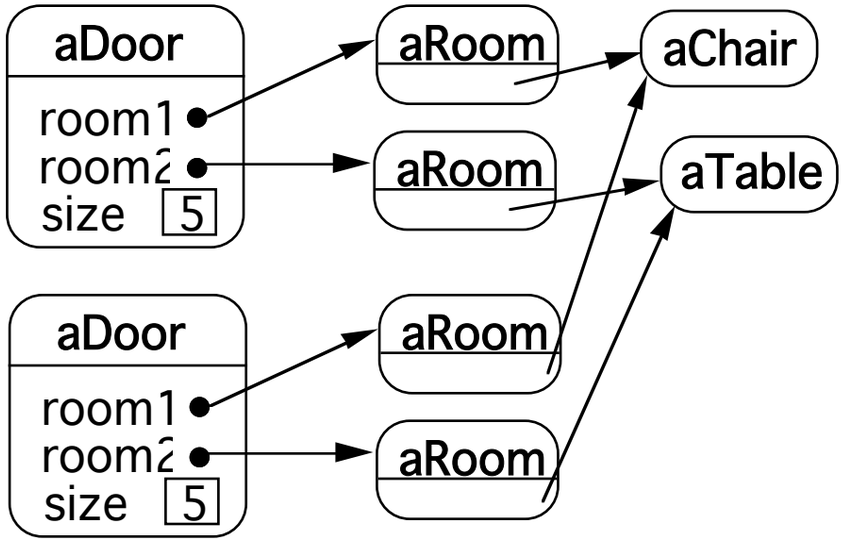


Shallow Copy Verse Deep Copy

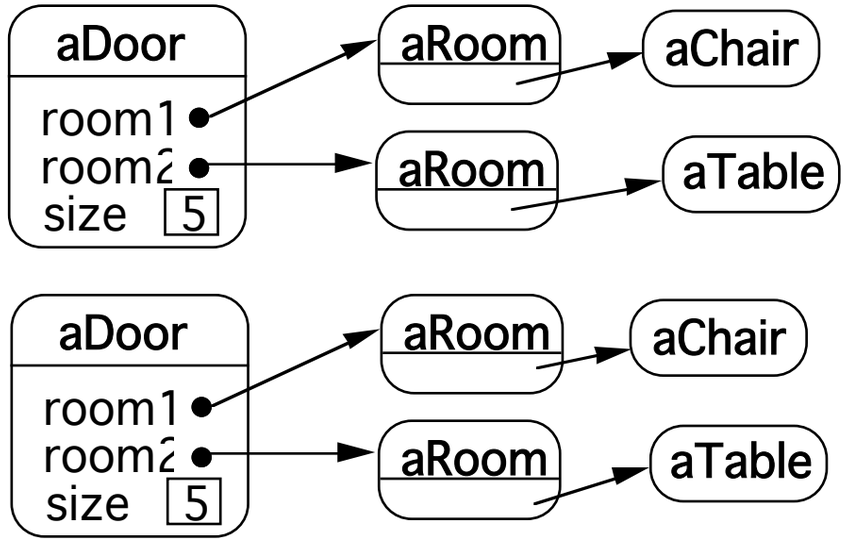
Original Objects



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