CS 635 Advanced Object-Oriented Design & Programming
Spring Semester, 2005

Doc 12 Command

Contents

References

Design Patterns: Elements of Reusable Object-Oriented Software, Gamma, Helm, Johnson, Vlissides, Addison-Wesley, 1995, pp. 233-242

The Design Patterns Smalltalk Companion, Alpert, Brown, Woolf, Addision-Wesley, 1998, pp. 245-260

Pattern-Oriented Software Architecture: A System of Patterns, Buschman, Meunier, Rohnert, Sommerlad, Stal, 1996, pp. 277-290, Command Processor

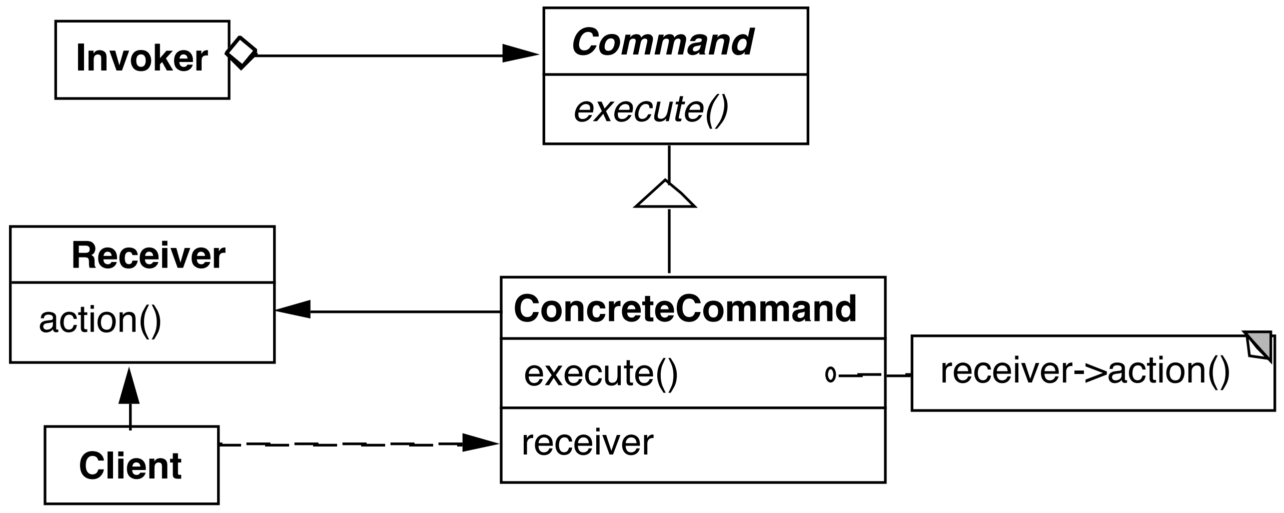
Command Processor, Sommerlad in Pattern Languages of Program Design 2, Eds. Vlissides, Coplien, Kerth, Addison-Wesley, 1996, pp. 63-74

Advanced C++: Programming Styles and Idioms, James Coplien, Addison Wesley, 1992, pp 165-170, Functor Pattern

Command

Encapsulates a request as an object

Structure



Example

Let

- Invoker be a menu
- Client be a word processing program
- Receiver a document
- Action be save

When to Use the Command Pattern

- When you need an action as a parameter
  Commands replace callback functions

- When you need to specify, queue, and execute requests at different times

- When you need to support undo

- When you need to support logging changes

- When you structure a system around high-level operations built on primitive operations

  A Transactions encapsulates a set of changes to data

  Systems that use transaction often can use the command pattern

- When you need to support a 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

Example - Menu Callbacks

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

class OpenCommand extends Command
   {
   private Application opener;

   public OpenCommand( Application theOpener )
      {
      opener = theOpener;
      }

   public void execute()
      {
      String documentName = AskUserSomeHow();

      if ( name != null )
         {
         Document toOpen =
               new Document( documentName );
         opener.add( toOpen );
         opener.open();
         }
      }
   }
```

Using Command

```
class Menu
  {
  private Hashtable menuActions = new Hashtable();

  public void addMenuItem( String displayString,
              Command itemAction )
    {
    menuActions.put( displayString, itemAction );
    }

  public void handleEvent( String itemSelected )
    {
    Command runMe;
    runMe = (Command) menuActions.get( itemSelected );
    runMe.execute();
    }

  // lots of stuff missing
  }
```

MacroCommand

```
class MacroCommand extends Command {
   private Vector commands = new Vector();

   public void add( Command toAdd ) {
      commands.addElement( toAdd );
   }

   public void remove( Command toRemove ) {
      commands.removeElement( toAdd );
   }

   public void execute() {
      Enumeration commandList = commands.elements();

      while ( commandList.hasMoreElements() )
         {
         Command nextCommand;
         nextCommand = (Command)
                     commandList.nextElement();
         nextCommand.execute();
      }
   }
}
```

Prevayler

http://www.prevayler.org/wiki.jsp

Prevalence layer for Java

Database that

- Serializes object to save them to disk
- Uses commands when modifying objects
- Keeps log of commands

Restaurant Example

```
import java.util.*;
import org.prevayler.implementation.AbstractPrevalentSystem;

public class Restaurant extends AbstractPrevalentSystem {
   private String name;
   ArrayList ratings = new ArrayList();

   public Restaurant(String newName) { name = newName;}

   public String name() {return name;}

   public void addRating( int newRating) {
      ratings.add( new Integer(newRating));
   }

   public float getRating() {
      if (ratings.size() == 0 )
         return -1;
      int total = 0;
      for (int k =0; k < ratings.size();k++)
         total = total + ((Integer)ratings.get(k)).intValue();
      return total/ ratings.size();
   }
}
```

Command

```
import java.io.Serializable;

import org.prevayler.Command;
import org.prevayler.PrevalentSystem;


public class AddRatingCommand implements Command {
   private final int newRating;

   public AddRatingCommand(int rating) {
      newRating = rating;
   }

   public Serializable execute(PrevalentSystem system) {
      ((Restaurant)system).addRating(newRating);
      return null;
   }

}
```

First Run

```
import java.util.*;
import org.prevayler.implementation.SnapshotPrevayler;

public class PrevaylerExample {

   public static void main (String args[]) throws Exception {
      SnapshotPrevayler samsDinerData =
         new SnapshotPrevayler(new Restaurant("Sams Diner"), "food");

      System.out.println( “Start”);
      Restaurant samsDiner = (Restaurant) samsDinerData.system();
      System.out.println( samsDiner.getRating() );
      samsDinerData.executeCommand( new AddRatingCommand( 5));
      System.out.println( samsDiner.getRating() );

   }
}
```

Output

```
Recovering system state...
Start
-1.0
5.0
```

Second Run

```
public class PrevaylerExample {

   public static void main (String args[]) throws Exception {
      SnapshotPrevayler samsDinerData =
         new SnapshotPrevayler(new Restaurant("Sams Diner"), "food");

      System.out.println( “Start”);
      Restaurant samsDiner = (Restaurant) samsDinerData.system();
      System.out.println( samsDiner.getRating() );
      samsDinerData.executeCommand( new AddRatingCommand( 10));
      System.out.println( samsDiner.getRating() );

   }
}
```

Output

```
Recovering system state...
Reading food/000000000000000000001.commandLog...
Start
5.0
7.0
```

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

One does have to be careful with the primitive types

```
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 Command Smalltalk Version

```
Object subclass: #PluggableCommand
   instanceVariableNames: 'receiver selector arguments '
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Whitney-Examples'
```

Class Methods

```
receiver: anObject selector: aSymbol arguments: anArrayOrNil
   ^super new
      setReceiver: anObject
      selector: aSymbol
      arguments: anArrayOrNil
```

Instance Methods

```
setReceiver: anObject selector: aSymbol arguments: anArrayOrNil
   receiver := anObject.
   selector := aSymbol.
   arguments := anArrayOrNil isNil
            ifTrue:[#( )]
            ifFalse: [anArrayOrNil]

execute
   ^receiver
      perform: selector
      withArguments: arguments
```

Using the Pluggable Command

```
| sample command |
sample := OrderedCollection new.
command := PluggableCommand
      receiver: sample
      selector: #add:
      arguments: #( 5 ).
command execute.
^sample at: 1
```

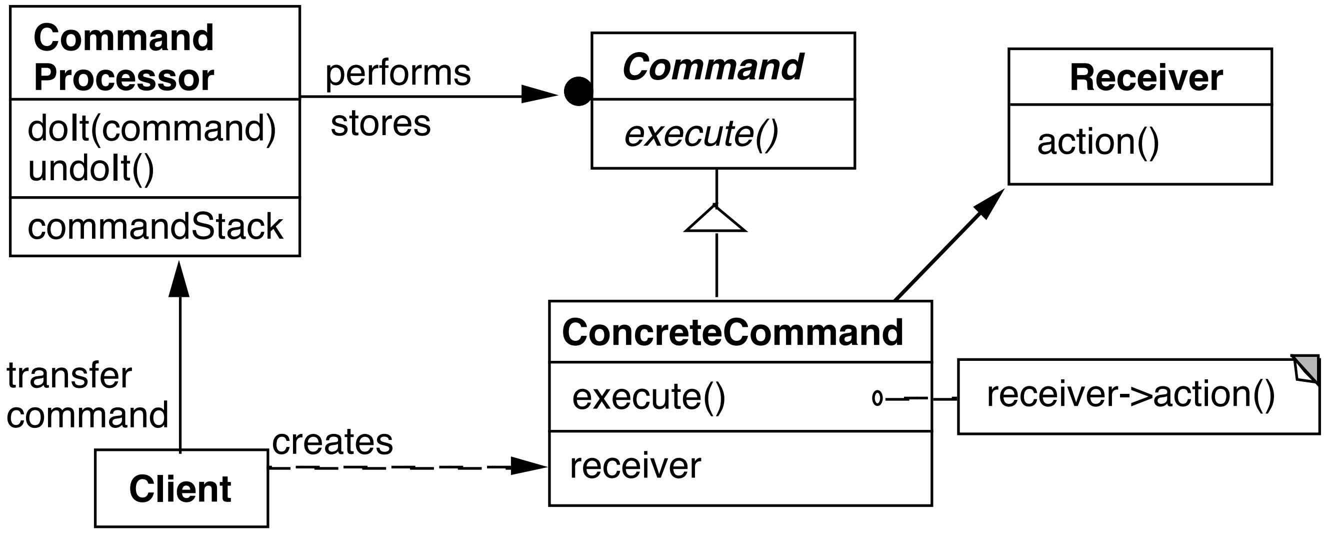
Command Processor

Command Processor manages the command objects

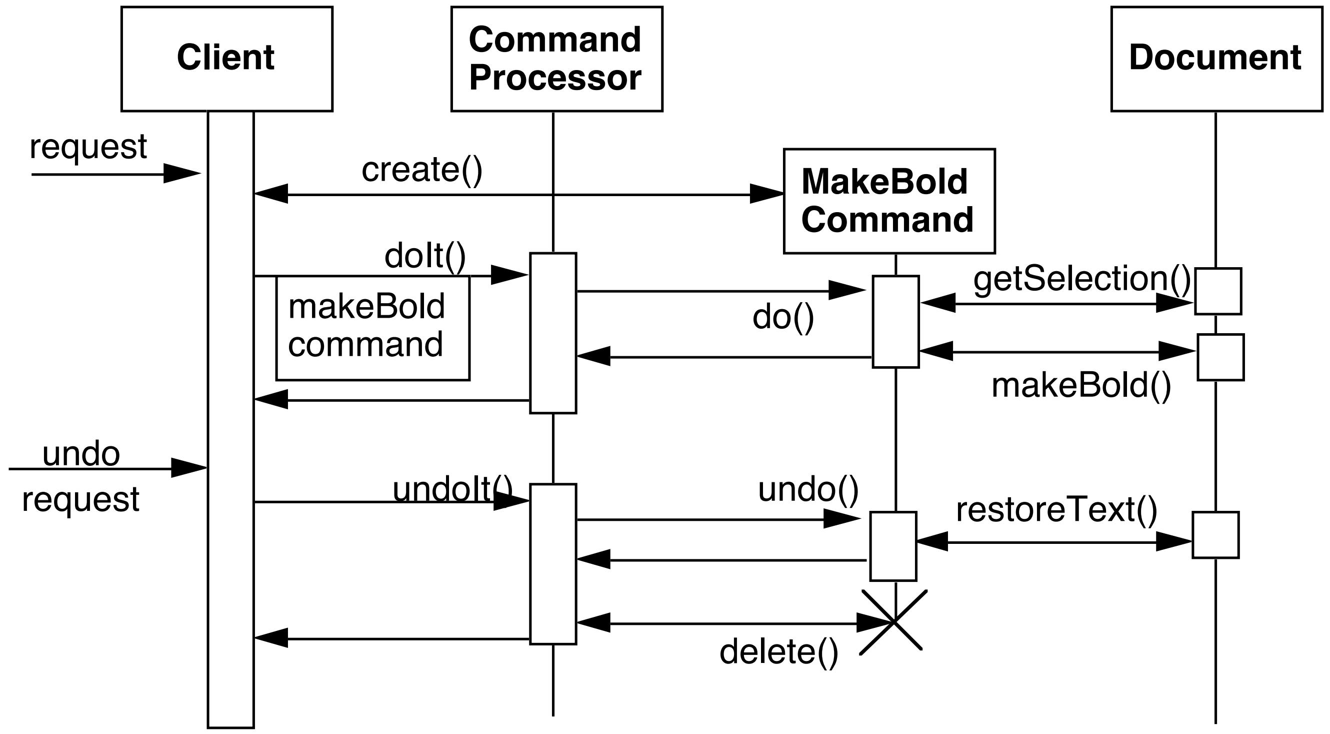
The command processor:

- 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



Consequences
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?

Functor
Functions as Objects

A functor is a class with

- A single member function (method)

Functors are functions that behave like objects

They serve the role of a function, but can be created, passed as parameters, and manipulated like objects

```
final class StudentNameComparator implements Comparator {

   public int compare( Object leftOp, Object rightOp ) {
      String leftName = ((Student) leftOp).name;
      String rightName = ((Student) rightOp).name;
      return leftName.compareTo( rightName );
   }
}
```

How Does a Functor Compare to Function Pointers?

- Using inheritance we can factor common code to a base class
- Same run-time flexibility as function pointer
- Lends it self to poor abstractions

How does A Functor compare with Command?

When to use the Functor

Coplien states:

> Use functors when you would be tempted to use function pointers

Functors are commonly used for callbacks