CSCI 253

Object Oriented Design:
Command Pattern
George Blankenship

Overview

Creational Patterns
- Singleton
- Abstract factory
- Factory Method
- Prototype
- Builder

Structural Patterns
- Composite
- Façade
- Proxy
- Flyweight
- Adapter
- Bridge
- Decorator

Behavioral Patterns
- Chain of Respons.
- Command
- Interpreter
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Template Method
- Visitor

The Elements of a Design Pattern

- A pattern name
- The problem that the pattern solves
  - Including conditions for the pattern to be applicable
- The solution to the problem brought by the pattern
  - The elements (classes-objects) involved, their roles, responsibilities, relationships and collaborations
  - Not a particular concrete design or implementation
- The consequences of applying the pattern
  - Time and space trade off
  - Language and implementation issues
  - Effects on flexibility, extensibility, portability

The Command Pattern: The Problem

Objects are used to represent actions. A **command object** encapsulates an action and its parameters.

- Building GUI would like command “add menu/item”
- Using GUI would like “display this message with time”
- Easy method to create solicitation panels
- Easy method to create boxes that validate the input

Also known as : Transaction

Command Pattern George Blankenship 4

addMenu

```
public void addMenu(String name, Menu menu) {
    Menu helpMenu = mainMenu.getHelpMenu();
    if(helpMenu!=null) mainMenu.remove(helpMenu);
    menus.setTail(new MenuList(name,menu)); // add to menu list
    mainMenu.add(menu);                     // add to the menu bar
    if(helpMenu!=null) mainMenu.add(helpMenu);
    repaint();
  }
```

Command Pattern George Blankenship 5

addMenuItem

```
public boolean addMenuItem(String name, MenuItem item) {
    MenuList l = (MenuList)menus.getFirst(); // find the menu
    while((l!=null) && (l.getName()!=name))
       l = (MenuList)l.getNext();
    if(l==null) return false;              // could not find menu
    l.getMenu().add(item);                   // add to the menu
    return true;
  }
```

Command Pattern George Blankenship 6

append

```
public synchronized void append(String t) {
    String s = "\n"+timeStamp();
    displayArea.append(s+" "+t);
  }
```


SolicitGUI

```
public SolicitGUI(String t, MainGUI g, ParameterPanel p) { // initialize the main window
    super(t);                       // define the main window
    screenSize = Toolkit.getDefaultToolkit().getScreenSize();
    title = t;
    configurationPanel = p;
    mainGUI = g;
    box = this;
    trace = new Trace(mainGUI,CODE_FILE);    // trace for debug
    trace.setActive(false);              // turn it off initially
    setupPanels();                  // set up the display panels
    doneButton.requestFocus();
    addWindowListener(new SolicitGUI.MainWindowEvents()); // wait for something to happen
setVisible(true);                   // make window visible
    windowSize = getSize();             // center the window
    setLocation((screenSize.width-windowSize.width)/2,
            (screenSize.height-windowSize.height)/2);
    trace.write("solicit GUI ready");
  }
```

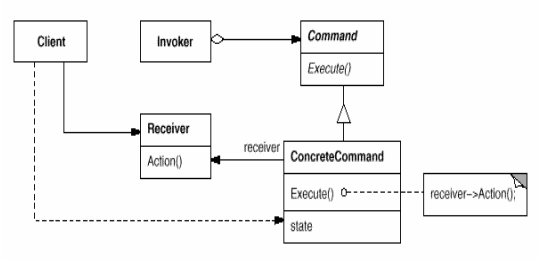

InputArea

```
public InputField(MainGUI g, String n, int c, String t, int f) { // create
  the input area
    super(c);                       // create the field
    mainGUI = g;
    setName(n);                     // listeners can identify events
    setText(t);                     // initial test in the field
    fieldType = f;
    addActionListener(new InputField.InputActions()); // wait for input
    addKeyListener(new InputField.KeyEvents());
  }
```

Undo

- Another of the main reasons for using Command design patterns is that they provide a convenient way to store and execute an Undo function.
- Each command object can remember what it just did and restore that state when requested to do so if the computational and memory requirements are not too overwhelming.

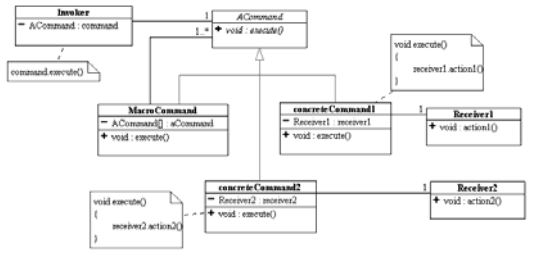
The Command Pattern: Structure



The Command Pattern: Participants

- Command
  - declares an interface for executing an operation.
- ConcreteCommand (PasteCommand, OpenCommand)
  - defines a binding between a Receiver object and an action.
  - implements Execute by invoking the corresponding operation(s) on Receiver.
- Client (Application)
  - creates a ConcreteCommand object and sets its receiver.
- Invoker (MenuItem)
  - asks the command to carry out the request.
- Receiver (Document, Application)
  - knows how to perform the operations associated with carrying out a request. Any class may serve as a Receiver.

Participant Map



Command Pattern George Blankenship 13

The Command Pattern: Collaboration

- The client creates a ConcreteCommand object and specifies its receiver.
- An Invoker object stores the ConcreteCommand object.
- The invoker issues a request by calling Execute on the command. When commands are undoable, ConcreteCommand stores state for undoing the command prior to invoking Execute.
- The ConcreteCommand object invokes operations on its receiver to carry out the request.

Command Pattern George Blankenship 14

The Command Pattern: Consequences

- The main disadvantage of the Command pattern is a proliferation of little classes that either clutters up the main class if they are inner or clutters up the program namespace if they are outer classes.
- Now even in the case where we put all of our *actionPerformed* events in a single basket, we usually call little private methods to carry out the actual function. It turns out that these private methods are just about as long as our little inner classes, so there is frequently little difference in complexity between inner and outer class approaches.

Command Pattern George Blankenship 15

The Command Pattern: Implementation

- It can improve API design. In some cases, code that uses a command object is shorter, clearer, and more declarative than code that uses a procedure with many parameters. This is particularly true if a caller typically uses only a handful of the parameters and is willing to accept sensible defaults for the rest.
- A command object is convenient temporary storage for procedure parameters. It can be used while assembling the parameters for a function call and allows the command to be set aside for later use.
- A class is a convenient place to collect code and data related to a command. A command object can hold information about the command, such as its name or which user launched it; and answer questions about it, such as how long it will likely take.
- Treating commands as objects enables data structures containing multiple commands. A complex process could be treated as a tree or graph of command objects. A thread pool could maintain a priority queue of command objects consumed by worker threads.
- Treating commands as objects supports undo-able operations, provided that the command objects are stored (for example in a stack)
- The command is a useful abstraction for building generic components, such as a thread pool, that can handle command objects of any type. If a new type of command object is created later, it can work with these generic components automatically.

Command Pattern George Blankenship 16