Command Pattern

GoF: object behavioral

Operational pattern

Lives at the boundary of two paradigms, functional decomposition and object orientation

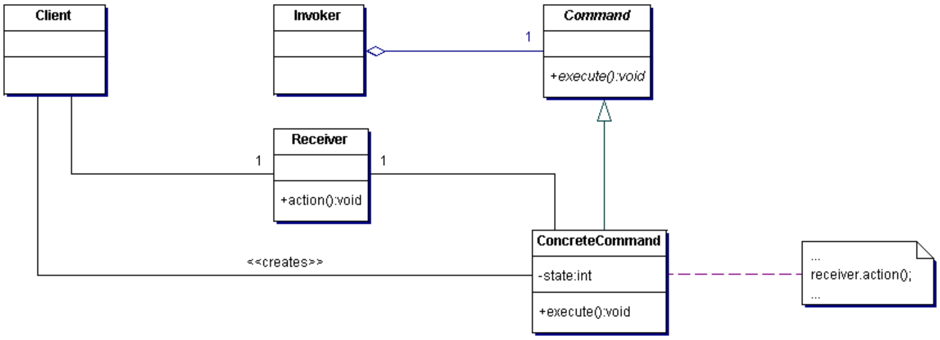
Background

- In *Advanced C++:Programming Styles And Idioms (Addison-Wesley, 1992)*, Jim Coplien introduces the term *functor* which is an object whose sole purpose is to encapsulate a function.
  - the term *function object* is often also used in this meanign.
  - The point is to decouple the choice of function to be called from the site where that function is called.
  - The term *functor* is mentioned but not used in *Design Patterns*. However, the theme of the function object is repeated in a number of patterns in that book.
- A Command is a function object in its purest sense: a method that's an object.
- By wrapping a method in an object, you can pass it to other methods or objects as a parameter, to tell them to perform this particular operation in the process of fulfilling your request.
- You could say that a *Command* is a messenger (because its intent and use is very straightforward) that carries behavior, rather than data.

Basic Aspects

- Intent
  - Encapsulate requests as objects, letting you to:
    - parameterize clients with different requests
    - queue or log requests
    - support undoable operations
- Problem
  - Need to issue requests to objects without knowing anything about the operation being requested or the receiver of the request.
- Applicability
  - Parameterize objects
  - Specify, queue, and execute requests at different times
    - replacement for callbacks
  - Support undo
  - Support for logging changes
  - Model transactions
    - structure systems around high-level operations built on primitive ones
    - common interface ⇒ invoke all transaction same way

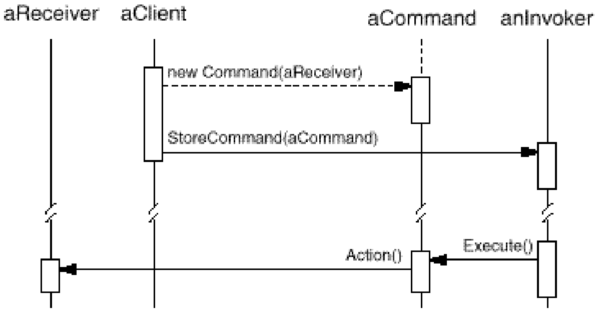
Structure



Participants

- Command
  - declares the interface for executing the operation
- ConcreteCommand
  - binds a request with a concrete action
- Invoker
  - asks the command to carry out the request
- Receiver
  - knows how to perform the operations associated with carrying out a request.
- Client
  - creates a ConcreteCommand and sets its receiver

Collaborations



- Example
  - Invoker is a menu
  - Client is an text editor program
  - Receiver is a document
  - Action is save

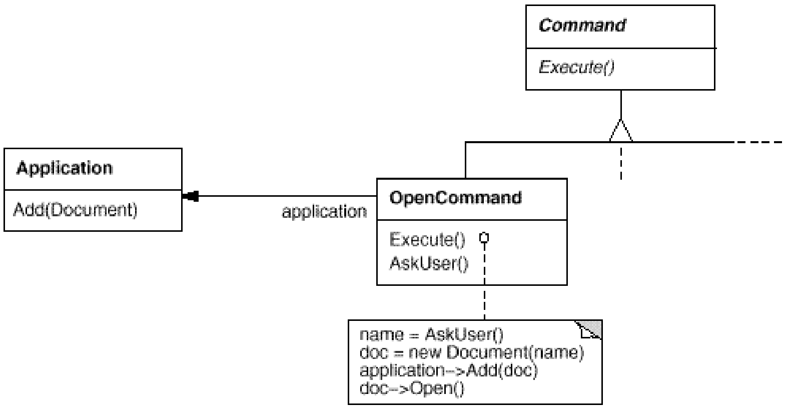
Consequences

- Command decouples the object that invokes the operation from the one that knows how to perform it.
  - To achieve this separation, the designer creates an abstract base class that maps a receiver (an object) with an action (a pointer to a member function). The base class contains an execute() method that simply calls the action on the receiver.
  - All clients of Command objects treat each object as a "black box" by simply invoking the object's virtual execute() method whenever the client requires the object's "service".
- Commands are first-class objects
  - can be manipulated and **extended**
- Composite Commands
  - Sequences of Command objects can be assembled into composite (or macro) commands
  - see also *Composite* pattern
- Easy to add new commands
  - Invoker does not change
  - it is Open-Closed

Intelligence of Command objects

- "Dumb"
  - delegate everything to Receiver
  - used just to decouple Sender from Receiver
- "Genius"
  - does everything itself without delegating at all
  - Related to proxy-pattern in intent
  - let ConcreteCommand be independent of further classes
- "Smart"
  - find receiver dynamically

Example: Menu Callbacks



Example – decoupling GUI elements from the program

- Suppose we build a simple program that has the functionality of selecting menu items File Open and File Exit, and a button Red that can be pressed.
- The program consist of the File Menu object with the mnuOpen and mnuExit MenuItems, and a button called btnRed.
- Clicking any of these causes an ActionEvent which generates a call to the *actionPerformed* method:

```
public void actionPerformed(ActionEvent e) {
   Object obj = e.getSource();
   if (obj == mnuOpen) fileOpen(); //open file
   if (obj == mnuExit) exitClicked(); // exit program
   if (obj == btnRed) redClicked(); //turn red
}

// one of the methdos that get called from actionPerformed (as an
   example)
private void fileOpen() {
   FileDialog fDlg = new FileDialog(this, "Open a file",
                                         FileDialog.LOAD);
    fDlg.show();
}
```

...decoupling GUI...

- The previous approach works fine as long as the GUI is simple, as the number of GUI elements increases the actionPerformed method gets complicated.
- Using command objects helps to solve this problem

```
// the simple interface that command objects must implement.
public interface Command
{
   public void Execute();
}
/* we make the GUI elements (menu items, buttons) containers
  for a command object that exists separately. This way we
  avoid the dependency that would result from binding
  command objects directly into elements that cause the
  action (invoker). */
// GUI elements will implement this interface
public interface CommandHolder {
   public void setCommand(Command comd); // put command
   public Command getCommand(); //fetch command to execute
}
```

...decoupling GUI...

- then we create the cmdMenu class to implement CommandHolder

```
public class cmdMenu extends JMenuItem implements
  CommandHolder {
   protected Command menuCommand; // internal copies
   protected JFrame frame;
//-----------------------
   public cmdMenu(String name, JFrame frm) {
      super(name);
      frame = frm;
   }
//-----------------------
   public void setCommand(Command comd) {
      menuCommand = comd;
   }
//-----------------------
   public Command getCommand() {
      return menuCommand;
   }
}
```

...decoupling GUI...

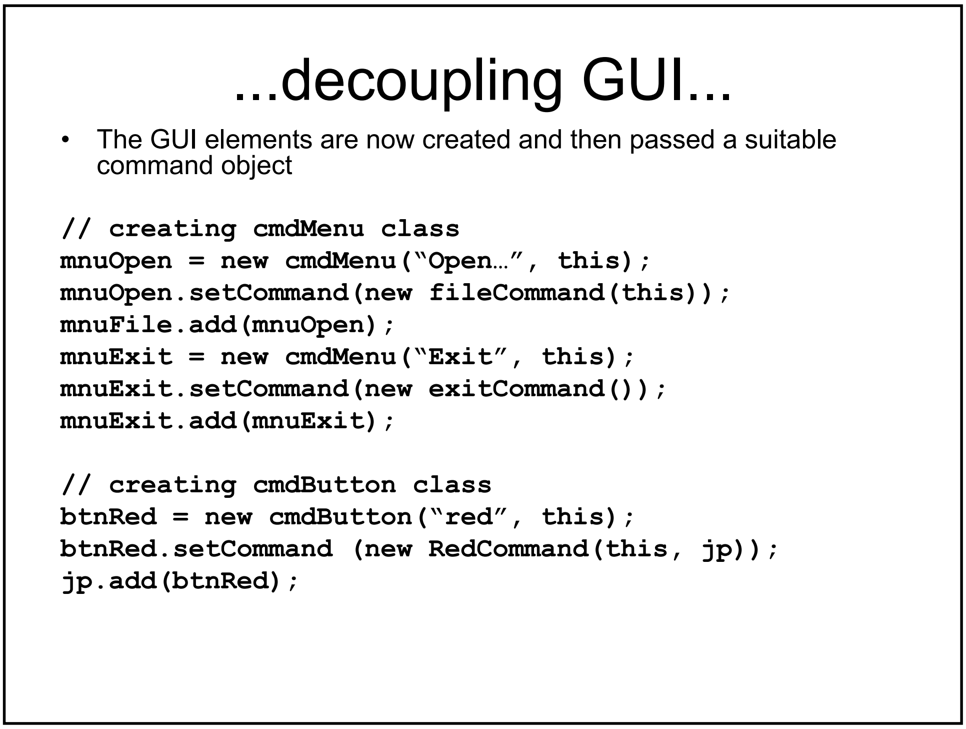
- and similarly we create the cmdButton class

```
public class cmdButton extends JButton implements
  CommandHolder {
        private Command btnCommand;
        private JFrame frame;

  public cmdButton(String name, JFrame fr) {
     super(name);
     frame = fr;
  }
  public void setCommand(Command comd) {
      btnCommand = comd;
   }
   public Command getCommand() {
      return btnCommand;
   }
}
```

...decoupling GUI...

- Now the command objects are separated from user interface classes. As an example, the FileCommand class is defined as:

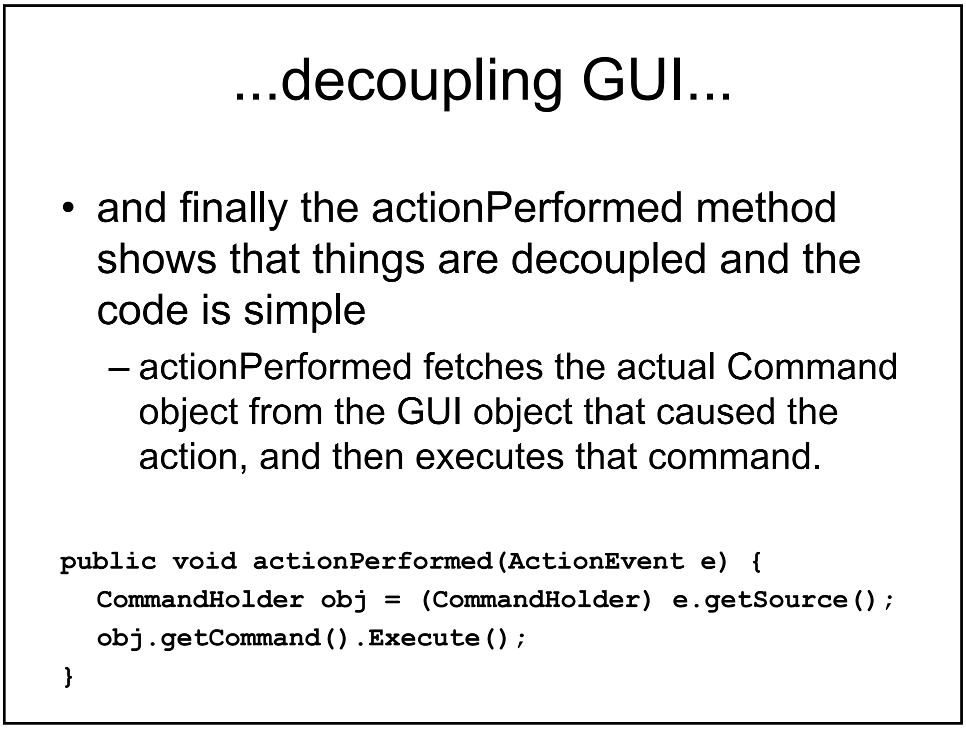
```
public class fileCommand implements Command {
   JFrame frame;

   public fileCommand(JFrame fr) {
      frame = fr;
   }
//------------------------------
   public void Execute() {
      FileDialog fDlg = new FileDialog(frame, "Open
  file");
      fDlg.show();
   }
}
```



...decoupling GUI...

- The GUI elements are now created and then passed a suitable command object

```
// creating cmdMenu class
mnuOpen = new cmdMenu(“Open…”, this);
mnuOpen.setCommand(new fileCommand(this));
mnuFile.add(mnuOpen);
mnuExit = new cmdMenu(“Exit”, this);
mnuExit.setCommand(new exitCommand());
mnuExit.add(mnuExit);

// creating cmdButton class
btnRed = new cmdButton(“red”, this);
btnRed.setCommand (new RedCommand(this, jp));
jp.add(btnRed);
```



...decoupling GUI...

- and finally the actionPerformed method shows that things are decoupled and the code is simple
  - actionPerformed fetches the actual Command object from the GUI object that caused the action, and then executes that command.

```
public void actionPerformed(ActionEvent e) {
  CommandHolder obj = (CommandHolder) e.getSource();
  obj.getCommand().Execute();
}
```