Command

The Command pattern encapsulates a request as an object and decouples the request from clients invoking the request. Encapsulating a request as an object allows executable code to be passed around and treated like any other object. A command object can be referenced from multiple locations, queued for batch processing, and saved to persistent storage. The command pattern also provides a convenient framework for implementing undoable operations.

Command Pattern

When the only method needed on an object is `run()` or `execute()`, that's a telltale sign you are dealing with a command object.

Introduction

Buttons that control devices are more useful when they are programmable. For example, consider some of the buttons found inside a modern automobile. The standard automobile radio has preset buttons that can be programmed to recall radio stations at specific frequencies.

Figure x. Radio buttons control the radio

Luxury cars typically have seat position memory buttons that can be programmed to return the driver's seat to a preset position.

Figure x. Seat position memory buttons control the driver's seat

The current design of programmable buttons in automobiles is somewhat limited though because the buttons are device-specific. Radio buttons only work with the radio and seat position memory buttons only work with the driver's seat. In programming terms, the buttons are tightly coupled to the components they control:

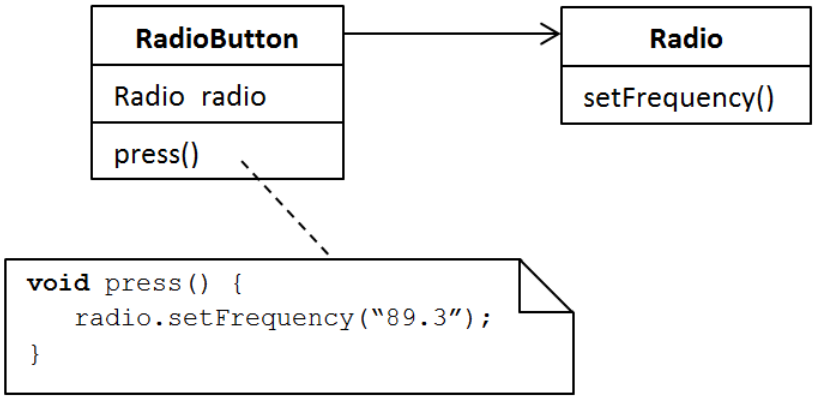


Figure x. Radio buttons are tightly coupled to the radio they control

A whole new level of automation would be possible in automobiles if manufactures offered generic device-independent programmable buttons capable of controlling any component within the automobile.

Imagine the possibilities. You could define a button to prepare your automobile for entering a dusty work zone. It would roll up the windows and adjust the heating and air conditioning system to recirculate the air inside the cabin. A convertible could have a "Summer Fun" button that would retract the top, roll down the windows and turn up the radio.

In order to provide for such flexibility, the buttons would have to be loosely coupled to the components they controlled. One way to accomplish this is to add a level of indirection between buttons and components. Buttons would no longer control devices directly but rather command objects that would encapsulate operations on one or more components.

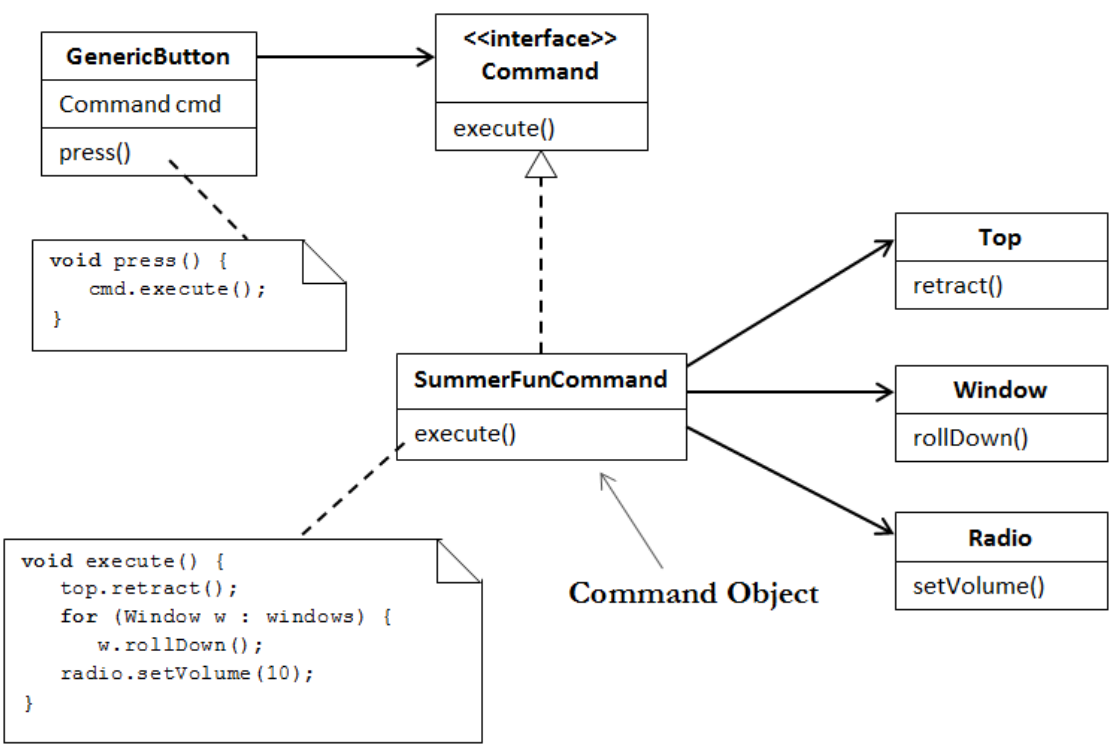


Figure x. Generic buttons are decoupled from the objects they control

In the design above, buttons no longer interact directly with components. Instead, they control command objects. A command object encapsulates one or more operations on one or more components. At runtime when the user presses a button, the button sends a message to a command object and the command object calls actions on encapsulated components. If all command objects implement the same abstract interface, any button can be made to work with any command object.

This design is essentially the command pattern. The command pattern shows how to encapsulate behavior (i.e. code) as a first-class object. When behavior is represented as an object, you have more control over how it is used. In this non-technical example the concept was used to implement more powerful and flexible device-independent buttons.

Intent

The Command pattern encapsulates a request as an object and decouples the request from clients invoking the request. To understand what this means, consider what it means to have an unencapsulated request:

```
class Invoker {
   Receiver r;
   . . .
   void f() {
      r.request();
   }
```

```
}
```

The method `f()` in the code fragment above is sending a request to an instance of `Receiver`. The request is unencapsulated because it doesn't have a separate existence or identity outside of the method `f()`.

We can give it a separate identity by encapsulating the request in a separate command class:

```
class SomeCommand {
   Receiver r;
   . . .
   void execute() {
      r.request();
   }
}
```

Clients wanting to invoke the encapsulated request would call the `execute()` method on an instance of the command:

```
class Invoker {
   SomeCommand cmd;
   . . .
   void f() {
      cmd.execute();
   }
}
```

The invoking object is now decoupled from the action invoked. However, since `SomeCommand` is a concrete class, we have just traded one instance of coupling for another. We can reduce the coupling between clients and commands by introducing an abstract interface for invoking commands:

```
// Abstract interface for invoking commands
interface Command {
   void execute();
}
```

Concrete commands will also need to implement this new abstract command interface:

```
class SomeCommand implements Command {
   Receiver r;
   . . .
   void execute() {
      r.request();
   }
}
```

Clients that access commands through this new abstract interface are loosely coupled to the commands they invoke.

```
class Invoker {
   Command cmd;
   . . .
   void f() {
      cmd.execute();
   }
}
```

Encapsulating a request as an object allows executable code (i.e. requests) to be passed around and treated like any other object. A command object can be referenced from multiple locations, queued for batch processing, and saved to persistent storage. The command pattern also provides a convenient framework for implementing undoable operations.

Solution

The following class diagram shows the static structure of the Command pattern.

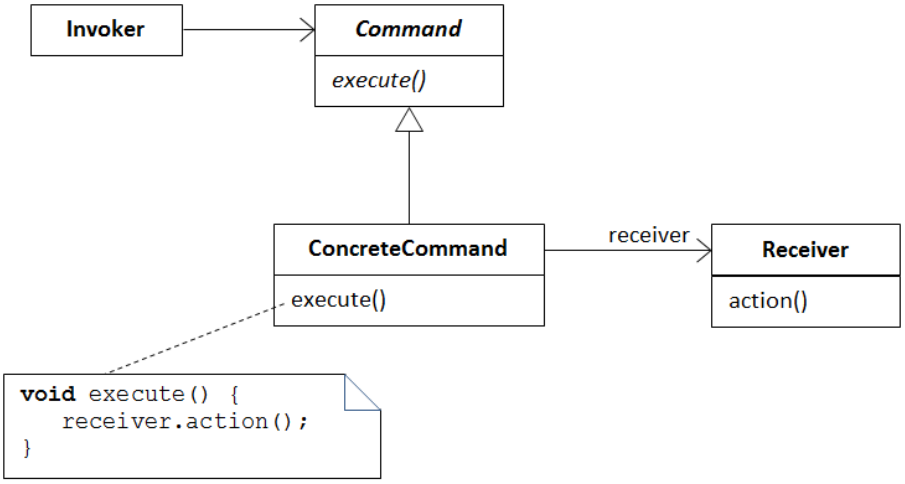


Figure x. Command design pattern

The four main components of the Command pattern are:

1. **Receiver** – The `Receiver` component represents existing functionality. The Command Pattern encapsulates a request. Typically the request is comprised of one or more operations being invoked on one or more existing receiver objects.
2. **ConcreteCommand** – `ConcreteCommand` calls operations on receiver objects in order to carry out the encapsulated request. For simple actions, `ConcreteCommand` may implement the action directly without the need for `Receiver` objects.
3. **Command** – `Command` is an abstract interface. It allows concrete commands to be loosely coupled to clients.
4. **Invoker** – `Invoker` objects initiate requests encapsulated in commands. Command objects are responsible for performing requests and invoker objects are responsible for initiating them.

Sample Code

Example #1

The first example in this section shows the command pattern being used to implement a simple menu-based text editor. Each feature or menu option is implemented as a command. The program's design follows the general structure of the command pattern introduced in the previous section:

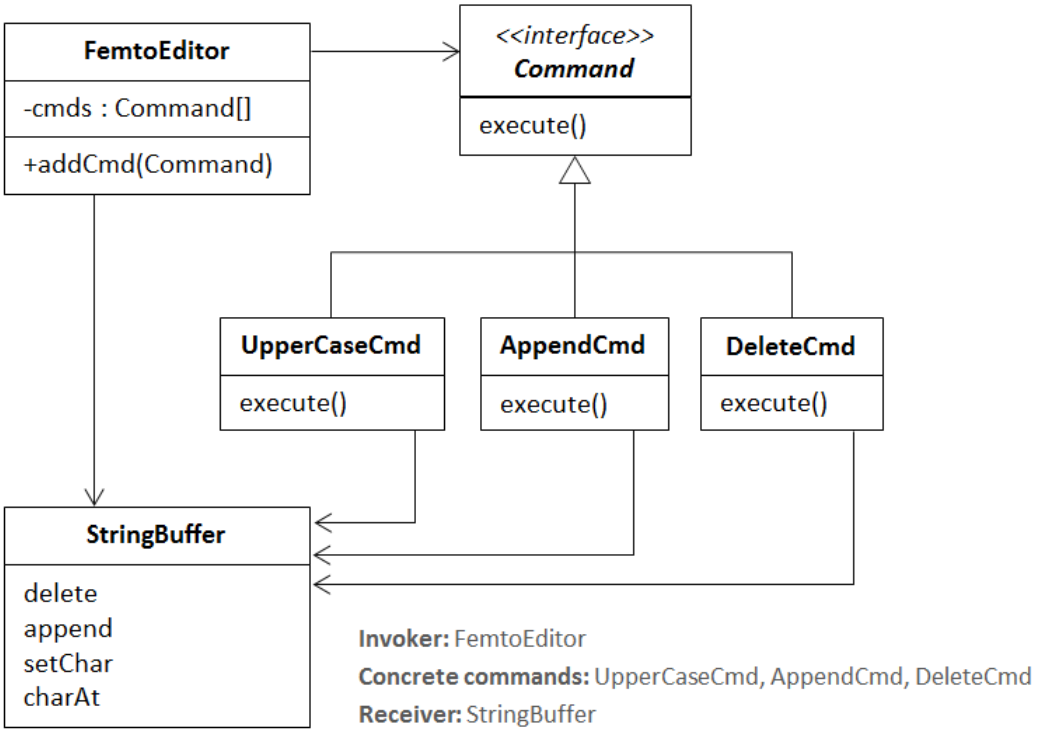


Figure x. Text editor design

During program startup, commands are created and added to the main control class: `FemtoEditor`. The runtime logic of `FemtoEditor` is simple. A menu option is displayed for each command. When the user selects a menu option, the corresponding command is executed. Commands operate on a shared reference to a `StringBuffer` which stores the state of the document being edited.

Here is the complete source code for the program:

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.ArrayList;
```

```
public class Startup {
   public static void main(String[] args) throws IOException {
      StringBuffer document = new StringBuffer("");
      FemtoEditor editor = new FemtoEditor(document);

      Command c;
      c = new UpperCaseCmd(document);
      editor.addCmd(c);

      c = new AppendCmd(document);
      editor.addCmd(c);

      c = new DeleteCmd(document);
      editor.addCmd(c);

      editor.run();
   }
}

interface Command {
   void execute();
}

class UpperCaseCmd implements Command {
   private StringBuffer document;

   public UpperCaseCmd(StringBuffer document) {
      this.document = document;
   }

   @Override
   public void execute() {
      for (int i = 0; i < document.length(); i++)
         if (Character.isLetter(document.charAt(i)))
      document.setCharAt(i, Character.toUpperCase(document.charAt(i)));
   }

   @Override
   public String toString() {
      return "Make upper case";
   }
}

class AppendCmd implements Command {
   private StringBuffer document;

   public AppendCmd(StringBuffer document) {
      this.document = document;
   }
```

```
   @Override
   public void execute() {
      BufferedReader br = new BufferedReader(
         new InputStreamReader(System.in));
      System.out.print("append> ");
      try { document.append(br.readLine()); }
      catch (IOException e) { }
   }

   @Override
   public String toString() {
      return "Append string";
   }
}

class DeleteCmd implements Command {
   private StringBuffer document;

   public DeleteCmd(StringBuffer document) {
      this.document = document;
   }

   @Override
   public void execute() {
      document.delete(0, document.length());
   }

   @Override
   public String toString() {
      return "Delete";
   }
}

class FemtoEditor {
   private ArrayList<Command> cmds = new ArrayList<Command>();
   private StringBuffer document;

   public FemtoEditor(StringBuffer document) {
      this.document = document;
   }

   public void addCmd(Command c) {
      cmds.add(c);
   }
```

```
   public void run() throws IOException {
      BufferedReader br = new BufferedReader(new
         InputStreamReader(System.in));
      for (;;) {
         System.out.println("\nText: " + document + "\n");
         System.out.println("Select an edit command:\n");

         for (int i = 0; i < cmds.size(); i++) {
            System.out.println((i + 1) + ". " + cmds.get(i));
         }

         System.out.print("\nmenu choice> ");

         int i = Integer.parseInt(br.readLine());
         Command c = cmds.get(i - 1);
         c.execute();
      }
   }
}
```

Here is a sample run of the program with input shown in bold:

```
Text:

Select an edit command:

1. Make upper case
2. Append string
3. Delete

menu choice> 2
append> Once upon a pattern

Text: Once upon a pattern

Select an edit command:

1. Make upper case
2. Append string
3. Delete

menu choice> 1

Text: ONCE UPON A PATTERN

Select an edit command:

1. Make upper case
2. Append string
3. Delete

menu choice> 3

Text:

Select an edit command:
```

```
1. Make upper case
2. Append string
3. Delete

menu choice>
```

Notice how the program follows the Open-Closed principle. `FemtoEditor` is open for extension but closed for modification. Adding a new edit feature such as spell check is as simple as creating a new command for spell checking and adding the command to `FemtoEditor` during startup. You can add a spell check feature to the editor without changing a line of code in `FemtoEditor` or any of the existing commands.

Example #2

The second and third examples in this section show real-world applications of the command pattern from the Java class library.

The command pattern has a prominent role in the Java class library. The language definition includes a generic command interface called `Runnable`:

```
public interface Runnable
   public void run();
}
```

The `Runnable` interface is used in the Java class library and user-level code whenever there is a need to pass computation around as an object. One situation where this capability is needed is when starting a new thread of execution.

In Java, a new thread of execution is created by instantiating an instance of the class `Thread` and calling its `start()` method. The constructor for the class `Thread` takes an instance of a class that implements the interface `Runnable`. Calling the `start()` method on the newly created `Thread` object causes the `run()` method of the `Runnable` object to be executed on a separate thread.

Here is an example:

```
public class HelloThreads {

   public static void main(String args[]) {

      Runnable helloCommand = new HelloCommand();
      Thread newThread = new Thread(helloCommand);
      newThread.start();

      System.out.println("Hello from main thread!");

   }
}
```

```
class HelloCommand implements Runnable {

   public void run() {
      System.out.println("Hello from new thread!");
   }
}
```

Output:

```
Hello from main thread!
Hello from new thread!
```

The order in which the print statements execute in the above program is unpredictable. Run the program again and the print statement in the new thread might execute first. The order of the output depends on what order the threads are scheduled. This is operating system dependent and may vary between runs.

Example #3

The command pattern is also used in the Java class library to pass code from one thread to another. Typically this is done through a queue. When using a queue, one thread adds work in the form of a command object to one end of the queue and another thread removes the command object from the other end of the queue and executes it.

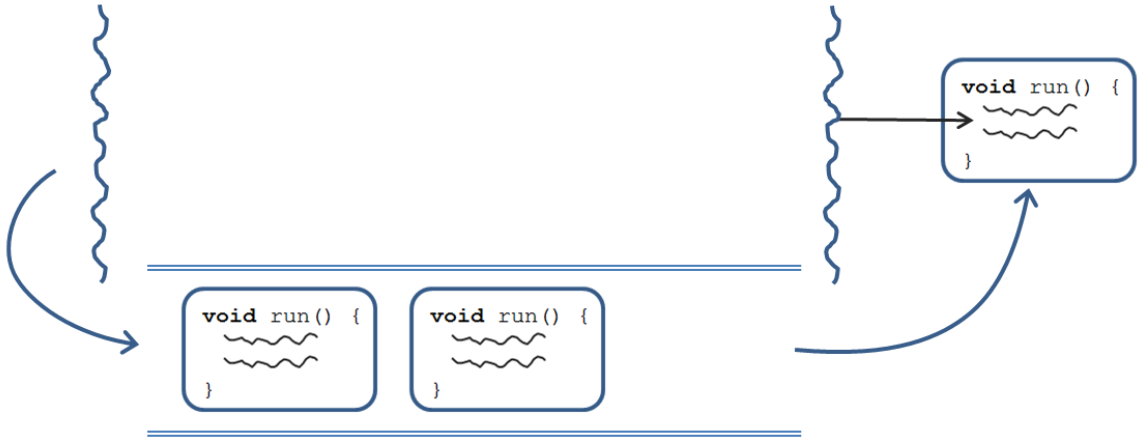


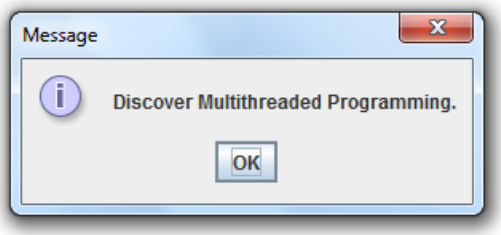
Figure x. Queue being used to pass code from one thread to another

This is a perfect opportunity to use the command pattern. Objects in the queue encapsulate work to be performed. From a thread's perspective, these objects need to support only one operation: `run()`. When the only method needed on an object is `run()` or `execute()`, that's a telltale sign you are dealing with a command object.

Scheduling code to run on a different thread in Java is surprisingly easy. The hard part is understanding why this is sometimes necessary. To get started, consider the following simple GUI program that displays a dialog box:

```
import javax.swing.JOptionPane;

public class CmdExample {
   public static void main(String[] args) {
      JOptionPane.showMessageDialog(null,
         "Discover Multithreaded Programming.");
   }
}
```

Output:



Although there is little chance of the program failing outright, as written the program is technically invalid. The problem is the call to showMessageDialog() being performed on the main thread of control.

Like most UI toolkits, Java's Swing UI toolkit is single-threaded. This means the creation and use of Swing components (e.g. buttons, text fields, labels, etc.) must be performed on a special UI thread created for every GUI application (the event-dispatching thread). The entry point of a GUI program (i.e. `main()`) executes on its own thread of control which is different from the special UI thread created for GUI programs.

To correct the error in the program above, the call to `showMessageDialog()` must be encapsulated in a command object and scheduled to run on the UI thread.

The Java API provides the following static method on the class `SwingUtilities` for scheduling work to run on the UI thread:

```
public static void invokeLater(Runnable doRun);
```

The following program displays the same dialog box as the one before but does so now without violating Java's GUI programming standards.

```
import javax.swing.JOptionPane;
import javax.swing.SwingUtilities;

public class CmdExample {
   public static void main(String[] args) {
      Runnable dialogCmd = new DialogCommand();
      SwingUtilities.invokeLater(dialogCmd);
   }
}
```

```
class DialogCommand implements Runnable {
   @Override
   public void run() {
      JOptionPane.showMessageDialog(null,
         "Discover Multithreaded Programming.");
   }
}
```

The program above is a good example of using the command pattern for queuing work to be executed by another thread. The thread taking work off the queue and performing it knows nothing about the details of the work queued.

Discussion

A common problem often solved with the Command pattern is allowing undo for user actions in an interactive application.

Most interactive applications support some form of undo. For example, Microsoft Word allows users to undo the last action or series of actions performed on a document:

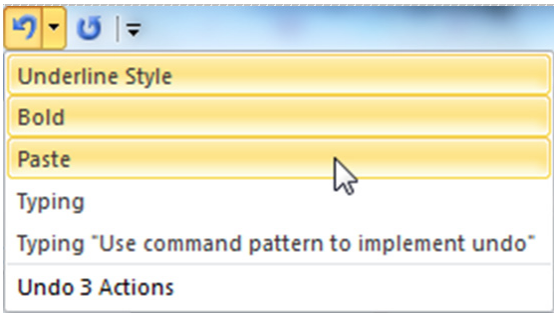
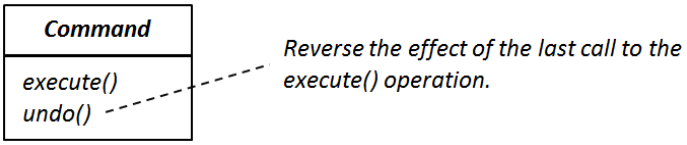


Figure x. Multi-level undo in Microsoft Word

Undo is a valuable feature with interactive applications because it encourages safe exploration [Designing Interfaces, page 11]. When users have the ability to undo actions, they feel more comfortable with trying new features and experimenting with different options.

Adding support for undo requires minimal extra effort when actions are implemented as command objects. The first step is adding an `undo()` or `unexecute()` operation to the abstract command interface:



The undo operation will be implemented in concrete command objects, but in order to implement `undo()` the `execute()` method of the command will most likely have to be updated first to store the

additional state information needed to undo the command. For example, in order to undo a debit command, the `execute()` method of the command will need to save the amount debited.

The following program shows an example of one level undo for the simple text editor introduced in the previous section. The following changes were made to the original program:

1. The `undo()` operation was added to the abstract command interface and implemented in each concrete command.
2. The `execute()` method of each concrete command was modified to save state information needed to reverse the effect of the command.
3. `FemtoEditor` was updated to keep track of the last command executed and accept an input of 'u' from the user in order to undo the previous command. When the user enters 'u', `FemtoEditor` calls the `undo()` operation for the previously executed command.

Here is the complete program:

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.ArrayList;

public class Client {
   public static void main(String[] args) throws IOException {
      StringBuffer document = new StringBuffer("");
      FemtoEditor editor = new FemtoEditor(document);

      Command c;
      c = new UpperCaseCmd(document);
      editor.addCmd(c);
      c = new AppendCmd(document);
      editor.addCmd(c);
      c = new DeleteCmd(document);
      editor.addCmd(c);

      editor.run();
   }
}

interface Command {
      void execute();
      void undo();
}

class UpperCaseCmd implements Command {
   private StringBuffer document;
   private String saveDocumentState;

   public UpperCaseCmd(StringBuffer document) {
      this.document = document;
   }
```

```
   @Override
   public void execute() {
      saveDocumentState = document.toString();
      for (int i = 0; i < document.length(); i++)
         if (Character.isLetter(document.charAt(i)))
            document.setCharAt(i, Character.toUpperCase(document.charAt(i)));
   }

   @Override
   public String toString() {
      return "Make upper case";
   }

   @Override
   public void undo() {
      document.replace(0, saveDocumentState.length(), saveDocumentState);

   }
}

class AppendCmd implements Command {
   private StringBuffer document;
   private int lengthOfLastAppend;

   public AppendCmd(StringBuffer document) {
      this.document = document;
   }

   @Override
   public void execute() {
      BufferedReader br = new BufferedReader(
         new InputStreamReader(System.in));
      System.out.print("append> ");
      try {
         String stringToAppend = br.readLine();
         document.append(stringToAppend);
         lengthOfLastAppend = stringToAppend.length();
      } catch (IOException e) { }
   }

   @Override
   public String toString() {
      return "Append string";
   }

   @Override
   public void undo() {
      int end = document.length();
      int start = end - lengthOfLastAppend;
      document.delete(start,end);
   }
}

class DeleteCmd implements Command {
   private StringBuffer document;
   private String saveDocumentState;
```

```
   public DeleteCmd(StringBuffer document) {
      this.document = document;
   }

   @Override
   public void execute() {
      saveDocumentState = document.toString();
      document.delete(0, document.length());
   }

   @Override
   public String toString() {
      return "Delete";
   }

   @Override
   public void undo() {
      document.replace(0, saveDocumentState.length(), saveDocumentState);
   }
}

class FemtoEditor {
   private ArrayList<Command> cmds = new ArrayList<Command>();
   private StringBuffer document;

   public FemtoEditor(StringBuffer document) {
      this.document = document;
   }
   public void addCmd(Command c) {
      cmds.add(c);
   }

   public void run() throws IOException {
      Command previousCommand = null;
      BufferedReader br = new BufferedReader(
         new InputStreamReader(System.in));

      for (;;) {
         System.out.println("\nText: " + document + "\n");

         System.out.println(
            "Select an edit command or 'u' to undo last command:\n");

         for (int i = 0; i < cmds.size(); i++) {
            System.out.println((i + 1) + ". " + cmds.get(i));
         }

         System.out.print("\nmenu choice> ");
         String input = br.readLine();

         if ("u".equals(input.trim())) {
            if (previousCommand != null ) {
               previousCommand.undo();
               previousCommand = null;
            }
         }
         else {
```

```
            int i = Integer.parseInt(input);
            Command c = cmds.get(i - 1);
            c.execute();
            previousCommand = c;
         }
      }
   }
}
```

Allowing muli-level undo in a program is a bit more complicated as the program needs to maintain a list of commands executed. If redo functionality is needed, previously executed commands must be stored in a list that can be traversed both forwards and backwards. Otherwise, a stack data structure is sufficient to store previously executed commands.

In addition, it may be necessary to make a copy of each command before placing it on the list of previously executed commands. Commands that save state and are reused must be copied before placing them on a list of previously executed commands. Otherwise, the state of a saved command would be wiped out the next time it was reused.

The following program demonstrates muli-level undo in the context of `FemtoEditor`. The following changes were made to the one-level undo version of the program:

1. `FemtoEditor` was updated to store previously executed commands on a stack. Because commands have state and are reused, a copy of each command executed is pushed onto the stack. When the user enters 'u', `FemtoEditor` pops the most recently added command from the stack and calls its `undo()` operation.
2. The `copy()` operation was added to the abstract command interface and implemented in each concrete command. The implementation of `copy()` is somewhat atypical. Each command holds a reference to the document being edited and state needed to undo the operation. The reference to the document is copied using a shallow copy and the reference to the state is copied using a deep copy.

Here is the complete program:

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.ArrayList;
import java.util.Stack;

public class Client {
   public static void main(String[] args) throws IOException {
      StringBuffer document = new StringBuffer("");
      FemtoEditor editor = new FemtoEditor(document);

      Command c;
      c = new UpperCaseCmd(document);
      editor.addCmd(c);
      c = new AppendCmd(document);
      editor.addCmd(c);
```

```
      c = new DeleteCmd(document);
      editor.addCmd(c);

      editor.run();
   }
}

interface Command {
   void execute();
   void undo();
   Command copy();
}

class UpperCaseCmd implements Command {
   private StringBuffer document;
   private String saveDocumentState;

   public UpperCaseCmd(StringBuffer document) {
      this.document = document;
   }

   @Override
   public void execute() {
      saveDocumentState = document.toString();
      for (int i = 0; i < document.length(); i++)
         if (Character.isLetter(document.charAt(i)))
            document.setCharAt(i, Character.toUpperCase(document.charAt(i)));
   }

   @Override
   public String toString() {
      return "Make upper case";
   }

   @Override
   public void undo() {
      document.replace(0, saveDocumentState.length(), saveDocumentState);
   }

   @Override
   public Command copy() {
      // Make a shallow copy of document.
      UpperCaseCmd aCopy = new UpperCaseCmd(this.document);
      aCopy.saveDocumentState = new String(this.saveDocumentState);
      return aCopy;
   }
}

class AppendCmd implements Command {
   private StringBuffer document;
   private int lengthOfLastAppend;

   public AppendCmd(StringBuffer document) {
      this.document = document;
   }
```

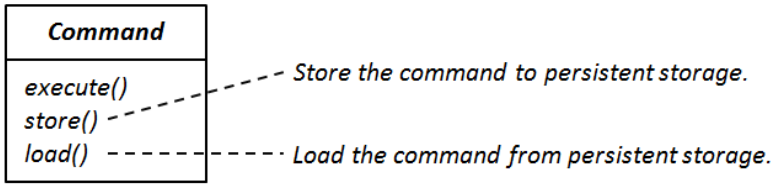
```
    @Override
    public void execute() {
        BufferedReader br = new BufferedReader(new
        InputStreamReader(System.in));
        System.out.print("append> ");

        try {
            String stringToAppend = br.readLine();
            document.append(stringToAppend);
            lengthOfLastAppend = stringToAppend.length();
        } catch (IOException e) { }
    }

    @Override
    public String toString() {
        return "Append string";
    }

    @Override
    public void undo() {
        int end = document.length();
        int start = end - lengthOfLastAppend;
        document.delete(start,end);
    }

    @Override
    public Command copy() {
        AppendCmd aCopy = new AppendCmd(this.document);
        aCopy.lengthOfLastAppend = this.lengthOfLastAppend;
        return aCopy;
    }
}

class DeleteCmd implements Command {
    private StringBuffer document;
    private String saveDocumentState;

    public DeleteCmd(StringBuffer document) {
        this.document = document;
    }

    @Override
    public void execute() {
        saveDocumentState = document.toString();
        document.delete(0, document.length());
    }

    @Override
    public String toString() {
        return "Delete";
    }

    @Override
    public void undo() {
        document.replace(0, saveDocumentState.length(), saveDocumentState);
    }
```

```
   @Override
   public Command copy() {
      DeleteCmd aCopy = new DeleteCmd(this.document);
      aCopy.saveDocumentState = new String(this.saveDocumentState);
      return aCopy;
   }
}

class FemtoEditor {
   private ArrayList<Command> cmds = new ArrayList<Command>();
   private StringBuffer document;

   public FemtoEditor(StringBuffer document) {
      this.document = document;
   }

   public void addCmd(Command c) {
      cmds.add(c);
   }

   public void run() throws IOException {
      Stack<Command> previousCommands = new Stack<Command>();
      BufferedReader br = new BufferedReader(new
         InputStreamReader(System.in));
      for (;;) {
         System.out.println("\nText: " + document + "\n");
         System.out.println("Select an edit command or 'u' to undo last
         command:\n");

         for (int i = 0; i < cmds.size(); i++) {
            System.out.println((i + 1) + ". " + cmds.get(i));
         }

         System.out.print("\nmenu choice> ");
         String input = br.readLine();
         if ("u".equals(input.trim())) {
            if (!previousCommands.isEmpty()) {
               Command mostRecentCommand = previousCommands.pop();
               mostRecentCommand.undo();
            }
         }
         else {
            int i = Integer.parseInt(input);
            Command c = cmds.get(i - 1);
            c.execute();
            previousCommands.push(c.copy());
         }
      }
   }
}
```

The command pattern can also be used to recover changes made to a large internal data structure after a system crash. For example, suppose you have an inventory control system that keeps track of parts on hand. Keeping the data structure in memory would improve performance but makes the data vulnerable

to a system crash. You could save the data structure to persistent storage after every update but that would have a negative impact on performance. Another option is to (1) model transactions on the data structure as commands, (2) take a checkpoint of the data structure at startup and (3) save to persistent storage the commands as they are applied to the internal data structure. Then, in the event of a system crash, the data structure can be restored by loading the original checkpoint and reexecuting the saved commands.

Command logging requires two new methods on the command interface:



The following program shows an example of command logging. An instance of StringBuffer is used as a proxy for a large internal data structure. Object serialization is used as the persistence mechanism.

```
import java.io.*;

// The receiver object was made global
//    for the convenience of this example
class Document {
   public static StringBuffer largeDataStructure = new StringBuffer(
      "This string is a proxy for a very large dynamic\n" +
      "internal data structure. Saving the complete data\n" +
      "structure after every update would be prohibitively\n" +
      "expensive. Instead, updates will be modeled as\n" +
      "commands, and after an initial checkpoint of the\n" +
      "data structure, just the commands will be saved. In\n" +
      "the event of a system crash, the saved commands will\n" +
      "be reloaded and reexecuted against the original\n" +
      "checkpoint.\n" +
      "Updates: 0");
}

public class LoggingExample {

   public static void main(String[] args) throws Exception {

      ObjectOutputStream os = new ObjectOutputStream (
         new FileOutputStream("checkpoint.data"));

      // Save initial state of data structure
      os.writeObject(Document.largeDataStructure);

      // Apply and save transactions
      for(int i = 1; i<=3; i++) {
         Command transaction = new UpdateTransaction(", " + i);
         transaction.execute();
         transaction.store(os);
      }
```

```
      os.close();

      // Simulate system crash
      Document.largeDataStructure = null;

      // Recover initial state of data structure and
      //   reexecute saved transactions
      ObjectInputStream is = new ObjectInputStream (
         new FileInputStream("checkpoint.data"));

      Document.largeDataStructure = (StringBuffer) is.readObject();

      for(int i = 1; i<=3; i++) {
         Command transaction = Command.load(is);
         transaction.execute();
      }

      System.out.print(Document.largeDataStructure);
   }
}

abstract class Command implements Serializable {
   public abstract void execute();
   public void store(ObjectOutputStream os) throws IOException {
      os.writeObject(this);
   }

   public static Command load(ObjectInputStream is) throws Exception {
      return (Command) is.readObject();
   }
}

class UpdateTransaction extends Command {
   private String stringToAppend;

   public UpdateTransaction(String stringToAppend) {
      this.stringToAppend = stringToAppend;
   }

   @Override
   public void execute() {
      Document.largeDataStructure.append(stringToAppend);
   }
}
```

Output:

```
This string is a proxy for a very large dynamic
internal data structure. Saving the complete data
structure after every update would be prohibitively
expensive. Instead, updates will be modeled as
commands, and after an initial checkpoint of the
data structure, just the commands will be saved. In
the event of a system crash, the saved commands will
```

```
be reloaded and reexecuted against the original
checkpoint.
Updates: 0, 1, 2, 3
```

Related Patterns

Sometimes it is useful to treat a sequence of commands as a single command. For example, applying a paragraph style in a word processor might change the selected paragraph's line spacing, indentation, font type, character style etc. Assuming there are existing commands for changing line spacing, indentation, etc., the command for applying a paragraph style could be modeled as a sequence of existing commands. A sequence of commands is sometimes called a MacroCommand. A MacroCommand is an example of the Composite pattern.

The design of a MacroCommand would look something like:

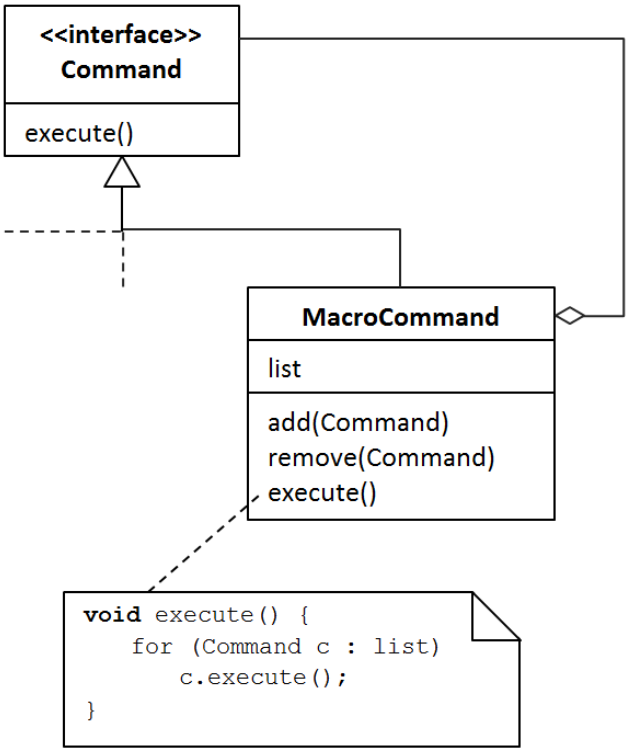


Figure x. Composite pattern being used to implement a MacroCommand