

---

## Reading

---

1. Mitchell, Chapter 12.

---

## Problems

---

1. (40 points) ..... Undoable Commands

The goal of this problem is to implement the core data structures of a text editor using the *Command Design Pattern*.

**Text Editor** In essence, a text editor manages a character sequence that can be changed in response to commands issued by the user, such as inserting new text or deleting text. Typically, these commands operate on the underlying character buffer at the current position of the cursor. Thus, if the cursor is positioned at the beginning of the buffer, typing the string “moo” will cause those letters to be inserted at the start of the buffer, and so on. This question explores the internal design of a simple editor.

Most text editors involve a GUI and the user issues commands to the editor by keyboard and mouse events. For us, however, the most interesting part of a text editor’s is what happens behind the scenes. Therefore, our text editor will just be a simple command line program that prompts you for edit commands. The program will print the contents of the text editor’s buffer, including a “^” to indicate the current cursor position, print the prompt “?”, and then wait for you to enter a command. (At one point in time, this was in fact how many text editors worked...) The following shows one run of the editor:

Sample Execution	Description
Buffer: ^	<i>Buffer is initially empty, cursor at start</i>
? I This is a test. Buffer: This is a test. ^	<i>Insert “This is a test.” and move cursor to immediately after inserted text</i>
? < 9 Buffer: This is a test. ^	<i>Move cursor 9 characters left</i>
? > Buffer: This is a test. ^	<i>Move cursor 1 character right</i>
? I n’t Buffer: This isn’t a test. ^	<i>Insert “n’t”</i>
? > 3 Buffer: This isn’t a test. ^	<i>Move cursor 3 characters right</i>
? D 4 Buffer: This isn’t a . ^	<i>Delete 4 characters.</i>
? I cow Buffer: This isn’t a cow. ^	<i>Insert “cow”</i>
? Q	<i>Quit</i>

Here is a summary of all available editor commands (including some described below). The term *[num]* indicates an optional number.

Command	Description
I <i>text</i>	Insert <i>text</i> at the current cursor, moving cursor to after the new text.
D <i>[num]</i>	Delete <i>num</i> characters to the right of cursor position. (If <i>num</i> is missing, delete 1 character.)
< <i>[num]</i>	Move the cursor <i>num</i> characters to the left. (If <i>num</i> is missing, move 1 character.)
> <i>[num]</i>	Move the cursor <i>num</i> characters to the right. (If <i>num</i> is missing, move 1 character.)
Q	Quit
U	Undo the previous edit command
P	Print the history of edit commands
R	Redo an undone edit command

I have provided a working program for all but the last three commands. Your job is to change `TextEditor` to support multiple levels of undo and redo using the *Command Design Pattern*.

Figure 1 shows an example that uses “U” (undo) and “P” (print history). (We’ll look at “R” (redo) at the very end of the problem.) Notice that you can undo multiple edits, not simply the last one. To support this, the text editor must keep track of an edit command history that permits you to undo as many commands from the history as you like. Undoing all commands will lead you all the way back to the original empty buffer.

The starter code for this problem is divided into two classes:

- **Buffer:** This class manages the internal state of the editor’s buffer (ie, character sequence and current cursor location), and it supports commands for getting/setting the cursor location and for inserting/deleting text. Refer to the javadoc on the handouts page for more details. *You should not change this class.*
- **TextEditor:** This class stores a `Buffer` named `buffer`. The `processOneCommand()` method reads in a command from the user and performs the appropriate operation on `buffer` by invoking one of the following methods:
  - `protected void setCursor(int loc)`
  - `protected void insert(String text)`
  - `protected void delete(int count)`
  - `protected void undo()`
  - `protected void printHistory()`

These methods are all quite simple. For example, the `insert` method simply inserts the text into `buffer` and repositions the cursor:

```
protected void insert(String text) {
    buffer.insert(text);
    buffer.setCursor(buffer.getCursor() + text.length());
}
```

**The EditCommand Class** To support undo, we first change the way the `TextEditor` operates on the underlying `buffer`. Rather than changing it directly, the `TextEditor` constructs `EditCommand` objects that know how to perform the desired operations and — more importantly — know how to undo those operations. All `EditCommand` objects will be derived from the `EditCommand` abstract class:

Sample Execution	Description
<pre> Buffer: ^ ? I Hello Buffer: Hello ^ ? &lt; 2 Buffer: Hello ^ ? D 2 Buffer: Hel ^ ? I p Buffer: Help ^ ? U Buffer: Hel ^ ? I ium Buffer: Helium ^ ? P History:   [Insert "Hello"]   [Move to 3]   [Delete 2]   [Insert "ium"] Buffer: Helium ^ ? U Buffer: Hel ^ ? U Buffer: Hello ^ ? P History:   [Insert "Hello"]   [Move to 3] Buffer: Hello ^ ? Q </pre>	<p><i>Undo the previous command.</i></p> <p><i>Print the command history.</i></p> <p><i>Undo the last command ([Insert "ium"]).</i></p> <p><i>Undo the last command ([Delete 2]).</i></p> <p><i>Print the command history.</i></p>

Figure 1: Sample run of the text editor with undo.

```

public abstract class EditCommand {
    /** buffer to operate on */
    protected Buffer target;

    public EditCommand(Buffer target) {
        this.target = target;
    }

    /** Perform the command on the target buffer */
    public abstract void execute();

    /** Undo the command on the target buffer */
    public abstract void undo();

    /** Print out what this command represents */
    public abstract String toString();
}

```

Here, the `execute()` method carries out the desired operation on the target buffer, and `undo()` would perform the inverse operation. For example, to make insert undoable, the first step would be to define an `InsertCommand` class:

```

public class InsertCommand extends EditCommand {

    protected String text;

    public InsertCommand(Buffer target, String text) {
        super(target);
        this.text = text;
    }

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

```

The `TextEditor` would then perform code like the following inside `insert`:

```

protected void insert(String text) {
    EditCommand command = new InsertCommand(buffer, text);
    command.execute();
    ...
}

```

Assuming `InsertCommand` is implemented properly, the insertion would happen as before. However, the `TextEditor` can now remember that the last operation performed was the `InsertCommand` we created, and we can undo it simply by calling that object's `undo()` method. In essence, an `EditCommand` object describes one modification to a `Buffer`'s state and how to undo that modification. Supporting undo is then as simple as writing a new kind of `EditCommand` object for each type of buffer modification you support.

And of course, to implement multiple levels of undo, you need to keep track of more than just the last command object created...

**Implementation Strategy** I suggest tackling the implementation the following steps:

- (c) Download the starter code from the handouts page. Compile the Java files with the command `javac *.java` as usual. I have added some `assert` statements to the `Buffer` class to aid in debugging. To enable assertion checking, run the program with the command
- ```
java -ea TextEditor
```
- You may find it useful to add similar asserts to your own code as well. (assert statements are new in Java 1.5. See the online documentation on the links page for more details.)
- (b) Implement `InsertCommand`, `DeleteCommand`, and `MoveCommand` subclasses of `EditCommand`. For each one, you must define: 1) `execute()`, 2) `undo()`, and 3) `toString()`. I recommend holding off on `undo()` for the moment. Change `TextEditor` to create and execute edit command objects appropriately.
- (c) Extend `TextEditor` to remember the last command executed, and change `TextEditor.undo()` to undo that command. Go back and implement `undo` for each type of `EditCommand`.
- (d) Once a single level of `undo` is working, extend `TextEditor` to support undoing multiple previous commands. Specifically, change `TextEditor` to maintain a history of commands that have been executed and not undone. Also implement the `printHistory()` method to aid in debugging. Your program should simply ignore `undo` requests if there are no commands are in the history. You are free to use any Java libraries you like in your implementation.
- (e) The last task is to implement redo. Specifically, if you undo one or more commands but have not yet performed any new operations on the buffer, you can redo the commands you undid:

| Sample Execution           | Description                      |
|----------------------------|----------------------------------|
| Buffer: hello<br>^         |                                  |
| ? D l<br>Buffer: helo<br>^ |                                  |
| ? D l<br>Buffer: hel<br>^  |                                  |
| ? U<br>Buffer: helo<br>^   | <i>Undo delete of "o"</i>        |
| ? U<br>Buffer: hello<br>^  | <i>Undo delete of "l"</i>        |
| ? R<br>Buffer: helo<br>^   | <i>Redo delete of "l"</i>        |
| ? R<br>Buffer: hel<br>^    | <i>Redo delete of "o"</i>        |
| ? I p<br>Buffer: help<br>^ |                                  |
| ? U<br>Buffer: hel<br>^    | <i>Undo insert of "p"</i>        |
| ? U<br>Buffer: helo<br>^   | <i>Undo redone delete of "o"</i> |

Note that redoing undone commands is no longer possible if the buffer is changed in any way. For example, if you insert text after undoing some command *E*, you should no longer be able to redo command *E*:

| Sample Execution | Description                     |
|------------------|---------------------------------|
| Buffer: ^        |                                 |
| ? I moo          |                                 |
| Buffer: moo ^    |                                 |
| ? U              |                                 |
| Buffer: ^        |                                 |
| ? I hello        | <i>Change buffer after undo</i> |
| Buffer: hello ^  |                                 |
| ? R              | <i>Redo will have no effect</i> |
| Buffer: hello ^  |                                 |

Also, redone commands should be able to be subsequently undone:

| Sample Execution    | Description                 |
|---------------------|-----------------------------|
| Buffer: ^           |                             |
| ? I 334             |                             |
| Buffer: 334 ^       |                             |
| ? I cow             |                             |
| Buffer: 334cow ^    |                             |
| ? I moo             |                             |
| Buffer: 334cowmoo ^ |                             |
| ? U                 | <i>Undo insert of "moo"</i> |
| Buffer: 334cow ^    |                             |
| ? U                 | <i>Undo insert of "cow"</i> |
| Buffer: 334 ^       |                             |
| ? R                 | <i>Redo insert of "cow"</i> |
| Buffer: 334cow ^    |                             |
| ? R                 | <i>Redo insert of "moo"</i> |
| Buffer: 334cowmoo ^ |                             |
| ? U                 | <i>Undo insert of "moo"</i> |
| Buffer: 334cow ^    |                             |
| ? U                 | <i>Undo insert of "cow"</i> |
| Buffer: 334 ^       |                             |
| ? U                 | <i>Undo insert of "334"</i> |
| Buffer: ^           |                             |
| ? R                 | <i>Redo insert of "334"</i> |
| Buffer: 334 ^       |                             |
| ? R                 | <i>Redo insert of "cow"</i> |
| Buffer: 334cow ^    |                             |
| ? U                 | <i>Undo insert of "cow"</i> |
| Buffer: 334 ^       |                             |

Extend `TextEditor` to support multiple levels of redo. You should not need to change any class other than `TextEditor` to implement this feature.

(f) Turn in your code using `turnin` as in the previous homeworks.

There are many extensions that would make our editor more “realistic”. One idea is listed below as an extra credit problem. It should not require more than a few additional lines of code and really highlights the elegance and simplicity of adopting this design pattern.

## 2. (30 points) ..... Assignment and Derived Classes

Mitchell, Problem 12.1

I put a working version of the code on the handouts page if you would like to experiment with it. Use `g++` to compile the program, and then run the executable `a.out` with the command `./a.out`.

## 3. (10 points) ..... Function Subtyping

Mitchell, Problem 12.3

## 4. (15 points) ..... Subtyping and Visibility

Mitchell, Problem 12.6

## 5. (15 points) ..... Dispatch on State

Mitchell, Problem 12.12

## 6. (10 points) ..... Printing

Programmers often want to print user-defined object types in a reader-friendly way to output streams (like `System.out` in Java). For example, we may wish to print a `Point` object `p` as `“(3, 4)”`. There are many ways to design a programming system to facilitate this.

(a) One not-so-great way is to have the stream class provide methods to print each kind of object. For example, the `OutputStream` class for whatever language we are using could be defined to have a `println(Point p)` method to facilitate writing the user-defined `Point` class to an output stream, and similar methods for other object types. What is the deficiency with this approach?

(b) In C++, this problem of writing user-defined types in a reasonable way is solved through operator overloading. C++ stream classes use the `<<` operator to write to streams. For example, `cout << 4` writes the number 4 to the terminal. To define how user-defined types are written to streams, one defines a new function like the following:

```
class Point {
    int x, y;
    ...
};

ostream& operator<<(ostream& o, Point p) {
    return o << "(" << p.x << ", " << p.y << ")" ;
}
```

With this definition, executing “cout << p”, would print “(3,4)” if p were a Point object with those coordinates.

Java does not use operator overloading. The method

```
public void println(Object o) { ... }
```

from `java.io.PrintStream` permits one to call “`System.out.println(o)`” on any object `o` and print its representation. How does the language and library ensure that this method can print different types of objects in the appropriate way? What methods must the programmer write, how are classes structured, and what language features are involved?

(c) Could the designers have taken the same approach in C++? Would that approach fit the C++ design criteria?

7. (10 points) ..... Subtyping and Exceptions

Mitchell, Problem 13.3

8. (15 points) ..... (Bonus Question) C++ Multiple Inheritance and Casts

Mitchell, Problem 12.10

9. (15 points) ..... (Bonus Question) Composable Commands

Here is one interesting extension to the basic Text Editor.

Most of the time, two consecutive commands of the same type are lumped together into a single command. Thus, if I type “hello” followed immediately by “there” into an editor (such as emacs), the editor lumps them together into a single insertion command that removes all of “hello there” from the buffer when undone. Similarly, if I perform two cursor movement commands in a row, that is recorded in the undo history as a single command. Here is an example:

| Sample Execution                                           | Description                              |
|------------------------------------------------------------|------------------------------------------|
| Buffer: ^                                                  |                                          |
| ? I hel                                                    |                                          |
| Buffer: hel ^                                              |                                          |
| ? I ium                                                    | <i>second insert composed with first</i> |
| Buffer: helium ^                                           |                                          |
| ? P                                                        |                                          |
| History:<br>[Insert "helium"]                              |                                          |
| Buffer: helium ^                                           |                                          |
| ? U                                                        |                                          |
| Buffer: ^                                                  |                                          |
| ? R                                                        |                                          |
| Buffer: helium ^                                           |                                          |
| ? <                                                        |                                          |
| Buffer: helium ^                                           |                                          |
| ? < 2                                                      | <i>second move composed with first</i>   |
| Buffer: helium ^                                           |                                          |
| ? D 2                                                      |                                          |
| Buffer: helm ^                                             |                                          |
| ? D 1                                                      | <i>second delete composed with first</i> |
| Buffer: hel ^                                              |                                          |
| ? P                                                        |                                          |
| History:<br>[Insert "helium"]<br>[Move to 3]<br>[Delete 3] |                                          |
| Buffer: hel ^                                              |                                          |
| ? I p                                                      |                                          |
| Buffer: help ^                                             |                                          |
| ? U                                                        | <i>undo insert of "p"</i>                |
| Buffer: hel ^                                              |                                          |
| ? U                                                        | <i>undo composed delete command</i>      |
| Buffer: helium ^                                           |                                          |
| ? U                                                        | <i>undo composed move command</i>        |
| Buffer: helium ^                                           |                                          |
| ? U                                                        | <i>undo composed insert of "helium"</i>  |
| Buffer: ^                                                  |                                          |

Implement composable commands by extending the `EditCommand` class and its subclasses to define the following method:

```
public boolean compose(EditCommand other)
```

This method either:

- returns `false` if the current command cannot be composed with `other`.
- returns `true` if the current command can be composed with `other`, because, for example, they are both insert commands. In this case, the method should also change the current command to be the composed command.

For example,

```
EditCommand command = new InsertCommand(target, "hel");  
boolean test = command.compose(new InsertCommand(target, "lo"));
```

would set `test` to `true` and change `command` so that `command.execute()` would insert “hello” into the target. You may find it useful to test whether an object has a certain type, which can be done in Java with a boolean test of the form “`other instanceof InsertCommand`”.