

# Homework # 6

## due Tuesday, December 4th, 12:30 PM

(As with previous homeworks, undergraduate students may do the work jointly in groups of 2, if desired. Discussions between any students is always permitted when credit for ideas is given.)

In this homework, you will identify design patterns and also provide some more extensions to our reservation program.

## 1 Design Patterns

The textbook has introduced the following design patterns:

- SINGLETON
- DECORATOR
- PROTOTYPE
- TEMPLATE METHOD
- STATE
- BUILDER
- STRATEGY
- FACTORY METHOD
- COMMAND
- ITERATOR
- (ABSTRACT)  
FACTORY
- ADAPTER
- COMPOSITE

Go through the solution to Homework #5, and find instances of each of these design patterns. For each design pattern, specify whether there are zero, one, or multiple instances, and give the *specific* UML for the pattern (if there are multiple instances, just give one specific UML diagram).

For any design pattern with zero instances, give at least one of the following:

- A refactoring (or sequence of refactorings) that could introduce the design pattern usefully.
- A good reason why the design pattern doesn't occur.
- An example extension that could profitably use the design pattern, and explain why.

## 2 Reading/Writing XML

Change the program so that it reads and writes XML of the following format:

```
<?xml version='1.0'?>
<RoomRes>
  <Room name="name" capacity="capacity">
    <Reservation>
      <Period day="DAYNAME" start="24 hour time" duration="number" />
      Description
    </Reservation>
  </Room>
</RoomRes>
```

```

    </Reservation>
    ...
  </Room>
</RoomRes>

```

It should also be able to read files in the original format.

### 3 Infinite Undo

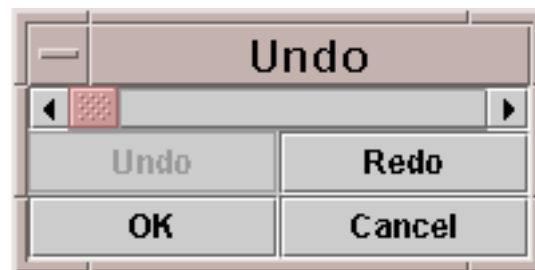
If you have used “emacs” you may be aware of one of its interesting features: you can undo any command, all the way back to the point the editor started up.

You should add an “Undo...” option to the top of the edit menu (separated by a line). This action should bring up an Undo dialog box with four buttons arranged in a square: Undo, Redo, Ok, Cancel. At first Ok and Redo are grayed out. Pressing Undo should undo

Figure 1: Starting undo dialog; Undo dialog in process.



Figure 2: Having undone everything.



the latest change (and make Ok and Redo pressable). Pressing Undo again will undo the previous action (if any), etc. Redo will undo the latest Undo. If you press Undo enough times, you come back to the point the program was started, and the Undo button should be grayed. Similarly if you redo everything undone, the Redo button should be grayed. Optionally you may provide a status bar to show the user how far back they are going. The undo and redo actions should have effect immediately on all views currently displaying. The Ok button stops the process where it is. The Cancel button redoes everything to the point where the Undo menu option was chosen.

Once the Undo dialog is completed (using Ok or Cancel), the undone actions should be added as actions themselves, so they can be undone themselves (or redone!). For example, suppose after starting the program we perform the following actions:

1. Add a one-hour reservation in EMS E270 on Friday at 9:00am “Lab Help.”
2. Change the reservation to 8:00am.
3. Change the room to EMS E280.
4. Remove it.

Then if bring up the Undo dialog and undo twice, and then press Ok, we will be back at the point where we have an 8am reservation in EMS E270. If we then bring up the Undo dialog again, pressing Undo multiple times will have the effect:

1. Undo Undo of changing room to EMS E280. In other words, change room to EMS E280.
2. Undo Undo of removing it. In other words, remove it.
3. Undo removal. Put it back.
4. Undo change room to EMS E280. Change room back to EMS E270.
5. Undo change time of reservation; change back to 9am.
6. Undo adding the reservation; remove it. (No more undo's possible.)

The following aspects of “Undo” can be tricky:

- If you add a reservation that is already there or if you remove one that isn't there, nothing happens. You must ensure that either (1) no action is logged, or (2) that undoing this NOP does nothing either.
- Undoing a “Change” action should cause the removal of the “new” reservation and the reinstatement of the old one in a single operation.
- Undoing a “Open” action should remove all the reservations that were added (and none of the ones that were already there) when the file was read. It should not require piecemeal removal.

The cleanest solution to the latter problems is to implement a generic “compound action” class.

**Exception!** The three “dangerous” operations (New, Clear and Quit) do not need to be undone. The warning message should mention this. If one of these operations happens, the undo log should be cleared. (The **Clear** operation can be made to work with Undo, but the others are fundamentally not undoable.)

## 4 Design Recommendations

You will find it easiest to implement “Undo” if you make reservations immutable (at least as far as the program is concerned). The solution already does that, in that changing a reservation actually causes a new one to be substituted. There is one case where a reservation is mutated (find out by removing the mutators and seeing where the error appears). Change it so that it creates a new reservation instead of overwriting an existing one. Then each state-changing action is either the addition or removal of a reservation, or a compound action made up of other actions.

You should design the code with the least possible coupling between the “undo” facility (which should be generic and live in its own package) and the rest of the system. The core undo system should be unaware of the Undo Dialog, which should itself be unaware of any of the scheduling classes. Your Homework grade will depend partly on this loose coupling.

For XML writing, you may find it useful to create a subclass of `PrintWriter` that has specialized routines for writing XML tags. For XML reading, you must use the SAX parser infrastructure.

## 5 Files

The solution files for Homework #5 are in

```
/afs/cs.uwm.edu/users/classes/cs552/solutions/homework5/
```

You may use our solution, or work with your own solution. However, your code must work at least as well as our solution. All your files should be in your `homework6` directory.

The programs must be runnable from the command line. In other words, on a grid, it must be possible to

```
grid.cs: cd homework6
grid.cs: cd bin                (optionally)
grid.cs: java Driver test.rml
```

The extra weekend is given because of the Thanksgiving holiday. Please make use of the time! Homework #7 will be due during finals.