

# Homework # 5

## Refactoring and Extensions

### due Thursday, November 15, 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.)

## 1 Change Log

In this homework, you will be making extensions to the GUI room reservation program. Extension is often easier after refactoring. In this assignment it is necessary that you use refactoring as much as possible *when it assists the immediate task at hand* which may be extension of features or removal of defects.

Keep an ASCII file `ChangeLog` (no DOC files, please) in the `homework5` directory that lists *all* changes that you make. It should *not* just list fields or methods changed; rather it should explain what the change accomplishes in overall terms. If your change requires more than three sentences to explain, you need to break it up into smaller changes. After each change (whether or not it is a refactoring), it should be possible to run the program.

Each entry in the log should thus have the following format:

**Number** An ordering number; it may be decimal. The number should increase for each change.

**DESCRIPTION** The description of the change

**REFACTORING?** Whether or not this is a refactoring and why. If it is a refactoring, say what the refactoring is. Possible names include

Change Reference to Value (Object with identity to immutable object), Change Value to Reference, Collapse hierarchy, Duplicate Observed Data (pull data out of GUI into domain object which is then “observed”), Encapsulate Collection, Encapsulate Delegate, Encapsulate Downcast, Encapsulate Field, Extract (delegate) Class (from a set of fields), Extract Interface/Superclass/Subclass, Extract Local Variable, Extract Method, Inline X (opposite of Extract X), Introduce Null Object (default object to replace a null reference), Introduce Parameter Object (collect parameters into an object), Move Field or Method (to delegate or back), Parameterize Method (Extract a new method with extra parameters that replaces two or more methods), Pull Up / Push Down Method/Field/ConstructorBody (Move things in the hierarchy), Rename (method, field or class), Replace Constructor with Factory Method, Replace Method with Method Object, Replace Inheritance with Delegation (and vice versa)

It’s not essential you get the name exactly right. These names mainly come from the book “Refactoring” by Fowler, who lists many more. You may also make up your own name (with additional explanation). Also explain whether you were able to use Eclipse refactoring menu to accomplish it.

**RUN CHECK** Whether the code runs after the change. If the change is very small, it is permitted to defer the run-check. Be honest. If something is broken, explain what happened and fix it. Describe the fix in the next change entry.

For example:

...

1.2 Make room name mutable

DESCRIPTION: make field non-final

REFACTORING? Yes: Make field mutable (new name)

RUN-CHECK: deferred to next change

### 1.3 Remove while renaming

DESCRIPTION: add setter for room that removes it from the container of rooms while changing the name

REFACTORING? No

RUN-CHECK: Yes, not surprisingly (new functions not used)

## 2. Prepare rooms for observation // a header for later changes

### 2.1 Use TreeMap

DESCRIPTION Put rooms in a `TreeMap` rather than a `HashMap`

REFACTORING? Yes, almost. Substitute `Container` (made-up name)

RUN-CHECK: Yes

Do **NOT** wait to write this log after you have changed the program! Always write the description and whether it is a refactoring first; then do the change, and then check to see if the program runs (and fix bugs). Part of this homework is figuring ways to change the software in small steps. Your log must include at least 10 different refactorings. You will be graded partly on

- whether the log shows understanding that refactorings are non-functional changes that make software more flexible, usually in preparation for functional changes;
- how well your refactorings were documented.

## 2 Structural Changes

This section describes some of the mostly structural (non-visible) changes that are required for this homework.

### 2.1 Reservations structure

The previous Homework required that reservations be stored in a sorted set. However, the recommended data structure (tree set) is awkward and inefficient to use with the `JList` widget. One of the problems is that it is difficult to keep track of all the places changes happen. Thus `Reservations` should no longer permit modifications (adds or removes) through subsets or iterators, but only using `add`, `addAll`, `remove`, `removeAll`, `retainAll`, or `clear`. Furthermore, a `Reservations` instance should permit “observers” to be added or removed. Each observer must implement a new interface `ReservationsObserver` which has three required methods:

**reservationAdded(Reservations,Reservation)** Called after every addition.

**reservationRemoved(Reservations,Reservation)** Called after every removal, except if the container is cleared.

**reservationsCleared(Reservations)** Called if the reservations set is cleared (which currently doesn’t happen).

This new functionality should be used by the window with the `JList` in it. This window should maintain its own `JListModel` that has an `ArrayList` that keeps a (redundant) sorted list of `Reservations` and which is updated when it observes changes. This information will then be relayed by firing the correct list model change events.

### 2.2 Generalizing the Observer

For this homework in an extension described below, you will also need to have a panel listing the rooms which will use `JList` and `JListModel` in a very similar way. Thus *after* getting the previous changes working, you should generalize first the `ReservationsObserver` as a `CollectionObserver<T>` (with more generic method names) and, once *that* is working, generalize the specialized `JListModel` as a `CollectionListModel<T>`.

These new type-parameterized classes have no connection to the `schedule` classes and thus should be moved to the `util` package. Eclipse should make this last refactoring easy.

### 2.3 Rooms

The room class will now be mutable: the name and capacity can be changed. The system must ensure that the room is removed from the map when renamed and replaced after the operation is complete. It should also be possible to delete rooms. You will also want to make rooms comparable so that you can reuse the `CollectionListModel` class.

The `Room` class should permit clients (that implement `CollectionObserver<Room>` to observe the container of rooms, without permitting the clients to make changes using the container. You will need to use `TreeMap` instead of `HashMap`, and will find the `values` method useful, especially in conjunction with `Collections.unmodifiableSet` wrapper.

### 2.4 Separating Reading from Display

The solution for Homework #4 combines the GUI code for the frame with code for reading reservations. This code should be refactored so that the reading code is done by the ADT being read. The reading code should return a list of conflicting reservations that were rejected so that they can be printed or used in a warning window, as appropriate.

## 3 Visible Changes

There are a number of visible changes for this homework.

### 3.1 Edit Dialogs

Currently we have three kinds of dialogs for editing reservations: add, change and remove. The latter two will be merged into a single dialog with four buttons: Reset, Change, Remove, and Cancel. Instead of bringing up a change dialog upon a double-click, it should bring up an edit dialog.

### 3.2 Block View

The current way of viewing reservations doesn't give a graphical view. For this Homework, you will design a new view. At the top will be controls for setting the day to view. There will be a previous day button, an indication of the current day and then a next day button. The buttons should be disabled if there are no more days in that direction.

Below this will be an area in which the reservations for this day are drawn as rectangles, with the description of the reservation vertically centered and horizontally left-justified. A time line should be drawn on the left with hour ticks. A scrollpane should be used to show portions of the schedule. Ideally, double-clicking on a rectangle should bring up an edit dialog for that reservation, but this optional.

### 3.3 New Main Window

The solution for Homework #4 has one main program window for every room. For Homework #5, these windows will no longer have menus. Instead we have a new main window. When this window is closed, the whole program will shut down (including all windows still open). This main window will list all the rooms currently known.

The main window will be the only window with a menu bar. The File menu will have the following items:

**New** The entire state of the system will be cleared, all rooms and all reservations. The system should confirm the choice if any changes have been made since the last **Save**, **Clear** or **New**.

**Clear** This option will remove all reservations but leave the rooms in place. It should require confirmation in the same situations as **New**.

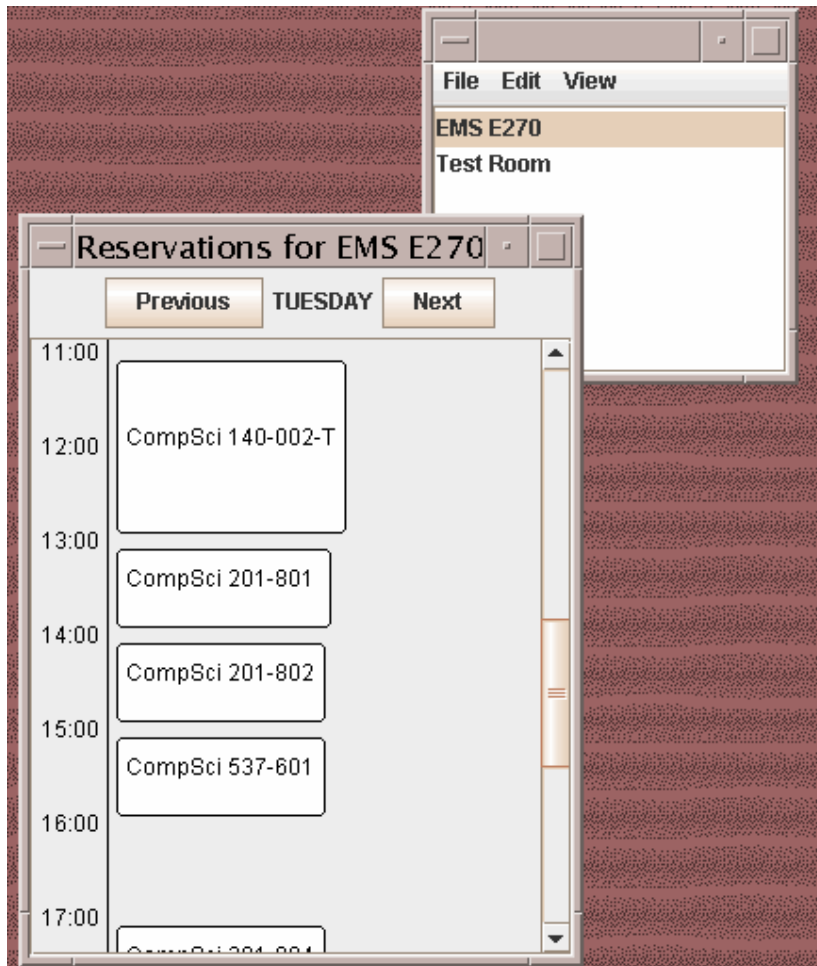


Figure 1: Example main window and new view

**Open** As before, this will read in a file. Unlike previously, no windows will be opened as a result.

**Save** This option will save the reservations for *all* rooms.

**Quit** This will terminate the program, upon confirmation if there have been changes since the last **Save**, **Clear** or **New**.

There will be an **Edit** menu with the following items:

**New Reservation** This will bring up the **Add** dialog design in Homework #4.

There will be a **View** menu with the following items:

**View as List** This will bring up a window with the reservations for the selected room as with the previous homework, but without menus, of course.

**View as Blocks** This will bring up the new view described above for the currently selected room.

A view (including the main window) must update itself when any change happens in the collection being viewed, using the new `CollectionObserver` technique. It must also close itself down if its room is deleted, or if the window is told to close itself. In this case it must also make sure that it removes itself from every collection it is observing. You should permit multiple views of the same room to be open simultaneously. The title bars on the windows should indicate the room whose reservations are being viewed.

## 4 Files

The solution files for Homework #4 are in

```
/afs/cs.uwm.edu/users/classes/cs552/solutions/homework4/
```

You may use our solution, or work with your own solution. However, your code must work at least as well as our solution. Please put your Homework #5 files in your `homework5` directory, with the normal Java directory structure.

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

```
grid.cs: cd homework5
grid.cs: cd bin    (OPTIONALLY)
grid.cs: java Driver Fall05.res test.res
```