

Homework #4

due Thursday, November 1, 12:30 PM

In this homework, you will update the room reservation system using Java library frameworks from Chapter 8, including collections and Swing. In particular you will implement a GUI front-end to the program. You will want to use the Swing tutorial available from the course web page.

1 Updates to the Base ADTs

We will be working with the abstract data types developed for Homework #2, but these classes need some more features in order to be useful. You may start with your solution, or (recommended) with the posted solution code.

1.1 Finding Rooms by Name

Currently although rooms have object-identity, there is no way for one to find a room that matches a given name or to find the room reservations for a given room. To address the first problem you will add two public class (“static”) methods to class `Room`:

`findRoom(String)` This method finds a room with the given name and returns it. If it cannot find such a room, it returns `null`.

`getRoom(String,int)` This method finds a room with the given name and capacity and returns it. If there is no room with this name, it creates and returns a new room. If a room with the same name but different capacity exists, then it returns `null`.

In order to control creation of `Room` instances, you will make the constructor private. Furthermore, your code will need to use an appropriate data structure to remember all rooms that have been created. You must choose an efficient container.

1.2 Reservations

You will add a new ADT `Reservations` that is a `SortedSet` of room reservations all for the same room. Reservations will be ordered first by day, then by starting time, then by duration, and finally by the string description. Each room will have a one-to-one relationship with a `Reservations` instance, and this relationship should be publicly navigable.

You may use any efficient way to implement this ADT; you should choose some technique that demonstrates good reliability, reuse and maintainability. (Hint: don’t implement your own data structure from scratch!)

An object invariant that all instances of this class should demonstrate is that all of the reservations in the set should be for the room associated with this instance and that none of the reservations conflict. You should implement a method that returns true if and only if this invariant is met.

You should **assert** this invariant at the end of the constructor and at the beginning of every public method that you write in the class. You should **assert** the invariant at the end of any method that modifies the set. (You are not required to have the assertion in code that is inherited from the superclass.)

Using **assert** is good design because checking this invariant on every access is inefficient and assertions are disabled by default.

However, whether or not assertions are enabled, the client should *never* experience an assertion violation—that would indicate a major bug in your ADT implementation. Rather, attempting to insert an instance that would conflict should raise an exception **ReservationConflictException** that you will declare as a subclass of **Exception**, not a subclass of **RuntimeException**. Choosing this exception as “checked” means that clients are not expected to check for conflict before attempting to add a reservation. On the other hand, inserting a reservation for a different room should cause an (unchecked) **IllegalArgumentException** to be thrown.

Checking insertions should be efficient (unlike the assertions, these checks are always enabled). You will need to use the property that in a container with no conflicts, a new reservation may only conflict with a reservation immediately before it or after it in the reservation ordering.

The **SortedSet** interface gives many ways in which the set can be modified, including through subsets requested. You don’t need to control the removals (because of the nature of the invariant, removals can never cause the invariant to be invalidated), but should be vigilant to control *all* insertions. This task is not easy. To make the task feasible, the ADT is *not* required to handle the fact that reservations may be changed while in the set. You will need to be very careful that reservations are never modified while they are in a reservations set. (You may understand why many programmers prefer immutable ADTs!)

2 Using Swing to implement a GUI

For this Homework, you will implement a “graphical user interface” (GUI) front-end to room reservation GUIs. As in Homework #2, the main program will read in room reservation files, but then will open a frame (window) for each room with reservations and enable editing. It will have two menus: the “File” menu has three actions (Open, Save and Close); the “Edit” menu has three actions (Add, Change and Remove). Clicking in the window should select one of the reservations. Double-clicking on a reservation should pop up the change dialog (the same as if the “Change” menu option were chosen).

The “Open” and “Save” choices will use a built-in class (**JFileChooser**) to request a file name. It should be possible to cancel the action. The file so indicated will (for “Open”) be read for reservations, possibly causing new windows to appear (for new rooms). Existing reservations are *not* affected; new reservations are only added if they do not conflict. If there are any conflicts, the user is noticed with a single warning dialog listing the rejected reservations. The “Save” option writes a file in the proper format listing *only* the reservations for the current room.

The “Close” choice should warn the user if the reservations for that room have been changed since it has last read from a file or saved (which an option to cancel the operation).

This should use a `showXXX` method of `JOptionPane`. The program should terminate when the last window for some room is closed.

The “Add,” “Change,” and “Remove” choices should bring up a dialog with the reservation selected, or an imaginary reservation at 9am on Monday. The dialog should give the ability to edit the parts of a reservation and should give three buttons:

Reset Reset the values in the dialog to the reservation (real or imaginary) that was used when the dialog started.

Add/Change/Remove Depending on the dialog, do the action requested. If an error happens, display the error message (using `JOptionPane`) and then keep the dialog up. Otherwise, close the dialog.

Cancel Close the dialog without performing an action.

You should use either the `TEMPLATE` or the `STRATEGY` pattern to reuse code between the three dialogs. The application needs to be robust, and properly repaint itself under all circumstances. For instance, after a new file is read, any window displaying reservations for rooms whose reservations have changed should be repainted. Exceptions due to user-inputs (or the file-system) must be caught at the appropriate place and displayed with warning messages.

When either the stop time or duration of a period changes (see figure), the other one should be updated. If the start time changes, the stop time should be changed too. If a change would cause an period to run into midnight (or beyond), the duration should be set to zero, to ensure the period is legal.

You need to be careful *not* to change the period of a reservation that is in a reservations container while using the dialog. All changes should be on a special period used by the dialog, and only copied over when the dialog is complete.

The solution creates instances of the following AWT/Swing classes:

```
BorderLayout BorderLayout JFrame JList JPanel JScrollPane JTextField
```

It uses other classes as well. It never overrides a `paint` method.

3 Files

The solution files for Homework #2 are in

```
/afs/cs.uwm.edu/users/classes/cs552/solutions/homework2/Part1
```

You may use our solution, or work with your own solution. However, your code must work at least as well as our solution.

There are no “parts” to this homework. Put all the files in the `homework4` directory of your AFS volume. As usual, the ADTs will be in their own subdirectory `schedule`.

The project must be runnable from within its directory. In other words, on a grid, it must be possible to

```
grid.cs: cd homework4
grid.cs: cd bin           (optionally)
grid.cs: java Driver
```

All work must be completed by 2pm, Thursday, November 1st, 2007. (The extra two days are because of the midterm.)

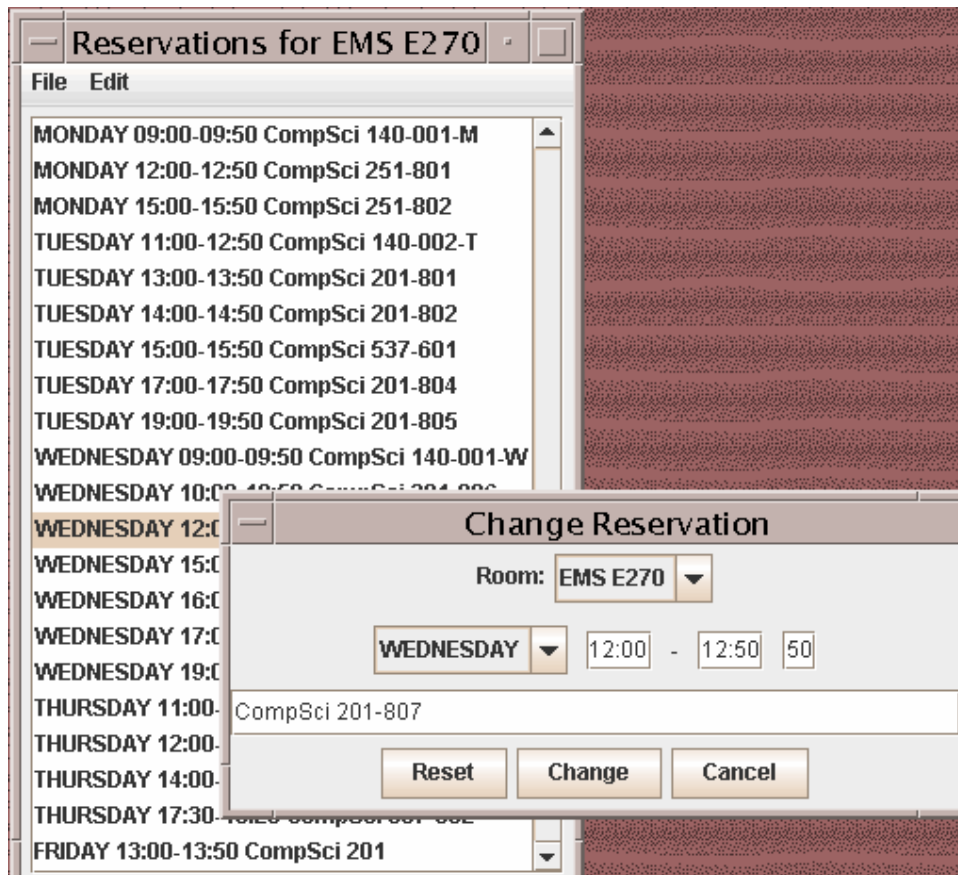


Figure 1: Running the GUI Test