CS5001 Object-Oriented Modelling, Design and Programming
Practical 4 – Room Booking System

School of Computer Science
University of St Andrews

Due Friday week 11, weighting 30%
MMS is the definitive source for deadlines and weightings.

In this practical, you are required to write a room-booking system with a text-based user interface and a graphical user interface using Java Swing. Your program should use either the Model–View–Controller (MVC) or the Model–Delegate (MD) design pattern. You should use JUnit to test your Model, and you should include these tests in your submission.

Requirements

This practical is more open-ended than the other practicals on this module. There are no automated tests that you have to pass, and the choice of classes, methods and output format that you use are up to you.

Your program will consist of a model, and two user interfaces.

Model

Your program should capture information about the people, rooms, timeslots and bookings involved in a room-booking system for a university. These entities relate to each other as follows.

The University consists of buildings and people. Each building contains rooms. Rooms can be *booked*, marking them as being occupied by someone at a specified time. Each booking is for one person, in one room, for a single contiguous block of time on a given date. This block of time must last at least 5 minutes, and its length must be a multiple of 5 minutes. A booking cannot stretch across multiple dates. Two bookings for the same room cannot have overlapping times.

Each person has an email address (which must be unique) and a name. Each building has a name and an address. Each room has a name, which might or might not be a number.

It should be possible to **add** and **remove** people, rooms, buildings and bookings in the system while it is running. Illegal bookings should be rejected, so it should be impossible for the user to put the model into an invalid state, such as having multiple users with the same email address.

It should also be possible to **save** the state of the booking system to a file, and load it again on a later run of the program. Only the model itself should be stored, with no details of the GUI being included in the file.

User interfaces

Your program must have **two user interfaces**: a command-line interface in the terminal, and a graphical user interface in a Swing window. When your program is started, both of these UIs should start running, and each one should feature full functionality for interacting with the model. Both UIs should be connected to the same model, so that changes made in one UI will display appropriately in the other.

In each UI, the user should be able to do the following things:

- Add and remove people, rooms, buildings and bookings;
- Choose a time, and see all rooms that are available at that time;
- Choose a timeslot (start and end times) and see all rooms that are available for that whole period;
- View the schedule for a given room, including bookings and free periods;
- View all bookings made by a given person;
- Save to a specified filename;
- Load from a specified file.

The two different UIs will handle these in different ways, but in each case it should be easy for the user to understand how to use the program and how to interpret what is displayed. In other words, the UI should be as *intuitive* and easy-to-use as possible.

Invalid input should be rejected by the program and a useful message should be displayed to the user in whichever UI they used to enter the bad input.

It should be possible for the user to **close** each UI. The program should continue to run until both UIs are closed, then it should terminate.

MVC/MD design pattern

As mentioned above, the two GUIs should be separated from the model using either the *MVC pattern* or the *MD pattern*, as described in lectures. In particular, this means that the model should be an explicitly separated part of the code, and should contain no references to any user interface code. The two UIs should have some kind of references to the model, but should not contain any code related to each other.

Testing

Along with your program, you should create a test suite that tests the functionality of your *model*. This test suite should be written in JUnit, and should test the broadest possible range of inputs to your model. You should aim to execute every line of code in your model somewhere in this test suite, and it should also include edge cases and invalid inputs. Your JUnit tests do not need to test either of your user interfaces, but you should test these thoroughly by hand to make sure that they work.

It should be possible to run these tests on the lab computers. Make sure to run your JUnit suite on the School infrastructure and ensure that they work before submitting. You should include everything necessary to run the tests in your submission, and your README should explain clearly how to execute them.

Deliverables

For this practical you **must** include a README file. This file must:

- List any features you may have implemented that are not part of this specification;
- Explain clearly how to **compile and run** your program from the command line on the School's Linux lab PCs;
- Give clear instructions on how to run the JUnit tests you have written.

When you are finished, you should submit a zip archive containing your source code, your JUnit tests, and your README, and submit it to MMS in the usual way.

Marking

Grades will be awarded according to the General Mark Descriptors in the Feedback section of the School's Student Handbook. The following table shows some example descriptions of projects that would fall into each band.

Grade	Examples
0	Nothing submitted.
1–3	Little or no working code, and little or no understanding shown.
4–6	Acceptable code for part of the problem, but with serious issues such as not compiling or running, or having no working GUI. Some understanding of the issues shown.
7	A working program that runs and allows the user to create bookings, with some kind of working GUI. Many features missing, major bugs, poor code quality and incorrect MVC/MD pattern.
8–10	Several requirements completed, such as the ability to add and remove rooms, users and bookings. A working GUI and some command-line output, but with bugs and with other requirements missing or broken. Major problems with code quality, documentation, or MVC/MD pattern. Some evidence of testing.
11–13	Most requirements implemented and working, perhaps with some bugs. Two working user interfaces, with reasonable use of MVC/MD, but code quality or doc that could be improved. A sensible set of JUnit tests.
14–16	Almost all requirements implemented and working, with only minor defects. Two user interfaces working well, with correct use of MVC/MD and good OOP, code quality and documentation. JUnit tests covering almost all code in the model, with edge cases considered.
17–18	All requirements implemented and working, with excellent design and code quality, great documentation and flawless use of MVC/MD. JUnit tests covering the whole model, with a wide range of edge cases considered.
19–20	Outstanding implementations of all required features, and possibly some extra features beyond what was required, with exceptional clarity of design, no code duplication, concise but full documentation, and an appropriate object-oriented structure. An exceptional test suite.

Any extension activities that show insight into object-oriented design, GUI design and test-driven development may increase your grade, but a good implementation of the stated requirements should be your priority.

Lateness

The standard penalty for late submission applies (Scheme B: 1 mark per 8-hour period, or part thereof) as shown at:

https://info.cs.st-andrews.ac.uk/student-handbook/learning-teaching/assessment.html#lateness-penalties

Good Academic Practice

The University policy on Good Academic Practice applies:

https://www.st-andrews.ac.uk/students/rules/academicpractice/