School of ITEE
CSSE2002/7023 — Semester 2, 2020
Assignment 2
Due: 23 October 2020 17:00 AEST
Revision: 1.2.2

Abstract

The goal of this assignment is to implement a set of classes and interfaces[1] to be used to create a simulation of a building management system.

You will implement precisely the public and protected items described in the supplied documentation (no extra package-private/public/protected members, methods or classes). Private members and Private methods may be added at your own discretion.

- Including extra package-private/public/protected members, methods or classes will cause the compilation for a class to fail, meaning no marks for functionality or unit tests will be awarded for the class. This strictly enforced (because the required specfication has not been followed). As noted below, you should utilise the assignment prechecks to help avoid this.

Language requirements: Java version 14, JUnit 4

Preamble

All work on this assignment is to be your own individual work. As detailed in Lecture 1, code supplied by course staff (from this semester) is acceptable, but there are no other exceptions. You are expected to be familiar with "What not to do" from Lecture 1 and `https://www.itee.uq.edu.au/itee-student-misconduct-including-plagiarism`. If you have questions about what is acceptable, please ask course staff.

All times are given in *Australian Eastern Standard Time*. It is your responsibility to ensure that you adhere to this timezone for all assignment related matters. Please bear this in mind, especially if you are enrolled in the External offering and may be located in a different time zone.

Introduction

In this assignment you will finish building a simple simulation of a building management system (BMS). A building management system monitors the structures of buildings, including their internal floors and rooms. A BMS uses different types of sensors to gather contextual information about the building.

In the first assignment you implemented the core model for the BMS. In the second assignment you will implement some of the more advanced logic to provide a very simple simulation for the BMS.

In addition to the ability to determine hazard levels in a room from sensors that were implemented in assignment one, you will add functionality to determine comfort level in a room from the sensors. You will need to integrate this new feature into the system. This is an example of a common situation when building a large system. New features need to be added to the system. A well designed system that uses interfaces to define an API means it should be simple to add the new feature.

[1] From now on, classes and interfaces will be shortened to simply "classes"

Monitoring and managing hazard levels requires sophisticated logic in a real BMS. In assignment one hazard levels were simply reported by each sensor. In assignment two you will implement logic for hazard level evaluators. These take the hazard level data from a set of sensors and determine overall hazard level for the Room where the sensors are located. The approach taken is to define a `HazardEvaluator` interface that provides an API. Different classes can implement this interface to provide different options for the logic of determining hazard levels. This is another example of a common approach to designing flexibility into the system's structure.

Rooms within a building require regular upkeep and maintenace. In assignment two you will implement logic for a `MaintenanceScheduler` which is able to organise a maintenance schedule for nominated rooms on a given floor. Maintenance times can vary depending on each room, and can maintenance be influenced by factors such as whether there is currently an active fire alarm.

Something which can be useful to know as a student is where you can go within a building to be able to study. In assignment two you will implement logic for a `StudyRoomRecommender` which is able to suggest a room within the building which is the most suitable for studying, based upon current conditions (this has some similarities to the screens which are present near the entrances in the UQ Library and provide information about current seating capacity in various location in the library).

When implementing the assignment you need to remember that it is implementing a simulation of the BMS and not the real BMS. Interfaces are provided for the sensors to allow easy replacement of sensor implementations in the program. You will not be collecting data from real sensors but will be implementing classes that demonstrate the behaviour of sensors. They store a set of data values that are used to simulate the sensors returning different values over time.

To manage simulation of time, there is a `TimedItem` interface and a `TimedItemManager` class, which you implemented in assignment one. Sensors implement the `TimedItem` interface, as they are items which need to react to timed events. `TimedItemManager` stores all the `TimedItem` objects in the application. The simulation's GUI tracks time passing in `View.run()` and it invokes `ViewModel.tick()` once per second. The `tick` method calls the `TimedItemManager`'s `elapseOneMinute` method, which sends the `elapseOneMinute` message to all `TimedItem`s. This approach of tracking the passage of time and invoking an action on all relevant objects once per second was the reason that `TimedItemManager` is implemented as a singleton[2].

A simple GUI has been provided to you as part of the provided code. It is in the `bms.display` package. *It will not work until you have implemented the other parts of the assignment that it uses.*

The GUI has been implemented using JavaFX and consists of three classes and an enum. `View` creates the main window for the BMS GUI. `BuildingCanvas` displays the structure of the building. `ViewModel` represents the BMS model that is to be displayed. The BMS application is initialised and started by the `Launcher` class in the `bms` package. It loads the building data and creates the GUI. Most of the GUI code has been provided to you. In `ViewModel` you need to implement some of the logic that is executed by events in the simulation and to handle keyboard input for the main application's window.

The functionality you need to implement in `ViewModel` is to:

- Save the state of the buildings to a file in response to the user selecting the save command. This is to be implemented in `ViewModel.save()`.
- Allow the simulation's execution to be paused and unpaused. This is to be implemented in `ViewModel.togglePause()`.
- Process time passing in the simulation. This is to be implemented in `ViewModel.tick()`.
- Keyboard input is handled by the `accept` method in the `ViewModel` class. It needs to process input from the user in the main window to perform actions in the simulation. Pressing the 'P' key will toggle whether the simulation is paused or not. The 'Q' key will quit the simulation. The 'S' key will save the current building to a file called "quicksave.txt" (inside the saves directory). A shell for this method has been provided because it is already hooked into the GUI.

[2] `https://refactoring.guru/design-patterns/singleton` provides a reasonably detailed description of the singleton pattern.

Persistent Data

Warning:
A common mistake made by students who are learning is to use hard coded directories when reading or writing to files. Please avoid using this approach.

You must use relative directories instead.

Using hardcoded directories instead of relative directories means that when we test your solution, it will fail, because your solution will not be able to locate the file on our system.

This is strictly enforced (because it is bad programming practice). A submission will not be remarked if hardcoded directories are used and the solution fails to read or write applicable files.

As an example (on Windows), please do NOT do this:
```
File myFile = new FileReader("C:\Programming\Java\ass2\saves\uqstlucia.txt");
```
or
```
File myFile = new FileReader("saves/" + filename);
```

The filename should be relative from the project root directory and should include "saves/".

Requirements:
You need to implement loading multiple buildings from a data file. The JavaDoc for the `loadBuildings` method in the `BuildingInitialiser` class describes the format of a building data file. Saving buildings is done by the `save` method in the `ViewModel` class. A building data file is structured using a hierachy as follows:

- First Building information
  - First floor in building information
    - First Room on floor information
      - First sensor in room information
      - Second sensor in room information
  - Second floor in building information
    - First Room on floor information
      - First sensor in room information
    - Second Room on floor information
      - First sensor in room information
- Second Building information
  - First floor in building information
    - First Room on floor information
      - First sensor in room information
      - Second sensor in room information
- etc.

The following lines are the building details.

- The first line is the name of the building.

  e.g. General Purpose South
- The second line is the number of floors in the building.

  e.g. 5

- A floor is described by the floor number, followed by a ':', then the width, followed by a ':', then the length, followed by a ':', then the number of rooms, followed by a ':', and a sequence of room numbers which are separated by commas.

  e.g. 1:10:10:4:101,103,102,104

- A room is described by the room number, followed by a ':', then the type of room, followed by a ':', then the area of the room, followed by a ':', then the number of sensors in the room, then optionally a ':' and the type of hazard evaluator in the room if it has one.

  e.g. 201:OFFICE:50:2:RuleBased

- A sensor is described by the type of sensor, followed by a ':', then a sequence of sensor readings which are separated by commas, then optionally any values required by the type of sensor, such as update frequency and capacity, separated by ':'.

  e.g. OccupancySensor:32,35,26,4,3,2,6,16,17,22,28,29:2:40

An example data file, called `uqstlucia.txt`, is provided in your repository in the `saves` directory.

Supplied Material

- This task sheet.
- An example building data file.
- Code specification document (Javadoc).[3]
- A Subversion repositiory for submitting your assignment called `ass2`.[4]
- A simple graphical user interface for the simulation, which is in the `display` package.
- A sample solution for the first assignment. You are to use this as the base for your implementation of the second assignment. As the first step in the assignment you should create a new project by checking out the `ass2` repository from Subversion.

Javadoc

Code specifications are an important tool for developing code in collaboration with other people. Although assignments in this course are individual, they still aim to prepare you for writing code to a strict specification by providing a specification document (in Java, this is called Javadoc). You will need to implement the specification *precisely* as it is described in the specification document.

The Javadoc can be viewed in either of the two following ways:

1. Open `https://csse2002.uqcloud.net/assignment/2/` in your web browser. Note that this will only be the most recent version of the Javadoc.
2. Navigate to the relevant assignments folder under `Assessment` on Blackboard and you will be able to download the Javadoc `.zip` file containing html documentation. Unzip the bundle somewhere, and open `docs/index.html` with your web browser.

Tags in the Javadoc indicate what code has been implemented in assignment one and what code you need to implement in assignment two. Some code from assignment one will need to be modified. There are tags indicating places where you can expect to modify the assignment one code but these are not guaranteed to be all of the places where you may end up modifying code from assignment one.

[3] Detailed in the `Javadoc` section
[4] Detailed in the `Submission` section

Tasks

1. Implement the classes and methods described in the Javadoc as being requried for assignment two.
2. Implement the indicated features of the user interface.
3. Write JUnit 4 tests for all the methods in the following classes:
   - `BuildingInitialiser` (in a class called `BuildingInitialiserTest`)
   - `MaintenanceSchedule` (in a class called `MaintenanceScheduleTest`)

Marking

The 100 marks available for the assignment will be divided as follows:

Symbol	*Marks*	*Marked*	*Description*
F	45	Electronically	Functionality according to the specification
R	35	Course staff	Code review (Style and Design)
J	20	Electronically	Whether JUnit tests identify and distinguish between correct and incorrect implementations

The overall assignment mark will be $A_2 = F + R + J$ with the following adjustments:

1. If $F < 5$, then $R = 0$ and code style will not be marked.
2. If $R > F$, then $R = F$.

For example: $F = 22, R = 25, J = 17 \Rightarrow A_2 = 22 + 22 + 17$.
The reasoning here is to place emphasis on functional code and to not to give marks to well styled code and well implemented JUnit tests when the code is not functional.

Functionality Marking

The number of functionality marks given will be

$$F = \frac{\text{Functionality tests passed}}{\text{Total number of functionality tests}} \cdot 45$$

Each of your classes will be tested independently of the rest of your submission. Other required classes for the tests will be copied from a working version of the assignment.

Code Review

Your assignment will be reviewed and style marked with respect to the course style guide, located under `Learning Resources > Guides`. The marks are broadly divided as follows:

Naming	5
Commenting	7
Structure and Layout	7
Code Design	8
Object-Oriented Practices	8

For Code Design we are looking for well thought out code that is easily followed and is maintainable. For Object-Oriented Practices we are looking for the course content to be applied to solve certain challenges present in the assignment.

Note that style marking does involve some aesthetic judgement (and the marker's aesthetic judgement is final).

JUnit Test Marking

The JUnit tests that you provide in `BuildingInitialiserTest`, and `MaintenanceScheduleTest` will be used to test both correct *and* incorrect implementations of their respective classes. Marks will be awarded for test sets which distinguish between correct and incorrect implementations[5]. A test class which passes every implementation (or fails every implementation) will likely get a low mark. This will be assessed by running your JUnit test classes on a number of correct and incorrect assignment implementations. Marks will be rewarded for tests which pass or fail correctly.

There will be some limitations on your tests:

1. if your tests take more than 20 seconds to run, or
2. if your tests consume more memory than is reasonable or are otherwise malicious,

then your tests will be stopped and a mark of zero given. These limits are very generous (your tests should not take *anywhere* near 20 seconds to run and in all probability could take well less than 1 second).

Electronic Marking

The electronic aspects of the marking will be carried out in a linux environment. The environment will not be running Windows, and neither IntelliJ nor Eclipse (or any other IDE) will be involved. OpenJDK 14 and JUnit 4.12 will be used to compile and execute your code and tests.

It is critical that your code compiles. If one of your classes does not compile, **you will recieve zero** for any electronically derived marks for that class.

Submission

Submission is via your Subversion repository. You must ensure that you have committed your code to your repository *before* the submission deadline. Code that is submitted after the deadline will **not** be marked. Failure to submit your code through your repository will result in it **not** being marked. Details for how to submit your assignment are available in the `Version Control Guide`. Your repository url is:
`https://source.eait.uq.edu.au/svn/csse2002-s???????/trunk/ass2` — **CSSE2002** students
or
`https://source.eait.uq.edu.au/svn/csse7023-s???????/trunk/ass2` — **CSSE7023** students

Do NOT submit to the ass1 directory!

Your submission should have the following internal structure:

`src/`	folders (packages) and `.java` files for classes described in the Javadoc
`test/`	folders (packages) and `.java` files for the JUnit test classes
`saves/`	`.txt` files for testing `BuildingInitialiser`

Source files submitted to the incorrect directory will not be marked; marks associated with the functionality or unit tests in the file will be 0. This is strictly enforced.

Please actively make use of the provided prechecks to verify if files are submitted to the correct directories.

A complete submission would look like:

```
src/bms/building/Building.java
src/bms/building/BuildingInitialiser.java

src/bms/display/BuildingCanvas.java
```

[5] And get them the right way around

```
src/bms/display/ButtonOptions.java
src/bms/display/View.java
src/bms/display/ViewModel.java

src/bms/exceptions/DuplicateFloorException.java
src/bms/exceptions/DuplicateRoomException.java
src/bms/exceptions/DuplicateSensorException.java
src/bms/exceptions/FileFormatException.java
src/bms/exceptions/FireDrillException.java
src/bms/exceptions/FloorTooSmallException.java
src/bms/exceptions/InsufficientSpaceException.java
src/bms/exceptions/NoFloorBelowException.java

src/bms/floor/Floor.java
src/bms/floor/MaintenanceSchedule.java

src/bms/hazardevaluation/HazardEvaluator.java
src/bms/hazardevaluation/RuleBasedHazardEvaluator.java
src/bms/hazardevaluation/WeightingBasedHazardEvaluator.java

src/bms/room/Room.java
src/bms/room/RoomState.java
src/bms/room/RoomType.java

src/bms/sensors/CarbonDioxideSensor.java
src/bms/sensors/ComfortSensor.java
src/bms/sensors/HazardSensor.java
src/bms/sensors/NoiseSensor.java
src/bms/sensors/OccupancySensor.java
src/bms/sensors/Sensor.java
src/bms/sensors/TemperatureSensor.java
src/bms/sensors/TimedSensor.java

src/bms/util/Encodable.java
src/bms/util/FireDrill.java
src/bms/util/StudyRoomRecommender.java
src/bms/util/TimedItem.java
src/bms/util/TimedItemManager.java

src/bms/Launcher.java

test/bms/building/BuildingInitialiserTest.java
test/bms/floor/MaintenanceScheduleTest.java

saves/uqstlucia.txt
```

Ensure that your classes and tests correctly declare the package they are within. For example, `CarbonDioxideSensor.java` should declare `package bms.sensors`.

Declaring incorrect packages may cause the class compilation to fail, meaning 0 marks will be allocated for the class functionality.

Do not submit any other files (e.g. no `.class` files), with the exception of additional `.txt` files in the `saves/` directory for the purposes of testing `BuildingInitialiser`. Note that `BuildingInitialiserTest` and `MaintenanceScheduleTest` will be compiled individually against a sample solution without the rest of your test files.

Prechecks

Prechecks will be performed on your assignment repository multiple times before the assignment is due. They will assess whether your folders and files are in the correct structure and whether your public interface aligns with the expected public interface. **Successfully passing a precheck does not guarantee any marks. No functionality or style is assessed.**

```
Precheck #1: Approximately 5pm on 7 October 2020.
Precheck #2: Approximately 5pm on 12 October 2020.
Precheck #3: Approximately 5pm on 16 October 2020.
Precheck #4: Approximately 5pm on 21 October 2020.
```

You should endeavour to have code written and in your repository before at least one of these prechecks, preferably the first, in order to make the most of them. No additional prechecks will be run for people who did not start the assignment in time, or who neglected to commit their code to their repository. Prechecks are valid only for currently released version of the Javadoc, if an update is made it may invalidate the precheck results.

Late Submission

Assignments submitted after the due date will recieve a mark of zero unless an extension is granted.

Do not wait until the last minute to commit the final version of your assignment. A commit that starts before 17:00 but finishes after 17:00 will **not** be marked.

Extension Requests

You may request an extension if exceptional circumstances affect your ability to complete the assignment on time. Circumstances that are within your control will not be accepted (e.g. multiple assignments due at the same time, losing work due to a computer crashing, ...). The ECP provides details of how to apply for an extension and the maximum allowed length of an extension. Your extension request must include appropriate evidence for the impact of the circumstances and the duration of that impact. Extensions will not be granted beyond the length of the duration of the impact.

You are expected to start work on the assignment well before the due date. Tutor support for the assignment is available up until the assignment due date. Some support will be provided after the due date for students who have an extension but you cannot expect the same level of support as is provided before the due date. You should not expect assignment help over a weekend. You should not expect an extra pre-check to be run on your submission after the last scheduled pre-check above. That is only two days before the assignment is due and you should have completed enough work to be checked for conformance by then, even if you have an extension on the assignment.

Remark Requests

To submit a remark of this assignment please follow the information presented here: `https://my.uq.edu.au/information-and-services/manage-my-program/exams-and-assessment/querying-result`.

Revisions

Version 1.2.2 (Tue 13 October)

- Clarified that submitting extra `.txt` files in `saves/` is allowed under Submission section of task sheet

Version 1.2.1 (Thu 8 October)

- Fixed explanation of sensor `encode()` representation under Persistent Data section of task sheet

Version 1.2.0 (Wed 7 October)

- Fixed example of sensor `encode()` representation under Persistent Data section of task sheet
- Clarified that `Floor.encode()` should append encoded rooms in insertion order in Javadoc
- Replaced references to a floor's 'height' with its length in Javadoc for `Floor.changeDimensions()`
- Clarified that `OccupancySensor.getComfortLevel()` should use floating point division, and round to the nearest integer in Javadoc
- Added assumption of weighting order returned by `WeightingBasedHazardEvaluator.getWeightings()` in Javadoc for `Room.encode()`

Version 1.1.0 (Sat 3 October)

- Removed all references to SimpleDisplay in the task sheet and Javadoc (this class was implemented in assignment 1)