Assignment 2

Scenario:

You have received new information from your client. They want you to update your system to handle multiple elevators. Each elevator should run on its own thread. Like before you are given the opportunity to modify the build if needs be.

Due Date:

Assignment 2 is to be demoed between **March 19th and March 23rd**. You must demo a working application even with bugs with all deliverables. If you do not demo you will receive a Zero with no exceptions. Since assignments are connected you will be given a chance to fix your errors in previous assignment every time you demo a new stage. This is only true if you have demoed your previous stage with acceptable progress (60% or higher in new assignment).

During the weeks before March 19th, you can ask question in labs regarding your assignment for advance feedback. This will allow you to fix and/or improve your application for better mark. However, during the demo week you will not be able to ask for help. You will have maximum of 4-5min to demo and receive you mark, no chance for you to improve and demo again.

A copy of your project with all UMLs and any other document if requested, must be submitted to BB by March 23rd midnight. Your submission will not change the mark given to you during the demo. However, **if submitted late you will receive 10% late penalty for every late day. This submission is used to find plagiarism, if found zero is assigned with no exceptions to all parties.**

Summery:

1. A working version of Assignment 1 is needed at this point.
2. Updated interfaces, Simulator and MovingState are provided.
3. Functionality of ElevatorImp is mostly unchanged.
    a. Some new basic methods added.
    b. Timing control is now inside of moveTo using Thread.sleep().
4. Most of the changes are in ElevatorSystemImp.
    a. callUp and callDown have threads for finding new elevator.
    b. requestStop and requestStops don't move the elevator anymore.
    c. runnable object in system is the new heart of ElevatorSystemImp
5. Using quick sort, sort the order in which the elevator will stop.
6. GUI needs to be able to show multiple elevators.
    a. Each elevator needs to have its own currentFloor, targetFloor, powerUsed and ID.
7. There is no focus on capacity or off state in this assignment.
8. People are assumed to get in after the elevator call and select all target floors before elevator starts using requestStops(int…floors)

Assignment 2

Code:

Below are the details of new changes in assignment 2. Look at the class diagram provided for more general look. As before all private methods are optional and can be ignored.

Elevator and ElevatorSystem:

Both interfaces have been updated with new methods and updated documentations. Download them and replace old interfaces. Read the documentations provided. Many details are explained in the documentations.

QuickSort:

Create a quick sort class which can sort in ascending and descending orders given a List<Integers>.

MovingState:

Two new methods are added to check if state is idle or if it is off.

Simulator:

This class will be provided as well. However, creating a new one is highly encouraged.

ElevatorApplication:

This class should remain mostly unchanged except for the new Nodes needed to add more elevators and info texts. The old design can be copied multiple times to add graphics for new elevators. EelevatorAnimation should remain mostly unchanged except there is no more need for timecheck condition as timing is controlled in ElevatorImp. There should be new functionality to handle the ID that each Elevator will have. ID is used to distinguish updates in queue. ID is passed to updated function through notifyObservers alongside of currentFloor, targetFloor and powerUsed.

ElevatorImp:

Some new methods from the new Elevator interface need to be implemented. The documentation in the interface explains the functionality of these new methods. MoveTo function remains the same aside from use of Thread.sleep() in moveto or other appropriate method (if design has been changed) which will pause the thread for some time after a new state is determined. This pause is the same timing as what was done in ElevatorAnimator. When notifyObservers is called ID of the elevator should be passed as well.

ElevatorImp constructor now takes 2 extra arguments. First, is the integer ID of the Elevator which will be used to distinguish elevators, must be stored in a final variable. Second, is a boolean than will determine if sleep should be used. During normal simulation it should be true but for testing should be false so JUnit can finish fast.

ElevatorImp must have equals and hashCode methods. Only ID is needed to be used inside of these two methods.

Assignment 2

ElevatorSystemImp:

Majority of the changes are in this class. Same as ElevatorImp there are new methods in ElevatorSystem interface that need to be implemented. Some already implemented methods might need to be updated as well.

Elevators are stored inside of a Map<Key, Value>, look up the documentation of this class on official JDK 8 API website (use google if already not bookmarked). This map object will have key type of Elevator and value of List<Integer>. Each key is an elevator in use and values for keys are lists of floors that key Elevator needs to stop at. In side of constructor this object needs to be initialized with new HashMap (look up the API 8). HashMap inherits from Map. To get a value from Map use get() and to put() to it use add method. To retrieve all keys, use keySet() which returns a Set of all keys. Set means a collection of data that are unique. To retrieve all values, use values() which returns a Collection of all values. Look up Collection API 8 for more details. Set, Map, List, ArrayList, LinkedList and more inherit from Collection Interface.

Every time an elevator is added to system use put method to and the elevator and new LinkeLlist to the map. When adding observer all keys should be retrieved from map and observer must be added to all elevators.

For threading ExecutorService is used. Look at code samples and lecture material for more details. Lookup examples for more practice. ExecutorService wraps all the functionality of threads in a very nice and clean package. There is an instance of ExecutorService object in system class that must be initialized in constructor using the static method newCachedThreadPool() from class Executors. This ExecutorService object will execute objects of type Runnable or Callable using the method submit(). ExecutorService has another necessary method called shutdown() that must be called during the exit or teardown of an object. If not, java program will not exit properly. Look at simulator as an example.

In the call method submit() a Callable task to ExecutorService. When using submit() an object is returned of type Future. This object will hold the result of Callable or Runnable object submitted to ExecutorService. Callable object unlike Runnable is not void and can return and object. Using method get() in Future will result in current thread beeing locked till execution of Callable is over. Then data returned from Callable can be retrieved from Furture object. Using this create a Callable lambda or an object and submit() it to ExecutorService. Store the return of submit() function and call get() on it. Finally return the result of get which is an available elevator. The Callable object should find the closest available elevator the calling floor.

A new method called requestStops is added that will take multiple targetFloors. All target floors are provided at the same time. Add all those floors to map using the elevator as key. Then quick sort the list in map for that elevator in ascending or descending order depending on if callUp or callDown was called.

It is important to be aware of which part of the code is access by multiple threads. For example, if map is accessed by multiple threads with some putting new data and some removing data, map must be protected. Using the synchronized(){} block sensitive parts of code can be prevented from being accessed by multiple threads. Create a final Object called REQUEST_LOCK and initialize it with new Object. Now when map is being populated or data being removed or read surround it with synchronized(REQUEST_LOCK){}. This way only one thread at a time can manipulate the map.

There is a need for a thread to actively check the map for next floor that elevator must move to. For this create an object of type Runnable that will have an infinite loop with a shutdown Boolean as condition. This Boolean is false and will be set to true when shutdown() method is called. Inside this loop content of the map is analyzed. Meaning it will find an elevator which is idle and has another stop in its list. Then call elevator.moveTo inside of a Runnable lambda submitted to ExecutorService to move the elevator to next target floor. Repeat this till all lists are empty.

Only one thread at a time should call moveTo method. To achieve this one solution is to use an AtomicInteger or AtomicBoolean for each elevator to keep track of active elevator. If AtomicInteger is not zero and AtomicBoolean is true a thread is already using moveTo. An array of this object can be created locally inside of Runnable object. Atomic object instantly updates the change of data in their memory instead of cache. Hence very useful for threading operations.

Design:

This assignment focuses more on quality of code not just functionality. There are many aspects in this code that can be improved or designed in different patterns. During the demos updates and changes to this design must be justified this is not to discourage change but to encourage proper research and reasoning.

Code must be clean as well. Here are few things:

1. Properly indented code, use ctrl+shit+f.
2. No massive commented blocks of code.
3. No big gaps of new lines.
4. Properly named Variables, Classes, Packages and Methods goes for JUnit as well.
5. Organize classes in proper packages. Any order is acceptable if it makes sense.

Assignment 2

Marking Scheme (113+5):

- Design Patterns (10, 4 marks for elevator reference, 2 marks each for rest, no partial)
  - Model View Controller is implemented properly. Easy mark as the structure is given.
  - Observer pattern, Observable and Observer are properly assigned to their right classes.
    - No reference of Elevator in update method
- Clean Code (8, 2 marks each, no partial)
  - Code is properly indented.
  - Naming conventions are followed.
  - Meaningful names for variables, classes, interfaces and methods.
  - Classes are in proper packages
- Documentation (25, 10 for sequence, 5 each for rest with no partial)
  - Variables, classes, interfaces and methods must have meaningful documentation.
  - Use inline documentation for complex algorithms when needed.
  - Sequence Diagram of ElevatorSystem Runnable object and requestStops with details of quick sort.
  - Full class diagram, no need to show functions.
- JUnit (20, 5 each, no partial)
  - All tests should be in a separate package called test.
  - All public methods from Elevator must be tested.
  - All public methods from ElevatorSystem must be tested.
  - All public methods from QuickSort must be tested.
- Error Handling (10)
  - There must be proper error handling in all public methods.
  - It is recommended to use exceptions for public methods.
- Execution (20)
  - Application must run successfully using the simulator provided.
    - Simulator floors must be unchanged.
  - Current and target floor, and energy consumed must be displayed as text.
  - Elevator animation or 4 elevators
- Code (20)
  - Justify any changes made to the original design if any.
  - Proper use if Java syntax, while loop is **not** a replacement for if condition.
  - Proper initialization, good code organization, proper used of helper methods.
- Bonus (5)
  - Reward is given based on impressive and/or creative designs.
  - Sequence and Class diagram done with pen and paper.