Project 1 Eric Sabelhaus

1. Problem Analysis
   1. The purpose of this project was to provide a solution which depicts the flow of passengers as the go through an airport security checkpoint. The main emphasis of the project, was to design a queue that efficiently and correctly uses list type objects in Java. The queue should be initialized with 20 passengers, and for every minute, 10 additional passengers should be appended to the queue. As the queue fills, every minute there will be 11 passengers checked in total by all 5 guards at their respective gates. The problem of the queue is separated into two problems, the line itself, and the efficiency of the security guard. By separating it into these two parts, it allowed me to easily define the constructor and modules of the two objects which make up the queue.
2. Design Considerations and Decisions
   1. When you consider what makes up a queue, the parts are not very different from one queue to another. Generally, you have the line, which ebbs and flows through a logical (or illogical in some cases) process towards the endpoint it is destined for.
   2. SecurityLine
      1. The line should be able to accept changes to its size by growing or shrinking dynamically based off how many elements it is comprised of.
      2. For visualization purposes, there should be some method by which you can iterate over the line in order to inform elements how many are in front of them, and also to record how many elements have been processed in total.
      3. The ArrayList object in java.util suits this perfectly, as the application will only be adding to the bottom of the queue, and removing from the top of the queue. And since the initial size is 20, and based on factor of reduction by 11 each minute, the object will not need to be resized.
      4. By not requiring resizing, an efficiency of O(1) is achieved for additions and removals from the top or bottom of the ArrayList.
      5. Also, by making the ArrayList private, it allows you to create separate list objects and manipulate those separately using public methods.
   3. SecurityGuard
      1. The process by which the line is managed before reaching their endpoint must also be configurable, as not all elements in the line will be the exact same necessarily.
      2. By initializing a private integer for the efficiency of the guard, it can then be modified by public methods, and allow for multiple guards with different efficiencies of people per minute.
3. Assumptions
   1. The first assumption is that no passenger would cut in line. Were a passenger to cut in line, there is no method implemented in the SecurityLine class to handle such an event. If this assumption were false, it would also create a bottleneck in the efficiency of the ArrayList of O(n).
   2. The second assumption is that there would only be 20 or less passengers in the line at one time. Were there to be more passengers in the line, it would not cause an issue, however it would create a bottleneck in the efficiency of adding new passengers of O(n).
   3. The third and final assumption, is that within the parameters of the requirements that no more than 10 passengers are being added every minute, there should be no IndexOutOfBounds exception at runtime.
4. Description of classes
   1. SecurityGuard
      1. Accepts arguments of a positive integer.
      2. Implements Private integer object to be set by argument
      3. Throws IllegalArgumentException if it is passed an integer of less than or equal to zero
      4. Implements Setter and Getter methods for the amount of passengers it can check per minute.
   2. SecurityLine
      1. Implements java.util.ArrayList and java.util.ListIterator
      2. Implements Private ArrayList, as well as Private Integer for initial passenger count, as well as the overall passenger count as more are added.
      3. Accepts no arguments, and initializes with 20 passengers.
      4. Implements Setter and Getter methods for adding and removing passengers from the ArrayList.
      5. Throws IllegalArgumentException if it is passed an integer of less than or equal to zero for both Setter and Getter methods.
      6. Implements a toS method, which returns the ArrayList concatenated into a string that has been delineated by new lines for every passenger. This method also appends a header to the string for visual aid in reviewing the output.
   3. TestAirportLine
      1. Implements org.junit.Assert.\*, org.junit.Test, and junit.framework.AssertionFailedError
      2. Invokes testPassing as a void which tests proper operation of a valid use for both SecurityGuard and SecurityLine objects
         1. This tests the required specifications per project requirements provided by the instructor.
      3. Invokes testGuardFailing as a void to verify that exceptions are thrown when SecurityGuard(int) and SecurityGuard.setNumberPersons(int) is invoked improperly.
      4. Invokes testLineFailing as a void to verify that exceptions are thrown when SecurityLine.addPeople(int) and SecurityLine.removePeople(int) are invoked improperly.
5. User Interface
   1. I chose not to implement a graphic user interface.
6. Testing and test cases
   1. Using Junit, I invoked three separate test cases in an effort to exercise all lines of code as well as all branches of logical operators implemented throughout SecurityGuard and SecurityLine.
   2. The first test case Invokes testPassing as a void which tests proper operation of a valid use for both SecurityGuard and SecurityLine objects
   3. The second test case invokes testGuardFailing as a void to verify that exceptions are thrown when SecurityGuard(int) and SecurityGuard.setNumberPersons(int) is invoked improperly.
   4. The third test case invokes testLineFailing as a void to verify that exceptions are thrown when SecurityLine.addPeople(int) and SecurityLine.removePeople(int) are invoked improperly.
7. error handling
   1. The main errors present would be whether a negative integer were passed as an argument to SecurityGuard(int), SecurityGuard.setNumberPersons(int), SecurityLine.addPeople(int), or SecurityLine.removePeople(int).
   2. In order to ensure that the arrays and integers were not manipulated directly, all objects which made up SecurityGuard and SecurityLine were cast as private. This alleviates any improper use of the classes when they are implemented.
8. lessons learned
   1. Something that I did learn as a result of testing whether an exception is exercised, was how to test them using the Junit testing framework.
   2. The primary lesson learned here, was how to decide which list type would benefit certain use cases. In the case of a queue, I have found that an ArrayList complements the nature of such an apparatus. Queues are expected to expand and contract to meet whatever load they are put under, this property is one of the main benefits of the ArrayList. The secondary benefit lies in the fact that adding or removing from the top and bottom of the ArrayList is an optimized functionality, if and only if the ArrayList is initialized with the largest expected size, otherwise it is a performance hit.
   3. In the process of figuring out whether to use an ArrayList or a LinkedList, I also learned a good deal of the benefits of the LinkedList list object. Should I ever have a fixed data set, and need to manipulate items into a correct order, or simply add/remove items from that data set, I will definitely use the LinkedList object type.