**COMP 2150 Assignment 2: More Fun With Airplanes**

**Due:** Monday, February 6, 2012, **before 11:00 am (precisely - electronic handin).** Note that this is before class starts. There's no reason to miss class the day of an assignment. As per assignment one, please make sure you are familiar with the electronic handin facility well before the due time of the assignment.

**Notes:**

• Please follow both the "Programming Standards" and "Assignment Guidelines" for all work you submit. Programming standards **1-25** are in effect, but see also the notes at the bottom of this assignment.

• Handin will be entirely electronic. Electronic handin instructions are on the website, and more specific details on how to arrange your files for handin will be provided before the due date. Official testdata will be provided several days before the assignment due date. Until then, work with your own testdata.

• This assignment is worth 65 marks in total. Question 1 is worth 20 marks. Question 2 is worth 45 marks. All assignments in this course carry an equal weight.

• Do **not** use built-in Java collection classes anywhere in this assignment. Plain arrays (not ArrayLists) may be used internally for the Seating Charts for each flight in question 1, and must be used to build an array of Security Lines in question 2, but should not be used to build other data structures. As discussed in class, arrays may also be used as temporary storage to support things like string parsing using split(), or for parameter passing in situations where you have multiple parameters of the same type (e.g., the cancellation and reservations in question 1 will include situations where these are being done for several individuals at the same time, and there is a need to store this data). Just don't use them to build data structures or for storage in data members in objects beyond these guidelines. If your solution requires ADTs other than those described above, these must also be your own linked structures.

**• You may not define or implement any Java Interfaces or Java Generics anywhere in this assignment.**

• Use polymorphism where possible: the ordered lists (or following my examples in class, the data that they store) and the event processing in question 2 must be polymorphic.

• Your program will be tested by the markers under Unix on the CCU machines, and must run with a simple javac/java as per the "Running Java" instructions on the course website (please make it obvious for each question what your main class is, to make this easier for the markers). This means that you cannot have any graphics (including file dialogs) generated by your program - just hard-code file names or ask for their names via console input. As stated many times, test it there yourself before handing anything in, and don't leave handins to the last minute!

**Question 1: Return of the Airline Seating**

Take your code from Assignment 1 (it must be YOUR code, not mine, though you are welcome to follow the general structure of the skeletal solution I have posted if you need to fix anything, and yours should be at least as good as mine in terms of OO - if it isn't, you will lose marks on this assignment for it as well). Beyond making sure your assignment works and is well-structured for what was asked for in the previous assignment, you must make the following changes (these are all pretty trivial if you've done a reasonable job of assignment 1):

• You must use abstract classes and methods for your ordered list. For example, my model solution defines what should be an abstract class for the data contained in an ordered list (similar to the unordered list example we did in class). That abstract class must have abstract toString() and compareTo() methods to allow the ordered list to operate properly. You will thus now be using polymorphism in a safe way. It is also possible (but not as convenient in terms of coding) to make the entire list an abstract class, but making the data it holds abstract is simpler and keeps the differences between classes simpler as well.

• Your data file will now be altered to define **emergency exit rows**. After each of the data lines that defines a section layout, there will be a second line with at least one entry on it. Each entry will represent the number of a row (**within the rows of that section**, starting at a **1 [one]** origin) that will have emergency exits at each end of the row. There may be many emergency exit rows in any section (or none, in which case this line will have the single entry **0** on it). Thus, the following becomes valid data: 565 //flight 565

• 2012/02/26 //date

• 2 //2 sections

• 2 3 3 //1st section: 2 rows: 3 seats/aisle/3 seats

• 1 2 //emergency exit rows in section 1: rows 1 & 2 (the 0th and 1st slots in your array of rows)

• 8 4 4 //2nd section: 8 rows, 4 seats/aisle/4 seats

• 5 //row 5 in section 2 is an emergency exit row. If this was 0, there would be no emergency exit rows.  
You can assume the defined emergency exit row(s) will be valid rows occurring in the particular section (or be 0, indicating no emergency exits in that section). In your seating chart output, use some obvious designation when you display an emergency exit row (e.g. a \* or E or some other additional character after the row itself).

• Minors are not allowed to be seated anywhere in emergency exit rows. Other than this, your seating algorithm should follow the same general principles as A1: try to seat everyone together in a block in the same row with preferences, then without, before finally resorting to individual seating if necessary. The addition of the emergency exit restriction means that any group with a minor will never be successfully seated as a block in an emergency row. It also complicates seating in a subtle way: previously, you could just count the number of available seats in a section and know that you could seat a group **somewhere**. Now, you can't just use a raw total, since some of those seats may be in rows where a minor couldn't be placed. You can't just discount those rows in the count entirely either, because if the group has to be split up, the non-minors could be placed there. So, you have to consider both the number of minors and non-minors separately in such a count to know whether things can work, or you will have situation where you will be rejecting groups that could still work if they were placed separately. Also, when you try and fail to seat a block, and are left with just seating individuals, you must ensure that minors are still seated away from emergency exits in the individual case. Be careful that you do know you can seat everybody separately if necessary (i.e. use the counts described earlier in this paragraph), or you may find a situation where you try forever to seat someone that could never be seated, or reject groups of passengers that could actually be seated individually according to our rules.

• You will be guaranteed test data that presents a situation where some individual seats can be considered both window and aisle seats, in at least two different ways. This is really a clarification of the last assignment, but we're going to be tighter on looking at this now. For example, the section description line: 4 1 3  
defines a section with four rows, each with a single seat (both window and aisle), an aisle, and then three more seats (including an aisle and window seat). More subtly, the section description: 2 2   
defines a section with two rows, each of which is a single block of two seats. This sesction **has** to have an aisle somewhere, since otherwise you wouldn't be able to get to the seats, but each end of the row would still have windows as well. For this assignment we will say that in the case of a single block like this, the aisle runs down the **left** (or lower-subscripted side) of each row - so while both of these seats would be window seats, the leftmost (seat 0) of the block is an aisle seat as well. This would only happen in a one-block row like this - otherwise, aisles are defined as above. My model solution from A1 (see the skeleton) solves both of these pretty easily by breaking up a row into a sub-array of Blocks, each of which contain Seats - so the subscripts of Seat and Block make it obvious what positions are windows and aisles (or both). That is not the only way of doing this, but it is a good one. If you had trouble getting your seating algorithm correct for preferences last time, you will gain a great deal of simplicity by viewing seat arrangements in this manner.

**Question 2: The Franz Kafka International Airport (FKI)**

The Franz Kafka International Airport wishes to do a study on the effects of their planned security arrangements on passengers. To do this study, you will write a Java program implementing an event-driven simulation according to the specifications that follow. **I am assuming you have never done any work with event-driven simulations, and so I have put some background on this in an appendix linked at the end of the assignment. This background will also be covered in class (just work on question 1 until then!), and will appear in your class notes. If you were not present in class when we went over this, I would suggest you read the appendix linked at the bottom first, and then come back here.**

Your code must be entirely your own, and you are also not allowed to use Java Generics or Interfaces of any kind (this includes writing your own Interfaces - write your own Abstract Classes and Abstract Methods instead). You must also use your own linked structures based on your own defined node class(es) to build all your data structures: no array-based data structures unless otherwise explicitly stated here, and no use of Java's built-in aggregate data classes.

Your simulation will involve only the security area of the airport: the station that separates the gates of the airport from the insecure check-in area. Thus, passengers will be arriving in your simulation after checking in with their airlines, holding anything they wish to take on the plane (carryons). Passengers *hope* to emerge from security and go to their gates. In this simulation, that may happen only after a long period of intervening activities (or possibly not at all).

Our simulated security area consists of a number (supplied in a data file) of Security Lines - these represent the familiar lineups you see at airports, where passengers get their carryons searched, and where passengers themselves pass through a metal detector and are questioned by security personnel. As in the real world, only one person can be processed at a time, and everyone else lines up behind them. As part of being processed in a security line, passengers may also be sent to a secondary screening area if a longer interview is deemed to be necessary. This is a separate holding area with another waiting room, where passengers wait their turn to be interviewed in one of a number (again, supplied in a data file) of interview rooms.

**Data File and Passengers**

In the data files for this assignment (update: Please read the filename in from console input, so it's easy to run this multiple times) you do not have to worry about bad or erroneous data anywhere. You are ***not allowed*** toread the entire file into memory ahead of time. Any data file will begin with a single line with two numbers on it: the first of these is the number of security lines that will exist (you must use an array of security lines, and security lines should be numbered from 0 to correspond with this), while the second is the number of interview rooms that exist (Hint: see the last point in the bulleted list of notes at the end of this document). There is guaranteed to be at least one security line and one interview room. After this first line, every data line describes a passenger's arrival, and is **ordered by passenger arrival time (just like our bank example in class and in the appendix!)**. Each line looks as follows:

14 2 24 SCF

The first integer is the time unit at which the passenger will arrive at the security area (i.e. after checking in with their airline, dumping checked luggage, and getting a boarding pass).

The second integer is the number of carryon items the passenger is attempting to bring through security (possibly 0, but with no upper limit).

The third integer is the simulation time at which the passenger's plane will take off (thus, this passenger has 24-14=10 units of time between when they arrive and when their plane takes off - this may or may not be enough time to clear security). These three integers will ALWAYS be present.

The remainder of the data line is entirely optional, and will be a series of character codes, in any order, describing attributes of the passenger.

If a capital "F" is present, it means this is a first class passenger.

If a capital "S" is present, it means this passenger will display suspicious behaviour to security personnel. Note that this does not mean the passenger is guilty of anything - just that security personnel will find them suspicious.

Finally, if a capital "C" is present, it means the passenger has an inappropriate carryon (i.e. items that cannot be carried on the plane - for any passenger, only on e carryon will ever be illegal, and you will never have a C for someone with no carryons. Any other letters than these should just be ignored.

In addition to these, there are two important attributes of passengers that are not contained in the data file.

First, each passenger has a number to identify them (everyone is only a number at the Franz Kafka Airport). This is generated when the passenger is created: start numbering passengers at **42** and go up from there.

Secondly, each passenger has a random chance of being a criminal (someone may be a criminal without displaying suspicious behaviour). In order to have one single right answer, we will set up random numbers in this program so that everyone will be generating the same random sequence, every time the program is run. To do this, you will create **one** **static** random number generator object in your passenger class (this will require importing java.util.\* to compile), **exactly** as follows:

**private static Random criminalGenerator = new Random(1001);**

When you create a passenger object as you read the data file, use this function in a conditional statement to test for criminality as follows:

**if ((criminalGenerator.nextInt(100)+1) < CRIME\_PERCENT) // if this is true, they're a criminal...**  
  
You must ensure that this is the only time this random number generator is called (i.e. once per passenger creation), so that the exact same individuals will be criminals each time the program is run, ensuring there is only one correct answer. Use a value of 20 for the **CRIME\_PERCENT** constant. While 20% of the population being criminals is highly unusual to you and I, this is what FKI security wishes to anticipate. It also ensures a greater likelihood of all possibilities being tested with a much smaller dataset and less output to wade through.

Beyond these, there will be several attributes you will want to define for passengers to keep track of the desired output statistics and to generate output according to specifications (see below for both of these).

**Events and Simulation**

You know that an event-driven simulation at its core uses a priority queue (referred to as the Event List here) which keeps track of future (pending) events. If you don't realize this, **read the appendix**. Your priority queue will be implemented using an ordered list (don't go as far as some of the elaborate tree-based implementations you saw in 2140) - this could be adapted from your code from question 1, for example.

All events in this simulation happen to individual passengers only. The simulation itself just removes the next (first) event from the event list (**which is kept in order by time, and if there is >1 event at the same time, ordered within that by passenger number, lowest # first**), and causes that event to make changes to the simulated world (which in part may be by creating events intended to occur during the current or future units and putting them in the event list). You will also need ordered list(s) elsewhere in this question, and so this list class should be made general through having a polymorphic data hierarchy, analogous to what you saw in class. Similarly, because there will be many types of events that will need to be treated similarly (making any event happen), you should also have a hierarchy of event types that uses polymorphism for this. In addition, you will need a linked-list based Queue (also possibly adaptable from your own previous work in this course) to use in the various places that people can wait. Get these data structures working first, and test them well (this should already have happened from assignment 1, for everything but the queue!)

The following describes the actions that may occur to a passenger, from which you must define appropriate event types. **Please pay close attention to these details**:

An **Arrival** **Event** can happen in **two** different ways in this world. The first is obviously from the data file - you will prime the main simulation loop by reading in the first arrival and inserting it, so that it is the first event that will occur in this universe. Creating an arrival event from the data file should also create a passenger (giving them a number). It is also possible for an arrival to occur for a passenger that already exists (you will see from later events that passengers can get sent back and must begin the security process again). In that case, you do not create a new passenger or change their passenger number, since they already exist. In either case, the passenger arriving at the security area must then choose the **shortest available security line** and enter it. If there is a tie in length between two lines, the passenger will choose the lowest-numbered line. Upon entering the line, if there is no one in the line, a **Start Search** **Event** (below) will occur on the **next** time cycle. This delay represents the time required for a person to lay their bags out in front of the workers, remove laptops, etc. If there is already somebody being searched in the line, the passenger will join the queue associated with that line (this is similar to the Bank example from the appendix/class, but with multiple lines). After processing an arrival, if this was from the data file, you must also read in the next arrival event and put it in the event list, since this is when that arrival might become relevant (again, just like the bank example). If it's not clear why, you need to go over that example again.

A **Start Search Event** happens after someone reaches the front of their security line. The length of a search will depend on two things: first, the number of carryons. A search will take one time unit per carryon (but define this as a constant so you can change it easily in future). The search time will also depend on whether the person is suspicious to workers (that is a passenger attribute). A person's search time is also one unit (again, make it a constant somewhere) and this will be multiplied (person only, not carryons) by a factor of 4 (another constant) if they are suspicious. When processing this event, this allows us to know the time that the search will finish, and create a corresponding event for that in the event list.

A **Finish Search Event** happens when the search is completed (**not** a time unit after, i.e. there is no time delay for picking up one's bags at the end). Finishing a search should move someone out of the lineup, set the next person to start their search if there is someone following them in the lineup (after the same one unit delay described above to lay their bags out) . The person having their search finished will always leave the line, but where they go from there has three possibilities. First, they may have no problems and be allowed to go (i.e. they will exit the simulation and you can tally the desired statistics described below). Secondly, the search may have found an inappropriate carryon bag (a passenger attribute), in which case they will be sent back to check their carryon. There will only be one inappropriate carryon bag for a passenger, and this action will cause the number of carryons for that passenger to go down by one, and the attribute in the passenger saying they have an inappropriate carryon to be cleared out. The time this takes in the real world is represented by another delay (a constant: 8), and will result in a new arrival event for that passenger at that future time (i.e. they get to start all over again, choose the shortest line, etc.). Finally, if they are suspicious (the passenger attribute), they will be sent for secondary screening - a future **Enter Secondary Area** event will occur after a fixed delay representing the time it takes for someone to escort them to the secondary holding area (2 units - a constant!). If a passenger is both suspicious and has inappropriate carryons, the inappropriate carryons will be detected first. They remain suspicious, and will later require the extended search this causes, when they return, even if they have been through the same line already (for maximum inconvenience).

An **Enter Secondary Area** event happens when someone has aroused suspicion during their search and was sent for secondary screening. This event puts them in a waiting room in the secondary holding area where they get to wait for an interviewer (i.e., this is a separate queue from all the security lines). There are a specific number of interview rooms in the simulator, as described earlier, and if any of these is available, the passenger will have a **Start Interview Event** occur during the same time cycle (i.e. unlike the security lines, there is no one-unit delay to start up here). If all the interview rooms are in use, they will wait in order of arrival in the waiting area (i.e. a regular queue). Note that this is a little different than the setup for a security line, since several people can be undergoing interviews at once (possibly, depending on the number of rooms defined), but there is only one lineup.

A **Start Interview Event** happens when someone is no longer waiting for an interview room. All interviews in this version of the program will be fixed at 5 time units (another constant). When a start interview event occurs, we know when the corresponding **Finish Interview Event** will happen, and can create such an event and insert it properly in the event list. Starting an interview means one more interview room is currently in use.

A **Finish Interview Event** occurs when the interview is completed. The outcome of the interview is also random in this simulation. **If the passenger is a criminal** (the attribute assigned when the passenger was created) there will be a given chance that will be uncovered by the interviewer. We will use a similar static Random object in this class:

**private static Random crimeCatcher = new Random(1001);**

and the test to see if someone is caught will be:

**if ((crimeCatcher.nextInt(100)+1) < CAUGHT\_PERCENT) { //caught!**

This test will **ONLY** be performed if someone is actually recorded as being a criminal in their passenger attribute (otherwise the number of calls to this routine in your code will differ and you will not get the correct output!). This also means that truly innocent people always get through in the end, preserving one small measure of sanity in this example. Use a value of 50 for the constant **CAUGHT\_PERCENT**.

If the person is a criminal and is caught, then they are arrested - this is the outcome for this passenger. If not, they will be allowed to leave to go to their gates. Departing from the interview is exactly the same as departing from the original security line, except that there are two outcomes here instead of three. Also, the actual departure will cause an interview room to be freed and someone from the secondary screening area queue to move into it (i.e. have a **Start Interview Event** defined in this same time cycle. In all cases where a new event occurs the same time cycle, you must still create the event object and put it in the event list, in order that the simulation proceeds appropriately.

You can see that there are some important differences from the example Bank simulation here: more than one place that one can leave the simulation, several outcomes when a passenger does leave, more than one possibility of arrival, as well as two stages to wait and several possible lineups at the first stage. There are also some strong similarities, however, in the basic simulation engine and file arrivals, and that pseudocode will be of use to you here.

**Output**

Your output should consist of two phases. First, as the simulation proceeds, you should have a line of output for every event that occurs (that is the reason in part for the breakdown of event types above, so that we have this generated at coherent times). Each line should state the time, passenger, and type of event that occurred, and specific types of events must also have additional items:

• An Arrival Event should state the number of carryon bags and the security line they chose, and should also flag whether this is a returning arrival as opposed to the first time the passenger has shown up in the security line. In addition, if the passenger is a criminal, this should be noted.

• A Start Search Event should indicate the security line number, the amount of time spent waiting to get to the front of this line (just this time through the lineup, and this does not include the one time unit it takes to lay bags out [described above] so this may be 0), the number of carryons, whether the passenger is suspicious, and the number of time units the search will take.

• A Finished Search Event should show the security line number, and the result/outcome of the search

• A Enter Secondary Area Event should indicate how many interview rooms were available on arrival

• A Start Interview Event should indicate how many empty interview rooms remain now that this one has been taken, and how much time the passenger spent waiting to have the interview (this does not include the time for the interview itself, or any waiting outside the secondary area - just the time spent waiting in the secondary area for an interview room, so this may be 0)

• A Finished Interview Event should state the result (arrested/released), the number of empty interview rooms that remain, now that this one has been opened up, and the total time the passenger spent waiting (earlier in the search lines and waiting for the interview, together).

For example, the following output follows the above description. Yours doesn't have to look precisely like this, but the more close it appears, the easier it will be for the marker to follow your solution.:

Welcome to the Franz Kafka Airport!

We have 2 security lines and 2 interview rooms for your inconvenience.

Time 1: Passenger 42 arrives (a criminal - 2 carryons, enters security line 0)

Time 1: Passenger 43 arrives (3 carryons, enters security line 1)

Time 1: Passenger 44 arrives (2 carryons, enters security line 0)

Time 1: Passenger 45 arrives (3 carryons, enters security line 1)

Time 1: Passenger 46 arrives (3 carryons, enters security line 0)

Time 1: Passenger 47 arrives (3 carryons, enters security line 1)

Time 2: Passenger 42 starts search (line 0, waited 0 - with 2 carryons plus suspicion, requires 6 units)

Time 2: Passenger 43 starts search (line 1, waited 0 - with 3 carryons plus suspicion, requires 7 units)

Time 2: Passenger 48 arrives (3 carryons, enters security line 0)

Time 2: Passenger 49 arrives (3 carryons, enters security line 1)

Time 2: Passenger 50 arrives (4 carryons, enters security line 0)

Time 2: Passenger 51 arrives (3 carryons, enters security line 1)

Time 2: Passenger 52 arrives (3 carryons, enters security line 0)

Time 2: Passenger 53 arrives (a criminal - 4 carryons, enters security line 1)

Time 3: Passenger 54 arrives (3 carryons, enters security line 0)

Time 4: Passenger 55 arrives (2 carryons, enters security line 1)

Time 4: Passenger 56 arrives (a criminal - 3 carryons, enters security line 0)

Time 5: Passenger 57 arrives (0 carryons, enters security line 1)

Time 5: Passenger 58 arrives (0 carryons, enters security line 0)

Time 5: Passenger 59 arrives (0 carryons, enters security line 1)

Time 8: Passenger 42 finishes search (line 0 - result: passenger is suspicious, secondary screening needed)

Time 9: Passenger 43 finishes search (line 1 - result: improper checked baggage found, passenger rejected)

Time 9: Passenger 44 starts search (line 0, waited 7 - with 2 carryons, requires 3 units)

Time 10: Passenger 42 enters secondary security area waiting room

Time 10: Passenger 42 starts interview (waited 0, 1 rm(s) remain)

Time 10: Passenger 45 starts search (line 1, waited 8 - with 3 carryons plus suspicion, requires 7 units)

Time 12: Passenger 44 finishes search (line 0 - result: released, spent 7 units waiting)

Time 13: Passenger 46 starts search (line 0, waited 11 - with 3 carryons plus suspicion, requires 7 units)

Time 15: Passenger 42 finishes interview (result: arrested, spent 0 units waiting, 2 rm(s) remain)

Time 17: Passenger 43 arrives \*again\* (2 carryons, enters security line 0)

Time 17: Passenger 45 finishes search (line 1 - result: passenger is suspicious, secondary screening needed)

Time 18: Passenger 47 starts search (line 1, waited 16 - with 3 carryons plus suspicion, requires 7 units)

Time 19: Passenger 45 enters secondary security area waiting room

Time 19: Passenger 45 starts interview (waited 0, 1 rm(s) remain)

Time 20: Passenger 46 finishes search (line 0 - result: passenger is suspicious, secondary screening needed)

Time 21: Passenger 48 starts search (line 0, waited 18 - with 3 carryons, requires 4 units)

Time 22: Passenger 46 enters secondary security area waiting room

Time 22: Passenger 46 starts interview (waited 0, 0 rm(s) remain)

Time 24: Passenger 45 finishes interview (result: released, spent 8 units waiting, 1 rm(s) remain)

Following this output, you must then print the summary information for the study. The FKI security people are especially interested in two subsets of passengers: those that they caused to miss their planes (this can be calculated by when they were released from security and, and those that were arrested. You must maintain an ordered list (by increasing passenger number) for **each** of these, adding each passenger in that category to the list as they leave the security simulation. After all events are complete you should print the contents of both of these lists, with appropriate headers. You need to print the following elements for each passenger in each list: their number, number of carryons, ticket class, status (suspicious/criminal or both), time when they first arrived at security, time when they finished their first search, time when they left the security area to either go to gates or to be arrested (i.e. time they left the simulation), and the total time they spent waiting. Note that the same passenger should not appear in both lists - if they were arrested, it is understood that they missed their flight.

For example, this part of your output might look like this:

...All events complete. Final Summary:

Passengers Arrested:

ID Flight Class Carryons Status Arrival Searched Exit Waiting

Number Time Time Time Time Time

-----------------------------------------------------------------------------------

42 24 First 2 SC 1 8 0 0

53 5 First 3 SC 2 80 0 53

Passengers Missing Their Flights:

ID Flight Class Carryons Status Arrival Searched Exit Waiting

Number Time Time Time Time Time

-----------------------------------------------------------------------------------

43 15 Coach 2 S 1 61 68 37

47 40 Coach 2 S 1 68 75 44

48 40 Coach 2 2 72 72 53

49 24 Coach 2 S 2 74 81 49

50 25 Coach 4 2 31 31 23

51 10 Coach 3 S 2 41 48 31

52 5 Coach 3 2 36 36 29

54 20 Coach 3 S 3 44 51 33

55 18 First 1 S 4 80 87 55

56 20 Coach 3 C 4 49 49 40

Total time spent waiting is defined as the number of time units spent "in line" - that is, the total time waited in both the security line (possibly more than one trip through this line, even) and the secondary waiting area if the passenger was sent there. It does not include the initial time cycle required to place carryons for searching (described in Arrival Events, above), or any extra time required to go back and check bags or be escorted to the secondary area.

Following these lists, you should print the total number of passengers processed through security (i.e. left the simulation in any way - just because somebody arrived a second time because they had to go back and check carryons does not mean they get counted twice), and the average time spent waiting per passenger (counting everyone through the whole simulation, no matter what the outcome of any particular passenger).

**Additional Notes/Hints:**

• It is important that the two Random objects be separate and static, and that the calls to these object's methods be identical in number - otherwise you will not get the same correct answer that is expected.

• You are not allowed to have more than one arrival from the input file in the event list at once (somebody returning after checking bags does not count in this)- you should understand why this is the case given the class notes and the Bank example.

• Start by getting the data structures working, and then JUST get arrivals working (i.e. people arrive, and just never leave). Then gradually flesh out the simulation by getting the other possible events working and interacting properly.

• Passengers never change lines (at least in this assignment), no matter how slow-moving a security line is.

• Note that you never care **which** interview room someone is in, only the number of free or occupied rooms. This can simplify how you treat interview rooms a great deal.

Background information on event-driven simulations in general is here.