CAB302 Software Development
Assignment 2: Test-Driven Development and Graphical User Interface Programming

Part 1: Test-Driven Development

Semester 1, 2016

Due date: Thursday, June 2, 2016 (late in Week 13)
Weighting: 35%. Part 1: 17.5% Part 2: 17.5%
Assessment type: Group assignment, working in pairs
Version: 1.0.1: May 8 11:26PM – fixed typo in Javadoc

Learning Objectives for the Assignment:

- To experience team-based program development using an approach similar to test- driven development (Part 1 of the assignment).
- To practice Graphical User Interface programming (Part 2 of the assignment).

http://www.qantas.com/travel/airlines/aircraft-seat-map-boeing-744/global/en
http://www.qantas.com/travel/airlines/aircraft-seat-map-airbus-380/global/en

The Scenario — Aircraft Bookings.

This assignment involves simulating the bookings for daily flights between the east coast of Australia and the west coast of the United States. We will base our simulation on QANTAS, which has at present (there have been some recent revisions):

- BNE > LAX : QF15 (Daily services: Boeing 747-400)
- SYD > LAX : QF11 (Daily services: Airbus A380-800)
- MEL > LAX : QF93 (Daily services: Airbus A380-800)

For those who haven't yet experienced it, LAX is the airport code for Los Angeles International Airport. Travelers' impressions of LAX are readily available on the web and we will not need to consider them here.

In Part 1 of this assignment you will play the role of a small programming team tasked with developing a set of Java classes to support an automated system to simulate passenger bookings, confirmed numbers flying and final numbers resulting from these flights over the period defined by the simulator. This will lead, naturally enough, to a GUI using these classes in the second part of the assignment. Part 1 (17.5%) will require that you complete the text based simulator and Part 2 (17.5%) will require that you use some of these ideas in a GUI environment. Automated testing is required for both parts of the assignment, although it is more heavily weighted in Part 2 as this is new to you.

An outline of the specific tasks required to complete Part 1 is given below. Details of the technical requirements for each of the classes to be produced are given in the Javadoc API specification. This will require a lot of exploration before you come to grips with what is required. Read the spec and the Javadoc carefully, but you should pay particular attention to the assumptions that we have made.

In what follows we will describe successively the problem domain, the structure of the simulation (this is written for you) and then the tasks that you have to complete in part 1. The requirements for the GUI are ***not considered at all in this document***.

The Problem Domain:

Computerised reservation systems have been in use for around half a century, and are remarkably effective. They are also remarkably complex. We will not mimic too much of their functionality: we are concerned only with simulating some of the numbers and looking at how passenger bookings vary as we change some of the simulation parameters controlling passenger flows. We will never record a passenger name or address – we are concerned only with their existence and how much they paid for their tickets: their fare class.

The Simulation:

The simulation is a bit complicated, but you don't need to understand every last detail of the approach. In practice, people may book their flights as much as a year in advance and we simply don't wish to consider that sort of time frame. We will thus have a short initial period of weeks during which there are no flights, and then there is an ongoing period in which we are both flying passengers and booking for the future. Each day, we cycle through the bookings for each plane for each day, from the earliest departure date to the last. Obviously, we do not take any bookings after the end date of the simulation, and we will show far less respect for our customers than QANTAS or another airline might in the circumstances, throwing away all the remaining passengers who cannot be accommodated on a flight.

Time is organized according to days, and we begin at time T=0, and there is a set routine for each day. The simulation operates roughly as follows, though as been noted, you do not need to understand every last aspect of this process initially:

- The simulation runs a fixed number of weeks, and we begin with an airline launch – i.e. we take bookings for a few weeks before we actually fly anyone.
- On any given day, we are approached by some random number of potential passengers. We assume that ***on the average***, this is much the same as or slightly above our overall capacity. But we model this number with a random process (Gaussian – more on this in the podcasts) and we have this number of passengers to process.
- We assume that people are trying to book for some flight over the ensuing weeks. There are a lot of complications here, but we keep it simple and assume that the available dates are equally probable.
- We will assume the same four class structure (First (F), Business (J), Premium Economy (P) and Economy (Y) for all flights. This is a simpler version of reality: for example, QF doesn't offer First Class directly from Brisbane, there are several fare levels across Business and Premium, and there are a huge number of fare gradations in Economy based on flexibility and discounting. We don't care.
- Our passengers are all associated with a given fare level – F, J, P, Y. We assign their preferences based on probabilities in the model – the probability of Economy is very high, the probability of First is very low (usually under 3%).
- Once we have created a passenger, and we have their preferred departure date, we

try to book them on a plane. If we can do this, we do it straight away. If we can't, then we go ahead and put them in a queue or waiting list for any flight up to 7 days after their preferred date. This is again simple, but it shows the basic idea.

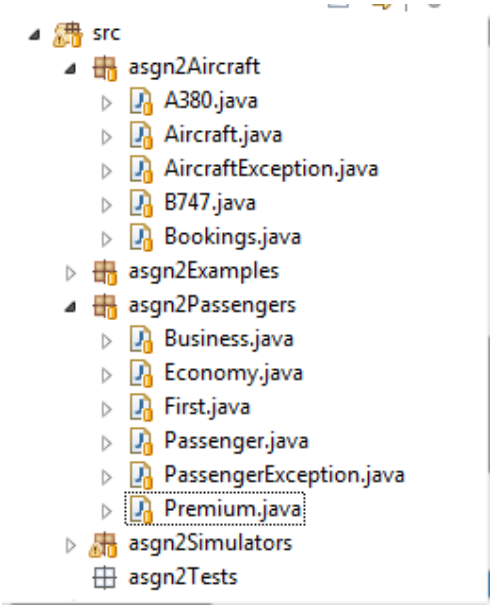
- If we cannot book the passenger on their preferred departure date – and we assume that they are just as happy to fly out of BNE, SYD or MEL – and we can't fit them in a queue, then we refuse the booking.
- Each day, if we can, we try to transfer people from the queue and fit them onto flights. If we can't do this by the time the deadline of 7 days is reached, their booking is refused.
- Over time, we fill up the aircraft over many weeks in advance. When we reach each departure date, we try to fill up every seat we can by upgrading, first J->F, then P->J, and finally Y -> P. Clearly we can't upgrade from First.

There are many more simplifications in this model. In particular, we don't care how passengers return, or even whether they do: we are not modelling the return flights.

Essential Parameters:
In the release version, the essential simulation parameters are *all* set up for you in the class `Simulators.Constants` and the aircraft capacities are given to you in the relevant `A380` and `B747` subclasses. Initially, we will not require you to vary any of these parameters, although you may do so in the source files directly.

Many of these things don't matter to you initially, but you do need to understand the hierarchies which underpin your work. Your tasks for part 1 are limited to the `Passenger` and `Aircraft` hierarchies. Each of these has at its root an abstract class which barely qualifies as abstract. Most of the working code for each hierarchy needs to be supplied in these classes. Your task is to implement these classes, their designated subclasses, and to write comprehensive unit tests for each of them.

The Source Packages:



The overall package structure is as shown in the image above. Your goal is to complete the hierarchies in the packages `asgn2Aircraft` and `asgn2Passengers`. The package `asgn2Examples` contains examples for the GUI work of part 2. The package `asgn2Simulators` contains code for controlling and running the simulation and you will

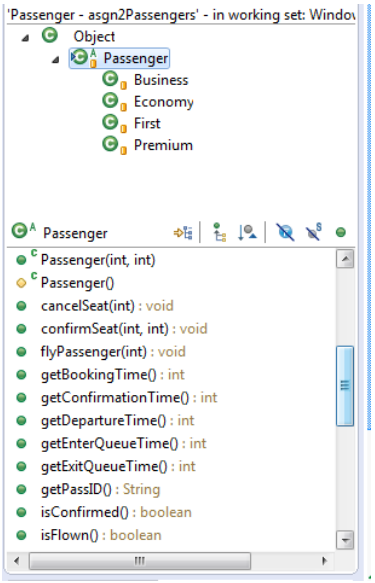
not contribute to it for part 1, though this is where you will place the GUI code for part 2. As usual, the asgn2Tests package is provided empty, and you must fill it with the designated unit test classes. Let us consider the two key packages in more detail.

Specific Tasks for Completing Part 1 of the Assignment

Your tasks in part 1 of the assignment are to complete the packages `asgn2Aircraft` and `asgn2Passengers` and to complete unit tests for the specified classes.

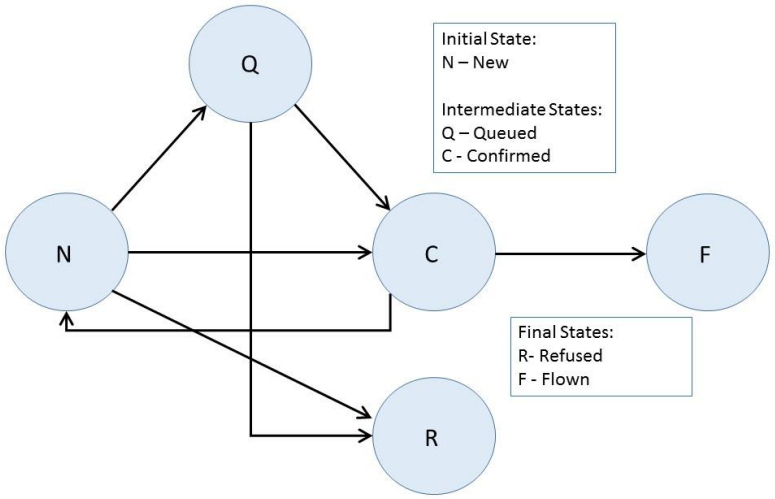
The `asgn2Passengers` Package:

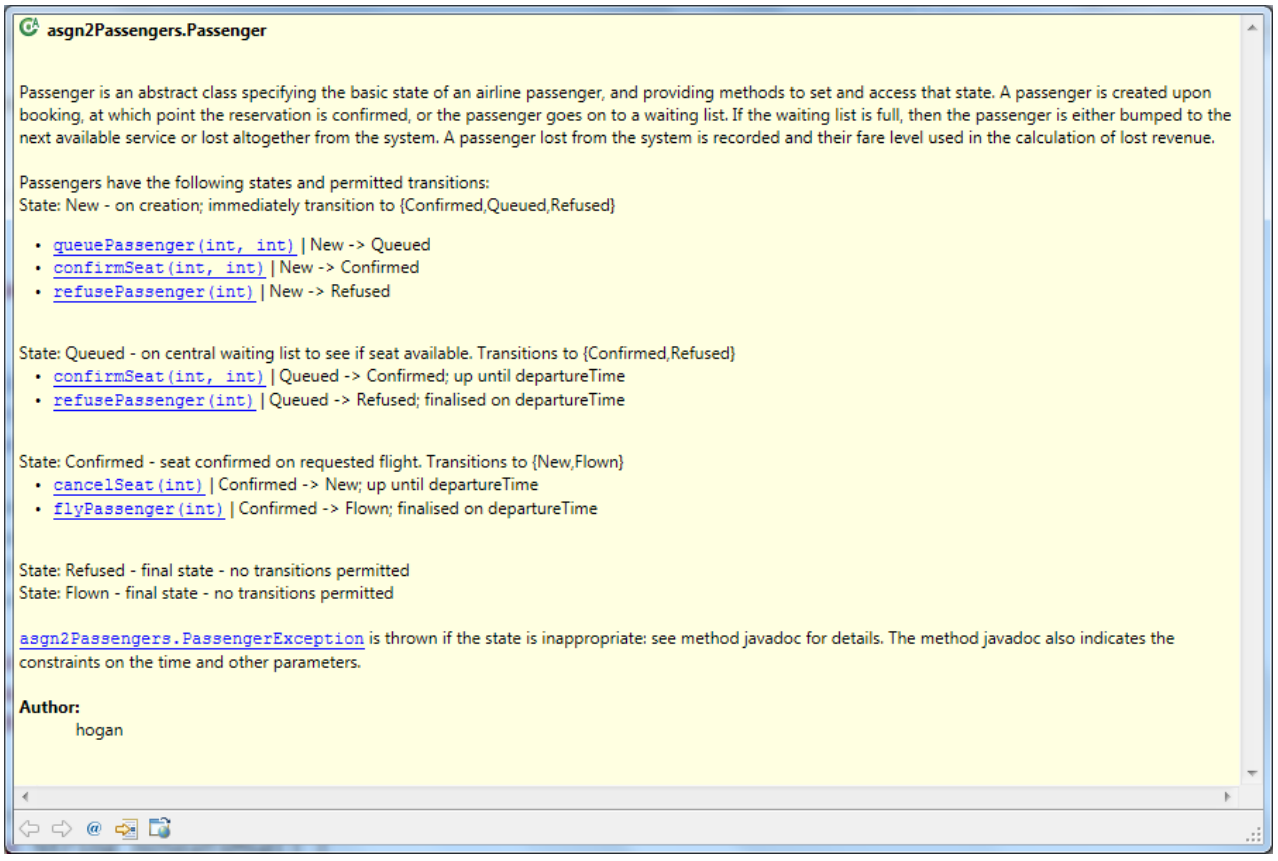
The `asgn2Passengers` package is built around the abstract class `Passenger.java`. The hierarchy, and most of the methods are shown in the type view from eclipse below. Almost all of these methods have single, concrete implementations in Passenger, and almost none of them are over-ridden below. You will, however, need to deal with polymorphism a lot in this assignment. We take Passenger arguments wherever possible, and we specialize the methods that use them wherever appropriate. This is the case for the simple `String` method `noSeatsMsg()` and for the more complicated `upgrade()`, but also for the passenger counting methods `in Aircraft` and its descendants.



As in the first assignment, I will supply the String methods, and you will not be required to test the getters and setters – as before I will also tell you explicitly which are to be tested and which you need not worry about. However, there is one crucial issue you need to understand here, and that is that the Passenger class supports a number of different states for a passenger depending on their progression in the booking process. These states have their own rules, and each of the state transition methods supplied here – `cancelSeat`, `confirmSeat` and so on – change the state fields of the Passenger, and are called from methods in other classes which move the Passengers around – especially those in the `Aircraft` hierarchy.

The available states and the permissible transitions between them are summarized in the figure below. The Javadoc for the Passenger class also provides very detailed information at the bottom of the next page.





asgn2Passengers.Passenger

Passenger is an abstract class specifying the basic state of an airline passenger, and providing methods to set and access that state. A passenger is created upon booking, at which point the reservation is confirmed, or the passenger goes on to a waiting list. If the waiting list is full, then the passenger is either bumped to the next available service or lost altogether from the system. A passenger lost from the system is recorded and their fare level used in the calculation of lost revenue.

Passengers have the following states and permitted transitions:
State: New - on creation; immediately transition to {Confirmed,Queued,Refused}

- `queuePassenger(int, int)` | New -> Queued
- `confirmSeat(int, int)` | New -> Confirmed
- `refusePassenger(int)` | New -> Refused

State: Queued - on central waiting list to see if seat available. Transitions to {Confirmed,Refused}
- `confirmSeat(int, int)` | Queued -> Confirmed; up until departureTime
- `refusePassenger(int)` | Queued -> Refused; finalised on departureTime

State: Confirmed - seat confirmed on requested flight. Transitions to {New,Flown}
- `cancelSeat(int)` | Confirmed -> New; up until departureTime
- `flyPassenger(int)` | Confirmed -> Flown; finalised on departureTime

State: Refused - final state - no transitions permitted
State: Flown - final state - no transitions permitted

`asgn2Passengers.PassengerException` is thrown if the state is inappropriate: see method javadoc for details. The method javadoc also indicates the constraints on the time and other parameters.

Author:
hogan

These transition methods, as you will see, also guard the parameter choices and the entry states very carefully, and throw many exceptions. These are the source of most of the exceptions thrown from the methods in the `Aircraft` hierarchy. DO NOT test them twice. I have usually indicated this by having a much less detailed description in the higher level method, and a link to the `Passenger` methods for details.

Your explicit tasks in the Passenger hierarchy are then to:

- Complete `asgn2Passengers.Passenger`
- Complete `asgn2Passengers.First`
- Complete `asgn2Passengers.Business`
- Complete `asgn2Passengers.Premium`
- Complete `asgn2Passengers.Economy`

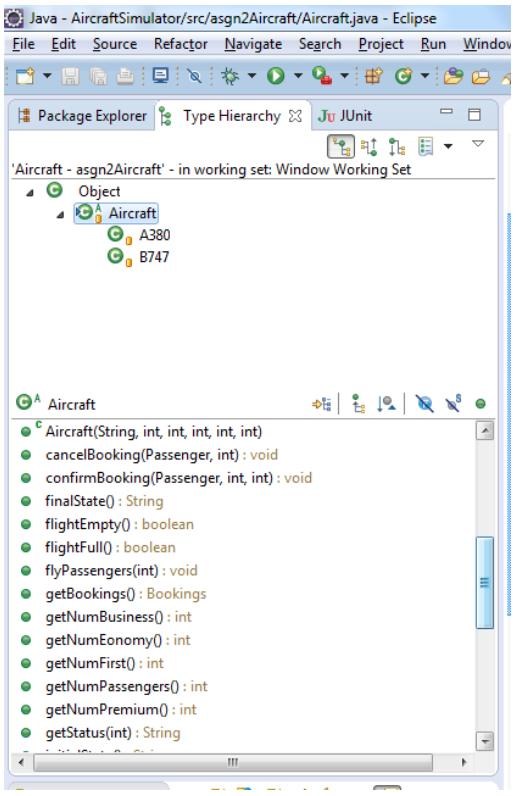
The class `asgn2Passengers.PassengerException` is supplied. You must also complete the following test classes in the package `asgn2Tests` (some of these will be trivial, but use separate files as the tools make it easy to do it this way):

- `asgn2Tests.FirstTests`
- `asgn2Tests.BusinessTests`
- `asgn2Tests.PremiumTests`
- `asgn2Tests.EconomyTests`

As in assignment 1, we will designate one of these subclasses as the focus for the tests of the root abstract class. Please place your tests for `asgn2Passengers.Passenger` in the class `asgn2Tests.FirstTests`.

The `asgn2Aircraft` Package:

The `asgn2Aircraft` package is similarly built around the abstract class `Aircaft.java`. The hierarchy, and most of the methods are shown in the type view from eclipse below. These methods have single, concrete implementations in the superclass. The subclasses are really there to allow easy specification of the default capacities and configurations and to include a name for the aircraft type. Here we have the two, A380 and B747.



Most of the methods in this class have a clear purpose and require little explanation. But we should discuss briefly a method such as `cancelBooking` as an example. Here I am interested only in the general structure – you have to work out the details

```
public void cancelBooking(Passenger p,int cancellationTime) throws … {
            //Some local exception stuff
            //Transition method on the passenger
            //Update of status string for the aircraft (see below)
            this.status += Log.setPassengerMsg(p,"C","N");
            //Remove passenger from the seat storage for the aircraft
            //Decrement the counts – this needs to be polymorphic
      }
```

So, here I have given you the string update for logging, and the rest by replacing some lines in my code with comments. This structure is the same or very similar for the other methods such as `confirmBooking`. Have a good look at the Javadoc for more information.

Your tasks in this package are then as follows:

- Complete `asgn2Aircraft.Aircraft`
- Complete `asgn2Aircraft.A380`
- Complete `asgn2Aircraft.B747`

We provide implementations for the following classes:

- `asgn2Aircraft.AircraftException`
- `asgn2Aircraft.Bookings` (a simple record class)

You are further required to provide comprehensive unit tests for the class `Aircraft` by implementing the test class: `asgn2Tests.A380Tests`. It is NOT necessary to provide additional tests for `asgn2Aircraft.B747` or for `asgn2Aircraft.Bookings`. The variations between the Aircraft classes are trivial and do not warrant any additional work.

Some Guidance:

Implementing these hierarchies is relatively straightforward, and most of the subclasses really are as trivial as they seem. Working through some of these transition methods, however, is not easy. You will need to understand the simulation loop properly in order to appreciate the role of some of the complex methods.

Source Control:

As stated previously, proper source control is mandatory for this assignment. That means a genuine source control system, including a remote repository accessible by both members of the team, and on request, by us. You will be required to include your commit logs as part of the final submission. We can and will make inferences about the relative contributions of team members if we see a substantial dominance of the commit log by one person. We have had people try to argue equality of contribution based on a single commit. This is not credible. Please ensure that there is clear evidence of your contribution.

A few pointers:

- Commit as often as you like – we really don't mind, and we will check the commits if we think it is ridiculous.
- Don't worry if the commit logs look messy. Good – it probably means that they are real.
- Finally, and let me make this ***very clear***, Dropbox is NOT a source control system.

Coding and TDD:

The basic coding tasks are laid out above. However, the reason we want you to work in pairs is to try to experience the TDD approach. For that reason, we require you to split the roles. The workload is roughly the same for each of the two hierarchies, so:

- For the `Passenger` hierarchy, assign student 1 to write the tests and student 2 to write the code to pass them. Mark these contributions with author tags and with specific commits.
- For the `Aircraft` hierarchy, reverse the order: assign student 2 to write the tests and student 1 to write the code to pass them. Again mark these contributions with author tags and with specific commits.

Unit testing is now something we expect you to do well, and for it to be a tool to be used as part of the development process. For assignment 1, it was something new and we required some extensive testing. Here, it is more restrained, and the classes are designed to be pretty straightforward to test. As previously advised, we shall also be sharper in our marking of code quality defects. The overall Java project comprises the packages as shown in the hierarchy earlier. Ensure that you follow these package and class names precisely (and do not introduce any classes, public methods or public fields other than those specified).

Academic Integrity

Please read and follow the guidelines in QUT's *Academic Integrity Kit*, which is available from the Blackboard site on the `Assessment` page. Programs submitted for this assignment will be analysed by the MoSS (Measure of Software Similarity) plagiarism detection system (`http://theory.stanford.edu/~aiken/moss/`).

Assessment

Submitted assignments will be tested automatically, so you must adhere precisely to the specifications in these instructions and on Blackboard. Your program code classes will be unit tested against our own test suite to ensure that they have the necessary functionality. Your unit test classes will be exercised on defective programs to ensure that they adequately detect programming errors. Apart from the quantitative results of these tests, the quality of your code will also be considered, so all of your classes must be presented to a professional standard. Further details about assessment criteria will appear in the rubric.

Submitting Your Assignment

Full details of required file formats for submissions will appear on Blackboard near the deadline.

You must submit your completed assignment as a complete source tree in a form to be specified closer to the due date. For now, you ***must*** respect the class name and package structures I have specified. You must submit your solution before midnight on the due date to avoid incurring one of the new and very firm late penalties. You should take into account the fact that the network might be slow or temporarily unavailable when you try to submit. **Network problems near the deadline will not be accepted as an excuse for late assignments.** To be sure of receiving a non-zero mark, submit your (perhaps incomplete) solution well before the deadline. Multiple submissions are permissible and we will mark the last successful submission ***ONLY.***

Appendix: Pair Formation and Guidelines:

Over the years we have run this assignment, there have always been people who have either requested that they be allowed to do the assignment alone (the right way of going about it) or simply handed in an assignment that is solely their own work and hoped that we'd be ok with it (the wrong way of going about it). Please ask if there is a good reason why you can't make it work, but generally the answer will be no unless there are specific and very significant obstacles in the way.

I have in the past had a number of students who work full time contact me to express their concerns about working with other people, with some worried about having opportunities to pair up with other students. I have successful collaborations which involve code with people on the other side of the world, and this is increasingly common. The key of course is to establish the pair. Once you have had a couple of face to face meetings, it will usually work. Mostly working full time is not a sufficient barrier – I have had people working for mining companies in remote locations with lousy internet connectivity, and I have allowed them to work alone. Those in the city should normally be able to connect sufficiently, with occasional face to face meetings.

I don't like randomly allocating people to teams, even if this might happen in real workplaces. So we usually operate an Assignment 2 'dating' page on the BB site to allow people to advertise and then meet up with suitable candidates, and I have mentioned the need to work in pairs earlier in the semester. Here are some guidelines on pair formation.

How to Find a Partner:

Most people seem to find a partner through contact from previous classes or tutorials in the current unit. But if you cannot find one quickly, please use the forum established on BB at Assessment> Assignment 2 Dating Agency. I require that you email me by ***Thursday, May 12*** with the outcome of your searches. To allow me to filter this, please use the subject lines indicated below.

If you have found a partner: Please send me one email from one of you, cc'ed to the other, confirming that you will be a pair for the assignment. The subject line should be [CAB302 Confirmed Pair] and the mail body should contain your names and student numbers.

If you have NOT found a partner: Please advertise on the forum if you have not already done so, and then send me an email to say you are still looking. The subject line should be: [CAB302 Need Partner] and the mail body should contain your names and student number, and the message you sent to the forum so I can help match you up if needed.

If you believe your circumstances justify working alone on this assignment, please email me with a detailed request to this effect by ***TUESDAY MAY 10***. I will be pretty ruthless in assigning people to pairs by the end of that week, so please try to sort this out as soon as possible.