CO531 Software Engineering Group Project 2010-2011

Establishment Planning System

Written and checked by:
Timothy Smith
Gonzalo Varela
Jack Offredi

Document History

Issue	Date	Author	Notes / Changes
A(1)	28/12/2010	Tim	First draft
A(2)	31/12/2010	Tim	Change of ordering, started building up sections.
A(3)	02/01/11	Tim	Added to Introduction, Project Specification and User Documentation.
A(4)	05/01/11	Tim	Completed first draft of sections 5.2 and 6.
A(5)	12/01/11	Tim	First draft of all sections. Second draft of Introduction, Project Specification and User Documentation.
A(6)	17/01/11	Tim	Second draft of all sections save testing.
A(7)	17/01/11	Gonzalo	Completed section 3.
A(8)	17/01/11	Gonzalo	General Revision of document. Changes in format.
A(9)	18/01/11	Jack	Changes to the GUI section.

Contents

1. Introduction

An effective crew rostering system is an important part of airline operations. These systems generate working hours operating under a set of restrictions over a period of time, and analysis of these types of systems have spawned an entire area of research ("operations research"). Solutions to these problems are usually non-trivial, given the combinatorial explosion involved.

The aim of this Software Engineering project is to develop such a system (an "establishment planning" system) for an airline. This system should receive a set of flights to be assigned, and fairly assign all of these flights to the minimum number of captains possible, given a set of restrictions specified in Section 2.

This document will be organised in the following manner:

Section Two will contain a more detailed project specification, including notes of the simplifications used and their justifications.

Section Three will contain a detailed project design which meets these specifications.

Section Four will be a record of project testing.

Section Five, the Project Management section, is intended to give an idea of the project flow and system evolution.

Section Six, the VCRI, will show evidence of the completed project specifications.

Finally, Section Seven will consist of a brief user guide for the project.

The project team would like to thank our supervisor Olaf Chitil for his continued assistance

throughout the project.

2. Project Specification

2.1 Introduction

The main project deliverable is a system for "establishment planning" within an airline. Given a flight timetable indicating planes (types), flight times, sources and departures over a given period of a week, the system should extend this timetable to four weeks and roster airplane crew for this period. A complete, industrial system may take into consideration:

- training and vacation requirements of crew
- expected illness and turnover of staff, aiming to optimise
- the overall cost, in terms of crew salaries and other costs such as crew overnight stays and transfers

2.2 Simplifications/Assumptions

Due to the combinatorial explosion present in any attempts to obtain a perfect solution within a reasonable timetable, the project does not consider the above, and hence only takes into consideration the following:

- legal, safety, and other rules, e.g. concerning maximum working hours and staffing of planes;

- requirements for standby staff;

- crew happiness (fairness, scheduling of days off and holidays)

- flexibility (accommodating changes to the timetable, crew request days off).

This project therefore includes a number of simplifications:

"1. We only consider captains, not entire crews. Captains are the hardest to roster anyway, as each

can fly only one type of plane.

2. The associations between captains and flights are called "pairings". We expect to use only simple pairings: captains go from A to B and then back. This may still incur a stopover if working both legs of the flight would lead to illegal working hours, but no transfers.

3. We do not explicitly consider sickness, vacation, training and turnover requirements separately - we just cover all of those jointly with standbys and extra captains.

4. We do not use the full set of legal etc rules, but a simplified version. This refers to the full rules document only for the definition of terms.

5. We only consider pairings starting and ending in (and therefore only captains based in) London Gatwick (LGW)." (https://moodle.kent.ac.uk/moodle/file.php/12707/Project/projv2.htm accessed 2/11/2011)

Further simplifications and assumptions:

- All flights entered into the system are flown by one type of plane, which every pilot can fly.
- In our project, we have treated standby flights the same as normal flights, albeit with modified beginning and ending times.
- We assume that the input .csv file is ordered by date and time, beginning with the earliest flight and ending with the most recent.
- All standby duties are assigned for the same length of time, regardless of the destination of the original flight. This means that it is assumed, for example, that a captain can travel from London Gatwick to Faro, Portugal in three hours. This may or may not be realistic depending on the destination.

The program should take the flight timetable and assemble legal pairings, then compute an anonymous roster detailing the minimum number of captains needed to operate the agreed pairings. In other words, the most important output is a *single* number for every departure airport (of which there is only 1 due to requirement 5 above): "We need 17 pilots in Gatwick". Of secondary importance is to make the roster available for on-screen browsing. This should be done in a way

which enables the following additions to the program in subsequent phases:

- on-screen modification of the roster

- check that a modified roster still satisfies legal (etc) constraints;
- given salaries, hotel costs, etc, compute total cost of roster.

2.3 Project Requirements

In summary, the functionality of the project is as follows:

Core functionality
Take a file to load, extract flights and legally assign these flights to the fewest amount of captains.
Display the number of captains required in London Gatwick.

Legal requirements for captain assignation: for each assignation of a flight or pair to a captain, the program needs to check that said captain:
Is not flying over the maximum allowed flight time
Is not flying more than three consecutive early or late duties
Is not doing more than four early or late duties in a seven day period
Has a holiday before flying two early duties
Does not already have a pair/flight assigned which clashes with the attempted

assignation
Has eight days off over a 28 day period
Is not flying more than 55 hours in 7 days
Has the minimum of either 12 hours or the length of the last duty off between duties
Is not flying more than 95 hours in 14 days
Has at least one day off in each 8-day period
Is not flying more than 190 hours in 28 days
Has at least 2 days off in a 14 day period
Has a reasonably fair timetable.

Secondary functionality
Allow on-screen modification and viewing of the roster, with these modifications satisfying legal requirements.

3. Design documentation

3.1 Program Design

As it was said in previous chapters, the proposed problem had a high complexity. Despite the fact that many simplifications were used, the flight assignation problem was hard to solve. In order to produce a solution for it, we saw the general program as two different issues. Firstly, we had to "create" captains and find some way to distribute the total amount of flights amongst them. The algorithm in charge for this distribution will be discussed in section 3.3, but briefly explained: it creates captains and tries to assign flights to them in some special manner. Here is where we saw the second separate issue.

The assignation problem as a whole, should create captains and check which captain could fly each of the different flights. However, we thought that an algorithm in charge of doing all of this was too complex. Instead, the main algorithm would create captains and try to assign the different flights to each of them. It is each captain that would then decide for himself if he can fly the proposed flight or not. By doing so, we separated the general problem in two: on one hand we had to solve how to distribute the flights, one the other hand, we had to create ways to check the availability of each captain (part which is totally independent from the first one). This solution allowed us to speed up development by splitting up the coding. Furthermore, as we will see afterwards, this solution has a high degree of maintainability due to the fact that a new rule can be implemented with low impact on the existing code.

If we look into the proposed design, the first thing that we note from the previous description is the existence of two main different classes: "Captain" and "Central". The first one is in charge of saving each captain's data, each captain's assigned flights and for checking if he can take an extra flight or pair (of flights). The second class has the "main algorithm" in charge of creating these captains and asking each of them if they

can fly the different flights. It does so in a special way, but basically it tries to assign a flight to a captain: if the captain can take it, he gets that flight and the algorithm will try to assign him another one (until he can fly no other flight); if he can't, the algorithm will try to assign him another flight until there are no more flights left (in that case a new captain is created).
In order to represent each flight, the "Flight" class was created. It saves all the data describing the different flights: departure or arrival GMT times, local times, airports, etc. Apart from this, each captain is assigned a pair of flights: out and back from/to Gatwick. For this, another class was created: the "Pair" class which holds a flight out, a flight back, the total flight time, etc.
Each flight departs from and arrives to airports, and each airport has a different local time. In order to represent this in the software, the "Airport" class was created. This contains the local time offset, the airport code, etc.
Finally, in order to separate some of the logic, all the code implemented to read the text file and create flights, was implemented on a different class called "Data". This last class together with the "GUI" class and the "Main" class, forms the total amount of classes our program.

3.2 Class Specification

In the diagram above, we can see all the relationships between the different classes. Each flight has a Pair, airports and captains. Data has a list of flights and list of captains. A pair has a captain and a flight. Central has a list of captains, a list of flights, a list of pairs and a "Data" object. In order to specify furthermore the role of each class, a brief summary is done on each one.

Captain

- Is an abstraction of an aeroplane captain
- Keeps all of his data.
- Has flights and pairs assigned to him.
- Checks his own timetable, including flying duty period. Controls that the flights assigned to him respect all of the rules.
- Can add and remove pairings.

Flight

- Is one half of a pair – a flight from location 'A' to 'B'
- Saves details of start time/date, finish time/date, destination and departure.
- Keeps track of its standby captain and the pair of flights in which it belongs.
- Has a method to compare itself to other flights, to check for clashes.
- Converts times to local times (using Airport class) and checks if it is an early/late flight.
- In order to assign standby captains, an 'artificial' flight is created for each pair of flight. This 'artificial' flight is a Flight object which has certain values in its attributes:
    - Has the arrival airport equal to the departure airport and equal to the first flight's departure airport.
    - Starts 3 hours before the departure of the first flight.
    - Finishes 3 hours after the departure of the first flight.

Airport

- Flights are to and from airports
- Has a local GMT offset, to assist in checking for early/late flights.
- The offset is hard coded. Each possible airport code gets mapped into a specific time offset. There is a fixed amount of possible existing airports.
- It allows successful allocation of flights.

Data

- Reads the .csv file.
- Creates Flight objects from the text file read.
- Creates Airport objects for each flight.
- Saves all created flights.

Pair

- Is made of two flights: a flight out and a flight back.
- It is the object that is assigned to each captain.
- It contains general info such as the captain, the standby captain, the duty start, the duty end, etc.
- It is an abstraction of a sector

Central

- Contains a list of Captains, each of which has their own timetable.
- Has the data class, where the text file is read and the flights are kept.
- Contains the main algorithm where captains are created and flights are assigned to them.
- Creates “standby flights” (concept that will be explained later, it is used for assigning standby captains). Assigns those standby flights to captains.

3.3 Code Flow

The program basically consists in one big operation. Regardless of the fact that changes can be made to the proposed timetable and that other operations are available, the main operation starts when the user selects the file to read and finishes when the total amount of captains needed is outputted. As a graphical application, everything flows from the GUI into the program. The following diagram illustrates most of the flow in the code:

The process described above is as follows:

1. Initialization – When the program starts, the GUI is created and showed. Besides this, the "Central" and "Data" classes are created, waiting for the user to select the file to open.
2. Once the user selects the file, the path of the file to read is sent to the "start" method, which drives the whole process from there onwards.
3. First, Data class is called and the file is read. Every 2 lines in the file, a Flight object and Airport objects are created.
    a. The airport object is created using its code. The system has all the possible airports hard coded, each with its out time offset. Therefore when the "Airport" constructor receives the code, an Airport object is directly constructed with its correct GMT offset.
    b. The Flight object is created using the data in the file. However there are some values that are calculated.
        i. Duty start and end are calculated subtracting 1 hour and adding half an hour respectively to the flights' departure and arrival times. T
        ii. he local times are calculated using the departure and arrival airports' GMT offset.
        iii. The flags indicating if it is a late or early flight are used if the flight starts/ends before/after certain times.

iv. The flight or duty durations are calculated using the arrival/departure or the duty start/end times.

4. The inputted timetable is for 1 week and it has to be used for 4 weeks. Therefore, each flight read has to be copied 3 times, each one of them one week after the other. This is done in the following way:
    a. After each flight is created, that flight is asked to clone itself a number of weeks later.
    b. A method "cloneLater" is run in Flight class which creates a new flight with exactly the same data as the original one. The difference lies in the flight departure and arrival, which is moved 'x' amount of weeks later. That flight is returned.
    c. This procedure is repeated 3 times for each flight in the .csv file. This means that for each flight in the file, 4 Flight objects are created. All of them are added to the list of flights in Data class.
    d. The procedure is repeated until the entire file is read.
5. The list of flights read (and the duplicates created), are returned to the "start" method. From these flights, pairs have to be created. Therefore the pairing algorithm is called (a method in Central class).
6. Before the pairing algorithm is actually called, "standby flights" are created. As it was stated before, these standby flights are used to assign standby pilots for each flight. Therefore a captain assigned to a standby flight is indeed the standby pilot of the original flight.
7. After that, the pairing algorithm forms pairs of flights. It creates "Pair" objects, with all the outgoing flights departing from London. Each pair has as a first flight, one going from Gatwick to airport 'X', and as a second flight one going from airport 'X' to Gatwick.
8. Once the pairs are made, these have to be assigned to captains, therefore the so called "main algorithm" starts. It consists in two parts: one for assigning captains and one for assigning standby captains. This will be explained later.
9. After the main algorithm finished assigning captains and standby captains, the program has the total amount of captains needed; each one of them knows his own timetable. The result is outputted in the GUI.

3.4 Algorithm Design

3.4.1 Data Class Algorithms

1. ***Algorithm One: Extracting information from .csv file and converting to class objects***
    1. For each record in the .csv file, append record to a String, including line breaks when necessary.
    2. Split the String around the line breaks. This creates an array of Strings, where each entry is a flight record.
    3. For each flight record, create a Flight object out of the details.
    4. For each Flight object, create three extra Flights, each a week later than the previous. This has the effect of extending the timetable across four weeks.

3.4.2 Central Algorithms

2. ***3.4.2.1 Captain Assignation***
    1. Obtain set of flights from Data class
    2. Order this set of flights by departure airport ***(?)***
    3. Create a set of standby flights from the flight data. Each pairing has one standby flight, beginning three hours before the pairing start, and ending three hours after the pairing start. This will be used when assigning standby captains: the set of flights from the Data class will be empty when the standby captain assignation begins, due to the nature of the algorithm.
    4. Pair up non-standby flights into Pair objects representing a flight out and back (section 3.4.3).
    5. Continue to main pairing assignation algorithm:
        - While there are still pairings to be assigned:

- Create a new captain
- Try and add largest flight to new captain. To be added, the flight pairing would have to satisfy all of the checks within the Captain class (Captain.doAllChecks()). Assignation of a pairing removes it from the set.
- Try to add smallest flight to captain. Again, to be added, the flight would have to satisfy all of the checks within the Captain class (Captain.doAllChecks()).
- This method will eventually result in one pairing remaining in the set. When this happens, the above method will not find the pairing. This last section finds the last remaining pairing and attempts to assign it to the current captain.
- Re-sort the remaining pairings by size.

This algorithm was our attempt at fairness. Assigned flights to captains linearly, such that the biggest flight was assigned to the first captain, then the next biggest to the same captain and so on, would have resulted in an excessively unfair system. The first captain would have all the long flights, and the last captain would have all the short flights. Our method gives on average the same total length of flights to each captain: the first captain has flights which are either large or small (coming from the beginning and end of the sorted flight list), and the last captain has flights which are of a medium length (coming from around the middle of the flight list).
Another method which we could have used is to randomise the flight order, and then assign the flights linearly. However, because this has the unavoidable element of randomness, we decided to make absolutely certain the flight allocation was as fair as we could make it.

1. Continue to standby pairing assignation algorithm. Standby flights are treated as being the same as normal flights.
   - For each captain currently existing:
     - Try and assign the largest flight to said captain.
     - Try and assign the smallest flight to said captain.
     - Again, this will result in one flight remaining. Attempt to assign the last remaining

flight to the captain.

When this algorithm completes, it is extremely unlikely that all the flights now have a standby captain. Because the algorithm removes flights from the set when they are assigned, if the set size is not zero, then some flights do not have a standby captain. In this case, a new captain is created and standby flights are assigned to him. If the captain cannot take all of the remaining flights, another new captain is created. This process continues until all of the flights are allocated.

Because we viewed the initial flight allocation as fair, we decided to extend the same method to the assignation of standby duties. However, we overlooked the fact that, in the standby algorithm, it may (and in most cases, must) happen that not all flights can be successfully allocated a standby captain from the previously existing captain. In this case, as noted above, a new captain is created. However, because all of the initial duties have been allocated, these captains will only have standby duties. If all the captains can fly all duties allocated to them, the captains only rostered for standby flights will not be flying any duties. Thus an element of unfairness has been introduced by using the same algorithm twice.

3. ***3.4.2.2 Pairing algorithm***

A Pair is a collection of two linked flights, from LGW to the destination and back. This algorithm pairs together the correct flights so that they can be assigned. The function is called with a LinkedList containing all the single flights obtained from the input file.

- Sort flights by departing airport to obtain a collection *col* of all flights leaving from LGW (section 3.4.2.3).
- Create a set of standby flights from this sorted list. This is a set of flights containing flights with the same name as in the original sorted list, but with their departure and arrival times modified to account for their standby status (section 3.4.2.4).
- For each flight *fl* in the collection:
  - check "orderFlights" method for errors. If errors are found, it means that there is no

flight leaving from the destination airport of the current flight, and that this flight cannot be paired up. This is however unlikely.

- If there are no errors:
  - Attempt to pair *fl* with another flight. For each flight *fl2* that could possibly be the correct flight (obtained by interrogating *col*) check if *fl2* is after *fl*, and whether the first six letters of the flight numbers match.
- If the pairing is successful , add the pairing to a collection. This collection will be returned when all the flights are successfully paired.

4. ***3.4.2.3 Flight ordering algorithm orderFlights()***

This algorithm returns a collection of array lists. Each of these array lists is a collection of flights which are departing from a certain airport, notated by a string associated with the collection.

For each flight in the unsorted list of flights:
Obtain the code of the departure airport
Obtain all flights leaving from that airport
Add a new array list to the collection and associate the array list with the code of the departure airport
Return the sorted flights

5. ***3.4.2.4 Algorithm creating Standby Flights createStandbyFlights(Arraylist<Flight> flightsFromLondon)***

Standby captains are incorporated into the program by the creation of a set of standby flights. These standby flights are the same as the original flights, except that the start and end times are modified to account for the different duty times of a flight and its standby.

For each flight in the original list:

Obtain the Gregorian Calendar *gregCal* used to store the beginning of the flight

Create a modified copy of *gregCal* , which begins three hours before *gregCal* and ends three hours after the beginning of *gregCal*.

Create a standby flight with this modified *gregCal.* This flight is equal to the original flight in all respects except the time of the beginning of the flight.

3.4.3 Captain Algorithms

6. ***captain.doAllChecks()***

This is a collection of unrelated checks that a "Captain" performs on his flights. For each assignation of a flight or pair to a captain, the program needs to check that said captain:

- is not flying over the maximum allowed flight time (belowMaxHours())
- is not doing more than three consecutive early or late duties (checkConsecutiveEarlyLates())
- is not doing more than four early or late duties in a seven day period (checkEarlyLateDuties())
- has a holiday before flying two early duties (checkHolidayBeforeEarlies())
- does not already have a pair/flight assigned which clashes with the attempted assignation (doesNotClash())
- has eight days off over a 28 day period (eightIn28())
- is not flying more than 55 hours in 7 days (fiftyFiveInSeven())
- has the minimum of either 12 hours or the length of the last duty off between duties (hasTimeBetween())
- is not flying more than 95 hours in 14 days (ninetyFiveInFourteen())

- has at least one day off in each 8-day period (oneInEight())
- is not flying more than 190 hours in 28 days (oneNinetyInTwentyEight())
- has at least 2 days off in a 14 day period (twoInFourteen())

7. *Function: belowMaxHours()*

Description: A captain is limited in the amount of hours he can legally fly over a period of twenty-eight days. This check makes sure he isn't flying more than he is allowed to.

"Absolute limit on Flying Hours for pilots is 100 hrs in 28 consecutive days, they cannot take off after 100 hrs."
(*Definitions and Rules for Establishment Planning Project,* CO531 Moodle page, accessed 14/01/2011)

Implementation Design Pseudocode:

Create a set of arraylists of flights, with each arraylist intended to hold the captains flights over a twenty eight day period

Until all flights have been assigned to an arraylist:

> Create an ArrayList to hold flights for this twenty eight day period..
>
> Create a GregorianCalendar object *weekLater* with the date set two weeks after the current flight, to ensure flights are only checked over two weeks duration.
>
> For each flight, if the flight ends before *weekLater,* add it to an ArrayList.

Add the current ArrayList to the set.

For each ArrayList, check that the total time spent flying is not over 100 hours.

8. ***Function: checkConsecutiveEarlyLates()***

Description: Early and late flights affects a captains ability to fly, if only by a small amount. A captain should not be allowed to fly too many of such flights consecutively. Hence, this functions checks that the captain is not flying more than three consecutive early or late flights.

Implementation Design Pesudocode: Each flight has a boolean “flag” that is set true if the flight is either an early or a late flight. These flags are used in the code below:

Take the first four flights in a captains flight table.

If all four flights are early flights, return false (that is, the captain cannot take this flight)

If all four flights are late flights, return false

Else, return true

9. ***Function: checkEarlyLateDuties()***

Description: Early and late flights can affect a captain's ability to fly even if they are not flown consecutively. This function checks that the captain is not flying more than four early or late duties in a seven day period.

Implementation Design Pseudocode:

For each flight that is assigned to the captain:

> Create two arraylists, one for early duties and one for late duties.
>
> Create another GregorianCalendar object with the date set a week after the flight, to ensure flights are only checked over a weeks duration.
>
> Place the flight duty in the appropriate array list.

Start another loop iterating through each flight assigned to the captain:

As long as the flight is not the flight being checked, and is within a week after the flight being checked, check if the flight is an early or late duty, and place it in the appropriate array list.

If either array list contains more than four flight duties, return false

If the first loop has completed, then a false result has not been returned. Return true.

10. ***Function: checkHolidayBeforeEarlies()***

Description: The flight rules stipulate that a captain should have a holiday before flying two consecutive early duties. This function enforces that.

Implementation Design Pseudocode:

For each flight assigned to a captain:

Obtain current flight and next flight

Check if both are early flights

If both are early, check if they have a day off assigned between them: if a day off has already been assigned (if there is a day between the end of the first and the beginning of the second duty) then there is no need for a day off before the first duty. Otherwise, obtain the flight before the two consecutive early duties, and check if there is a day off between the end of that duty and the beginning of the first early duty. If there is not, return false.

If all flights have been checked and no result has been returned, then there have been no problems with the current timetable, and the function returns true.

11. ***Function: doesNotClash()***

Description: A basic part of successful flight allocation is to ensure that a pilot is not scheduled to

fly two conflicting flights. This function ensures this.

Implementation Design Pseudocode:

For each pair assigned to a captain:

Obtain the next pair in the list

Check that the first pair does not clash with the second (pair1.isNotDuring(pair2)). If it does, return false

For each flight f1 assigned to a captain:

If f1 is not part of a pair (i.e. is a standby flight):

f2 : the flight after f1

If f2 is not after f1, or f1 is not before f2, then there is a clash: return false

If both checks have been run without returning false, there are no clashes: return true.

12. ***Function: eightIn28()***

Description: Another stipulation in the flight rules regarding pilot holidays is that a pilot has to have at least eight days holiday in any twenty-eight day period, the checking of which is the purpose of this function.

Implementation Design Pseudocode Unfinished:

For every flight assigned to a captain, increment an integer *i*:

Obtain the first flight, and calculate the date twenty-eight days later. This is used to ensure that the flight checks are only done over a twenty-eight day period.

From point *i* within the flight list until the end of the flight list:

Obtain the flights at points *i* and *i+1*

If the second flight leaves before the end of the twenty eight day period

Check how many days off there are between the two flights

13. ***Function: fiftyFiveInSeven()***

Description: A captain can fly for a maximum of 55 hours over a period of 7 days: this is implemented here.

Implementation Design Pseudocode:

For every flight *fl* in the captains flight list:

Calculate the date a week after the flight start, to ensure the checks are done over a week

Obtain the duration of the flight

For every flight *fl2* in the captains flight list

If *fl2* is not *fl*:

Check if *fl2* is in the same week as *fl* (due to the input .csv file being multiplied by four for the different weeks, a check is necessary to ensure that a flight on Sunday ,Week 1 is not compared to a flight on Monday, Week 2).

If so, obtain the duration of the flight and add to a counter variable.

Transform the counter variable to hours

Check if the counter is over 55 and return false if not.

Otherwise, if all flights have been checked without returning false, return true.

14. *Function: hasTimeBetween()*

Description: A captain cannot walk off one plane and immediately start flying another. All captains must have a rest time between flights, minimum of 12 hours or the length of the last duty.

Implementation Design Pseudocode:

Initialise a pair for use in the algorithm.

Because all of these checks are performed every time a flight or pairing is assigned to a captain, this check is performed from the first captain assignation, which is not needed. Hence, if the size of the captain's pair list is over 1:

Obtain the next pair

Obtain the time period between the end of the previous pair and the start of the current

Convert this time period into hours

Return false if the time period is over 12 hours or the length of the last duty

The checks for the standby flights are separate from the checks for the main pairings. Each flight has a flag which indicates whether it is part of a pair. Standby flights are modelled as individual flights, and have these flags set to false. For each such standby flight:

Obtain the next flight

Check if the flight is part of a pair: if so, skip to the next flight

Obtain the time period between the end of the previous flight and the start of the current

Convert the time period into hours

Return false if the time period is over 12 hours or the length of the last duty

If the algorithm has not returned a false boolean result, the flight table is sound, and returns true.

15. *Function: ninetyFiveInFourteen()*

Description: An extension of 55in7, this checks that a captain is not flying for more than 95 hours in 14 days.

Implementation Design Pseudocode:

For every flight *fl* in the captains flight list:

Calculate the date two weeks after the flight start, to ensure the checks are done over two weeks

Obtain the duration of the flight

For every flight *fl2* in the captains flight list

If *fl2* is not *fl*:

Check if *fl2* is during the two-week period after *fl1*.

If so, obtain the duration of the flight and add to a counter variable.

Transform the counter variable to hours

Check if the counter is over 95 and return false if not.

Otherwise, if all flights have been checked without returning false, return true.

16. *Function: oneInEight()*

Description: Holiday allocation is very important in a system of this type. Within this system, there a number of checks to make sure that pilots are allowed a certain amount of holiday leave. This checks that a pilot has at least one full day off over every eight days of flying.

Implementation Design Pseudocode:

For each flight in the captains flight list:

Obtain the flight

Calculate the date eight days later, to ensure the checks are restricted over an eight day period

17. ***Function: oneNinetyInTwentyEight()***

Description: Another extension of 55in7, this check ensures that a pilot is not assigned more than 190 hours over 28 days.

Implementation Design Pseudocode:

Obtain the first flight in the captains list, and add twenty eight days to it. This new calendar *c* is used to limit the scope of the check.

Iterate through each flight in the captain's pair list, and create a new array list containing all pairs which end before the twenty eight day limit (before *c*).

Iterate through this array list and and retrieve the total length of duty allocated. Transform this number into hours.

Return whether this number is over 190.

18. ***Function:twoInFourteen()***

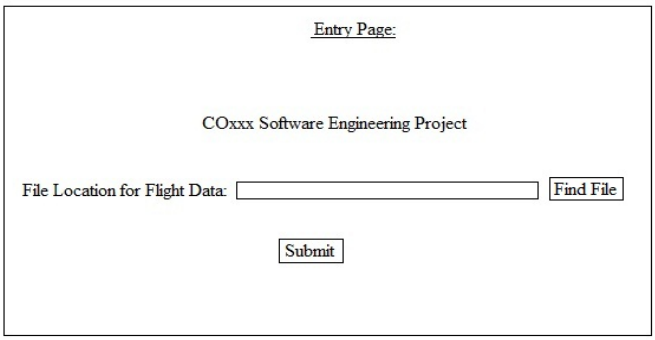
Description: An extension of oneInEight, this checks that a pilot has at least two full days off over

every fourteen days flight.

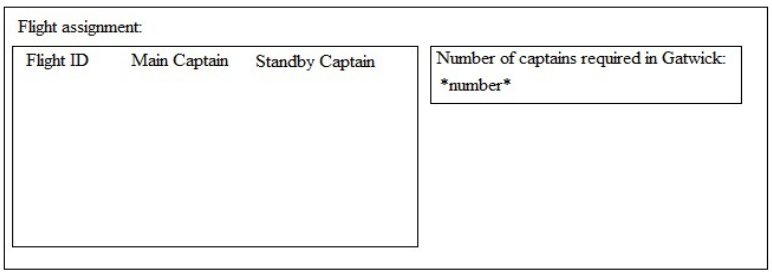
Implementation Design Pseudocode:

3.5 GUI Design

The initial GUI design is shown below.



which takes the user to this page:



This was then modified when we realised that this would create a window with possibly hundreds of entries. It made more sense to group flights by captain, and we also decided to implement reallocation of flights.

Therefore, we needed a GUI that fulfilled the following criteria:

- Begin with an entry point into the project. Hardcoding the file location into the program would not be user-friendly; it would require that the file have a certain name and be in a certain location. To reload the program with a different file would require the old file be replaced with the new file. Hence, the program should have some form of file chooser.
- When the file is loaded and the flight allocation process running, there should be some form of feedback to the user.

- When the flight allocation process has finished, the allocated flights should be grouped by captain.
- It should be possible to reallocate flights once assigned.
- It should be possible to reload the program once completed.
- Flight data should be presented clearly.

The revised GUI design was hence:

Above, the “Open File” button would load a file chooser window, which would allow the user to choose their own file. Reloading the project would be done by File->Open;once a valid file is chosen, the program re-executes.

3.6 Changes to design during project

The initial project design assumed that references to the standby captains would be stored in the flight itself. The name of the standby captain could presumably be accessed by filling the flight window with buttons for each captain as opposed to labels. These buttons could then be clicked to find the name of the standby captain.

The final project split *pairings* from *flights*. A pairing in the final project corresponds with a flight in the initial project design: a flight from LGW to the destination and back. In the final project, flights include both the two flights in each pair, as well as any flights the captain is standby for (indicated by a “-Standby” at the end of the flight name).

The final GUI also included a summary of the total time a pilot spends flying a) their active duties (the duration of all pairs allocated) and b) both their active and standby duties (essentially the duration of all flights assigned to the captain).

3.6.1 Ideal changes to project design and final project deliverable

19. #### 3.6.1.1 GUI

With the project as it currently stands, the GUI is closely coupled to the main flight allocation

algorithm – every time a change was made to the GUI, the central algorithms had to be rerun before an updated GUI was displayed. To make future development of the project easier, it would have been beneficial to separate the GUI from the central algorithms. A possible method of doing this would be to have the GUI class take an object of a class which contained the central algorithms, making both classes subordinate to a superclass (providing access to the project) which then calls the GUI once the central algorithm class has run to completion.

Currently, it is not immediately obvious what the structure of the full roster for a captain looks like. Both the pairs and flights frame only immediately display dates rather than times, so it is not easy to build up a picture of when a captain has time off. A calendar-style captain roster would be much more user-friendly.

Finally, the parts of the program dealing with captain allocation have minimal user feedback. A message box appears:

a) Before captain-to-flight allocation begins

b) When the initial set of flights have been allocated (when all flights have been assigned to captains).

c)When the standby set of flights have been allocated.

The time elapsed between steps (a,b) and (b,c) tends to be lengthy. Because the current implementation does not provide any feedback as to what stage the program is in, or how long it is expected until the program completes, it is easy to assume that the project has somehow failed, crashed, or is otherwise nonoperative. Ideally, a progress bar would be displayed and a threading mechanism implemented to provide constant feedback to the user.

20. **3.6.1.2 Algorithms**

The current flight allocation system is not entirely fair. The initial flight allocation algorithm is fair (see section 3.4.2), but the standby flight allocation algorithm is not – it creates new captains that are only assigned standby flights. If all main captains can take all their flights over the four-week

period, any company using this crew rostering system is effectively paying a proportion of their staff to do nothing.

21. **3.6.1.3 Adding locations through the GUI**

The current project has a set of destination airports, taken from the example .csv file, hard-coded into the Airport class. There is no way to add or modify these locations from the GUI in the current design. This appears to be an ideal possible extension to the project.

4. Test Documentation

Test: Adding labels to pair frames detailing total length of all pairs
Expected result: Credible positive integer between 10 and 99.
Actual result: Occasional negative integer.
Action taken: Project checked, problem found in initial flight creation. Problem fixed.

4.1 Unit testing – functions within modules

Testing individual functions

4.2 Unit testing – modules within project

Testing individual modules (classes)

4.3 System testing

Testing the entire system from the GUI

5.

Project Management

5.1 Project Organisation

The project team consisted of Gonzalo Varela-Leizagoyen, Nick Wridgeway, Timothy Smith and Jack Offredi. Each member took part in coding the project, with Gonzalo as nominal project leader.

In more detail, Gonzalo was mainly responsible for:

Timothy Smith was mainly responsible for: the main algorithm assigning flight and standby duties to captains; the GUI.

Nick Wridgeway was mainly responsible for:

Jack Offredi was mainly responsible for: GUI design and implementation. GUI usability testing.

Selection of a software development process was based on the way our team actually worked. We decided to develop the project using the incremental delivery software development process, which allowed our team to split up user requirements and assign responsibility of each requirement to a team member. This method allows individual members to work relatively autonomously; there were only a small number of cases in the early stages of the project where project development by one member was halted pending development (of, for example, a certain function) by another member.

5.2 Meeting Documents

5.3 Version Control

Version control was carried out by using a combination of three tools: project space on CSProjects, a centralised source code management service based on Subversion and Trac for projects related to the University of Kent, Subversion (and the client TortoiseSVN) and the version control built into

BlueJ.

5.3.1 CSProjects Timeline

1. ***5.3.1.1 Key Dates***

31/10/10: Initial project shared. This contained a basic set of shared classes that we could develop.

3/11/10: Code added to read in the .csv file.

14/11/10: Initial version of central algorithm to assign flights to captains added

10/12/10: Initial version of central algorithm to assign standby captains added

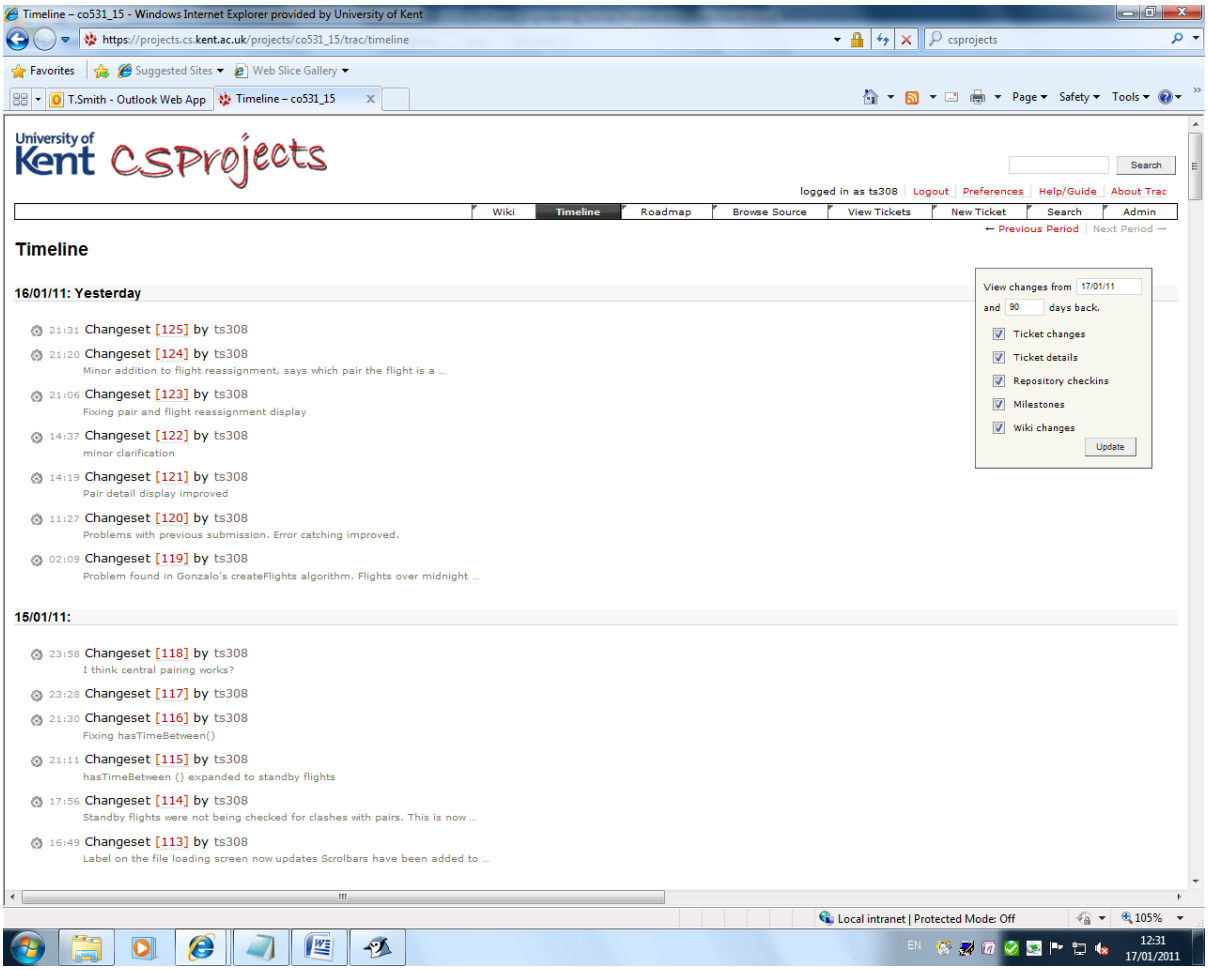
17/12/10: Final version of central algorithm added

4/1/10: Initial version of project with complete functionality added

16/1/10: Final version of project added

2. ***5.3.1.2 Proof Of Use***

CSProjects was used to examine changes to the project. It was especially useful on occasions when faulty versions were uploaded to the project space: the changes could be examined and fixed or removed as required. If this resulted in a project which could not be fixed easily , the local projects could be updated to an earlier version ("rolled back").


Timeline – co531_15 - Windows Internet Explorer provided by University of Kent
https://projects.cs.kent.ac.uk/projects/co531_15/trac/timeline
csprojects
Favorites
Suggested Sites
Web Slice Gallery
T.Smith - Outlook Web App
Timeline – co531_15
Page
Safety
Tools
University of Kent CSProjects
Search
logged in as ts308 | Logout | Preferences | Help/Guide | About Trac
Wiki
Timeline
Roadmap
Browse Source
View Tickets
New Ticket
Search
Admin
← Previous Period | Next Period →
Timeline
16/01/11: Yesterday
21:31 Changeset [125] by ts308
21:20 Changeset [124] by ts308
Minor addition to flight reassignment, says which pair the flight is a ...
21:06 Changeset [123] by ts308
Fixing pair and flight reassignment display
14:37 Changeset [122] by ts308
minor clarification
14:19 Changeset [121] by ts308
Pair detail display improved
11:27 Changeset [120] by ts308
Problems with previous submission. Error catching improved.
02:09 Changeset [119] by ts308
Problem found in Gonzalo's createFlights algorithm. Flights over midnight ...
15/01/11:
23:58 Changeset [118] by ts308
I think central pairing works?
23:28 Changeset [117] by ts308
21:30 Changeset [116] by ts308
Fixing hasTimeBetween()
21:11 Changeset [115] by ts308
hasTimeBetween () expanded to standby flights
17:56 Changeset [114] by ts308
Standby flights were not being checked for clashes with pairs. This is now ...
16:49 Changeset [113] by ts308
Label on the file loading screen now updates Scrolbars have been added to ...
View changes from 17/01/11
and 90 days back.
Ticket changes
Ticket details
Repository checkins
Milestones
Wiki changes
Update
Local intranet | Protected Mode: Off
105%
12:31
17/01/2011

5.3.2 Subversion

Subversion was used to enable us to roll back to a previous version if any committed changes broke the project. An example is shown below. SVN Update allowed us to obtain code from any of the versions placed in the CSProjects directory.

6. Verification Cross-Reference Index

6.1 Core functionality

Functionality	Page documented/illustrated

Take a file to load, extract flights and legally assign these flights to the fewest amount of captains.	
Check that these assignations are fair	
Display the number of captains required in London Gatwick.	

6.2 Legal requirements

Legal requirements for captain assignation: for each assignation of a flight or pair to a captain, the program needs to check that said captain:	Page documented/illustrated
Is not flying over the maximum allowed flight time	
Is not flying more than three consecutive early or late duties	
Is not doing more than four early or late duties in a seven day period	
Has a holiday before flying two early duties	

Does not already have a pair/flight assigned which clashes with the attempted assignation	
Has eight days off over a 28 day period	
Is not flying more than 55 hours in 7 days	
Has the minimum of either 12 hours or the length of the last duty off between duties	
Is not flying more than 95 hours in 14 days	
Has at least one day off in each 8-day period	
Is not flying more than 190 hours in 28 days	
Has at least 2 days off in a 14 day period	
Has a reasonably fair timetable	

6.3 Secondary functionality

Functionality	Page documented/illustrated
Allow on-screen modification and viewing of the roster, with these modifications satisfying legal requirements.	

7. User Documentation

Overview

This system is reasonably simple to use. A .csv file containing flight details is presented to the system, which extracts the necessary data. These flights are then assigned to the fewest amount of captains possible, with each flight having a single standby captain. The results are then displayed in a GUI which allows flight checking and reallocation. At any point after captain allocation, brief instructions as to the use of the project can be found under Help->Instructions.

Quick Start

The project will be submitted as a .jar file. Double-click on the project file to load the GUI. This should present a screen similar to the one below:

Click on the “open” button, and choose a file. The program will then execute using that file.

Rostering with the GUI

Once the program has finished executing, it should present a screen similar to the image below:

The anonymous captains are assigned numbers for identification. Each captain has a *pair list* and a *flight list*, which contain pairs and flights respectively. Pairs are linkings of two flights, one from Gatwick to the destination, and another from the destination back. Both flights within these pairings are also entered onto the flight list, which also contains any flights that the captain is standby for

(denoted by a *Standby* suffix).

Below: an example pair list and flight list for Captain 21. Note that most of the flights have the -Standby suffix – the four non-Standby flights correspond to the two pairs this Captain has been allocated. The "-1", "2" and "-3" show that those flights are respectively one, two and three weeks after the original flight.

Reallocating flights

Reallocation is the same for flights and pairs. To reassign, select the "Reassign pairing/flight" option next to the pairing or flight that needs to be reallocated. This will present a bar containing a set of buttons corresponding to the captains currently existing in the system. Select a captain to attempt reallocation; if the reallocation fails, you will be notified via a message box.

Viewing pair/flight details

In the above screen shots, it can be seen that a button has been created which has the name of the flight/pair as a label. If selected, these buttons open a new window containing the details of the flight or pairing, including the time and date of start and end.

Reassigning pairs/flights

Essentially the same method is used to reassign both pairs and flights. With both, reassignation is performed by opening the captain's pair/flight list and selecting the “Reassign flight/pair” button next to the relevant duty. Flights that are part of a pair cannot be reassigned individually: they have be reassigned by finding the pair which it is part of and reassigning that pair. Reassigning flights is only possible for reassigning standby duties: active duties must be reassigned from the Pair menu.

Removing pairs/flights

There may be occasions where flights are cancelled and pilots are no longer required to fly them. In this case, they may be removed from the captain's roster by selecting the “Remove pair” or “Remove flight” button from the appropriate menu.

Dealing with errors

An error may occur if the file to be loaded is corrupt or unreadable. In this instance, the program should catch this error and display a message:

The “technical details” section is intended to provide more details if the program crashes, which may help determine the cause of the problem. If the file to be loaded is corrupt, unreadable or in the wrong format, “null” will be displayed here, as above.