Analysis, Design & Testing

SEG2012GP9

|  |  |
| --- | --- |
| Seabrook, Edward J F | [ejfs1g10@ecs.soton.ac.uk](mailto:ejfs1g10@ecs.soton.ac.uk) |
| Elliott, Kristian | [ke1g10@ ecs.soton.ac.uk](mailto:ke1g10@soton.ac.uk) |
| Yu, Brian | [by2g10@ ecs.soton.ac.uk](mailto:by2g10@soton.ac.uk) |
| Mariani Elola, Oscar N | [onme1g10@ ecs.soton.ac.uk](mailto:onme1g10@soton.ac.uk) |
| Chan, Yun Chak | [ycc1g11@ ecs.soton.ac.uk](mailto:ycc1g11@soton.ac.uk) |

# Notes to the Authors:

* Max 12 sides A4 in total for this report, excluding "wrapper" pages (e.g., title page, contents page, document control page) and appendices
* This document shall be written in first person plural, future tense. i.e. “We will destroy the Earth”, “Our project shall really suck” etc.
* ^^^ Look at all that Latin ;) ^^^ We will be Sasan Malekis’.

# Document Control

|  |  |  |
| --- | --- | --- |
| **Version** | **Author** | **Changes** |
| 1 | ejfs1g10 | Initial document |
| 2 | onme1g10 | Added conceptual model of HCI |
| 3 | ke1g10 | Added the approach taking |
| 3 | ejfs1g10 | Various stylistic alterations |

# Table of Contents

Notes to the Authors: 1

Document Control 2

Table of Contents 3

1 Approach Taken 4

1.1Technical 4

1.2 Modelling 5

2 Analysis Documentation 6

2.1 Requirements Analysis 6

2.2 Design Analysis 6

2.2.1 Model 6

2.2.1.1 Airport 6

2.2.1.2 Email 7

2.2.1.3 Loading and Saving XML 7

2.2.2 View 8

2.2.3 Controller 8

2.3 Interface Design 9

3 Testing 10

3.1 Test Plan 10

3.2 Test Report 10

4 Screenshots 10

# 1 Approach Taken

## 1.1Technical

We plan to make use of many different tools in throughout our project:

1. **Java** - The project specification requires that we work in the Java programming language; this works well as we have all had experience working with Java and are familiar with it.

1. **Eclipse** - has been chosen by our group as our preferred development environment. The reason for this choice is that it is a good, extensible Java IDE (Integrated Development Environment) with features such as autocomplete that enable us to write code faster than in a standard text editor. Eclipse is also cross platform which is a great benefit since we use a range of Operating Systems within our group. We all have experience using Eclipse.
2. **JUnit** – we will create tests for our application in JUnit to ensure we produce a robust code base.
3. **Google Window Builder Pro** - this Eclipse plugin shall be used to speed up the creation of Graphical User Interfaces for the program.
4. **Microsoft Word 2010** – Shall be used to produce the formal documentation and reports for the project. Word is good at handling medium sized documents and is easy to style to the house guidelines. We shall however submit our reports as PDF files to ensure good compatibility.
5. **Google Docs -** will be used as scratch space for rough notes and ideas, and also to circulate agendas before meetings. Google Docs was chosen because it has been designed to facilitate collaboration. Google Docs also offers a spread sheet application that we shall be using to log out time spent.
6. **ECS ugForge** - We shall use this service to host our version control repository. We shall make regular commits of our work to ensure high levels of transparency on the work being done, and allow rollbacks if a major error is found in our work.
7. **SmartSheet[[1]](#footnote-1)** – We shall be using this web based service as it produces clean, maintainable charts. Using this software will allow us to monitor our progress and decide which tasks to perform next. Frequent updates to the chart will ensure it reflects the real world.
8. Many types of diagram will be used to help us understand various aspects of the project:
   1. **Use Case diagrams** - produced in **Visual Paradigm**, shall help us identify and comprehend all of the use cases from the requirements.
   2. **Class diagrams** - produced using **PlantUML** (a text based UML tool), allow us to break the software into classes that can be coded.
   3. **Sequence diagrams** – also created using **Visual Paradigm** show how the classes are meant to interact within the system.
   4. **Wireframes** - of the user interface will be produced using **Balsamiq** to help us develop a user friendly user interface.
   5. **State charts** – also produced using **PlantUML**, shows the various states of the software

## 1.2 Modelling

To produce good quality, extensible software, a number of different abstractions and architectural styles have been adopted throughout our application:

1. The project specification mandates the use of Model-View-Controller architecture. This architecture separates the classes of the system into three different categories.
   1. The **Model** classes are concerned with the problem domain – all the business logic and calculations for the application are stored in the Model classes.
   2. The **View** classes perform all of the drawing to the screen. In Java these are the Swing classes that produce the user interface. There will be classes for the various windows and controls used in the application. We shall extend many standard controls to meet our requirements.
   3. The **Controller** classes interact with both the View classes and the Model classes. In essence, the Controller tells the Model to update and lets the View know that the Model has been updated so it can redraw the data.

We have chosen to keep the three groups of classes very separate, splitting into pairs to develop them, this allows the program to be more extendable, as the coupling is very low.

1. We shall make extensive use of the Iterator Pattern through the use of the Java enhanced-for loop. This construct allows us to easily traverse over all elements of container classes consisting of Lists of airports, runways and other classes.
2. The Observer Pattern will be used to ensure that every object that is concerned with the values of Airport will be kept consistent when a new Airport is loaded in from a file. An interface AirportObserver shall be implemented by these classes, completing a method notify(Airport) that performs the required behaviour when the airport changes. These classes shall then be added to a List of classes that are observing the Airport. Whenever the airport is changed, the method that changed it shall call a method that iterate through all of the interested objects, calling their notify method.
3. We chose to base many of the Model classes in our project on real life objects
   1. The Runway class is based on the abstract declaration of a runway and, along with another instance of the same class, is contained inside a PhysicalRunway class that represents the real world runway. As many as required of these objects can then be stored inside an Airport class.
   2. For the email section of our program everything is wrapped up in classes. The functionality of sending an email, is contained within a class called Email. This makes it easy for other objects to call on this function without in depth knowledge of how it must be set up. To ease sending emails to commonly used email addresses, the classes Contact, and it’s container AddressBook shall be available.

# 2 Analysis Documentation

In producing our software a lot of thought and discussion was undertaken to decide on the best design, ensuring our application would be able to fulfil its needs to an excellent standard.

## 2.1 Requirements Analysis

The first thing we needed to do was understand the problem that we had been given. We started by reading the specification and various pieces of background information provided by the CAA (Civil Aviation Association). In our early meetings, we discussed what we had read and came to agreements on our interpretations of the information.

We identified the use cases of our project and compiled them into a use case diagram[[2]](#footnote-2) to help us communicate with one another the requirements. The diagram shows that the intended user of the software is Air traffic controllers and engineers; this was taken into consideration when making later design decisions. The number of the original requirements that each use case intends to help satisfy are also show, in brackets, on the diagram. It should be noted that not all of the requirements have been shown to be satisfied by the use cases, this is because they are non-functional requirements that must instead be satisfied by the qualities of the software.

We identified the extensions we would be doing. We chose to do x y z. I guess addressbook, and help?

## 2.2 Design Analysis

Once we had a good understanding of what the software was required to do, we needed to decide how the software would do it. As MVC was mandated, we broke the problem down into three different packages: Model, View and Controller. We aimed to design the software to ensure these three packages, and the classes within them, have as little coupling as possible, along with a good level of cohesion. The interfaces of the classes were defined such that they hide their inner workings from one another. We produced a class diagram to show how the architecture of the program is structured.[[3]](#footnote-3)

### 2.2.1 Model

To identify the classes needed for the model, we began by looking at the problem definition, and identified the nouns. This gave us a good set of classes that represented the problem. These classes made up the bulk of the model package.

#### 2.2.1.1 Airport

The most obvious objects of the model were those that represent the airport, runways and obstacles. We decided to build a class to represent the declared parameters of a runway. We called this class runway. It stores all of the values (TORA, TODA, etc.) that are needed by the user, and provided accessor and mutator methods to each of them. Each value was stored in an array of length two, the first index containing the declared value and the second index the redeclared value. Two public named constants (DEFAULT and REDECLARED) were provided to raise the level of abstraction away for these “magic” numbers.

Since the calculations always require the values for both declared runways that take up one piece of concrete (e.g. 09L and 27R), we created a class PhysicalRunway to represent a real life runway. This class contains two Runway objects. It also contains an Obstacle object. The PhysicalRunway class aims to contain all of the common features of the two runways to reduce redundancy and increase code reuse. All of the methods that calculate the redeclared runway parameters are in PhysicalRunway.

As an airport can consist of multiple runways (e.g. Heathrow) we the class Airport was designed to contain as many runways as required – A List was chosen as the container for the runways, an ArrayList was chosen as the concrete implementation for this list as it is very fast and items never need to be inserted.

The Airport class also acts as a single point of access for other packages to this part of the model; A class needs only instantiate an airport to obtain access to all of this functionality. This single access point reduces coupling and ensures consistency throughout the application.

The final class in the core business logic section of the model is the Obstacle class. As previously mentioned, it lives inside a PhysicalRunway object. When an obstacle is placed on a PhysicalRunway, using the method placeNewObstacle, the redeclared parameters are calculated.

#### 2.2.1.2 Email

The email sending is performed using Java EE’s JavaMail API. However, since it was felt that JavaMail is very hard to use as it offer far more functionality and customisation than we need, a wrapper class, called Email, was created. This class encapsulates all the functionality required to send an email using a predetermined server. We chose to create a Gmail account to demonstrate this, but it would be possible to set this up with any server that offers SMTP access.

Because most of the emails that the application is likely to send are to a small selection of people, we designed an address book system to store commonly used email addresses. The Contact class is a “Plain ol’ data” class that stores the users email address and their full name. The AddressBook class consists of a List of Contacts and methods to add remove and search for contacts.

To send an email, an Email class must be instantiated, a Contact, or List of Contacts passed in. The subject and message body are set and finally the send method is called.

#### 2.2.1.3 Loading and Saving XML

One important feature of the application is the ability to load and save the various parts of the model to file. To satisfy this need, two classes were made, one to save to XML and another to save to XML. The classes are named eponymously: SaveToXMLFile and LoadXMLFile. These classes make use of the DocumentBuilder classes in Java. A third class, named XMLFileFilter, was also created - this class simply sets open file dialogs to show only XML files.

To save an XML file, a SaveToXMLFile object is constructed, passing in the object to be saved as an argument. Airports, Obstacles, Contacts and Contact Lists can be saved as XML by the class. The constructor takes care of displaying the dialogs and creating the correct XML schema.  
Insert lies about why this is good?

To load an XML file, a LoadXMLFile object is created. One of the methods loadObstacle, loadAirport, loadContact or loadContacts is called. The method will then present the user with a dialog to pick a file to load. Finally the method shall return the required object. These classes also provide silent alternatives in which a filename can be passed in as an argument and no dialog will be displayed; this feature was needed for the open recent menu.

### 2.2.2 View

The classes in the view package are concerned with displaying to the screen all the information and visual control elements that the user needs. We chose to implement the application as a Java swing desktop application as this allows good cross platform support, good familiarity and we are all experienced in using swing.

After a discussion with our supervisor, we decided to use a Graphical User Interface builder. Online research showed that the most critically acclaimed GUI builder was Google Window Builder Pro (GWBP). Unlike other tools that would have prevented us from editing the generated code by hand (and probably filled up our classes with useless junk) GWBP was quite happy for us to edit its underlying code which it would then parse to show us the design view.

Unfortunately not all of our GUI could be created using the GUI Builder. The more dynamic elements had to be coded by hand. We decided that to make finding a recently view airport or obstacle easier, a menu listing the recently opened airports and obstacles should be shown. As it wouldn’t be known at compile time which files would have been recently opened, these Menu Items needed to be dynamically generated both from a persistent file of the previously recently opened files, and any files that are opened while the application is running. A few helper functions were written to perform these tasks.

Another set of dynamic menus had to be created for the runway selection. It would not be known which runways exist within an airport until that airport has been loaded in. This was dealt with in the same way as the recent files.

To draw out a visualisation of the runways redeclared parameters, a panel was extended and used as a canvas for drawing lines to. The classes TopView and SideView were created to display the runway from a bird’s eye view and a worm’s eye view respectively. A very simple interface was created for using these components; they are simply instantiated, passing in an airport, and then added to some other component on a frame. The Views are registered as AirportObservers and receive notifications whenever a new airport is loaded in or created.

### 2.2.3 Controller

The controller classes were implemented in the form of ActionListeners. Their constructors took as parameters as little as possible to ensure weak coupling. Many of the listeners were registered as AirportObservers, others acted as sources of notifications to the Observers taking a List of AirportObservers as an argument and then notifying each member of the list when their event occurs, and they have performed their action.

Explicit listener classes were chosen over anonymous classes as we believed it would lead to more maintainable code. The Listeners were placed in a package Controller.

## 2.3 Interface Design

**This section has been flagged for a rewrite because I haven’t read it but it looks a bit smelly:**

The user interface is important. Needs to be easy to use etc.

This software product will provide visual and graphical tools to recalculate and present revised runway parameters, visualizations and summarized calculations to the customer, given and obstacle located in it.

This team will take advantage of the widely known components of Java’s Graphical User Interface libraries and their high affordance and ease of use (e.g. drop down menus, menu bars, buttons, text areas, etc.) and allowing focus on those considered as the main usability goals.

While designing this system, the following usability goals will be regarded as most important:

* ‘Effective to use’; the aim is to build a set of tools that effectively ‘solve the problem’.
* ‘Safe to use’; the aim is a system with low error rate and easy recovery.
* ‘Good utility’; the system will count with a very appropriate set of functions to aid the customer when dealing with obstacles in a runway.

The system assumes a set of users familiarized with the concepts that they will be exposed to, including CAA’s (UK Civil Aviation Authority) rules and regulations, runway redeclaration, and runway parameters, including TORA (Take-off run available), ASDA (Accelerate Stop Distance Available), TODA (Take-Off Distance Available) and LDA (Landing Distance Available).

With the aid of this software product, rapid indication of the effects of an obstruction can be obtained and used to decide if operations in the runway can continue and if performing the manual calculations is worthwhile.

# 3 Testing

Testing is an important part of software development, ensuring that software meets the user’s needs.

## 3.1 Test Plan

Our testing has been broken down into several phases:

1. **Unit Testing** - Unit tests shall be performed by the authors of each class. Normal, Boundary and Error data shall be used to ensure that each class does what is expected. The unit testing will focus mainly on the Model as the rest of the application assumes it is functionally correct.
2. **Integration Testing** - Once each of the Classes has been unit tested, they shall be assembled and integration tests will be performed to ensure that they work together in the desired manner.
3. **System Testing** - The integrated application will be tested on a variety of different operating Systems. This System test will ensure that the GUI and functionality is consistent on different platforms.
4. **User Acceptance testing** - This will come last and will ensure that the application meets the requirements of the User. The program shall be black box tested against the requirements.

## 3.2 Test Report

Partial test report including a good representative selection of test results, including UAT. We understand that it is difficult to complete your testing by this date but expect coverage of at least 75% of plan. Ideally, this and the test plan will be combined in tabular format.

A table might look like, this one doesn’t have any legs ☹ :

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Input | Expected Output | Actual Output | Type | Comments |
| Click on the picture of a plane | The universe implodes | La Bibliotheca es la muñeca | Boundary | What a shock, totally wasn’t expecting that |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

# 4 Screenshots

This section has been flagged for renaming. The final document shall include this section as a short narrative, possibly integrated into the rest of the document, that will reference screenies in the appendix.

A small number of screenshots to enable your examiner to make the link between your demonstrated and your paper submissions (as an appendix).

1. Can be found at [www.smartsheet.com](http://www.smartsheet.com) [↑](#footnote-ref-1)
2. Found in appendix x [↑](#footnote-ref-2)
3. Can be seen in appendix y [↑](#footnote-ref-3)