Coursework B Group 2 Report - Check In

Simon Grange, Safa Al Ameri, Mohamed Serry, Bruce Tauro, Rania Ballout

12th June 2020

1.Introduction

This involved the addition of threads and design patterns, and the development of a suitable GUI to show the state of the simulation. The work was broadly divided into two ‘Epics’. The first related to Core functional requirements and the second to extended functional requirements.

Epic 1 - Core Functional Requirements

These are the core functional requirements met in this assignment. These are addressed first with the extended requirements section below in epic 2. Table 1 Lists the developments, the primary developer and the state of readiness.The application simulates the arrival of passengers at the airport. This simulation has control functions vs. operate + manage a separate control panel consists of the following steps;

Step	Functional Requirements	Software Requirements - models employed	Lead Author
	Passenger entry to airport - Read in details of passengers and flights when on start up	File upload and read	Safa
1)	Multiple check-in desks	Multithreaded desks	Rania
2)	Single queue of passengers - Joins the end of the queue - Assign passenger to a check in desk	Random selection of passengers who have not yet checked in	Bruce
3)	Group of flights waiting to depart	List of flights read to separate lists	Rania
4)	When passengers arrive at the airport, they join the back of the queue	List / Hash?	Rania / Bruce

5)	When they reach the front of the queue, they will be processed by the next available check-in desk	Managed in Sim	Bruce / Rania
6)	'Check In' - Check in and check Bag ● Weight ○ ( ) Kg ● Size ○ Length ○ Width ○ Height	Each passenger has up to one piece of baggage, and the dimensions and weight of the baggage can be chosen randomly when the customer joins the queue GUI shows the list of **passengers** waiting in the **queue** and the details of what each **check-in desk** is currently doing	Bruce
7)	Passenger assigned to the appropriate flight	Based on the queue	Safa / Rania
8)	Check In desk Closes	period of time has elapsed - check-in desks close	Rania
9)	Flights Depart	planes depart - remaining in the queue will not be able to board their flights	
10)	Log Generated	Log of control processes	Simon

Table 1: Functional Requirements Delivery

At each stage this is displayed on the dashboard by Mohamed Serry

Software Engineering Requirements

Software Components

The functionality that our system meets the specification and is outlined in table 1 above. Extension to the core requirements is detailed separately at the end of this section and the comments in table 1 relating to these are *italicised - additional functions.*

- UML Class diagram

GUI

This shows the list of passengers waiting in the queue. It provides details of what each check-in desk is currently doing;

List of data displayed		
Screenshot		

Figure 2. Screenshot of Data displayed on the dashboard

a) Processing a particular passenger
b) Charging for excess baggage
c) Status of each flight
    i) Number of passengers checked-in
    ii) Weight of baggage *etc.*

The GUI was completely redesigned from stage 1, with completely different code. This employes the MVC pattern.

Log

Logging refers to the recording of the application activity. Whilst logging can be used to store exceptions, the primary purpose here is a record of the information, and is extended to provide warnings as messages that occur during the execution of the check-in process.
This records the following events in near real time via a singleton pattern;

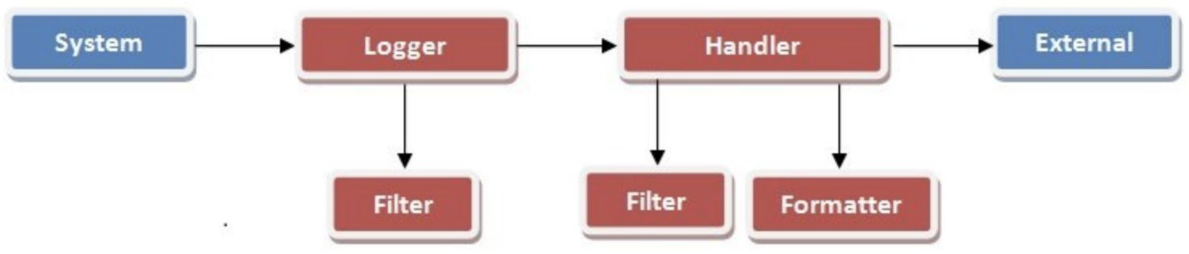
a) Passengers joining the queue
b) Passengers checking in
c) Passengers boarding flights

The output log is written to a file on closing the application. These processes record our application's activity. Logging is used to store exceptions, information, and warnings. These messages occur during the execution of a program and are reported to a file on closing the check-in process. Whilst logging is used in the debugging process of a program, on this occasion the purpose is to provide evidence of completed activity. Using the java.util.logging package, the Logger object is used to log such messages.

The Logger object generates a LogRecord object which stores the message to be logged, which is then forwarded to the [XXXX Airport Log] handler assigned to the Logger object. The check-in system uses an AirportLogFilter to structure the log messages. The handler then publishes the logged messages to the output file; [filename].

AirportLogOutput

The Logger allows the application to log independently of where the output is sent. The application logs the message by passing an object (+/- exception with an optional severity level) to the logger object under the identifier; [log id].



Redraw with the revised singleton pattern

Log Formatter

The Formatter is an object that consists of taking the binary object and converting it to a String representation for the three records above.

Appender - Extended functionality

The appender listens for messages at or above the specified minimum severity level. This includes passengers who missed flights or did not arrive. The Appender takes these messages and posts them appropriately. Message dispositions include:

a) Passengers failing to join the queue
b) Passengers failing to check in
c) Passengers failing to board flights

The output of these [‘warning’] logs

d) Display on the console via the syslog
e) Write to the AirPortLog file on exit

Other options such as appending to a database table, distributing via Java Messaging Services (to the other services), sending via email to notify those who will act on the results or writing to a socket if they have JDBC access to a database recording passenger flow were beyond the scope of this assignment.

Extending the core requirements

The team experimented with agile methods and ultimately agile DevOps. The first Epic focused on core functionality and had iterations every two weeks and features were developed in each iteration according to the backlog. Team planning via Webex (Cisco systems, USA) managed the SCRUM meetings where design, development and testing of each iteration was completed before discussing the future iterations. Some refactoring of code was performed at the end of each iteration based on the most recent ‘pushes’ to branches in the GitHub repository. https://github.com/raniazqt/HW_Airport/tree/feature/logger.
Our team implements pair programming (SG and AS for GUI development and SG and SA for Log / IO development), ‘stand-up meetings’ were managed virtually via the ‘What’s App’ (Facebook, CA) communications tool for the group - see figure 2 below;

- Constraints

- Needs GUI refinement (currency display)
- Error messages (*e.g.* >3 letters)

MS added: GUI?

Design

Use of Threads

These ensure that they are synchronized and do not interfere with one another. For the core functional requirements, we have one thread for each of the check-in desks, and one thread to add passengers to the queue. We [* which] added more threads to the application for developing extensions and also considered more advanced scenarios like the producer-consumer model. Thread-safe versions of some collection classes [Which?] were not adopted so as to demonstrate knowledge of threading.

Use of Patterns

The following design patterns were used in our application architecture; Singleton, Observer and MVC.

Singleton Pattern

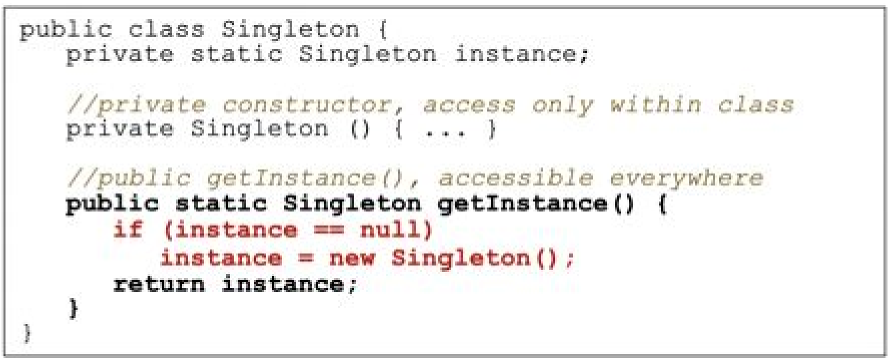
Our team employed the Singleton pattern to implement our log class. It is used to ensure only one instance of a class exists in our system, whilst allowing other classes such as the handler and formatter to get access to this instance. The log class is accessed by various objects in a system such as the queue, check in and the register of who is eligible to board flights. This is a common use for the singleton pattern, with the added benefits of access control for the user to see the reports.

Singleton implementation - code snippet

```
public class NextNumber {
private static NextNumber instance = new NextNumber();
```

private int number; // stores current number
private NextNumber() { this.number = 0; }
public int getNext() { // returns next number number++;
return number; } public static NextNumber getInstance() {
return instance; } }

This also offers lazy instantiation, which since the logging system must always generate an output on exit, ensures automation of the process for AirportLog: + ur generation.



```
public class Singleton {
   private static Singleton instance;

   //private constructor, access only within class
   private Singleton () { ... }

   //public getInstance(), accessible everywhere
   public static Singleton getInstance() {
      if (instance == null)
         instance = new Singleton();
      return instance;
   }
}
```

based on this

Observer Pattern

The observer and MVC patterns are used in the GUI. The purpose of the observer pattern in this scenario is to define a one-to-many dependency between the objects when multithreading so that when one object changes state, all of its dependents are notified and updated automatically. The main motivation for this is to reduce coupling between the classes and increases reusability as each of these dependents publish their output to the GUI.
Effectively this allows observer objects to register interest in other objects' state (subjects), so that whenever the subject changes state, the subject informs all the observers.
The [XXXX] interfaces is used to add generality and flexibility, and further reduce coupling between classes, so the subject interface, which is implemented by the [XXXX] subject class updates the Observer interface, which is implemented by each [XXXX] observer class.

Clock class, holding data about time from the queue
3 Observers – objects of the AnalogDisplay, DigitalDisplay, and Counter classes, react when they are told the time has changed so the display objects react by altering their display;
[screenshot - GUI Queue registration time]
The counter class increments a counter and shows a message
[screenshot - Next passenger - check-in time]
The Subject (here the Clock object) knows who to tell when it has changed? By;
[source code - updates]
The Subject class keeps a list of all the objects (Observers) which need to be told that the information has changed
Whilst the Subject class has methods to add or remove Observers from the prescribed list, it remains static in this system.

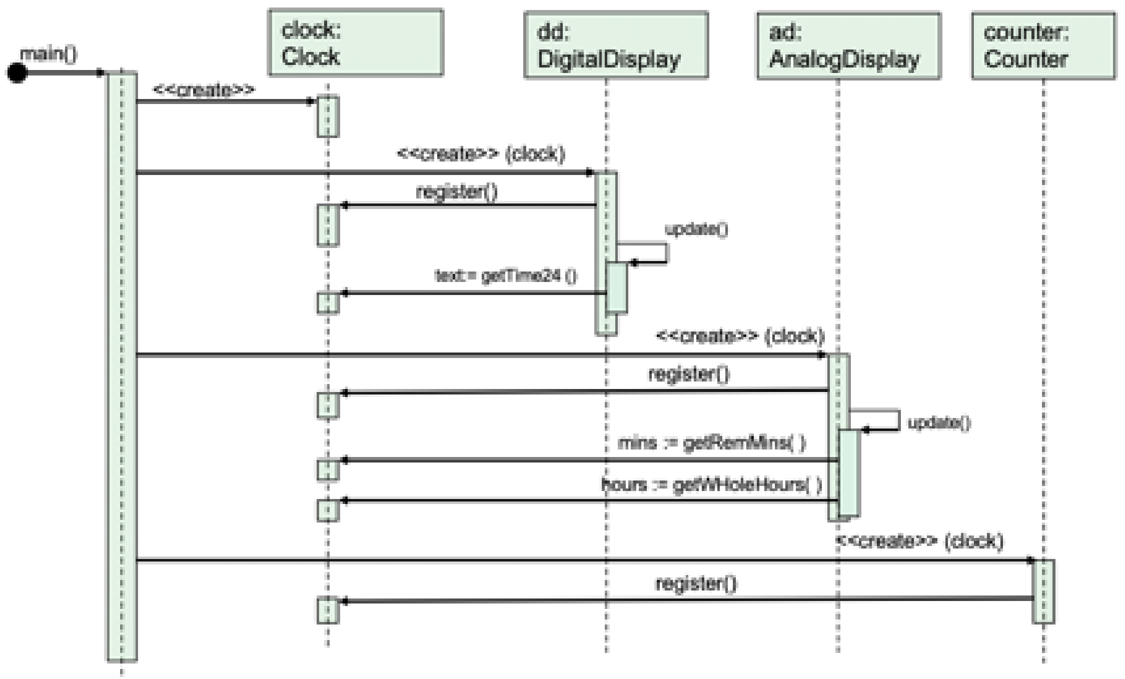


Figure X: Sequence diagram - The observer pattern is used for the time management

MVC Pattern

The architectural design pattern refers to the compound 'model, view, controller' pattern separates a program's data, its display, and how it handles user interaction;

The Model

The underlying data of the program and any associated logic (i.e. methods)

The View

The visual display/presentation of the model

The Controller

Handles user interaction with the model

This pattern says that you should implement the model, view and controller as separate modules that are largely independent of one another. MVC uses three other design patterns, the 'Observer' outlined above as a glue between View and Model. The Strategy pattern acts as a glue between View and Controller. Finally the Composite pattern handles interface updates, since the GUI is a composite of components. MVC therefore makes it easy to update the interface of our program, since the interface code is already separated

Other Design Patterns

No other design patterns were used in the preparation of this system.

Packages

The following package structure was adopted;

- Src
    - GUI
    - Logger
    -
    -

Version Control

Version control was managed via GitHub;

Repository structure -branch management

TypeNameLatest commit messageCommit time

A single example is provided her but the rest of the commit list is available at;
[URL]

Delivery

The final application was exported to a jar file, to run. Due to the change in program requirements consequent on the COVID-19 Pandemic restrictions our group will demonstrate virtually by submission of the necessary materials.

Jar file at;

[Address]

Epic 2 - Extended Functional Requirements

You are also asked to extend the core requirements.

Marks will be awarded based on the complexity of your extension(s)

knowledge and understanding required to implement them.

1. Allow the user to alter the speed of simulation using runtime controls.

2. Allow the user to open or close individual desks as the simulation proceeds, or the simulation could automatically open new desks (up to a limit) when the queue gets to a certain length (10 passengers).
3. Your simulation could have more than one queue, with each queue feeding passengers to more than one check-in desk. You could limit the size of queues. You might introduce different cabin classes (*e.g.* economy, business) and have queues with different priorities for these.
4. Rather than the check-in desks closing at a particular time, each flight could have a specified departure time, and check-in desks will only accept passengers who arrive before their flight departs. You could show flights departing by removing them from the GUI.
5. If you're feeling even more ambitious, you could simulate more details of the airport experience, e.g. queuing for security. However, bear in mind that a small number of good quality extensions are better than lots of rushed extensions.

UML Diagrams

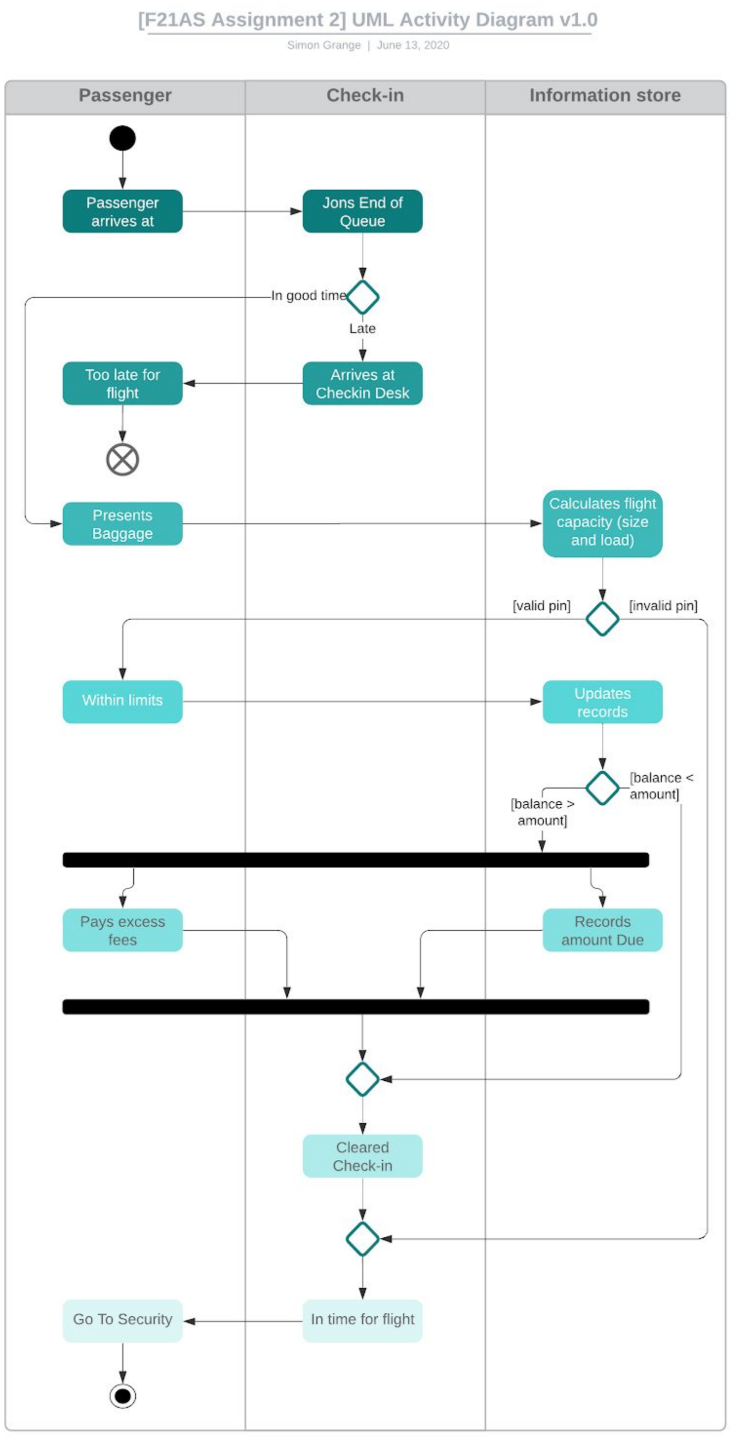


Figure 1: **[F21AS Assignment 2] UML Activity Diagram -** This shows the main activities of the workflow, and the contents of each stage of the process

These show the associations between the top levels of classes for the Check-in process with related classes, and the contents of each class being detailed in Appendix I.

Top level package diagram demonstrates how the different classes are integrated into packages and then into modules though for this level of project this extra layer of complexity can be avoided - using the single HWAirport package.

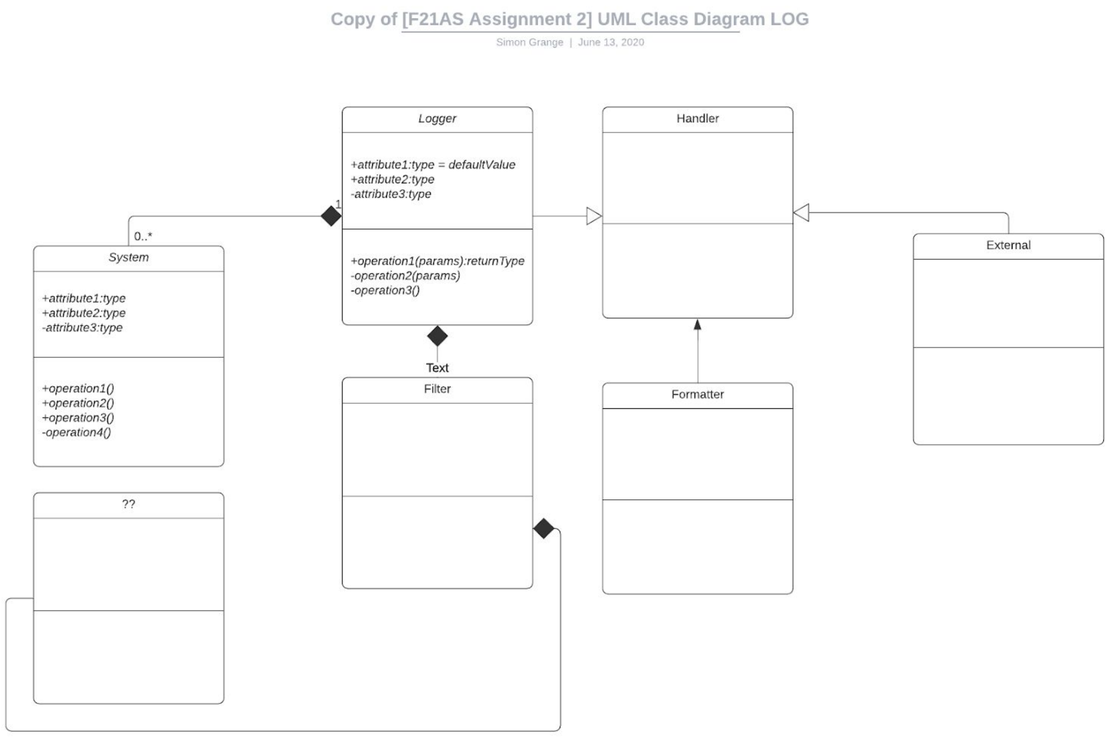


Figure 3: Class diagram demonstrating the relationship between the different classes

Figure 2: **[F21AS Assignment 2] UML Class Diagram in code -** This shows the associations between the classes, and the contents of each class

UML (lecture notes)

Agility

According to the Manifesto for Agile Software Development:

•We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

•Individuals and interactions over processes and tools Working software over comprehensive documentation Customer collaboration over contract negotiation Responding to change over following a plan

•That is, while there is value in the items on the right, we value the items on the left more.

[see www.agilemanifesto.org]

In the spirit of this, and using 'state of the art' methods; our team developed the program using agile processes. JIRA was adopted as the established shared sprint work environment. Details of this are available at; [JIRA view only address with name and login?]

Agile techniques and tools

Flexible hierarchy;

- RoadMap
- Epics
- Storyboards
- Stand Up Meetings
- SCRUM

Group Report

A stage 1 / 2 comparison reflects the necessary changes in work practice from a 'waterfall' traditional method to an agile & ultimately an Agile DevOps one. These are outlined in the table 2 below;

The aim is to move away from the mythical man month (Fred Brooks) which is an approximation of work volume and complexity [Ref] and has little insight into the likely complexities

	Stage 1 Approach	**Stage 2 Approach**
Advantages	Face-to-Face team building, made workshops with white boards easier, more spontaneous and even time	Completely virtual infrastructure allowed the project to continue under the terms of strict physical isolation of staff members

	efficient	
Disadvantages	Rigid approach generated a ‘time crunch’ when deadlines were superimposed from independent activities such as other projects.	Time frames were less precise, Agile methodology maintained activities through multithreaded linear progression of service development through challenges with a single fixed deadline

Table 2: Comparison of Stage 1 and stage 2 work methodologies

Arbitrary deadlines such as internal demos are therefore added to act as milestones by the team.

Sections of agile technique are detailed in [section XXXX] below.

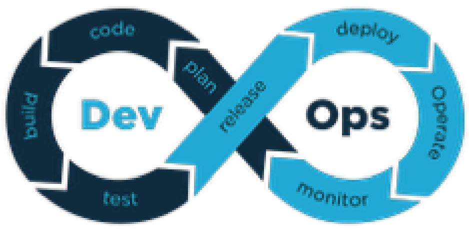
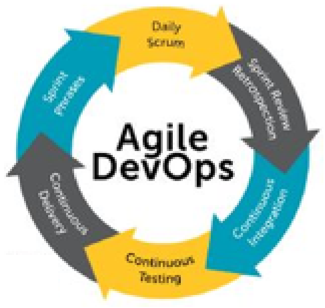
	Description		**Pros**	**Cons**
Dev Cycle				
DevOps	 code plan build Dev release Ops deploy Operate monitor test			
Agile DevOps	 Daily Scrum Sprint Review Retrospection Sprint Phrases Agile DevOps Continuous Delivery Continuous Integration Continuous Testing			

Table 4. The preferred Agile DevOps plan based on a team working on different sites asynchronously

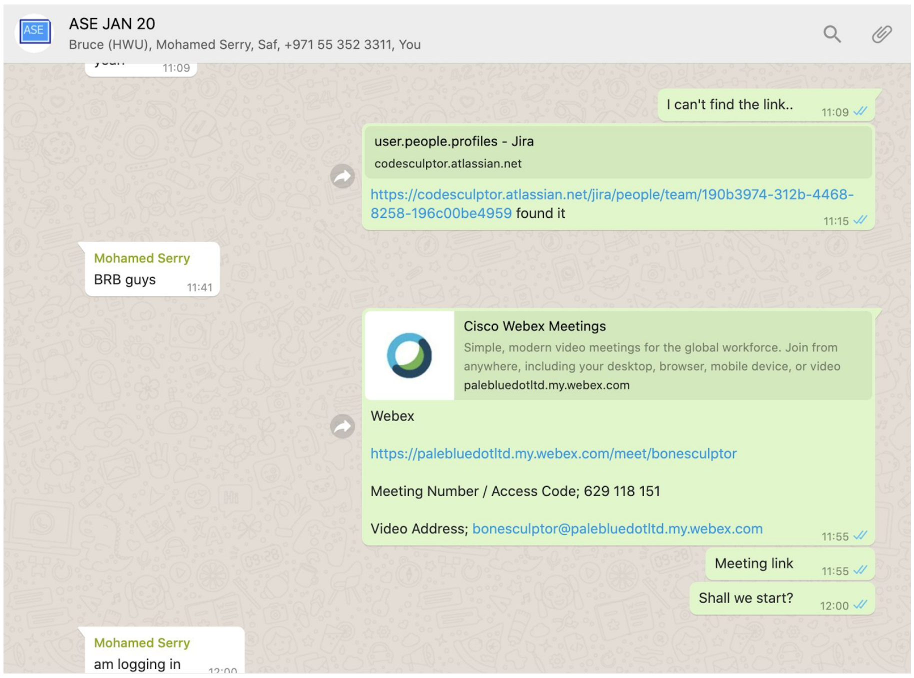
The following steps are managed by the team;

1. **Daily Scrum:**
2. **Sprint Review:**

3. **Continuous Integration:**
4. **Continuous Testing:**
5. **Continuous Delivery:**
6. **Sprint Process:**

This is best to think of the overall flow so that the activities list in table 1 expanded to reflex the processes in the activity diagram.

Stand Up meetings



ASE JAN 20
Bruce (HWU), Mohamed Serry, Saf, +971 55 352 3311, You

11:09

I can't find the link.. 11:09

user.people.profiles - Jira
codesculptor.atlassian.net
https://codesculptor.atlassian.net/jira/people/team/190b3974-312b-4468-8258-196c00be4959 found it 11:15

Mohamed Serry
BRB guys 11:41

Cisco Webex Meetings
Simple, modern video meetings for the global workforce. Join from anywhere, including your desktop, browser, mobile device, or video
palebluedotltd.my.webex.com
Webex

https://palebluedotltd.my.webex.com/meet/bonesculptor

Meeting Number / Access Code; 629 118 151

Video Address; bonesculptor@palebluedotltd.my.webex.com 11:55

Meeting link 11:55

Shall we start? 12:00

Mohamed Serry
am logging in 12:00

Figure 2: ‘Stand-up’ meetings managed via ‘What’s App’.

With regular time-boxing, though, the latter became somewhat practical to use for our project due to the changes in project schedule and remote working restrictions.

SCRUM

Agile and DevOps serve complementary roles: several standard DevOps practices such as automated build and test, continuous integration, and continuous delivery originated in the Agile world, which dates (informally) to the 1990s, and formally to 2001.[22] Agile can be viewed as addressing communication gaps between customers and developers, while DevOps addresses gaps between developers and IT operations / infrastructure.[23] Also, DevOps has focus on the deployment of developed software, whether it is developed via Agile or other methodologies.[22].

Project Development Philosophy

Starting with traditional waterfall in part 1 we progressed to an agile methodology in part 2 and finally progressed this towards DevOps methodology towards the end of the project. DevOps is;

"a set of practices intended to reduce the time between committing a change to a system and the change being placed into normal production, while ensuring high quality".[6]

The term DevOps, however, has been used in multiple contexts.[7

This relies on completing the following steps;

As DevOps is intended to be a cross-functional mode of working, those who practice the methodology use different sets of tools—referred to as "toolchains"—rather than a single one.[17] These toolchains are expected to fit into one or more of the following categories, reflective of key aspects of the development and delivery process:[18][*unreliable source?*][19][*unreliable source?*]

1. Coding – code development and review, source code management tools, code merging.
2. Building – continuous integration tools, build status.
3. Testing – continuous testing tools that provide quick and timely feedback on business risks.
4. Packaging – artifact repository, application pre-deployment staging.
5. Releasing – change management, release approvals, release automation.

6. Configuring – infrastructure configuration and management, infrastructure as code tools.
7. Monitoring – applications performance monitoring, end-user experience.

Some categories are more essential in a DevOps toolchain than others; especially continuous integration (e.g. Jenkins, Gitlab, Bitbucket pipelines) and infrastructure as code (e.g., Terraform, Ansible, Puppet).[20][*unreliable source?*][21][*unreliable source?*]

RoadMap

[Screenshot - JIRA]

Epics

Storyboards

Our group did not rely on a roadmap for a short project like this as the specifications were clearly defined in the scope.

Epics

There were two main phases of the coursework and of the second part there were two epics;

Epic 1 - Related to the core functionality

Sprints

Sprint 1 - Design of the application including

Sprint 2 -

Sprint 3 -

Epic 2 - The additional extensions to the core program.

Sprints

Sprint 1 -

Sprint 2 -

Sprint 3 -

Version Control was managed via GitHub

Agile techniques explored included pair programming (Simon Grange & Mohamed Serry - GUI, and Safa Al Ameri & Simon Grange Log I/O) stand-up meetings (performed via Webex) and time-boxing using separate Sprints. This was necessary due to the team being under lockdown for the duration of the project in separate cities due to the presence of the COVID-19 pandemic. This imposed constraint demonstrated the effectiveness of the methods employed to deliver management services for the software development.

Observations

4. An explanation of how and where threads are used in your application.

    Threads

        Multithreading for different check-in desks

6. Sample screen shots.

    Screenshots of different stages of the application

    Figure. Sample of the GUI screen

7. A brief comparison of your development experiences in Stage 1 and Stage 2.

    Did plan-driven or agile development work better for your group?

        plan-driven (Pre-COVID) worked well with regular meetings 'in the office' but would not have worked during the COVID-19 outbreak as it was necessary to have irregular meetings due to different commitments.

    What problems did you encounter, and what might you do differently next time?

        In future the model of using

        GUI design can use dedicated GUI generation tools

    Diagrams

        UML diagrams where appropriate

    snippets of code where relevant

        Patterns

            Singleton

            Observer

Sections to Integrate in the final report

Functionality

Core requirements complete

Extensions

User Interface

Demonstration

- All features shown to work in demo

Implementation

Code quality,

- Readability
- Comments

Software Documentation

Technical Writing

UML

Report

Iterations

Development Tools

- Agile techniques

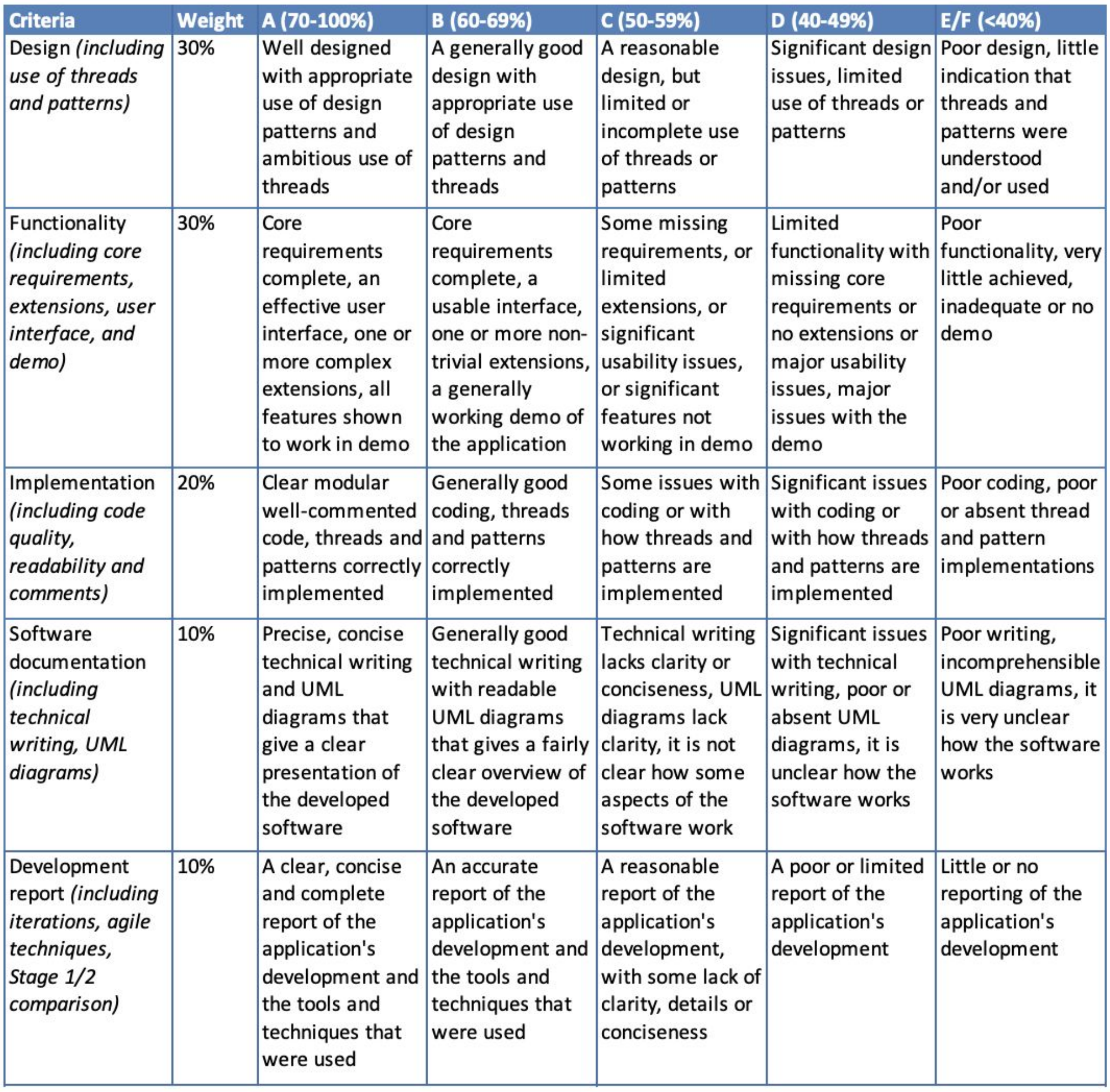
Stage 1 to 2 comparison

Appendix I

UML Diagrams

The UML class diagram for the singleton pattern employed in the logging system allows for a single point to access it, and in so doing controls access to the instance. This avoids the use of a global variable.

Appendix II - Marking criteria



Criteria	Weight	A (70-100%)	B (60-69%)	C (50-59%)	D (40-49%)	E/F (<40%)
Design *(including use of threads and patterns)*	30%	Well designed with appropriate use of design patterns and ambitious use of threads	A generally good design with appropriate use of design patterns and threads	A reasonable design, but limited or incomplete use of threads or patterns	Significant design issues, limited use of threads or patterns	Poor design, little indication that threads and patterns were understood and/or used
Functionality *(including core requirements, extensions, user interface, and demo)*	30%	Core requirements complete, an effective user interface, one or more complex extensions, all features shown to work in demo	Core requirements complete, a usable interface, one or more non-trivial extensions, a generally working demo of the application	Some missing requirements, or limited extensions, or significant usability issues, or significant features not working in demo	Limited functionality with missing core requirements or no extensions or major usability issues, major issues with the demo	Poor functionality, very little achieved, inadequate or no demo
Implementation *(including code quality, readability and comments)*	20%	Clear modular well-commented code, threads and patterns correctly implemented	Generally good coding, threads and patterns correctly implemented	Some issues with coding or with how threads and patterns are implemented	Significant issues with coding or with how threads and patterns are implemented	Poor coding, poor or absent thread and pattern implementations
Software documentation *(including technical writing, UML diagrams)*	10%	Precise, concise technical writing and UML diagrams that give a clear presentation of the developed software	Generally good technical writing with readable UML diagrams that gives a fairly clear overview of the developed software	Technical writing lacks clarity or conciseness, UML diagrams lack clarity, it is not clear how some aspects of the software work	Significant issues with technical writing, poor or absent UML diagrams, it is unclear how the software works	Poor writing, incomprehensible UML diagrams, it is very unclear how the software works
Development report *(including iterations, agile techniques, Stage 1/2 comparison)*	10%	A clear, concise and complete report of the application's development and the tools and techniques that were used	An accurate report of the application's development and the tools and techniques that were used	A reasonable report of the application's development, with some lack of clarity, details or conciseness	A poor or limited report of the application's development	Little or no reporting of the application's development

Unlinked References