Onerous Android Application

Architecture/Design Documentation

Table of Contents:

1. Introduction
2. Specification
3. Logical View
   1. High-Level Design
   2. Mid-Level Design
   3. Low-Level Detailed Design

1 Introduction

This document describes the architecture of the Onerous App developed by the Modelling and Simulations Group of Cranfield University based at the Defence Academy Shrivenham. The Onerous App will be used to teach and demonstrate students about stochastic discrete event simulation and their use in operational analysis. It models helicopters stationed at the frontlines with engines that need to be regularly maintained.

The purpose of this document is to describe the architecture of the Onerous App clearly and accessibly to meet a few requirements:

1. Reduce time taken for future developers/programmers to understand the code at a level needed to be able to maintain, modify, enhance and perhaps even port the app to another platform such as iOS.
2. Allow students to explore and understand how the simulation is built and how it works.
3. Provide multiple views to support various different specialised interests in accordance with IEEE std. 1471.

What this document will not do is explain the Android or Java framework. Understanding of either should not be necessary in understanding the architecture of the application. However, competence with both Java and Android will be necessary for code maintenance or modification.

It will not detail how the individual methods or functions operate but will summarise their jobs and input/outputs. If further technical detail concerning the implementation of individual methods and functions are required, the commented source code can be found in the project files.

2 Specification

The Onerous Android App based on the original Onerous V2.0 desktop application, which will be referred to as Onerous 2008. Onerous 2008 is a desktop application built using Microsoft Visual Studio 2008 using the Visual Basic language.

The objective was to port Onerous 2008 so that it could run on tablets running the Android OS. The specific model that was available at the SSEL was the Samsung Tab 2 10.1 running android version 4.1.1 (16 Jellybean). All functionalities had to be retained and the app UI had to be updated to one that is geared towards tablet usage with a touchscreen as opposed to Desktop usage with mouse and keyboard.

For further information on the specification and details leading to the creation of the original Onerous 2008, file OA 1229 and the presentation file FMS\_SSE\_16\_7\_Onerous\_Responsibility can be referred to.

System Behaviour overview

This section aims to give a brief summary of how the app should behave and what it functions it should be fulfilling so there is context for the underlying logic and architecture.

The Onerous apps functions can be split up into three core functions:

1. Viewing the current simulation run settings and changing them if desired.
2. Running the simulation with the option to view the events animated and visually represented on an activity cycle diagram, or to turn them off and complete the simulation instantly.
3. The ability to view statistics and graphs relating to each entity and its state e.g. the amount of helicopters that are operational.

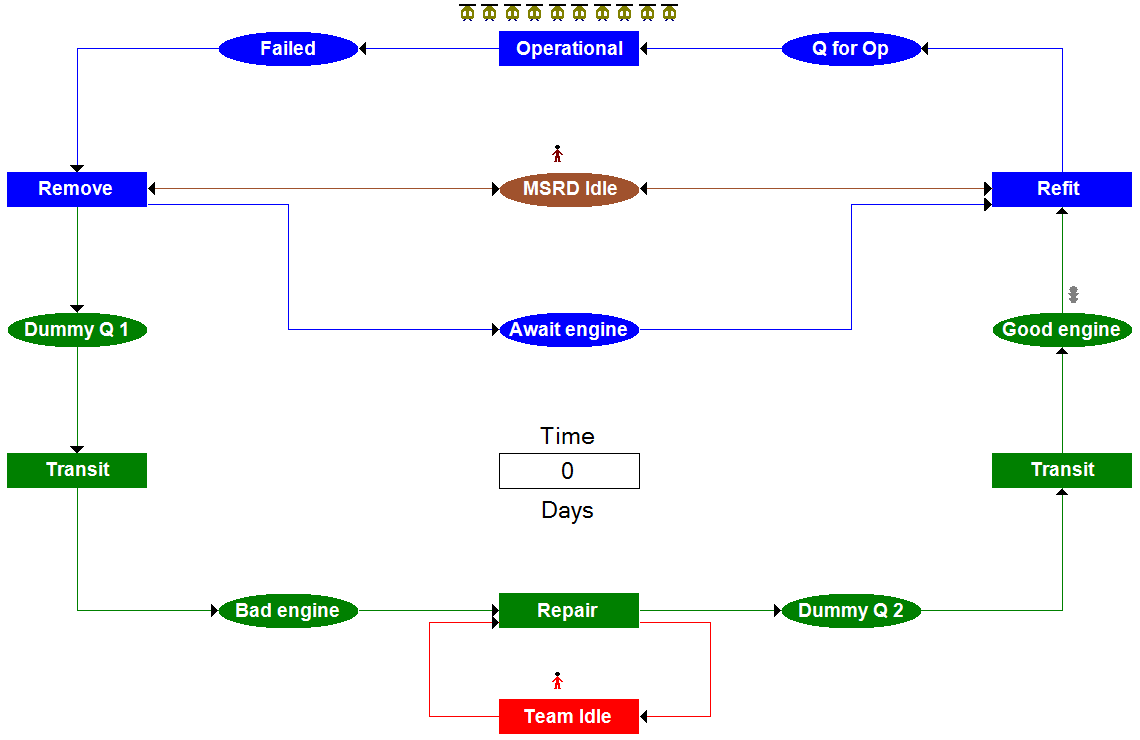


Fig 1. Activity cycle diagram of Onerous 2008

Fig.1 shows the activity cycle diagram that illustrates the model the simulation will be using for the simulation. There are six main activities that occur: helicopters operational (until engine failure occurs), engine removal, transit to manufacturer, engine repair, transit to operation and refit time.

**3 Logical View**

The logical view describes the components of the app and how they work together to provide the overall functionality of the application. The logical view will begin at a high level view and work its way down to low level. High level will explain and show how the major components work together at an architectural level and will then progress towards more detail by showing smaller components and then eventually to individual methods and functions, but any further detail will require a look at the source code and the comments included with it.

**3.1 High-Level**

The three core functions of the Onerous 2008 application has been split up into three different UI screens for the android application. They have been chosen to be logically intuitive for the user and we can use these as the major components that build up the high level architecture view.

**Memory**

View current run settings.

Edit run settings.

Restore default settings.

**Variables**

Show statistics of each activity.

Show graph of each activity.

**Results**

Run simulation.

View animation/graphics.

Toggle graphics on/off.

**Simulation**

*Read*

*Write*

*Read*

*Write*

*Read*

***Fig.2 Diagram showing major components of Onerous Android App 2017***

The function of each component is shown in the diagram and what kind of interaction the component has with the memory. The read/write interactions are expanded on further below:

* Variables
  + Read: Reads the current simulation settings stored in the memory to display to the user and accesses it every time the settings are edited to update the display for the user.
  + Write: Writes the users inputted changed settings into the memory. This overwrites the previous settings.
* Simulation
  + Read: Reads the simulation run settings and runs an instance of the simulation accordingly. Reads run variables used to keep track of entities and other values, such as runTime*,* of the specific instance of the simulation that is running.
  + Write: Writes any changes to the run variables as the instance of the simulation progresses. Stores data that will be used to generate statistics and graphs as the simulation progresses. Calculates various statistics and datasets at the end of the run and stores it in the memory.
* Results
  + Read: Reads the data generated and stored during the simulation run that to generate a table of statistics and a graph of each node of the activity cycle diagram.
  1. **Mid-level View**

Variables Tab

The Variables Fragment contains simple sequential logic with no loops or ‘if’ statements. For this reason a multilevel list should suffice in describing the process the code follows. There are three different events that trigger a process to start: creation of the Variables tab, the ‘Update Variables’ button being clicked and the ‘Restore Default Variables’ button being clicked.

1. onCreateView()
   1. Inflate layout file (variables\_fragment.xml)
   2. Initialise buttons’ onClick functions.
   3. RefreshCurrentVarValues()
      1. Refreshes the display showing the current simulation run settings so they display correct values on start-up.
   4. Setup spinners and what happens when each item is selected: focus (nextFocusDown and nextFocusForward) behaviour, textView text and editText/TextView visibility.
2. ‘updateVarBtn’ onClick Event
   1. Calls updateVars()
      1. Changes the simulation run settings saved in the memory to values entered by the user.
      2. Calls refreshCurrentVarValues()
         1. Refreshes the display showing the user the current simulation run settings with the newly updated settings.
3. ‘restoreDefaultBtn’ onClick Event
   1. Sets the simulation run settings saved in the memory with the default values.
   2. Calls updateVars()
      1. Refreshes the display showing the user the current run settings with the newly updated default values.

Simulation Tab

The Simulation Fragment UI contains the controls for the main part of the program which contains the most complex part of the code of the Onerous App; the run simulation function. The run simulation function will be covered by itself in its own section.

1. onCreateView()
   1. Inflate the layout file (simulation\_fragment.xml)
   2. Initialise buttons’ onClick functions.
   3. Initialise the toggles/switches functions when toggled.
   4. resetImage()
      1. Resets the activity diagram icons so that it shows the starting points of the entities correctly.
2. ‘Run Simulation’ button onClick
   1. On new thread call function SimMethods.runSim()
      1. runSim() will run all aspects of the simulation i.e. graphics, animations, data and the event handling of the simulation itself. It will do so from a different thread to prevent locking the interface up which will cause lag or crashes.
3. ‘Reset Image’ button onClick
   1. Calls Graphics.resetImage()
      1. Resets the icons placed on the Activity cycle diagram to their correct starting location and incorrect quantity.
4. Graphics switch/toggle
   1. If switch is toggled on will set Variables.graphicsOn to ‘true’
   2. If switch is toggled off will set Variables.graphicsOn to ‘false’
5. Animations switch/toggle
   1. If switch is toggled on will set Variables.animationsOn to ‘true’
   2. If switch is toggled off will set Variables.animatinsOff to ‘false’

Results Tab

The Results Fragment UI is very straight forward. It has a table, a graph and 15 buttons that will populate the table and graph with the relevant data relating to each activity state from the simulation.

1. onCreateView()
   1. inflate layout file (results\_fragment.xml)
   2. Initialise Buttons onClick functions.
   3. Initialise empty table and empty graph.
2. On any ‘activity state’ Button click
   1. clearTable()
   2. show\_\_\_\_\_Stats()
      1. Setup the appropriate stats to show on the table for the requested activity state.
   3. populateGraph()
      1. Clear graph and populate it with the appropriate coordinates for the requested activity state.

Variables Class

The variables class is the class that contains all of the static variables for Onerous to run successfully. It does not contain any methods or functions of its own. The variables stored in this class can be logically split up into their different jobs and grouped logically.

* **Simulation run settings**
  + Run parameters
  + Entity numbers
  + Activity distribution variables.
  + Graphics on and Animations on.
* **Internal Variables (used by the app during a run of a simulation. Invisible to the user)**
  + Event Data
  + Sim Time, end flag and stop now.
  + Activity state number values.
  + Graph array fill number values.
* **Statistics/Graph Data (generated during sim run, used to generate results output)**
  + Statistics results data.
  + Graph results data.

Below is a diagram showing how the variables from the Variables Class is accessed and utilised.

**Sim run settings**

Run parameters

Entity numbers

Activity Distributions

Graphs/Animations toggles

**Internal Sim Variables**

Event Data

Sim Time, end flag & stop now

Activity state number values

Graph array fill number values

**Statistics/Graph Data**

Statistics results data

Graph results data

- **Read**

- **Write**

**Memory**

***Fig. x?***

This overview should explain how the flow of data works in Onerous.

Run Simulation

The SimMethods.runSim() function is a core function of the app. It is the top level function that controls the execution of the lower level functions required for the simulation to run successfully. Below is a flow chart of the runSim() function that summarises how the simulation algorithm works.

Call next event function (e.g. Events.engineFails, Events.endRepair etc.)

initVariables()

initList()

resetAllStats()

resetAllGraphData()

Set DaysPassed to simTime

(simTime = 0 at start)

endFlag = true?

getNextEvent()

set simTime to nextEventSimTime

If (graphicsOn = true)

Set DaysPassed to simTime

Set simTime to runTime

Set DaysPassed to simTime

finaliseStats()

Return

Yes

simTime > rumTime?

Set endFlag to true

Yes

No

No

The flowchart should give a rough overview of how the code works to have the simulation run. To go into a little bit more detail, essentially what is happening is that the simulation presumes that you start with all helicopters operational at the start. It then starts things off by creating an ordered list using initList()to simulate these operational helicopters breaking down or being forced to repair after 25 days.

**Event Kinds Legend**

1. Engine Failure
2. Engine removal complete
3. Transit to manufacturer complete
4. Engine refit complete
5. Engine Repair complete
6. Transit to operation complete

**EVENT LIST**

[7.903659904462676 , 1]

[10.60605503862791 , 1]

[12.05211199111656 , 1]

[12.13203440726710 , 1]

[12.38969114357327 , 1]

[17.15016351725560 , 1]

[19.27481299837936 , 1]

[20.95945467054718 , 1]

[21.21235508803384 , 1]

[22.63522666609157 , 1]

***Fig. x***

This is a typical list that the simulation would initiate at the start. The first value is the ‘time’ the event occurs and the second value is the ‘kind’ of event that it is. To begin with all the events are kind ‘1’: Engine failure. The default distribution of the engines breaking down is a normal distribution with a maximum of 25 and minimum of 0 therefore the values appear to be correct.

The ‘for’ loop condition checks whether or not the endFlag value is true or not every repetition. If not then the next event at the top of the list is executed. This process repeats until the simulation runTime is exceeded and the simulation terminates.

**EVENT LIST**

[8.403659904462675 , 2]

[10.60605503862791 , 1]

[12.05211199111656 , 1]

[12.13203440726710 , 1]

[12.38969114357327 , 1]

[17.15016351725560 , 1]

[19.27481299837936 , 1]

[20.95945467054718 , 1]

[21.21235508803384 , 1]

[22.63522666609157 , 1]

**EVENT LIST**

[8.903659904462675 , 4]

[9.003659904462674 , 3]

[10.60605503862791 , 1]

[12.05211199111656 , 1]

[12.13203440726710 , 1]

[12.38969114357327 , 1]

[17.15016351725560 , 1]

[19.27481299837936 , 1]

[20.95945467054718 , 1]

[21.21235508803384 , 1]

[22.63522666609157 , 1]

**EVENT LIST**

[7.903659904462676 , 1]

[10.60605503862791 , 1]

[12.05211199111656 , 1]

[12.13203440726710 , 1]

[12.38969114357327 , 1]

[17.15016351725560 , 1]

[19.27481299837936 , 1]

[20.95945467054718 , 1]

[21.21235508803384 , 1]

[22.63522666609157 , 1]

Call event kind 1 function: engineFails()

Call event kind2 function: endRemoval()

***Fig. x***

You will notice that the event kinds that are called from the list are all terminal events. This means they are all events that mark the end of an activity as opposed to events that mark the start of an activity. In fact these events all have their complementary start events associated with them. However, these start events are not called from the events list in the runSim() function. These other start events are called by the terminal events that precede them. These start events then carry out their own tasks and then schedule the end event and add it to the ‘Event List’ for the end event to be called at the appropriate time. Below is a diagram showing what happens when a terminal event is called from the ‘Event List’.

**Terminal Event Function**

* General event tasks such as animations, graphics and gathering data for statistics and storing graph points.
* Changing the appropriate **numVariable** values. E.g. numOperational -1 after an engine fails.
* Calls start events that follow the particular terminal event. E.g engineFails() calls startOperational() and startRemoval()

**Start Event Function**

* General event tasks such as animations, graphics and gathering data for statistics and storing graph points.
* Changing the appropriate **numVariable** values. E.g. numOperational +1 after an engine is refitted.
* Create new terminal event and add it to the ‘Event List’. E.g. startOperational() schedules and adds engineFails() to list.

**Event List**

* Calls next scheduled event

***Fig. x***

Following the

**Class references**

**ActivityDistribution**

The ActivityDistribution class is a user defined type that holds values that will be used to calculate how long each ‘activity’ (e.g. engine removal, engine refit, bad engine transit etc.) takes.

**EventData**

The EventData class is a user defined type that is used to hold two bits of data: the time of an ‘event’ and the ‘kind’ of event.

**Events**

The class Events contains the methods used for performing the actions required when a certain scheduled event occurs.

**MainActivity**

The MainActivity class is the very first class that launches when the app is launched. It is responsible for initialising the UI view and setting up the tabbed fragments that will hold the majority of the UI elements.

**PlaceholderFragment**

Tabbed placeholder fragment that can hold the place of a tab in a tabbed activity.

**ResultsData**

The ResultsData is a user defined type that is used to hold data and statistics when the simulation runs.

**ResultsFragment**

The ResultsFragment class initialises the UI elements of the ‘Results’ tab, and contains the methods required for populating the results table and the graph with the appropriate data.

**SimAnimations**

The SimAnimations class contains all the methods for performing the various animations when visually displaying the simulation running.

**SimMethods**

The SimMethods class contains much of the backbone for actually simulating the scenario.

**SimulationFragment**

The SimulationFragment class initialises the UI elements of the “Simulation” tab.

**Stats**

The Stats class contains the methods used when generating the simulation run stats.

**Variables**

The Variables class contains all the global variables that the application needs to access from multiple other classes. Many of these global variables are also the variables that control the simulation run settings.

**VariablesFragment**

The VariablesFragment class initialises the UI elements of the “Variables” tab which allows the user to change many of the simulation settings.