Module Code
COM00008I

BSc, BEng and MEng Degree Examinations 2019–20

DEPARTMENT OF COMPUTER SCIENCE

Software Engineering Project (SEPR)

Open Group Assessment

Module	**Software Engineering Project (SEPR)**
Year	**2019/20**
Assessment	**2**
Team	**The Dicy Cat**
Members	**Michele Imbriani** **Dan Yates** **Luke Taylor** **Isaac Albiston** **Marta Cartwright** **Riju De** **Sean Corrigan**
Deliverable	**Software Architecture**

Similarly to our previous architecture, we have once again chosen to represent our architecture using a flowchart to model the runtime of our game and a UML 2.0 diagram to represent the static structure.

Static UML Diagram
Runtime FlowChart

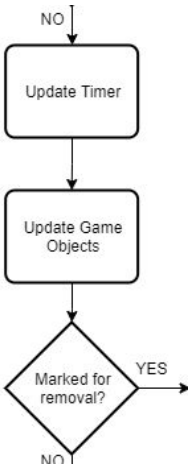
We decided to use UML 2.0 again as it's a well-known standard to model object-oriented languages, such as Java. We used StarUML [6] to actually create the diagram as it has built-in features to help with the correct formatting and style.

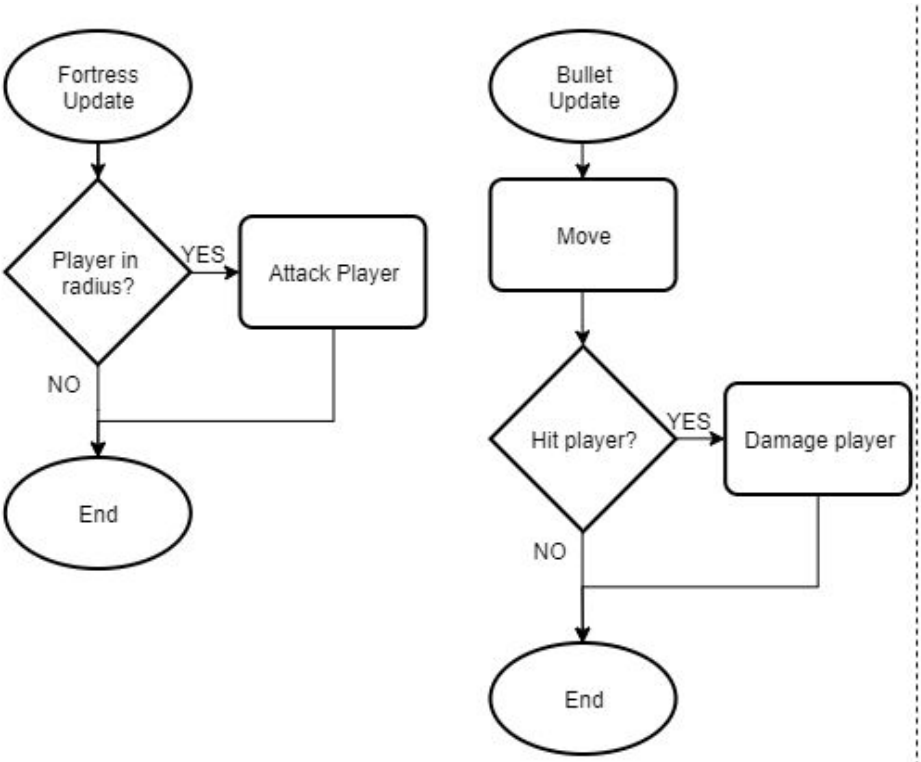
Our UML does not describe all attributes and methods classes have available to them as in order to do that they would lose some usefulness and make the diagram too crowded and difficult to understand. The attributes removed generally concerned the positioning of certain UI features, such as the coordinates of buttons a particular screen needed to place. We excluded these details as they made the UI classes look too significant when these classes didn't add much to model as they do not include any logical details only aesthetics ones instead. After removing them, the model focuses on the details of the logic and how the program works instead of the easily changeable and arguably less important aesthetics.

With these changes in mind, the model we made creates a good image to understand the workings of the code we produced. It transforms the project from a somewhat difficult to follow collection of java files to a more understandable and generally useful.

We used a flowchart to represent the runtime as it provides a good visualisation of the sequence of steps the program runs to meet the requirements and is generally easy to understand and interpret. To create this we used the free tool "Draw.io" [7] as it is a free plugin to google drive which provides a useful method to collaboratively work on the design. Similarly to our initial architecture, we chose to utilise 'draw.io' which is a free diagram creation program that has dedicated flowchart tools. The reasoning for this was that it seamlessly integrates with Google Drive allowing us a useful tool to work on the design collaboratively. We stuck to the same style of flowchart as previously used which follows the ISO 5807 standard but with some extra symbols which we felt added better descriptions.

The model is split into the main logic, subroutines and then methods for when each individual object needs updating or dying. This is because in our architecture we don't specify an order for the game to process the entities in so to show this within the model we use a general "Update Game Objects" [2] (as seen in the picture) with subroutines of how each entity would update separately (pictures below show the update subroutines for the Fortress and Bullet classes).




Fortress Update
Player in radius?
YES
Attack Player
NO
End
Bullet Update
Move
Hit player?
YES
Damage player
NO
End

In this section, requirements will be referred to based on their ID in our requirements Table. A link to the full table can be found below.

The architecture of the system can be split up into 3 main components for which most of our classes could be placed [4]:

Screens, (including “GameScreen”, “GameOverScreen” and “MenuScreen”, blue [4])

These classes implement the libGDX screen interface and provide the main backbone for displaying much of the features of the game. They are initialised and stored statically within the “Kroy” class to keep them accessible but unchangeable throughout the program. They handle much of the logic of the program indirectly because while many of the calculations and updates happen locally within each class, they are called and controlled by the screen classes. The class “GameScreen” is instantiated once per instance of the game and stores all the details and operations that that particular instance needs to run. Benefits of doing this include the ability to store the entire game currently as a save which would be possible to load up in a future version if it were implemented as one class will contain all attributes regarding that instance. This relates to the requirements as it means that each instance of the game will only ever have 4 lives (SFR_FIRETRUCKS_SELECTION, UR_FIRETRUCK_MIN_START)

GameObjects, (including “GameObject”, “FireTruck”, “Bullet”, “StatBar” etc, green[4])

These classes implement the actual objects that move or the user can interact with. “GameObject” is the superclass to them all and stores basic necessary information about them all like their texture on screen. They range from more complex classes such as “FireTruck” to simple indicator classes such as “StatBar”. These classes are vital for providing most of the features specified by the requirements, such as the ability to heal (UR_FIRETRUCKS_REPAIR) and replenish water (UR_FIRETRUCKS_REPLENISH) when near the FireStation. Even simple classes such as “StatBar” provide a simple solution to the requirement to display indicators of health and water at all times. (SFR_WATER_SUPPLY_BAR).

UI/ Setup, (including “FireTruckSelectionScene”, “OptionsWindow”,”HUD”, purple[4])

These classes make up most of the UI the game uses. They provide necessary features which the game needs in order to run and meet it’s requirements, such as how “FireTruckSelectionScene” is used to define which truck the user will be playing as which is needed to meet the requirement (SFR_FIRETRUCK_STATS)

Our concrete architecture takes influence and the basics from the abstract architecture we designed in assessment 1 but completes it to contain the full details needed for our one particular implementation. This means that much of the logical ideas from the abstract[5] have been carried forward with the details needed to create a libGDX based system that runs the game we were tasked to do.

Examples of this include how we took the main idea of having all on-screen objects that need particular properties such as movement or player detection and having them all inherit

from a particular superclass. This was seen with the “Entity” class in the abstract [5] and while this class exists in the concrete the superclass “GameObject” [1] was created which better fit the idea.

One major change between the two is seen by the migration of much of the logic of the game from the “Kroy” class to the “GameScreen” class. We have done this as in our implementation, the “Kroy” class manages the game as a whole rather than just one instance of the game. This includes storing attributes like “highScore” and making sure that there is only one instance of the game being run at once.

Another change we have made from our abstract is the removal of the “Interactivity” interface. This interface was originally designed to add features such as “isPlayerInRange()” to the classes that need it (ie FireStation, Fortress and UFO) but without having to create a whole abstract class for them. However, with more thought, we decided that this was unnecessary as it was simpler to put the features in the “Entity” class than it was to create an entirely new interface just for a few simple methods.

One change made regarding the runtime of the system is how in the concrete architecture there is no fixed order to how entities get processed [2] compared to the abstract in which it was specified as 1, update movement 2, check interactions and then 3, check collisions.[5] The reason for this change was to move away from having too much logic being directly performed in a central class. The way the implementation turned out, we decided it would be best to instead let each individual entity deal with its own collisions and interactions rather than all at once.

References,

[1] Static UML Model
[2] Runtime Flowchart
[3] Requirements
[4] Architecture Justification
[5] Arch1.pdf
[6] StarUML
[7] Draw.io