Module	Engineering1 (Eng1) - COM00019
Assessment Title	Assessment 2, Cohort 2
Team	Dragonite (Team 21)
Members	Omar Omar, Rhianna Edwards, Okan Deniz, Craig Smith, Omar Galvao Da Silva, Joel Wallis
Deliverable	Architecture

Architecture - Part A

Language used to describe the architecture and tool used to create it

When deciding on our architecture, we chose an object-oriented design (see Part B for the justification for this decision). As it's often used with OOP, we used UML diagrams [1] [2] to graphically document the architecture. These provide:

- a better understanding of the system for team members through the use of visuals,
- a recognisable methodology for the other teams to follow in Assessment 2, and
- the ability to indicate where reusability of code has been implemented.

LucidChart was used to create our diagrams as it had good reviews, plentiful libraries, a free basic subscription, the ability to share files and was easy to learn. We did also consider Gliffy but research [3] gave LucidChart the edge with reviews and features.

Abstract representation of the architecture

Our abstract architecture consists of two views - one behavioural and one structural [4] - to give a broader overview of the solution. Further explanations and justifications for these views are given in Part B.

Behavioural View - a UML activity diagram (flowchart) was used to create a blueprint for the gameplay process. (See Chart 1)

Structural View - a basic class relationship diagram [5] was used to map relationships and show aggregations and inheritance between classes. (See Chart 2)

Concrete representation of the architecture

The basic UML class diagram created during the abstract phase has been expanded on and now fully describes each class. (See Chart 3)

Chart 1 - Abstract Behavioural View

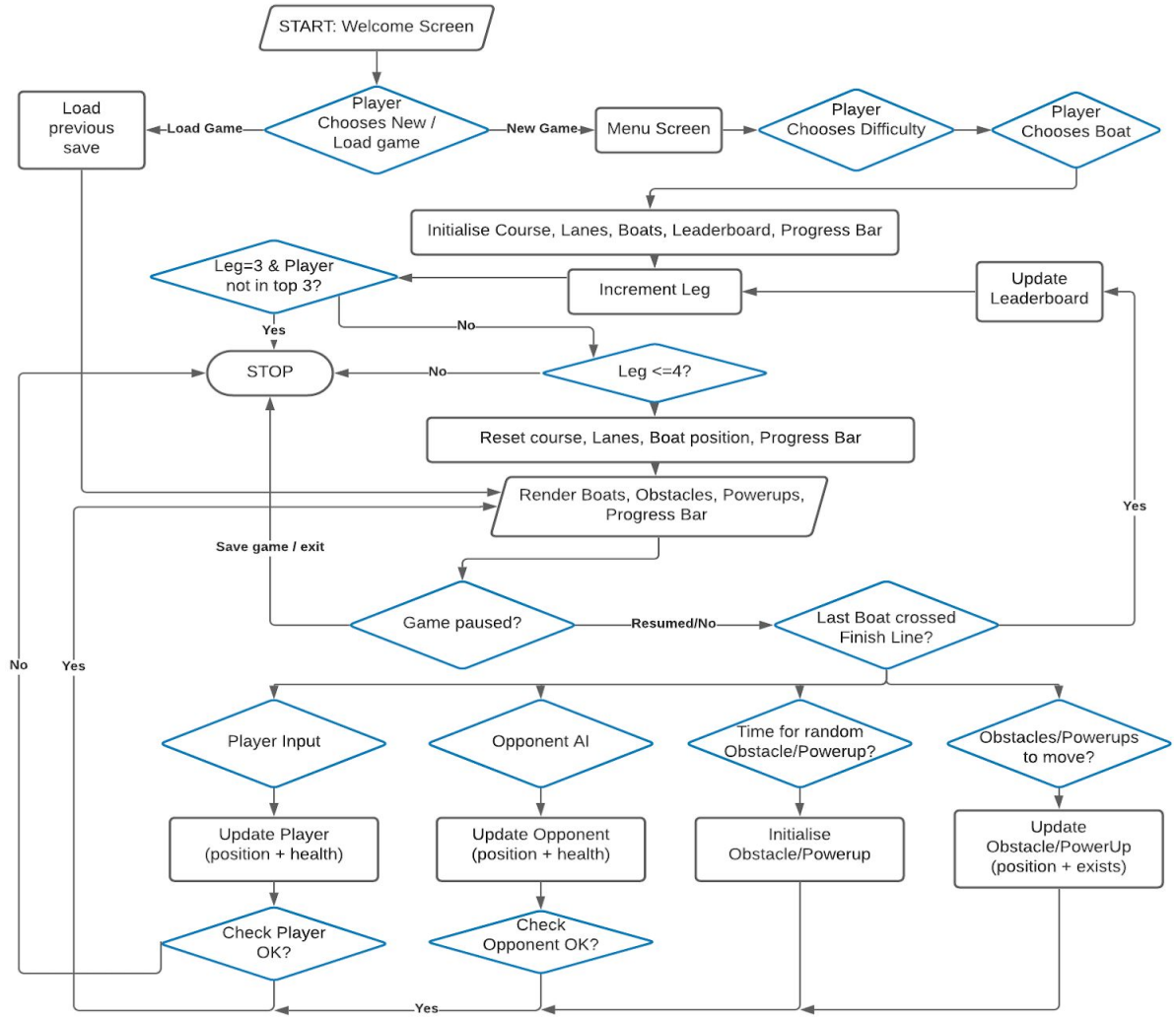


Chart 2 - Abstract Structural View

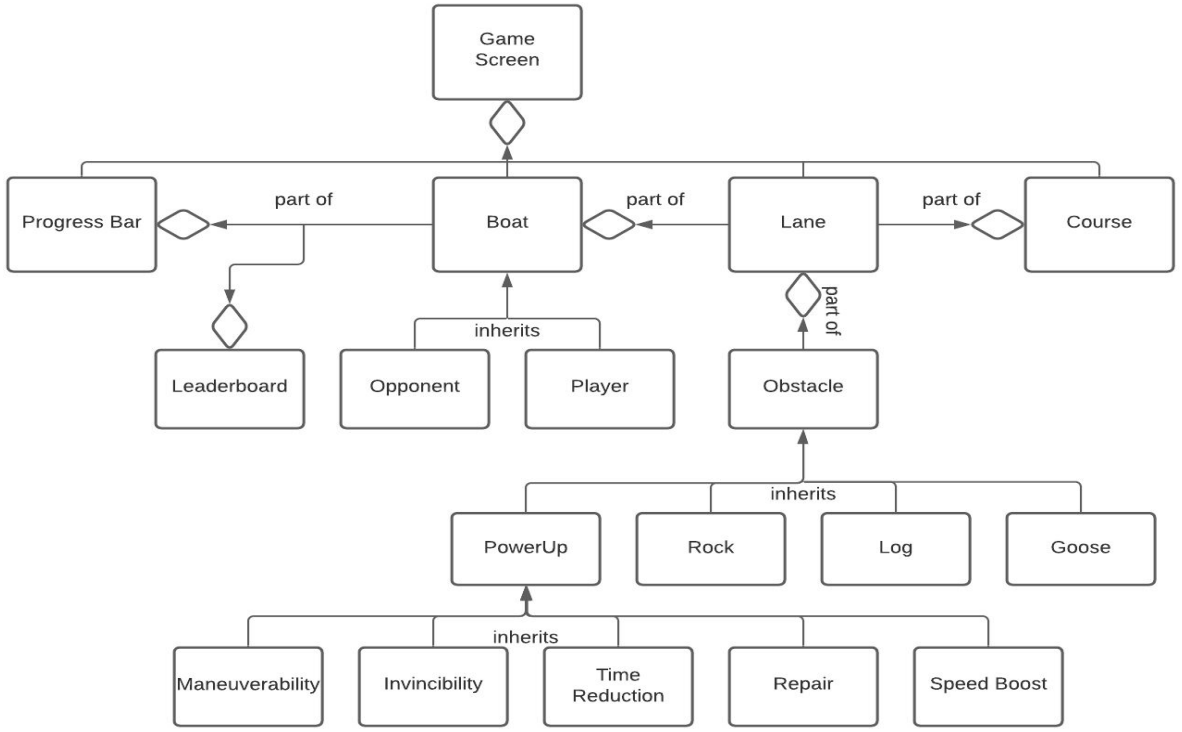
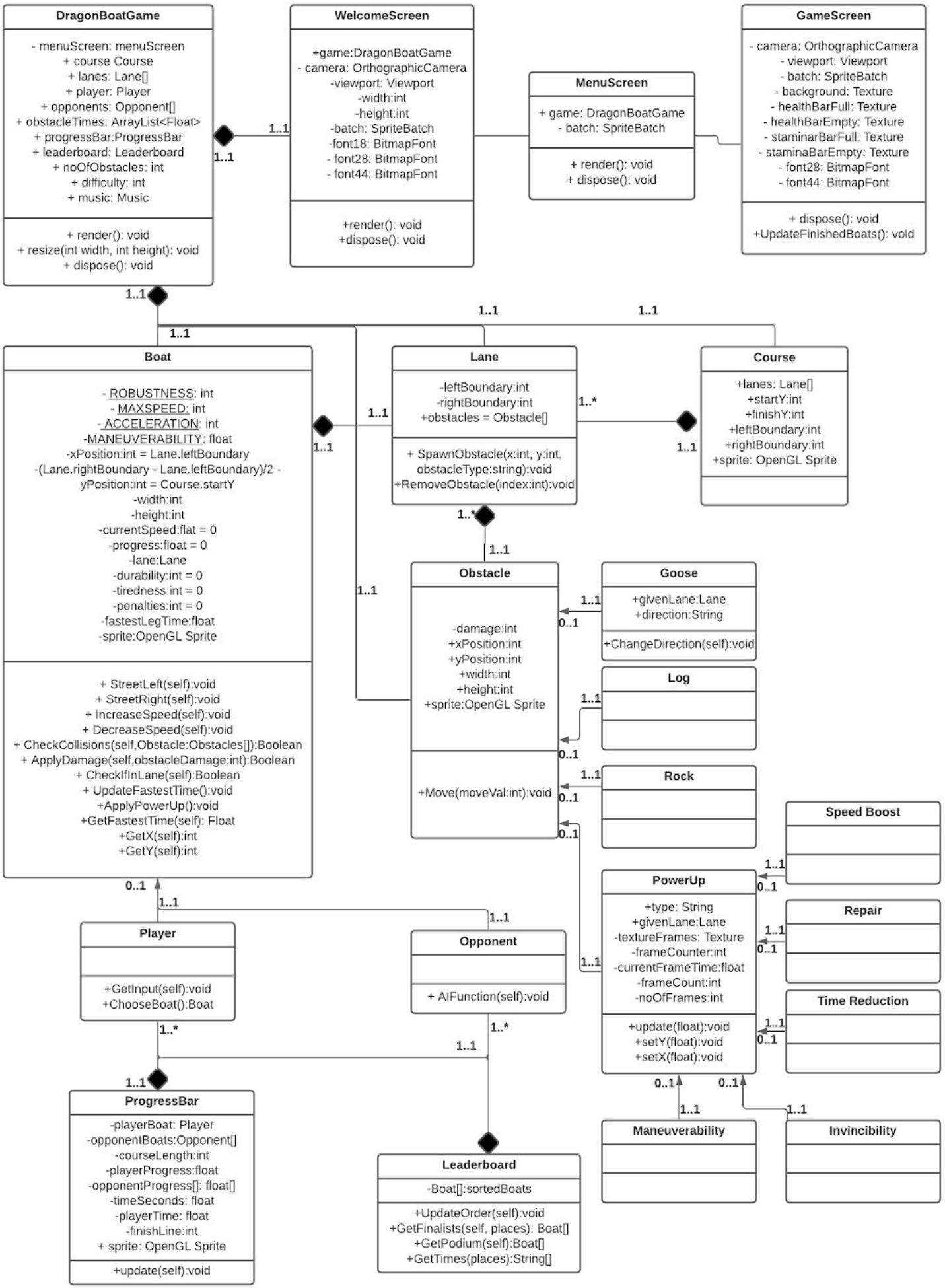


Chart 3 - Concrete Architecture



Architecture - Part B

Systematic justification for the abstract and the concrete architecture

Key decisions that shaped the architecture

OOP v ECS

In the requirements phase, we identified all elements needed to create the game. Based on these, we decided that the game leant itself to Object Oriented Programming (OOP) as our real-world objects (eg, Boats) have both data and behaviours. We also saw the opportunity to implement inheritance (eg, Player inherits from Boat) and polymorphism (eg, Goose can be treated as its parent class, Obstacle), both of which lend themselves to code reusability.

We decided to use the OOP architecture style [6] [11] rather than the other notable architecture for game development, entity-component systems (ECS) [7] [8] due to:

The small scope of the project:

- The bigger the project, the more its architecture suits ECS. As this is a small project, with few classes that interact, an OOP hierarchical approach is more appropriate.

The inherent hierarchical structure of the project:

- Our classes would be designed to maximise the use of inheritance.
- The game's main classes inherit from parent classes, Boat or Obstacle, because they only vary from their parent class by a few methods or attributes.
- Were we to use ECS, we'd end up with (1) a plethora of Component classes that are shared by what are instead child classes of Boat or Obstacle, and (2) a series of Component classes that each belong to only one entity. This results in a complicated replica of the OOP design, just with more Java classes and repeated code.

The experience of the team:

- All team members have experience of Java and OOP, making it the instinctive choice. Most do not have experience with ECS making this the riskier option.

Choosing sequential over concurrent programming

An issue common to many video games is how to deal with concurrent activities [9], eg, the simultaneous movement of 7 boats and numerous obstacles. The use of threading was dismissed due to possible synchronisation issues and lack of team experience with the concept. Instead it was decided that game processes would be performed sequentially whilst leaving the rendering of sprites until last, to give the impression of concurrency.

Design of abstract architecture

Our abstract architecture consists of two models of the game based on specific details identified during the requirements phase.

- These behavioural and structural views give a high-level outline of our solution.
- They give team members a better understanding of the system as specified below.
- They are complete, having been mapped to requirements in the table below.
- They allow individual programming tasks to be (1) identified for planning purposes and (2) implemented independently to save overall project time.

Behavioural view: a UML activity diagram (flowchart) was used to create a blueprint for the gameplay process, detailing the key steps and decisions. It shows how the player will make their way through the game, and how the main components, such as the Boats and

Obstacles, will interact and come together as a whole. It facilitated the production of the structural view and will help ensure game prototypes flow as expected.

Structural view: a basic UML class relationship diagram was created using information from the activity diagram. All required classes were identified from the activity diagram, and then any

aggregations and inheritance relationships between them were mapped. Deconstructing the activities into independent classes meant different team members could work on each one in isolation, and then the game screen would link them to interact with one another.

Progression from abstract to concrete architecture

The final concrete architecture builds from the abstract. The latter grew incrementally as we updated it with more details during both the planning and implementation phases:

- We expanded on the basic UML class relationship diagram previously created.
- As well as relationships, each class is now complete with attributes and methods.
- These further details were incorporated: (1) based on the original decision to use OOP, (2) using specific information from the requirements (eg, a Boat needed attributes to include robustness, current speed etc), (3) from the abstract activity diagram (eg, an opponent needed an AI method), and (4) based on trial and error during coding, for example we missed the Game Screen class originally.

The resulting concrete architecture adds to the commented code to create our system documentation that will be useful for another team in Assessment 2 or future development.

Relate the concrete architecture to the requirements: each class in the final concrete architecture can be traced back to how it satisfies the original requirements as follows:

CLASS	**JUSTIFICATION & REQUIREMENT ID (in brackets)**
Dragon Boat Game	Class implemented to describe the game's structure, ie, which menus to show on screen, and to implement other systems, like the background music. (FR_MUSIC)
Welcome Screen	Class implemented for the user to choose a new game or load a previous save as well as decide on the difficulty (FR_PAUSE_SCREEN) (FR_GAME_MODES)
Menu Screen	Class implemented to allow the user to assign a specific Boat as the Player. (FR_TITLE_SCREEN)
Game Screen	Class implemented to describe the gameflow, ie, the core game loop. (UR_DIFFICULTY, FR_OBSTACLE_INCREASE)
Course	Class implemented to pass Lanes into the Game Screen. (FR_COURSE_BOUNDARIES)
Lane	Class implemented to connect Obstacles and a Boat to their corresponding Lane. (FR_OBSTACLES)
Boat	Parent class implemented to describe all the attributes and methods shared between the Player and Opponent/PowerUp classes through inheritance. (UR_BOATS, UR_MIN_BOATS, FR_UNIQUE_BOATS, FR_TIREDNESS, FR_ANIMATIONS, FR_COLLISION, FR_PENALTY, FR_LANE_WAR) (FR_UNIQUE_POWER_UPS)
Player	Child class implemented to allow a Boat to respond to player input methods. (UR_CONTROLS, FR_UI, FR_SPEED_CONTROL, NFR_RESPONSIVE, NFR_INFORMATION_TIME)
Opponent	Child class implemented to allow a Boat to be controlled by an AI method. (FR_AI)
Obstacle	Parent class implemented to describe all the attributes and methods shared between the Goose and Log classes through inheritance. (FR_COLLISION)
Goose	Child class implemented so it would override the Move() method to create a more complex movement pattern, using the ChangeDirection() method.
Log,Rock	Child class implemented so it could be distinguished from Goose and future Obstacles.
Progress Bar	Class implemented to illustrate the Boats' progress in the race, and the time it takes the Player to complete the leg. (FR_TIMER, FR_FINISH)
Leaderboard	Class implemented to store each Boat's fastest leg times, and from these select the finalists. (UR_FINAL_PLACE, UR_FINAL_RACE, UR_LOSS)
PowerUp: Invincibility, Repair, TimeReduction, SpeedBoost, Maneuverability	PowerUp class implemented to generate PowerUp texture frames and graphics. Each individual PowerUp such as Repair and Invincibility are child classes implemented to create unique instances. (FR_UNIQUE_POWER_UPS)

Bibliography

1. *What is Unified Modeling Language (UML)?*, Visual Paradigm. Accessed on: October 23, 2020. [Online]. Available: https://www.visual-paradigm.com/guide/uml-unifiedmodeling-language/what-is-uml/
2. *What is Unified Modeling Language*, Lucidchart. Accessed on: October 24, 2020. [Online]. Available: https://www.lucidchart.com/pages/what-is-UML-unified-modelinglanguage
3. *Lucidchart vs Gliffy Diagram*, Capterra. Accessed on: October 23, 2020. [Online]. Available: https://www.capterra.com/diagram-software/compare/145714-146136/Gliffy-vs-Lucidchart
4. *UML Types*, TutorialRide.com. Accessed on: October 24, 2020. [Online]. Available: https://www.tutorialride.com/software-architecture-and-design/umltypes.htm
5. *What is Class Diagram?*, Visual Paradigm. Accessed on: October 23, 2020. [Online]. Available: https://www.visual-paradigm.com/guide/uml-unified-modelinglanguage/what-is-class-diagram/
6. *Object Oriented Architecture*, TutorialRide.com. Accessed on: October 23, 2020. [Online]. Available: https://www.tutorialride.com/software-architecture-anddesign/object-oriented-architecture.htm
7. *OOP vs. ECS*, Kata Learns To Code. January 26, 2016. Accessed on: October 30, 2020. [Online]. Available: https://katatunix.wordpress.com/2016/01/26/oop-vs-ecs/
8. *Videogame Architecture*, JRL. December 18, 2018. Accessed on: October 30, 2020. [Online]. Available: https://jarlowrey.com/blog/game-architecture
9. *Space Invaders*, Flylib.com. Accessed on: October 29, 2020. [Online]. Available: https://flylib.com/books/en/2.752.1.89/1/
10. N. Nordmark, *Software Architecture and the Creative Process in Game Development*, Norwegian University of Science and Technology. June 2012. Accessed on: October 22, 2020. [Online]. Available: https://core.ac.uk/download/pdf/30863405.pdf
11. *I. Sommerville, Software Engineering*, 8th Ed., Addison-Wesley, 2007, Chapter 14