Module Code
COM00008I

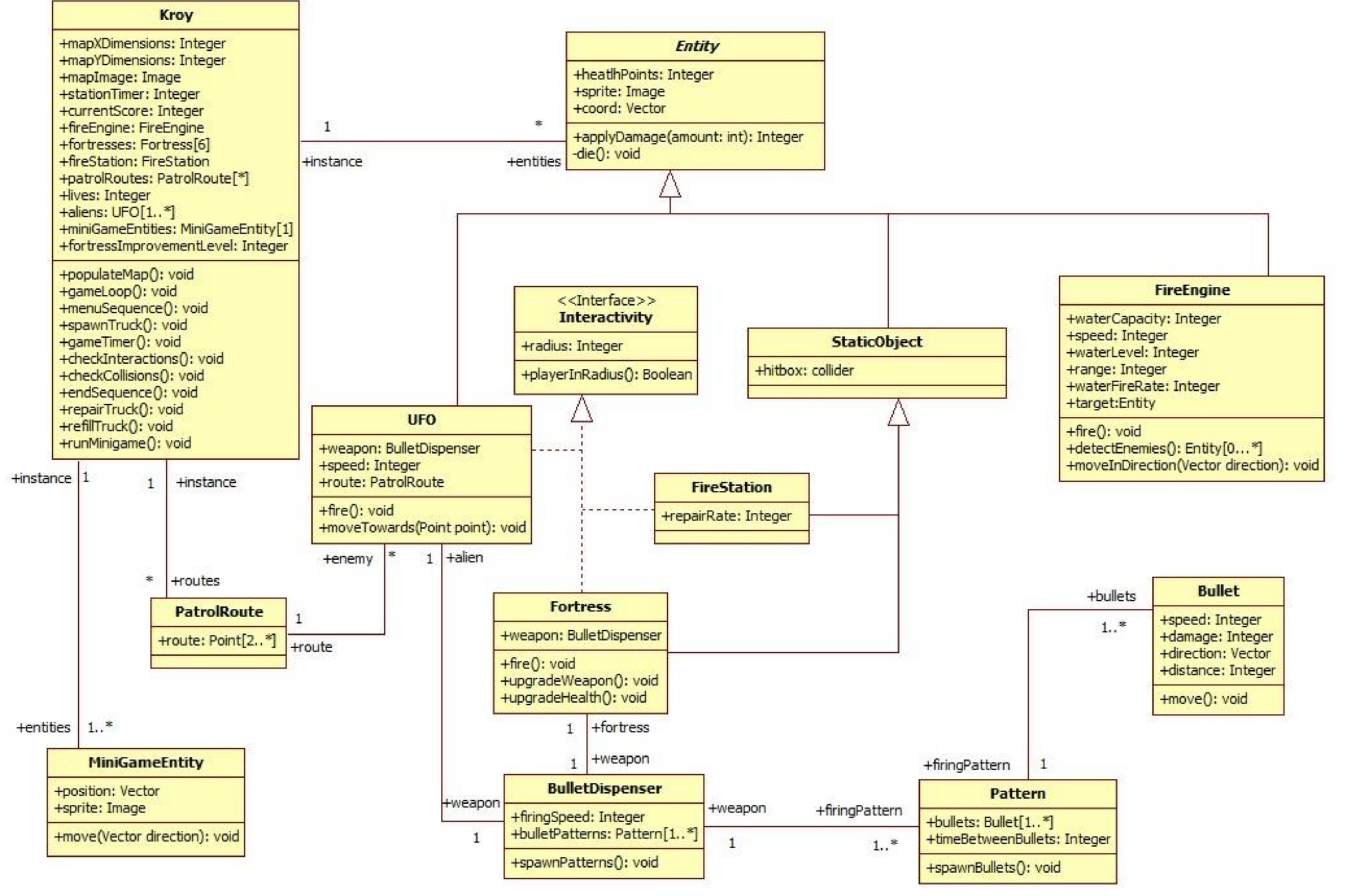
BSc, BEng and MEng Degree Examinations 2019–20

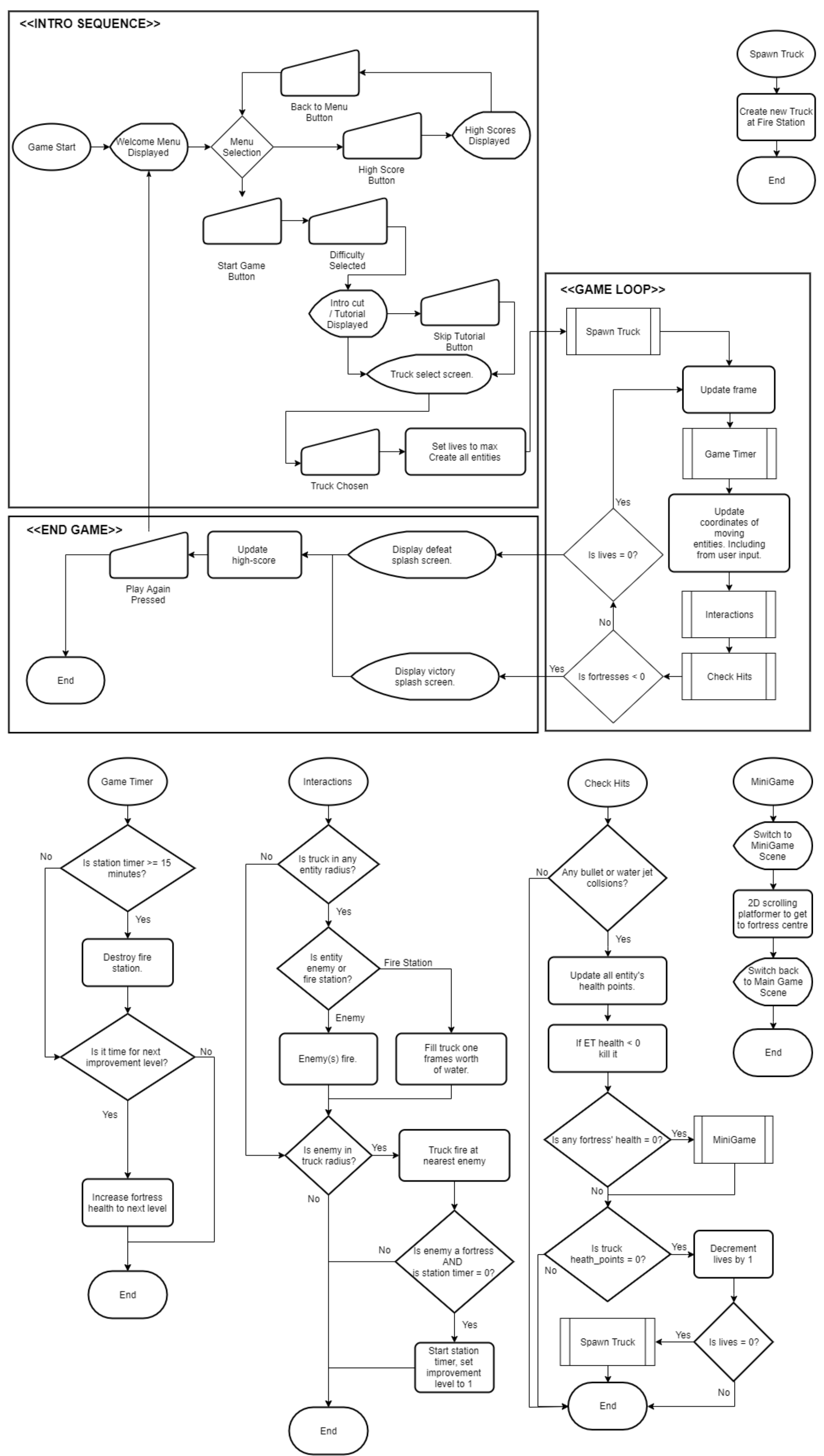
DEPARTMENT OF COMPUTER SCIENCE

Software Engineering Project (SEPR)

Open Group Assessment

Module	**Software Engineering Project (SEPR)**
Year	**2019/20**
Assessment	**1**
Team	**The Dicy Cat**
Members	**Michele Imbriani** **Daniel Yates** **Luke Taylor** **Isaac Albiston** **Martha Cartwright** **Riju De** **Sean Corrigan**
Deliverable	**Software Architecture**


Kroy
+mapXDimensions: Integer
+mapYDimensions: Integer
+mapImage: Image
+stationTimer: Integer
+currentScore: Integer
+fireEngine: FireEngine
+fortresses: Fortress[6]
+fireStation: FireStation
+patrolRoutes: PatrolRoute[*]
+lives: Integer
+aliens: UFO[1..*]
+miniGameEntities: MiniGameEntity[1]
+fortressImprovementLevel: Integer
+populateMap(): void
+gameLoop(): void
+menuSequence(): void
+spawnTruck(): void
+gameTimer(): void
+checkInteractions(): void
+checkCollisions(): void
+endSequence(): void
+repairTruck(): void
+refillTruck(): void
+runMinigame(): void
1
+instance
*
+entities
Entity
+heatlhPoints: Integer
+sprite: Image
+coord: Vector
+applyDamage(amount: int): Integer
-die(): void
<<Interface>>
Interactivity
+radius: Integer
+playerInRadius(): Boolean
StaticObject
+hitbox: collider
FireEngine
+waterCapacity: Integer
+speed: Integer
+waterLevel: Integer
+range: Integer
+waterFireRate: Integer
+target:Entity
+fire(): void
+detectEnemies(): Entity[0...*]
+moveInDirection(Vector direction): void
UFO
+weapon: BulletDispenser
+speed: Integer
+route: PatrolRoute
+fire(): void
+moveTowards(Point point): void
FireStation
+repairRate: Integer
+instance
1
1
+instance
*
+routes
PatrolRoute
+route: Point[2..*]
1
+route
+enemy
*
1
+alien
Fortress
+weapon: BulletDispenser
+fire(): void
+upgradeWeapon(): void
+upgradeHealth(): void
1
+fortress
1
+weapon
BulletDispenser
+firingSpeed: Integer
+bulletPatterns: Pattern[1..*]
+spawnPatterns(): void
+weapon
1
+weapon
1
+firingPattern
1..*
Pattern
+bullets: Bullet[1..*]
+timeBetweenBullets: Integer
+spawnBullets(): void
+firingPattern
1
+bullets
1..*
Bullet
+speed: Integer
+damage: Integer
+direction: Vector
+distance: Integer
+move(): void
+entities
1..*
MiniGameEntity
+position: Vector
+sprite: Image
+move(Vector direction): void



<<INTRO SEQUENCE>>
Game Start
Welcome Menu Displayed
Menu Selection
Back to Menu Button
High Score Button
High Scores Displayed
Start Game Button
Difficulty Selected
Intro cut / Tutorial Displayed
Skip Tutorial Button
Truck select screen.
Truck Chosen
Set lives to max Create all entities
<<GAME LOOP>>
Spawn Truck
Update frame
Game Timer
Update coordinates of moving entities. Including from user input.
Interactions
Check Hits
Is lives = 0?
Yes
No
Is fortresses < 0
Yes
<<END GAME>>
Display defeat splash screen.
Display victory splash screen.
Update high-score
Play Again Pressed
End
Spawn Truck
Create new Truck at Fire Station
End
Game Timer
Is station timer >= 15 minutes?
No
Yes
Destroy fire station.
Is it time for next improvement level?
No
Yes
Increase fortress health to next level
End
Interactions
Is truck in any entity radius?
No
Yes
Is entity enemy or fire station?
Fire Station
Enemy
Enemy(s) fire.
Fill truck one frames worth of water.
Is enemy in truck radius?
Yes
No
Truck fire at nearest enemy
Is enemy a fortress AND is station timer = 0?
No
Yes
Start station timer, set improvement level to 1
End
Check Hits
Any bullet or water jet collsions?
No
Yes
Update all entity's health points.
If ET health < 0 kill it
Is any fortress' health = 0?
Yes
MiniGame
No
Is truck heath_points = 0?
Yes
No
Decrement lives by 1
Is lives = 0?
Yes
No
Spawn Truck
End
MiniGame
Switch to MiniGame Scene
2D scrolling platformer to get to fortress centre
Switch back to Main Game Scene
End

We have chosen to represent our architecture in two diagrams which describe our game from a runtime and static point of view and collectively describe a structure we could use to implement our program in an orderly manner that will meet our requirements.
For our runtime model, we chose to utilise ‘draw.io’, a free diagram creation program that has dedicated flowchart tools. The reasoning for this was that it seamlessly integrates with Google Drive, which not only allowed us to collaborate on the same work at the same time; but also it streamlines adding it to our website, which is created using ‘Google Sites’ and links with Google Drive. We used draw.io to create a flowchart (see above) to model the processes and states the game will cycle through and while we tried to stick to standards of ISO 5807, some symbols used we could not find any official documentation for and so a full list of symbols is provided below.

Symbol	Meaning
	Decision - diverts the flowchart depending on the question in the symbol
	User input - anything inputted by the user, in most cases represents waiting for a user to press an on-screen button
	Start - symbol to start a subroutine or main start.
	Stop - terminates either the main program or a subroutine
	Subroutine - this symbol is used to call a subroutine found elsewhere in the program
	Process - this symbol is used to process any information or call any methods not specified in the flowchart
	Display - this is used to show a major change to a screen not inside the game loop, mostly to show the end screen or menu screen
	Arrow - this is used to show the transitions between one symbol to the next

Similarly, our game is to be implemented in the Java language which is Object-Oriented therefore we used the Unified Modelling Language (UML) version 2, which provides a convenient standard to model and design the classes and methods of the program. Our static model provides a useful insight into one possible structure of how the game could be implemented to make sure it meets the criteria set out in the requirements. To create our model, we used ‘StarUML’, as it provided far better tools for modelling the UML syntax than other alternatives as well as being cross-platform to allow for group members with different OS’s to work on the model. Also through extensions, it gives us the option for code generation to save us setting up the basic framework of the program if we wished.

Our runtime model is a flowchart that sets out the main states the application can be in which will provide us with a guide to make sure implementation stages are carried out properly to meet the requirements. This is described by splitting the model into the “Intro Sequence” and the “Game Loop” sections which allows us to focus on the logical and functional aspects of the game while not neglecting the steps needed to provide a good user experience.
The intro sequence section explains how we want our menus to function and the important information they need to display, such as having a high score button and a start game button, however in this stage many of the processes are a simple set of sequential steps executed based on user input. Some important features determined at this stage are the options to choose which type of engine the user wants to play and the difficulty for the game. These features allow for a much more enjoyable and re-playable experience as different trucks can provide different in-game experiences and the ability to change the difficulty will allow the game to cater to a wider player base with a wider range of gameplay experience. Similarly, the possibility of implementing accessibility features in future versions could also be added to this section, which is a feature also discussed in the requirements.

The game loop section of the model illustrates the iterative set of steps and decisions the program needs to make based on the current state of the game. This is done by breaking the loop into 4 main subroutines, each representing a key process that must be carried out during each iteration, such as “Interactions” which checks to see how close entities to each other and if any weapons need to be fired or the engine needs to start healing and refilling. Each iteration in the game loop more or less represents the processing needed to be done for each frame of our game. During each loop, it also deals with the end of game scenarios by checking if either the fire truck has lost all its lives or if all the fortresses have been destroyed. When this situation occurs the loop ends and enters the “end game” section of the diagram. This model, especially the game loop, is essential as whilst our static model demonstrates the structure of our program, it doesn’t show how the game actually runs nor how it plays.

However, what the static model does provide is a clear structure and a more in-depth look at one possible implementation of the program. This model is useful as it sets out the classes the program will need allowing for a simpler implementation in the future, especially when combined with the table below which provides an additional explanation of the desired functionality of each class.

Class	**Subclass of...**	**Explanation**
Kroy	N/A	This is the master class which will control functionality such as the game loop and all the subroutines marked out in the runtime model.
Entity (abstract)	N/A	Entity is the superclass for the majority of our other classes. It defines fields that all generated classes have in common, such as taking damage and the sprite the object should be. It is abstract as no copies of it should be initialised.
StaticObject	*Entity*	While not included in the static model, this will be

		utilised for objects such as trees, buildings and rivers that the fire engine cannot traverse. It also provides the logic for the hitboxes which interactive objects (such as the FireStation and Fortress) will inherit from.
<<Interface>> Interactivity	N/A	This interface allows us to pass the FireEngineDetect methods to the classes that need them.
MiniGameIEntity	N/A	Entity used in the minigame (platformer)
FireStation	InteractiveObject	This allows us to make a distinction between the two main static, interactive objects, since the fire station will have a repair rate, unlike the fortress.
FireEngine	Entity	FireEngine is the blueprint for the multiple engines in the game. It allows us to specify different stats for each engine, such as differing water capacities, flow rates and speeds in order to meet the requirements of making different engines.
Fortress	InteractiveObject	The fortress class will allow us to populate the map with multiple distinct fortress objects, which each have a unique bullet dispenser. They inherit a radius from the interactivity interface as well as a custom amount of health points from the entity class. Both of these characteristics will allow the fortresses to have customisable stats as set out in the requirements.
UFO	Entity	The UFO class represents the patrolling ET's on the map. They will patrol a given patrol route at its own speed. When the truck enters its radius it will fire its weapon.
Patrol	N/A	Stores a set of waypoints.
Bullet	N/A	Used to create an individual bullet object, with unique speed, direction and damage inflicted upon impact with the fire engine.
Pattern	N/A	Pattern specifies an array of bullets and the time in between spawning them. This allows for the assigning of unique patterns both each firing entity, but also to each specific attack.
BulletDispenser	N/A	Each firing entity will have a bullet dispenser, which specifies the location to spawn bullets from, as well as an array of patterns that it will fire out.