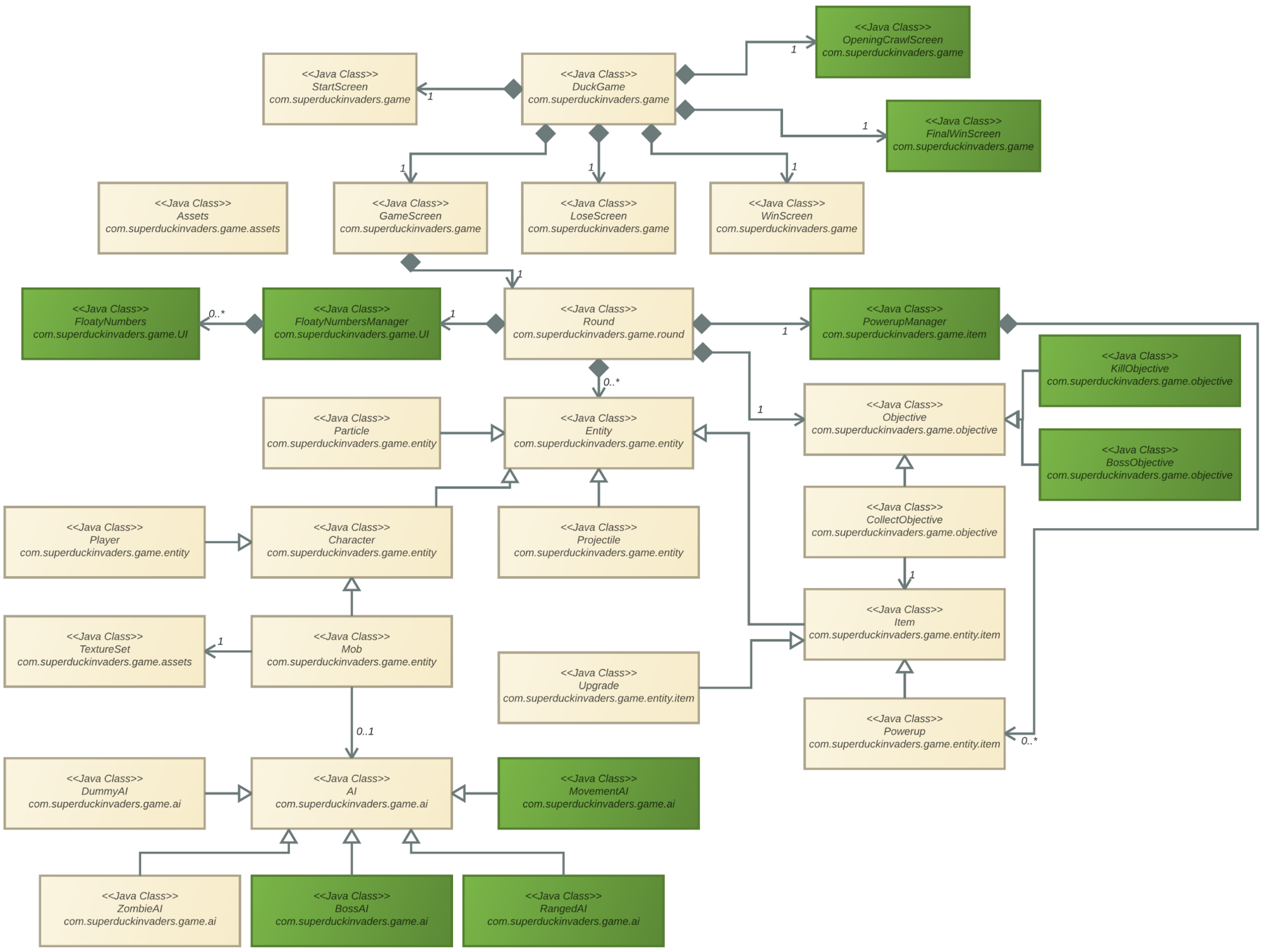
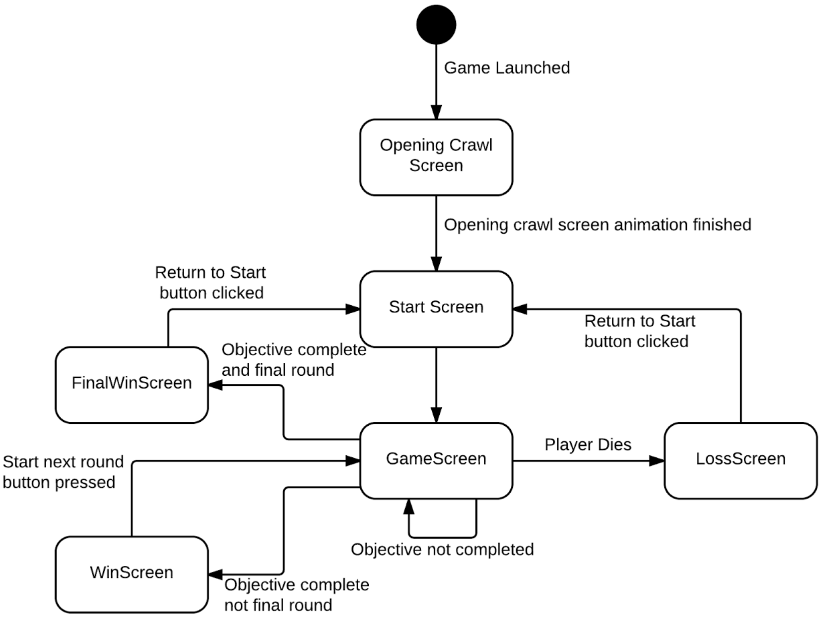
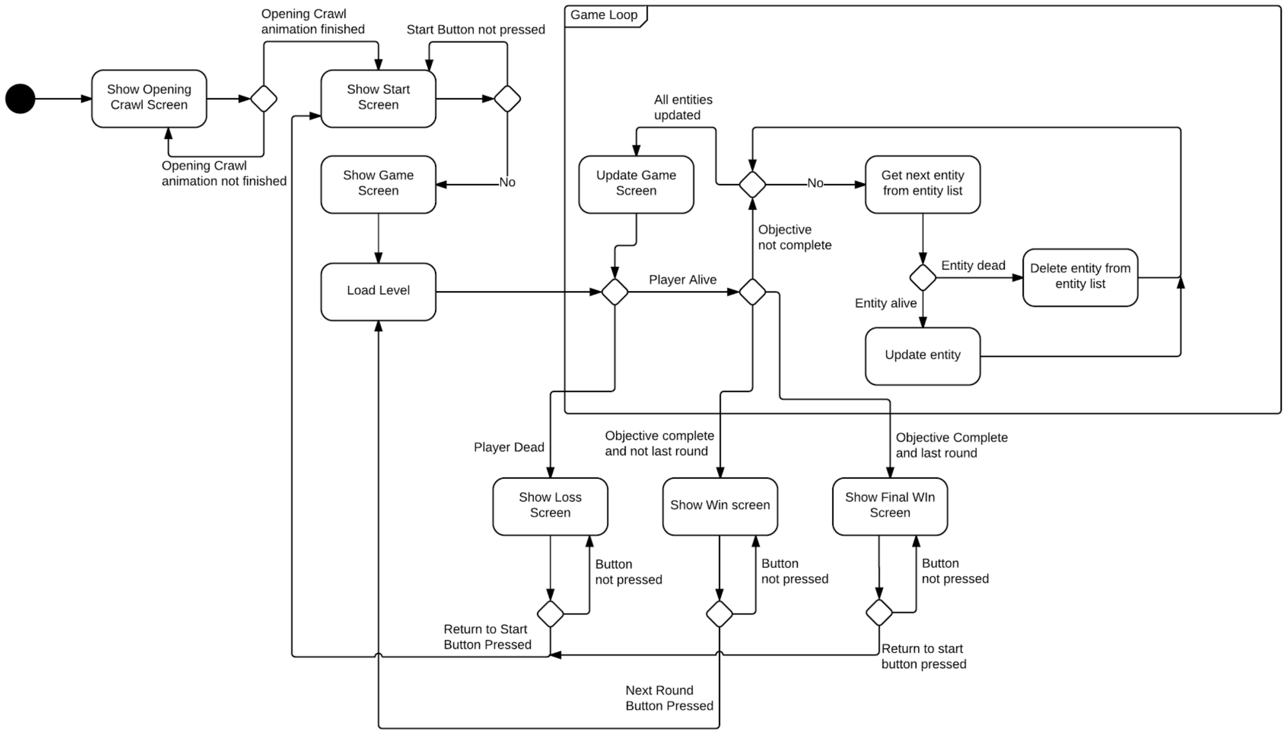
Class Diagram


<<Java Class>> OpeningCrawlScreen com.superduckinvaders.game
<<Java Class>> StartScreen com.superduckinvaders.game
<<Java Class>> DuckGame com.superduckinvaders.game
<<Java Class>> FinalWinScreen com.superduckinvaders.game
<<Java Class>> Assets com.superduckinvaders.game.assets
<<Java Class>> GameScreen com.superduckinvaders.game
<<Java Class>> LoseScreen com.superduckinvaders.game
<<Java Class>> WinScreen com.superduckinvaders.game
<<Java Class>> FloatyNumbers com.superduckinvaders.game.UI
<<Java Class>> FloatyNumbersManager com.superduckinvaders.game.UI
<<Java Class>> Round com.superduckinvaders.game.round
<<Java Class>> PowerupManager com.superduckinvaders.game.item
<<Java Class>> KillObjective com.superduckinvaders.game.objective
<<Java Class>> Particle com.superduckinvaders.game.entity
<<Java Class>> Entity com.superduckinvaders.game.entity
<<Java Class>> Objective com.superduckinvaders.game.objective
<<Java Class>> BossObjective com.superduckinvaders.game.objective
<<Java Class>> Player com.superduckinvaders.game.entity
<<Java Class>> Character com.superduckinvaders.game.entity
<<Java Class>> Projectile com.superduckinvaders.game.entity
<<Java Class>> CollectObjective com.superduckinvaders.game.objective
<<Java Class>> TextureSet com.superduckinvaders.game.assets
<<Java Class>> Mob com.superduckinvaders.game.entity
<<Java Class>> Item com.superduckinvaders.game.entity.item
<<Java Class>> Upgrade com.superduckinvaders.game.entity.item
<<Java Class>> Powerup com.superduckinvaders.game.entity.item
<<Java Class>> DummyAI com.superduckinvaders.game.ai
<<Java Class>> AI com.superduckinvaders.game.ai
<<Java Class>> MovementAI com.superduckinvaders.game.ai
<<Java Class>> ZombieAI com.superduckinvaders.game.ai
<<Java Class>> BossAI com.superduckinvaders.game.ai
<<Java Class>> RangedAI com.superduckinvaders.game.ai
1
0..*
0..1

State Diagram


Game Launched
Opening Crawl Screen
Opening crawl screen animation finished
Return to Start button clicked
Start Screen
Return to Start button clicked
FinalWinScreen
Objective complete and final round
GameScreen
Player Dies
LossScreen
Start next round button pressed
WinScreen
Objective not completed
Objective complete not final round

Activity Diagram


Show Opening Crawl Screen
Opening Crawl animation finished
Opening Crawl animation not finished
Show Start Screen
Start Button not pressed
No
Show Game Screen
Load Level
Game Loop
All entities updated
Update Game Screen
No
Get next entity from entity list
Objective not complete
Player Alive
Entity dead
Delete entity from entity list
Entity alive
Update entity
Player Dead
Objective complete and not last round
Objective Complete and last round
Show Loss Screen
Show Win screen
Show Final WIn Screen
Button not pressed
Button not pressed
Button not pressed
Return to Start Button Pressed
Return to start button pressed
Next Round Button Pressed

The above UML 2.0[1] diagrams illustrate the concrete architecture of the initial implementation of our game. UML 2.0 was chosen as it is a well-defined standard way of expressing software architectures, and as such using it likely negates the need for any future maintainers of our code to learn another modelling language in order to understand the diagrams. The class diagram shows how the concrete Java classes in our game are related, including how they inherit from each other (*"is-a"* relationships) and where classes are composed of other classes (*"has-a"* relationships). The state diagram shows a representation of the different screens a user could be presented with in the game, and what is required to transition between them. The activity diagram gives a high-level representation of the operation of our game, and what happens when the victory/loss conditions are met.

All diagrams were created using an online chart drawing tool called Lucidchart[2]. Lucidchart supports a large range of UML symbols, as well as other notations. Note that the omission of end symbols from both the state and activity diagrams was due to there not being an exit button implemented in the game. A user can still exit the game at any time by closing the game window, however we did not indicate this on the diagrams as it is not strictly part of the system.

Changes for Assessment 3

The changes in the concrete architecture are all highlighted in green in some way. The introduction of the PowerupManager (CA6) class handles multiple powerups at the same time instead of just having one. The introduction of a new objective, KillObjective which inherits from Objective in the same way CollectObjective does. The other objective type is BossObjective (CA10) which also inherits from Objective. The introduction of FloatyNumbers (CA1) and FloatyNumbersManager are used to keep track of the numbers such as score and damage which will be rendered when the player damages or gets damaged or scores points FloatyNumbersManager manages this for the Round class. OpeningCrawlScreen (CA9) is a class whose sole purpose is to render the opening animation and then move onto the StartScreen when finished, the state diagram has been changed to show this, the activity diagram has been left and just continues from after the OpeningCrawlScreen switches to the StartScreen. The last additions are three new AI classes. RangedAI is for a constant follow and shoot type of enemy, MovementAI (CA3) is for a stop and shoot type of enemy and BossAI (CA10) is the AI responsible for the final boss enemy. FinalWinScreen is used to take the player back to the start screen and display player score. With the addition of rounds being fully implemented in the game the state diagram and activity diagram both required changing to reflect this. Now when you win a round you will be taken to the GameScreen again to play the next round or if all rounds have been completed you will be taken to the FinalWinScreen.

Justification

For the most part, the concrete architecture follows what was set out in the abstract architecture, with several key changes. In the class diagram for the abstract architecture, there are classes for *Tile*, *Spritesheet*, *Sprite* and *Font* which are not present in the concrete architecture. This is because the abstract architecture was created with implementing our own game engine in mind. However, after consideration we decided not to implement our own game engine and instead use an existing one that has map and sprite rendering built in, thus eliminating the need for these classes. The class named *Game* in the abstract architecture was changed to *DuckGame* as there was already a class called *Game* in the game engine library (which *DuckGame* inherits). *DuckGame* is the entry point to the game and contains all of the screen classes. *Display* has been changed to *GameScreen* in order to be consistent with the new classes *StartScreen*, *WinScreen* and *LoseScreen*. These classes represent the possible different screens the user could be displayed with, including screens for when the winning/losing conditions have been achieved (as per requirement G12). *GameScreen* itself is responsible for rendering the actual gameplay part of the game and for maintaining the game loop, which is shown in the activity diagram. The transition between these screens is shown in the state diagram. *Menu* has been removed as the initial implementation only has one map and objective type, so there is no point in having a menu to choose from; however, the game architecture allows for a menu to be added at a later stage.

The *Round* class contains the core logic for the game, maintaining a list of entities as well as storing the current objective and game map. This is where the progress toward the objective is checked and all entities are either updated or deleted each game loop. When the associated objective is complete, the round is over and the player wins (according to requirement G1). Each *Entity* is an object within the game, having a position and optionally a velocity at which it moves. Collision detection also takes place in the *Entity* class. *Entity* is an abstract class, as having just an entity without other properties is meaningless. It is inherited by four other classes: *Character*, *Item*, *Particle* and *Projectile*. The *Character* class adds health, facing direction and functionality for attacking. Both *Player* and *Mob* inherit from *Character*. *Player* represents the player controlled duck in the game, and stores the player's current score (requirement G4) and any possible powerups/upgrades the player may have obtained (requirements G9 and G10). The *Player* class also handles the player's movement and attack via input from the keyboard (requirement C1). The *Mob* class replaces *NPC* in the abstract class diagram. This is because we originally planned to have a shopkeeper character in the game where the player could buy new weapons, but this was not implemented in the initial implementation, so *NPC* was changed to *Mob* to reflect the fact that the only non-player characters in the game were aggressive towards the player. This is also the reason that the *Shop* class from the abstract architecture is not present in the concrete architecture. The *Mob* class represents an enemy that will attack the player, serving as an obstacle between the player and the objective (requirements G7 and G8). Each *Mob* has an associated *AI* which defines its attacking behaviour. The two types of *AI* present in the initial implementation are the *ZombieAI* and *DummyAI*. This decoupling between the *Mob* and its attacking behaviour allows for easy implementation of different attack behaviours in the future without having to modify the *Mob* class. *Projectile* is another class that inherits from *Entity*. It represents a bullet in the game that applies damage upon impact with a *Character*. This could be fired from the player's gun upgrade or potentially by an enemy.

Particle represents a small animation with a limited duration used for graphical effect, e.g. when a *Projectile* hits a wall. *Item* represents an object on the map to be collected by the player. This could be an objective, a health kit, etc. Two special types of items, *Upgrade* and *Powerup* inherit from *Item*. *Upgrade* is an item on the map which when collected grants the player an upgrade, which alters their weapon (requirement G10). Upgrades are permanent. *Powerup* is an item which when collected grants a temporary boost to one of the player's stats (for example double speed for 10 seconds), therefore making it easier to achieve the objective. (requirement G9). Progress towards the objective is stored in the *Round* class.

The objective itself is represented in the code by the *Objective* abstract class. This class contains methods to retrieve and update the current status towards the objective - these are both called in the game loop (as per the activity diagram). The *Objective* class is abstract in order to support different types of objective (requirement G2). The objective that we have implemented in the game is represented by the *CollectObjective* class, which involves collecting an item while avoiding/killing the enemies attacking you. Other types of objective e.g. "kill 100 enemies" could be easily implemented in the future by extending the *Objective* class. The *Assets* class is never instantiated, but instead provides a central location for all of the game assets to be loaded in. This way of loading assets ensures that all required assets are loaded before the game starts, and allows assets to be shared between all classes. *TextureSet* brings together all of the idle and walking textures used by *Character*s and provides a method to get the appropriate texture to use depending on the facing of the character and whether it is moving or not. This prevents the *Assets* class from being cluttered with too many idle/walking textures.

[1] Object Management Group, "Unified Modeling Language: Infrastructure", 2006. [Online]. Available: http://doc.omg.org/formal/2005-07-05.pdf. [Accessed: 12- Feb- 2016].
[2] Lucidchart, "Flow Chart & Diagram Maker - Lucidchart". [Online]. Available: http://www.lucidchart.com/. [Accessed: 10- Feb- 2016].