Architecture Report

Changes are shown in blue text.

Class Diagram

Leading on from the abstract architecture for the game presented in Assessment 1's architecture report, we have made a number of changes as we progressed with the development of our game. Specifically, this saw additions to our class diagram, with attributes and operations added to the majority of classes as we began designing algorithms and discussing what each class would need to be able to do. This architecture document is far more in-depth than the document we inherited; this is partly due to the fact that we implemented the game completely throughout the assessment period and so could faithfully recreate the architecture in the diagram. While names were somewhat inconsistent previously, final names for all classes, included attributes and methods have been provided accurately.

Instead of using https://www.gliffy.com/, we used jsUML2 to create the diagrams to describe our architecture; this is because this is the tool we used in the previous assessment and we are more familiar with its use.

Before we discuss the attributes and operations, we should note the changes that have been made to the structure of the class diagram. The 'Boss Room' class, which was previously a subclass of 'DungeonRoom', has been removed. We decided that there was no real need for a specific class for boss rooms. Because it would still be fundamentally the same as a standard room, it would have one enemy and a number of obstacles associated with it - similar to a normal room (which would likely have more than one enemy instead of just a single one). The functionality for denoting boss rooms was added to our enum 'RoomType'.

Similarly, the 'NonDestructible' subclass of 'obstacle' was removed as was not going to add any additional functionality to the 'Obstacle' class. Furthermore, we have removed the 'Overworld' class, as a result of design changes moving from a 'Legend of Zelda' style overworld to one similar to the 'Super Mario Bros.' series.

Class Breakdown

We did not use the enum gameState which was previously explained here. This enumerable passed all the responsibility of rendering screens into the MuscovyGame class, enlarging the class and failing to separate the screens sufficiently. Instead, we implemented separate classes for each screen; due to the progression of the game, the game state was implied by what screen was on display. We could still implement requirement 10 ("The game must end when every area of the university has been conquered, or, when the user dies.", found at https://github.com/teal-duck/teal-duck/raw/gh-pages/Assessment%202%20swap%20files/req2.pdf), by adding a WinScreen, to be displayed to the user upon completing the game.

We no longer have the class Building. Instead, we will build the level (i.e. the layout of the rooms) in the class LevelGenerator. Each room is generated through the DungeonRoom class, which takes its templates from the DungeonRoomTemplateLoader class. This class will load the templates from a csv file; therefore, the code will be neater. Using dedicated

classes for these jobs is better practice as it sufficiently separates different tasks, therefore making it easier to test and to code reliably.
Building on from above, the 'DungeonRoom' class now has some attributes to allow the player to navigate between them. This is in the form of upDoor, rightDoor, downDoor and leftDoor, which will represent where doors exist in the room (each attribute corresponding to one of the room walls). We will only draw (and allow players through) doors where we have specified the location of one when we generated the building. The functionality for the movement through the levels described here is provided by the EntityManager and GameScreen classes.

The 'PlayerCharacter' class representing the user's in-game character will require a number of attributes in order for the game to be able to monitor various things such as health and what abilities the player has access to. Health will be determined by 'maxHealth', holding the maximum amount of health the player can have, and 'currentHealth', representing the player's current health (as they will take damage over the course of gameplay). We did not implement limits on ammo for the player's ranged attack. To keep track of the player's acquired abilities, we used the enum PlayerShotType and ProjectileType to denote the modifiers on the player's projectile attack. The enum ProjectileDamager indicates whether the projectile instance is damaging to the player or to the enemies. The Action enum lists the actions the player can take, providing a foundation to base the input logic upon.

We did not implement money as a resource. We interpreted requirement 9 (https://github.com/teal-duck/teal-duck/raw/gh-pages/Assessment%202%20swap%20files/req2.pdf) slightly differently; health and bomb pickups were implemented. Health is purely a resource, while bombs are also an ability the player can acquire; however, they are limited. Both of these are an instance of class Item; the particular item in question is decided by enum ItemType.

Player movement now uses the same logic as enemy movement; they both extend the class MoveableEntity. Variation in speed occurs when the player uses their flight ability (their speed increases for a limited period of time), and the friction increases when the player plays the swimming level. Enemies may be static, or they may move randomly or follow the player – the logic for these system is embedded in MoveableEntity, and the type of movement is decided by enemy enum 'MovementType'.

The player character will moved via our movementLogic() function in the PlayerCharacter class, using the Action enum and logic implemented in the MoveableEntity class. AbilitySelect() no longer needs to be included in the architecture; abilities collected by the player will replace their previous special ability. The exception is bombs, which are added to the player's abilities when acquired.

The Enemy class, as discussed before, will also extend the MoveableEntity class. The other attributes of a given instance of an enemy will depend on their enums; ProjectileType and ProjectileDamager work the same as explained before when used in the PlayerCharacter class. AttackType shows whether the enemy is a touch-damage or ranged-damage enemy – therefore eliminating the need for additional 'Melee' and 'Ranged' classes, which have been removed from our architecture. EnemyShotType denotes whether the enemy fires one projectile at a time towards the player, two towards the player, four at a time in all directions, in random directions, etc.

Each projectile, represented by the ‘projectile’ class will need to have its own attributes in order to damage targets properly. Therefore, its speed, damage and range will be represented by the attributes ‘speed’, ‘damage’ and ‘maxLifeTime’ – the life time denotes how long the projectile has on the screen before it is destroyed automatically, and therefore decides its range.

Generally, obstacles (represented by the ‘Obstacle’ class) won’t need many attributes or functions, as they are static. The ‘damage’ attribute will be used in a similar manner as melee enemies, and inflict that damage on contact. Objects that don’t damage the player will simply have a damage of 0. Destructible obstacles are no longer needed.

We refer to what the previous team called ‘collectibles’ as ‘items’. These items can be picked up by the player on collision. The ItemType enum describes the item as either: a power-up (triple-shot, flamethrower, etc.), or health or bomb increase. Depending on what the effect of the item is, the function will use one of the attributes of whichever subclass the item is of to change the player's stats. For example, if the player picks up a health pack, the function would use the health pack’s ‘HEALTHPACK_REGEN’ attribute to increase the player’s ‘currentHealth’ by its value (of course, remembering to cap the increase to the maxHealth).

Classes representing functionality we would newly implement were also included in the new architecture. The abstract classes Saver and BaseSerializer are used to implement the saving and loading. Serializers for all the required game state data inherit from the BaseSerializer class. Similarly, the abstract class BaseScreen and the different Screen classes which inherit from it provide the previously discussed screen rendering functionality.

Appendix

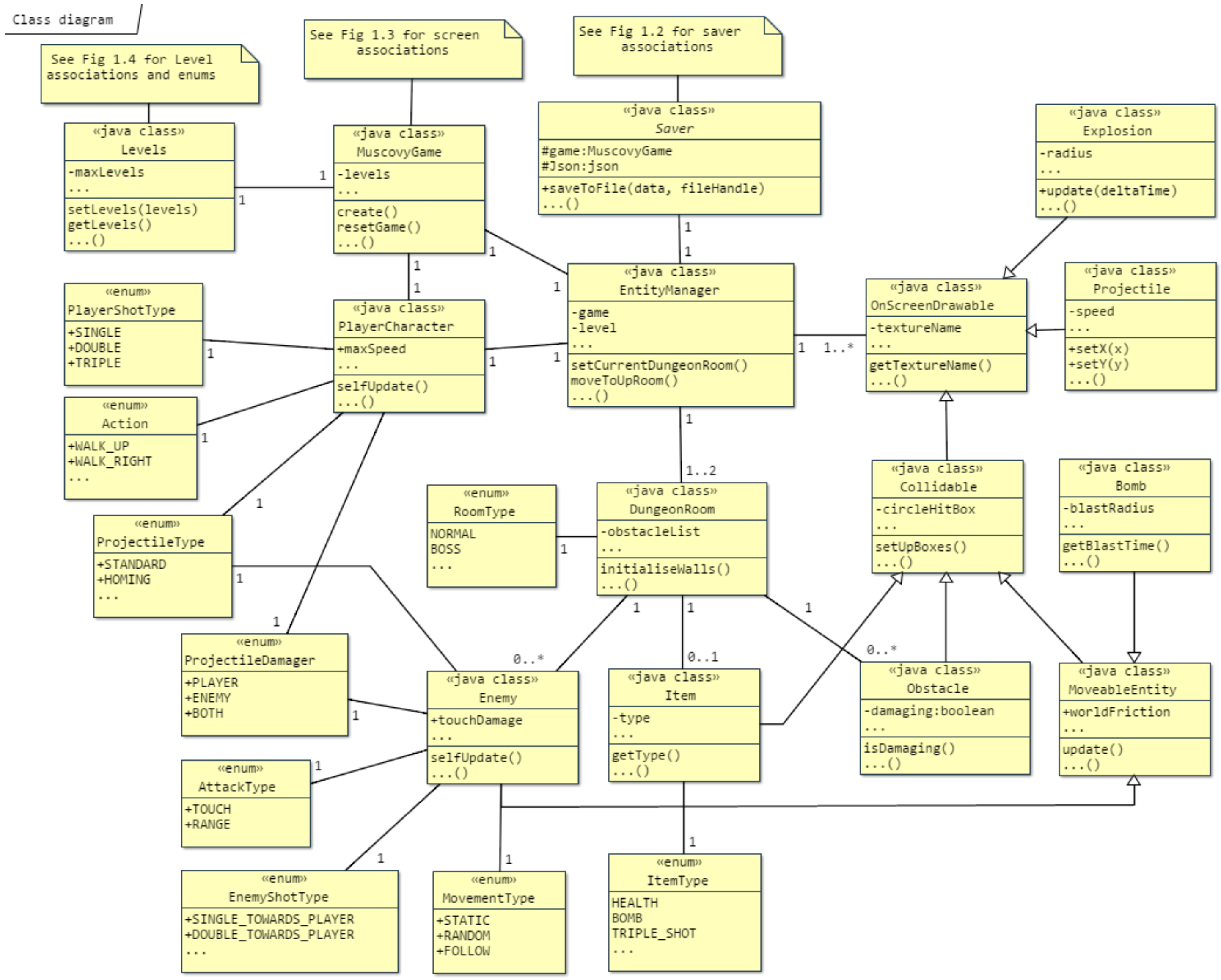

Fig 1.1

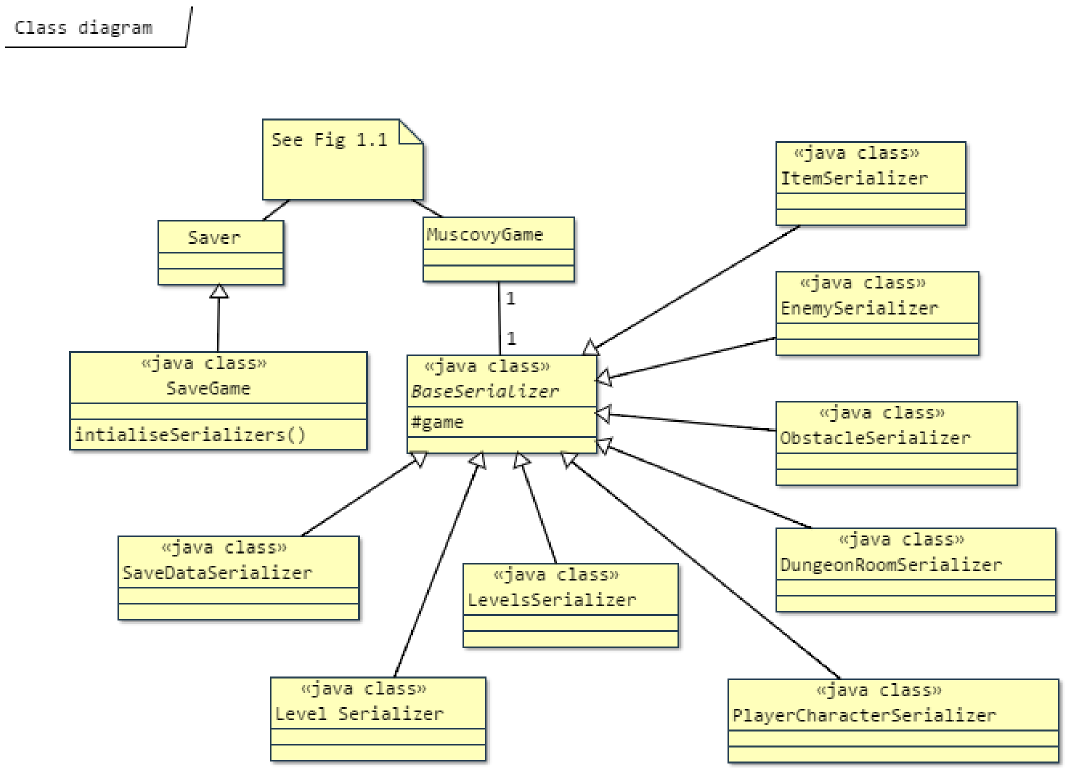
Class diagram
See Fig 1.1
Saver
MuscovyGame
1
1
«java class»
SaveGame
intialiseSerializers()
«java class»
BaseSerializer
#game
«java class»
ItemSerializer
«java class»
EnemySerializer
«java class»
ObstacleSerializer
«java class»
DungeonRoomSerializer
«java class»
SaveDataSerializer
«java class»
LevelsSerializer
«java class»
Level Serializer
«java class»
PlayerCharacterSerializer

Fig 1.2

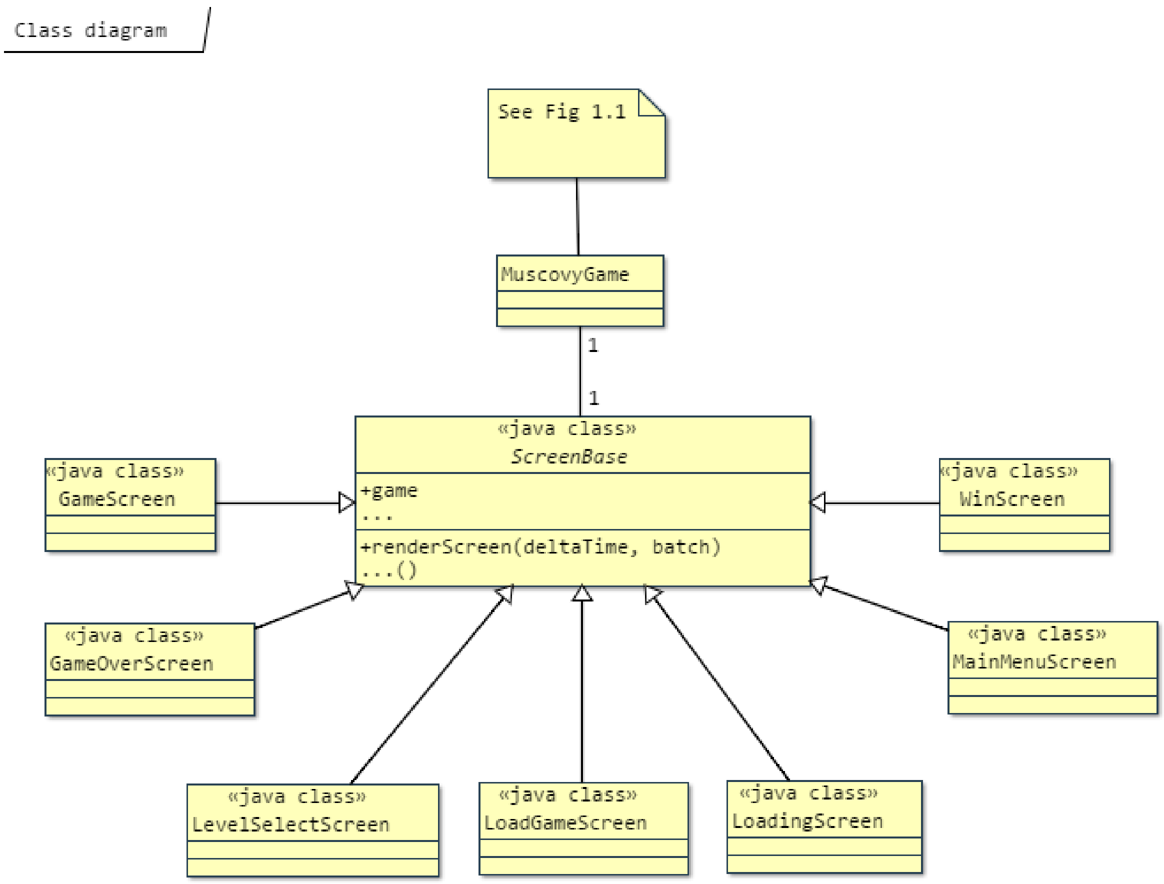
Class diagram
See Fig 1.1
MuscovyGame
1
1
«java class»
ScreenBase
+game
...
+renderScreen(deltaTime, batch)
...()
«java class»
GameScreen
«java class»
WinScreen
«java class»
GameOverScreen
«java class»
MainMenuScreen
«java class»
LevelSelectScreen
«java class»
LoadGameScreen
«java class»
LoadingScreen

Fig 1.3

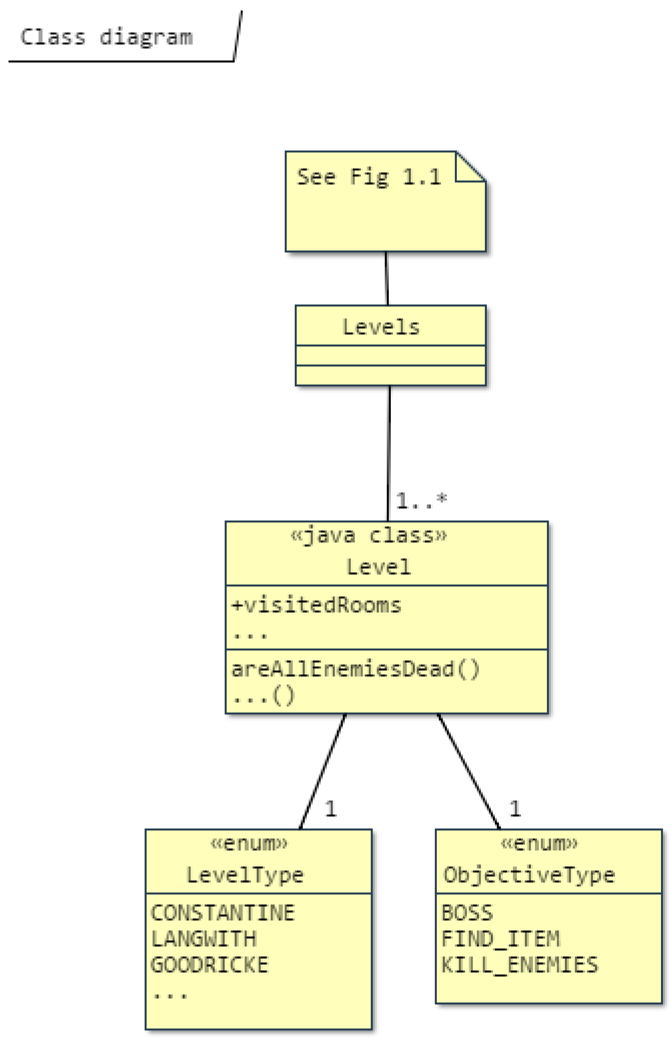
Class diagram
See Fig 1.1
Levels
1..*
«java class»
Level
+visitedRooms
...
areAllEnemiesDead()
...()
1
1
«enum»
LevelType
CONSTANTINE
LANGWITH
GOODRICKE
...
«enum»
ObjectiveType
BOSS
FIND_ITEM
KILL_ENEMIES

Fig 1.4