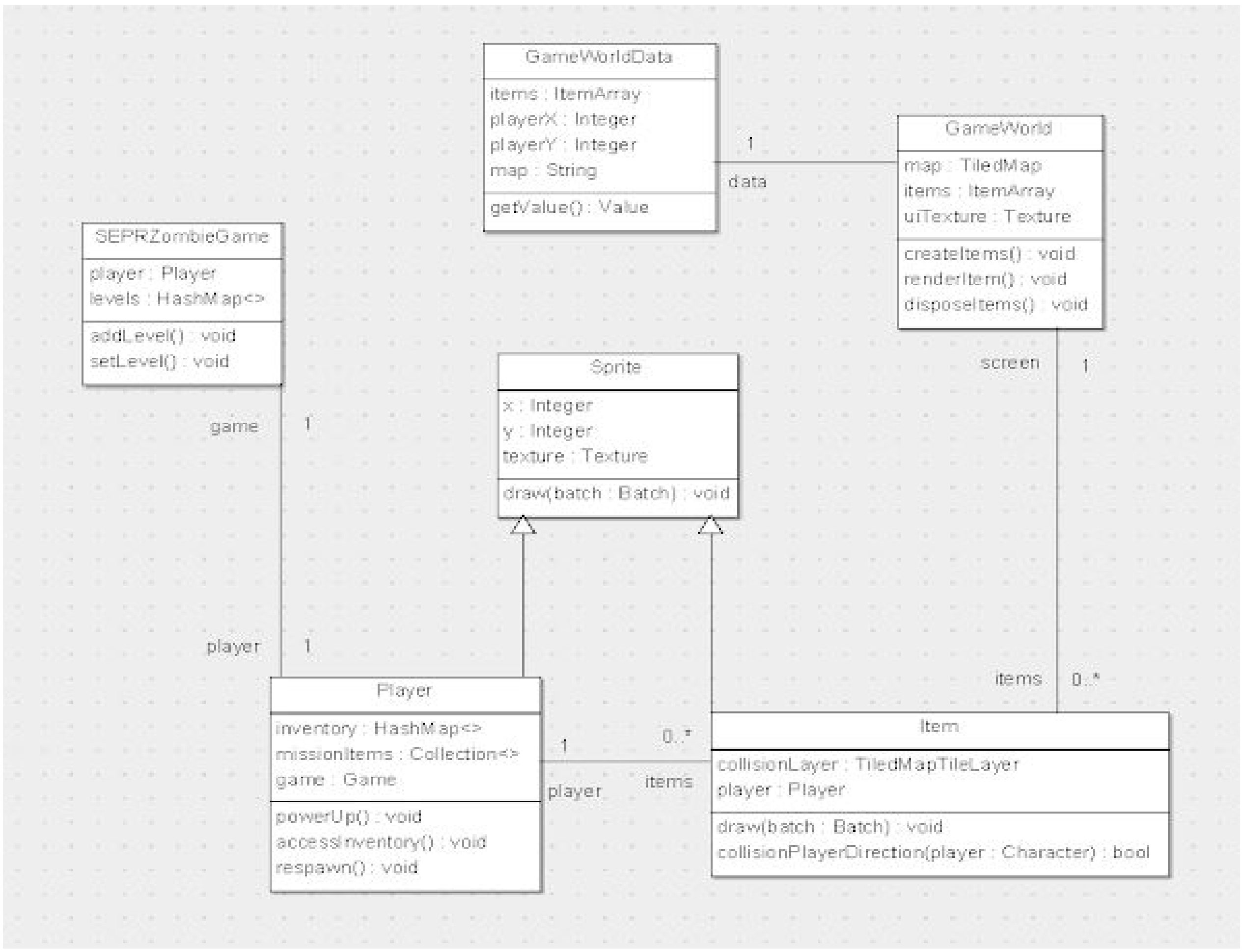
Architecture

To outline the general structure of our concrete model, we have provided a concise, simplified UML diagram using the ArgoUML tool. This diagram demonstrates the game's core classes and the associations between them, providing a good basis for discussion of our concrete architecture. To provide our code's complete structure, we have also used the ObjectAid UML plugin to extract a detailed UML diagram from our source code [1].

Simplified UML Diagram



Architectural Languages

Both diagrams follow the UML 2.0 standard, our simplified diagram only provides the class properties and operations that we decided were the best representation of the functionality of each class, this includes the omission of getters/setters and platform specific methods such as rendering as well as the generalisation of some properties. For example, the GameWorldData's instance variables for each Item present in the GameWorld has been condensed to one generalised ItemArray, as opposed to an Item array for each class of Item.

Only the classes that we deemed to be essential to the game's structure are included.

Our full exported UML 2.0 diagram details all classes, methods and instance variables in full and as such is more appropriate as an external document.

Tools

For our simplified diagram was created using the ArgoUML tool used for our assessment 1 UML diagram. We feel ArgoUML is the best fit for the SEPR module as it is free and accessible to all SEPR groups as it is installed on the department's computers. Our detailed and more complex UML diagram was produced using ObjectAid UML plugin. The main motivation for this was that we felt using an automated tool would reduce human error in our diagram and save us time. The tool is also heavily supported by Eclipse, our IDE of choice.

Modifications to our abstract architecture

When designing our game's initial architecture, we aimed to create a system that was versatile enough to be independent of implementation platform but also constrained to the Java programming language to meet the demands of the specification. Taking this approach, we had to make significant additions and amendments to our abstract model. Changes to the basic structure and relationships of our main game objects were minimal, many of the architectural additions were in the larger, broader classes such as GameWorld and GameObject as LibGDX provided pre-existing classes that offered similar functionality with the added benefit of better integrating into other elements of our game such as TiledMaps and Sprites. Throughout this section, we will reference our requirements specification [2].

It is also worth noting that there were features of our initial model that proved impractical or we simply didn't have the time to implement and these deductions can also be considered changes to our original model. These are specified in the missing requirements section [3].

The idea of a GameObject, a class that is common between all objects in the game that require basic properties such as position, image and collision mechanism, is consistent between the old and new models. In the new architecture, we use libGDX's built-in Sprite class to implement much of our conceptual GameObject class. The Sprite class is created so that we can build our objects on top of its core properties, affording us the ability to build and integrate a wide range of game objects, satisfying a number of our functional requirements including F1.0, F5.0 and F9.0. Similarly, the functionality of our abstract GameWorld class is now shared between implementations of libGDX's Game and Screen classes (in the form of SEPRZombieGame and GameWorld, respectively).

We made significant additions to the Character class, which now handles player and enemy collisions and animation. We removed the Key class, keys are now implemented using the MissionItem class, which is equivalent to our initial model ObjectiveItem class. Furthermore, we decided the Environment and Map classes provided no necessary functionality. Removing these classes was once again influenced by game engine alternatives. Environment, for example, was supposed to handle all environment objects. This proved to be a task more efficiently completed by using the LibGDX TiledMap class. Additionally, using TiledMaps allowed us to create more interesting and detailed map designs using the Tiled software. We used this power to create 2D moulds of sections of the university campus, satisfying F4.2 and NF1.0. Similarly, the role of the Map class from our initial model is completed by the Screen class, which also affords us more design choice in other ways such as camera position and displaying UI.

Justification of Architecture

SEPRZombieGame:
An extension of the LibGDX Game class. When a LibGDX project is created this class is created as a rough equivalence to the main class. By default, this main class extends a LibGDX ApplicationAdapter, allowing us to create a listener. We changed this to inherit from the Game class as this provides us with the ability to change screens using the setScreen method. Making use of this method we are able to switch seamlessly between a wide range of game screens without difficulty. This allowed us to satisfy F4.0 as well as F3.0, as we can simply switch to a mini-game screen from anywhere in our game.

GameWorld:
An implementation of the Screen class, GameWorld is essentially a template for each of our game levels where GameWorldData handles to data that makes each of these levels unique. GameWorld implements much of the game’s core features. It allows us to create UI elements, satisfying F8.0. GameWorld also adds to our game’s functionality in the form of creating and rendering each level’s objects and giving the user the ability to pause the game, meeting functional requirement F7.0. GameWorld also has an OrthographicCamera instance variable that maintains the 2D, player-focused camera, allowing us to meet F6.0.

Menu:
Menu is another implementation of the Screen class that we added to afford the user some personal choice in their character type, satisfying F1.0 through F1.3. Menu is the first screen our game switches to and acts as the portal to the main game. Menu also allows us to provide some context to the game, possibly in the form of static text or an animation, giving us the means to meet F10.0.

GameWorldData:
A class for storing data for each level, where each instance of GameWorldData represents the complete data set required to create a unique GameWorld level. We can utilise the modular structure that this class provides to design a range of unique levels very simply, helping us meet F4.0 through F4.3 as well as non-functional requirement NF5.0, as we can create a much more detailed and visually interesting game in the time we have.

Character:
The Sprite class provides us with the core properties and operations of any game element including texture, location, rendering and disposing. We have built upon this using the Character class, which provides a range of functions including animation, handling collisions and attacking. Character is an abstract class and therefore is more of a template for our Player and Enemy classes, providing the functionality that the two share, such as combat. The Character class facilitating combat allows us to adhere to functional requirement F11.0.

Item: Item also inherits from the Sprite class. Item is an abstract class that provides the basic collision functionality for all stationary game elements. This includes consumable items, doors and weapons.

Player:
As the only user-controllable object, our Player class is designed to handle key inputs and integrating them into actual player movement and action, including pausing (F7.0). Player also consists of the player's inventory and various procedures to access this inventory. Lastly, Player includes the respawn functionality, placing the player away from harm after they have died, this directly correlates to F8.1.

Enemy:
Like Player, Enemy inherits from the Character class. As such, Enemy controls the random movement of enemy instances, the range and power of enemy attacks and a boolean representative of the enemy's awareness of the player.

DramaPlayer and SportsPlayer:
DramaPlayer and SportsPlayer are concrete classes representing our game's two main player types. Each type comes with its own set of animations as well as a signature special ability, including a cooldown period for each ability. Creating these classes brings us closer to satisfying F1.0 as we have an established template on how to create these characters. It also directly meets F1.2 and F1.3, providing exactly the functionality specified.

Door:
An extension of the Item class, Door is a very important class that acts as a portal between games screens. Each Door has a Screen that going through the Door brings the player to as well as a progression system in the form of Door keys required to access the next level. Having a Door class and a key system allows us to satisfy F4.8 by making the progression condition incrementally more difficult to meet.

Consumable:
Consumable is a very simple abstract class that acts as a medium between the Player and Consumable items by providing methods that each Consumable child class must implement namely, the abstract consume method.

HealthConsumable, SpeedConsumable, ShieldConsumable:
Each an extension of the Consumable class, these classes provide specific functionality when interacted with by our player. Each provides a unique implementation of the Consumable consume() function. This gives us a scalable way of adding a wide range of player consumable items, allowing us to accommodate F2.0.

MissionItem:
MissionItem is a class that controls the progression aspect of the game. MissionItem instances such as keys are the items a player must possess in order to pass through to the next level. MissionItem instances have a unique ID that is used to search for them in the player's inventory in an efficient way.

References

[1] SEPR “Detailed UML” Sean of the Devs [Online]. Available:
https://lloydbanner.github.io/SEPR-Team-7/UML2.png

[2] SEPR “Requirements” Sean of the Devs [Online] Available:
https://lloydbanner.github.io/SEPR-Team-7/Req1.pdf

[3] SEPR “ Requirements that haven’t been fully implemented” Sean of the Devs [Online]
Available: https://lloydbanner.github.io/SEPR-Team-7/Impl2.pdf