CSC207 Design Document - Internet Explorers (Group 75)

Additional Functionality / Updated Specification

The main additional functionality that we added was the collision and level systems from phase 0. The level system allows us to render, advance and run the gameplay of the level, as well save and load progress from the game. The collision system allows us to advance the game by attacking and defeating enemies. These two features significantly advance the gameplay and move the program more in line with the specification.

There are many ways to extend the level system in the future, namely by creating more difficult levels. As of now, the code is set for just one basic level, but can be implemented to increase in difficulty, whether that be through the time limit or enemies spawned. Once more features are added to our levels, we can also expand the level saves, to apply serialization to the additional functionality across our game.

Major Design Decisions

World Entities: One of the major design decisions we made was to abstract the world entity class. Initially, each object in the world had an associated world entity, meaning we would have 2 objects per 1 item on screen. Although it was easier to code as it required less specification when spawning, it led to having a lot of unnecessary data being stored within the world, which in the long run we believe would likely make it significantly more difficult to extend the code and develop larger levels. The decision we made was to abstract the worldEntities and make the other entities child classes to give them the functionality while having to store less objects. This is also in line with the liskov substitution principle as the code we initially wrote allows us to replace WorldEntity with a child class and still run without error. Additionally, it better aligns with the single responsibility principle as the one class is no longer responsible for catering to every type of entity individually, now instead it gives basic functionality and then allows the children to add what's needed on top of it.

SOLID

Character Classes: Overall, we believe our design of the character systems adhere strongly to the SOLID principles. Specifically, the single responsibility principle, which is exemplified in the CharacterManager class, as its sole responsibility is to update the characters based on inputs and nothing else. By keeping its responsibility solely to altering characters, we ensure the class does not do too much.

Further, the GameCharacter exemplifies the open/close principle, as we can easily extend its behaviour through altering health, level or adding items to inventory, which will alter the gameplay and allow us to add new levels, features or characters in the future.

Finally, it follows the Dependency inversion principle: Similarly, the implementation also closely follows a clear hierarchy between the InputController, the CharacterManagers, and GameCharacters, representing the control class, the use case, and the entity class correspondingly. The CharacterManagers doesn't call on inputHandler and the GameCharacter class does not call on CharacterManager or the inputHandler Class

One area the design can be improved is in following the dependency inversion principle, as there is limited abstraction within the character classes, and as such increases the coupling between layers.

Hud and Inventory Window Classes: We believe the HudManager and inventory window classes exemplify the open/close principle. In our project, the InventoryWindow and the item class implements this principle. The item class is open for extension while the InventoryWindow class is closed for modification. For example, the item class has an (abstract) method createInventorySlot() that generates the corresponding item's sprite, so the InventoryWindow will be able to display any of its subclasses without the need for modifications.

In contrast, we believe the HUD and inventory window classes can be improved to follow the single responsibility principle better. Our HudManager class is responsible for operating the display of the inventory window and displays such as the time and score of the game. Although we demonstrate some of the single responsibility principle by splitting the inventor window into a separate class, this can be improved further by also decentralizing the other displays from the HudManager class. Thus, ensuring that any changes in the inventory window will not affect the other displays and vice versa.

Inventory System: In addition to following the single responsibility and open close principle, it also follows the Liskov substitution principle as weapons, a subclass of items, can replace items without breaking the game, especially in the current build as they are the main item type currently carried.

Additionally, the inventory system follows the interface segregation principle as we avoid the declaration of methods that are not needed and ensure each is required by any and all of the classes using the interface. For example, all of the items would find the getTexture(), getSize(), and getID() methods useful.

Level System: We designed the level system for our program to carefully follow SOLID principles, and believe it is properly aligned, the same as the other systems. The level system follows the single responsibility principle, by limiting the responsibilities of each class to one. The LevelManager itself is responsible only for game actions relating to progressing the level, such as rendering the physics of the objects on screen, elapsing time, and keeping score throughout the game.

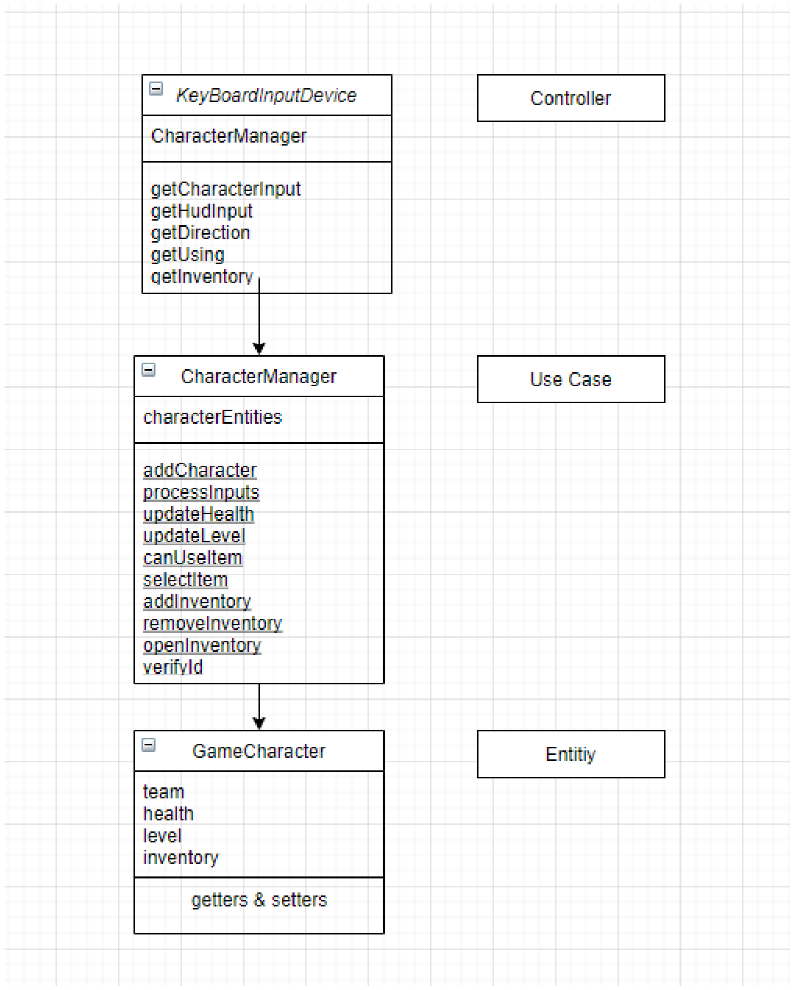
Further, the level system also follows dependency inversion principle, with the structuring between level related classes. Between our low level LevelState and higher level LevelGameplayController, a middle layer LevelManager is used as an abstraction layer, to manage changes in the lower levels. In this way, the LevelState can freely be changed without having compounded effects across the program.

World, Entity, and Spawn System: Our WorldEntity system's class hierarchy follows the open/closed principle. The base WorldEntity class implements the minimum amount of functionality needed to represent entities in the game's World, and its WorldEntityWithSprite subclass adds the functionality to draw the entity's representation on-screen. By extending either class, client code can effectively implement desired features while minimizing superfluous methods and properties.

Summary: The above highlights some examples of SOLID within the program, however, we believe the entire program adheres as we leveraged pull requests and code reviews to ensure design consistency throughout the project.

Clean Architecture

Overall, we believe our design follows clean architecture very well as we made a significant effort to ensure that the different levels interact through each other. One example which we believe highlights our designs adherence to clean architecture is the character systems.



As seen in the diagram, we segregated the responsibilities by level to ensure the dependency rule is not violated. A typical scenario in the game that illustrates this is when the human player presses a button on their keyboard, w, for example. The inputController will take that input through the KeyboardInputDevice and then convert it into a vector. The inputController will then pass on the vector to the CharacterManager and call its method, which will then update the position of the game character. This demonstrates clean architecture as each level maintains its responsibility and only communicates with the level directly underneath to ensure the chain of dependency is maintained. There are no clear violations as the imports from the external library we use, LibGDX are consistent with the level of architecture we are at, maintaining the consistency. Finally, the dependency rule is consistently followed as the inner entities have no knowledge of outer layers. As described in the scenario walkthrough, all the entities know is their position, which is updated by the CharacterManager following an input being packaged into a usable form by the InputController, which follows the dependency rule cleanly.

Overall, we believe our project adheres well to clean architecture as we tried to ensure the depency rule was maintained across the project as with the above example.

Design Patterns

Command Pattern: We leveraged the command pattern to implement our game's enemy spawn system. The LevelState object specifying each level stores: Which/When/How many enemies spawn. Effectively, we're storing a queue of operations that involve spawning some enemy. The controller class which orchestrates each level as it's played, LevelManager, doesn't care about details such as which enemy to spawn and where. LevelState could create and store a queue of Spawner Command objects describing the enemies to spawn, so that the LevelManager only needs to call each Spawner 's spawn method. Here, Spawner is the Command object. The LevelState 's World object is the receiver that handles actually creating the enemies' WorldEntity representations. The method to be applied should be a member of the World .

Factory: We used the factory method to create different types of world entities within the program to make it as easy as possible to handle new entity types being added. We used this design pattern within our spawner class to allow us to spawn different types of world entities without having to specify exactly which type. This is effective because it allows us to avoid making multiple functions to handle different types and also makes the code more extendable as it will still work for future subclasses of WorldEntity.

We also plan to implement a factory to create weapons. Currently, the Weapon abstract class and its subclasses use a semi-factory model as they are created, however, this can and will be turned into a complete factory in phase 2. Further, an idea that we have considered implementing is the builder pattern to create world entities. This would be an alternative to the above mentioned design decision for world entities to make the characters more easily extendable than they already are.

Github Features & Refactoring

Pull Requests: PRs are the github feature we made the most use of. Prior to merging any of our code into the main production branch, we made a pull request and had 1-2 people review the code to ensure there were no critical errors and find code that could be refactored. Specifically, Following pull request 19 on the CharacterDevelopment branch, we did significant refactoring to the CharacterManager in order to simplify the id system and make better use of the collections built in methods. Specifically, we simplified the id system by randomizing all rather than tying them to a specific type of entity to avoid redundancy between the id and team attribute on GameCharacter and used the .get method to update the GameCharacter values.

One code smell that we have in our code is the ability to generate world entities as we need to create a body, which requires a world. This leads to excessive coupling between classes, something we will look to fix in phase 2.

Issues: Issues were the second feature our group used. We utilized the kanban board to stay on track of each others' progress, as well as an understanding of what else needed to be completed. We used this to follow the most important development tasks, specifically the item implementation and spawn controller as those two represent significant functionality required for the final game.

Projects: We used the kanban board in the projects tab on GitHub to track our TODOs and give each other clarity into what tasks were completed and what still needs to be done. We found this extremely helpful as writing out all of the classes we needed to create on the kanban board made the overall project significantly less daunting as we knew exactly what we had to do to get to our final product.

<u>**Code Style and Documentation**</u>

As a team, we made sure to utilize Javadoc to clearly explain each method, and what can be expected for parameters, as well return values. However, when working on our project, we still found blocks of code hard to understand, and made it a point to increase single line comments throughout our methods to ensure readability of our code even within the method. This helped out a lot when reviewing each others' pull requests, as we could focus on making comments and fixes, rather than being caught on trying to understand the code.

For warnings across our code, we tried to fix them as we go. With each commit, we fixed the warnings that we could, and gradually reduced the number throughout our work process, instead of simply leaving it for the end.

<u>**Testing**</u>

We tested the critical components of our project during this phase, namely the characters, world entities and items. We focused our tests around ensuring the program could be integrated well, and are looking to focus on the underlying functionality of the code before how it is represented on the screen as we believe a strong back-end will help us deliver a better user experience. One portion of the code that is difficult to test is the characters due to the above code smell, however, moving into phase 2 we will look to simplify the instantiation. Another part that's difficult to test is the windows as it is hard to write tests for a GUI screen rather than simply running the program to test work.

Moving into phase 2, we will also make a significant effort to cover a greater percentage of our code with our tests.

Code Organization and Package Structure

We decided to package the classes by feature as we thought it was the easiest way to understand the code and find the necessary classes when they were needed. For example, keeping all of the character classes within the same package allows them to use each other without having to import. Further, when a class needs to be imported, little thought needs to be put into where it would be found as the packages are named in an easy-to-understand fashion. For these reasons, we believe organizing by feature worked best for us, and it has paid dividends by making it easy to add and find new classes as we work together.

Functionality

Our code follows our specification as the player is able to load the game and is met with the main menu with the ability to start a game. A tower is spawned in the middle of the map (the cannon image) which the player must defend. The player can then move around the map. Enemies currently spawn in waves of 1, every 15 seconds and then begin moving left. If you have survived for 100 seconds, meaning that all enemies have spawned and you are still alive, then the level is won.

We believe our functionality was very ambitious, as developing the functionality in the back end and then presenting it to the screen proved to be more difficult than anticipated. For example, the ability to save and load state is implemented under src -> main -> java -> core -> levels, with the LevelManager handling the saving of the saving of the current level state to a text file, and the LevelLoader loading in the saved levels from the text file. This is not yet an option on screen, the on screen interaction will be implemented in phase 2. Additionally, the ability for the character to place entities on the map proved more difficult than expected and will be implemented in phase 2.

Progress Report

Open Questions: What's the best way to test the code to ensure the game works the way we want to? How can we simplify or extract methods to be more common and more easily extendable? Can we implement design patterns in more areas to make our code more efficient?

Positives About Design: It is very clear how we will extend the design in the future (ex. Add levels, items, enemy types, etc.), follows clean architecture well, easy to split work and ensure everyone has an important part to work on

Individual Work:

- **Akshay:** Implemented the inventory system, including items and weapons. Will continue to flesh out the system and add more types of weapons such as health potions to add depth to the overall gameplay.
- **Ian:** Developed the level system and ability to save the game state. Will work on extending the level functionality and add more levels in the future.
- **Philip-Nicolas:** Implemented the collision system allowing objects to collide with each other and take/inflict damage. Will continue to work on collisions and more collidable objects to the world.
- **Roy:** Implemented the game display window and added the inventory window functionality. Will continue to work on enhancing the functionality of the UX/UI.
- **Andy:** Developed the inputHandler class to take inputs from the user and communicate it to lower levels of the architecture. Will continue to add functionality and character types.
- **Michael-Anthony:** Developed the input handler for the AI and fleshed out the CharacterManager class from phase 0. Will continue to add functionality and character types alongside Andy.