# Design Document
# CSC207 Course Project Phase 2

**Internet Explorers**

# Contents

## Summary

Whiteknights is a defense game in which the goal is to stay alive for all waves of enemies spawned into the game before time runs out. Through the course of the semester, group 75 - Internet Explorers, has worked diligently to develop the game adhering to the principles of software design. Ultimately, we created a game we believe follows SOLID and clean architecture strongly, has a clean, well organized code base and is a joy to play.

## Progress Report

**Akshay:** Further fleshed out the game's inventory system by allowing the player to hold defenders in their inventory and place them onto the map. Additionally, completed the implementation of the itemUsageDelegate class to allow items to be used in game. This includes the usageDelegates and ItemManager. Pull Request is significant because it allows us to actually use items, a core component of the gameplay.

**Ian:** Refactored the level system to better adhere to clean architecture. Additionally, made SavedLevel and ActiveLevel extend LevelState, in order to have full serialization of the level, including entities on the map. Pull request: was significant for its fixtures to rid of

**Phil-nic:** Fully implemented the collision system to allow entities to take damage when they collide with enemies or weapons to allow core gameplay loop to be played. Pull request was significant as it implemented the collision system which allowed us to deal and take damage to and from enemies.

**Roy:** Incorporated serialization into the main game by adding it to the presenter classes. Additionally, refactored the presenter classes to better adhere to clean architecture. Pull Request was a significant contribution as it removed the render functions from the LevelManager and LevelGameplayController. This helped eliminate a significant and unnecessary violation of clean architecture and make the code cleaner overall.

**Andy:** Oversaw the testing and helped to ensure adequate coverage across the entire program by writing and editing tests. Pull request: was a significant contribution as it implements a new way of testing through the "mock" function, which enables testing for several presenters such as LevelGamePlayPresenter that we were unable to test prior.

**Michael:** Refactored the Character, CharacterManager, WorldEntityManager and WorldEntity classes to reduce coupling and better adhere to the single responsibility principle. Pull request: was a significant contribution as it removed the need to multiple entity arrays and made the code more extendable by casting within the verifyId method. Further, both the CharacterManager and

WorldEntityManager now better adhere to the single responsibility principle as their jobs are more distinct.

## Design Considerations

### Major Design Decisions

**Inventory:** One of the main design decisions we had to make for phase 2 was how to handle the defenders. In our initial specification, we stated that a defender would be able to be placed on the screen by the player and be used to help the player defend against enemies. This led to us having to make a key design decision of whether to treat the defender as an item or as a Character. We decided the best way was to distinguish the inventory representation from the in game representation. We made a defender item which allowed us to have it in the inventory with minimal changes. We then leveraged the weapon use case class to implement the behaviour that happens when the defender item is used. The use method spawned a world entity on screen. By designing it in this way, we ensured the classes each had a single responsibility and also utilized existing classes to avoid unnecessary work.

**Level System:** The next major design decision we made was related to redesigning the old level system to adhere to the single responsibility principle, as well support serialization of the level. By extending the LevelState entity class with two further classes, SavedLevel and ActiveLevel, a serializable SavedLevel class was made. This class holds all the information needed to recreate an ActiveLevel, making for easy exit and re-entry into gameplay for users. To do so, attributes had to be distributed across the three classes, keeping in mind the bare minimum for which would be necessary only when actively playing the level, what was needed to recreate such a level when saved, and what would be needed in both.

**World Entities:** The final major design decision 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 Design

Overall, we believe our design follows SOLID well, as overall, each class has a single responsibility, is easily extendable without directly altering, is substitutable and they use interfaces and abstraction to reduce dependencies. Below we highlight some specific instances we believe strongly adhere.

**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 Character exemplifies the open/closed 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 Characters, representing the control class, the use case, and the entity class correspondingly. The CharacterManagers doesn't call on any of the input classes and the Character class does not call on CharacterManager or any of the input classes. One area in which this principle is slightly violated is within the CharacterManager. Upon further inspection, it's dependence on the concrete Character type makes it more difficult to change and evolve over time. We tried to minimize the impact by containing the casting within a single method, however, we believe it was necessary in order to simplify the overall code base and allow for the use of a single WorldEntity hashmap.

In phase 1, one of the things we noted could be improved upon was the level of abstraction we use. While there is still room to improve, we further abstracted the Character from the WorldEntity class by delineating between attributes and characteristics that are more general and those that are specific to the Character.

This allows these classes to more closely follow the liskov substitution principle as they now behave in a more consistent fashion.

**Hud Presenter and Inventory Window Classes:** We believe the HudPresenter and InventoryWindow 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, elapsing time, and keeping score throughout the game, while LevelGamePlayController and LevelGamePlayPresenter are responsible for presenting this to the user, such as rendering the physics of the objects on screen. Previously, LevelState had the responsibility of both saving and storing active levels, but in phase 2 we separated these features to better uphold the single responsibility principle. Further, in our phase 2 extension, we added LevelLoader, which continues to follow SOLID principles as it is exclusively responsible for loading the different levels.

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 only from the LevelManager, 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.

## 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. In the UML file, which can be found in our github repo in phase2/uml_diagram (given how large it is), it is clear that the dependencies move cleanly between each layer, with limited backwards dependencies, a significant improvement from phase 1.

One example which we believe best 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 demonstrates this well is the inventory system. The input from the keyboard will be taken in by the input controller classes and then be passed to the CharacterManager which will then get the inventory from the player using their ID. This straight, downward line of dependencies exemplifies clean architecture as we ensured no class knew more than necessary during the process.

Additionally, 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 current inventory, which is accessed by the CharacterManager following an input being packaged into a usable form by the InputController, which follows the dependency rule cleanly.

One area of our design where we infringe on clear architecture is in the input system for character movement. While it is easy to think that one could get inputs from a controller, pass them to a use case and have that alter entities position, in practice it becomes significantly more complicated. Specifically, having to pass inputs to the manager, which then had to find the specific entity using the id proved to be extremely inefficient for the AI given that we knew all of the spawns and wanted them all to have the exact same behavior. Similarly, we knew the player, and as such, it made sense to us to directly edit the position. While this does violate clean architecture, we believe this decision enhances the readability and user friendliness of our code. The main drawbacks outside of the violation are that it may make the code more difficult to extend in the future, however, the enemy and character inputs will likely remain similar in the long run, hence our decision.

Despite the small infringement, 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

Spawner Command: The LevelState object specifying each level stores a list of Character spawners which specify where and which enemies to spawn during the level. Effectively, we're storing a queue of operations that involve spawning some enemy. The controller class which orchestrates each level as it's played, LevelGameplayController, doesn't care about details such as which enemy to spawn and where. When its step method is called, LevelManager calls the appropriate Spawner's spawn method. Spawner<T extends WorldEntity> is the Command object. Since the command is instantiating objects, the type T is the receiver class, and its constructor is the targeted method. We extend the Command pattern by allowing client code to provide callbacks which perform additional configuration on the resulting WorldEntity object. Another useful feature is that we set up getters for the WorldEntity parameters, instead of static objects. Thus, parameters to the constructor can be computed immediately before its invocation, so they can rely on the actual state of the game.

LevelEvent Commands: The LevelEvent command encapsulates all events that are conducted in the level into an object. In doing so, a queue of events are made, allowing for events to be conducted in order of the queue. The determinant for when an event is conducted is dependent on time, where the attribute time, which is unique to a single event, is compared to currentTime of the ActiveLevel. If the time of the ActiveLevel has passed the time of the last queued LevelEvent, that LevelEvent is to be performed. In this way, the LevelEvents are conducted, as populated in the queue of events.

WorldEntity Factory: We used this design pattern to remove WorldEntiy's dependence on their WorldEntityManager. We turned the manager into a factory for all current and future WorldEntitys by making the manager have constructor methods for the different world entity subclasses. The factory prevents subclasses from altering the constructor signature by leveraging call backs for additional configuration. Additionally, it limits the use of private and final properties within the entities. Originally, we chose not to do this, however, after familiarizing ourselves with generics, we realized this was the best way to create entities in the game while maximizing future extendability.

LevelState Memento: In phase 2 we realized that serialization presents an excellent opportunity to implement the memento design pattern. In phase 1, LevelState represented both mid-gameplay active and saved levels, a violation of the single responsibility principle. We split LevelState into SavedLevel, a serializable Memento class and ActiveLevel, which can be created from a saved level. The memento captures the state and stores it externally as a .txt file which can be used to restore the game to the same state at a later time. In this way, a played level can be loaded back into the game, without violating encapsulation.

Item Factory: Similar to WorldEntity Factory, this design pattern was used to remove the item's dependence on ItemManager. The createItem method delegates the instantiation of new weapon objects to the manager class instead of client code.

This allows client code to seamlessly create the specific type of item without needing to understand its implementation details.

## Development Considerations

### Use of GitHub Features

**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. In phase 1, we found pull requests to be an excellent way to find code to 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. We simplified the id system by generating a random UUID upon instantiation rather than tying them to a specific type of entity to avoid redundancy between the id and team attribute on Character and used the .get method to update the Character values.

**Issues:** Issues were the second feature our group used. We used these heavily following phase 1 to outline the main feedback we needed to address before we began extending functionality. Linking issues to pull requests helped us systematically address problems and correct them efficiently. Additionally, tagging issues and adding tasks within helped to guide our workflow making it significantly easier to progress on the assignment.

**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.

### Code Style and Documentation

As a team, we made sure to utilize Javadoc to clearly explain each method, and what can be expected for parameters, as well as 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.

## Refactoring

In our phase 1 code, it was clear that there was a lot of extra data being stored and that the CharacterManager and WorldEntityManager shared a lot of responsibilities. To decrease the overlap, in PR Character_Refactoring #49, we removed the separate list of character entities so that we have a single list for all the entities in the world. We then added checks to make sure the entity is an instance of Character within the CharacterManager class to avoid runtime errors. Finally, we moved general methods, such as updating position outside of the CharacterManager, only keeping one's unique to the Character class.

Additionally, in commit b2f5c0a we refactored the packaging structure of our code. The code is still packaged by feature, as we believe it is the best way to organize code for a game, however, we tried to take a more zoomed out approach to lessen the overall number of packages that we have. We decided to do this following the feedback that classes such as CharacterManager and WorldEntityManager are often needed together, yet were part of separate packages. Now, they are within the same main package, with the character classes nested for further organization. One unexpected but positive side effect from this was that the protected methods and attributes in a lot of our classes became significantly less restrictive making the code a lot easier to work with.

From the phase 1 feedback, we found that some use case classes had presenter methods. For example, LevelManager had renderMap and renderWorld, which we refactored to LevelGameplayController. We also renamed many of our classes to better represent their roles; such as ScreenManager to ScreenController, and HudManager to HudPresenter.

## 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. As mentioned in the refactoring section, we took a zoomed out approach to include more like classes within the same parent package and then used sub packages to further organize the code. For example, having all of the WorldEntities within the same package makes them easier to find, and then having the Character and CharacterManager within a sub-package makes it extremely easy to find should one need to. 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.

## Testing

In phase 1, we had minimal testing as we only tested the character systems, WorldEntities and inventory. We believed these were some of the most important components as they defined the core components of the game. In phase 2 we heavily expanded the breadth of testing by testing the underlying systems that run the game. Specifically, we added tests for the spawners and levels to help us find problem areas within our code.

By decoupling the WorldEntity and by extension Character, writing tests for those classes and related ones became much simpler. While entities became easier to test, the presenters, controllers and GUI remain difficult to test as their output is visual, and as such we struggled to write tests for these classes. To work around this difficulty, we added a config setting to the command line. This allows us to see in real time what's happening when certain things interact within the game, such as the character taking damage. This made it possible for us to write tests for the on screen portions of the game and ultimately allowed us to increase our test coverage.

## Discussion

### Updated Specification

From Phase 1, the key additional functionality added was the completion of the collision system which allows the player to both take and deal damage. This is a core element of gameplay. In the game, once all of the enemies spawn, the player is able to win the game.

During this phase we extended the core functionality of our game by adding two additional levels. Given that we also had to complete the remaining portions of our specification, we believe that this was a significant addition as it creates new functionality beyond the initial specification of having only a single level to play on. Further, functionality was added to accommodate saving levels, even mid-gameplay. In this way, users could exit the game and come back right to the same progress point they were at, with the same time elapsed, position of player character, position of defenders spawned, enemies defeated and current location of enemies still on the map.

### Functionality

Improving on phase 1, our code now more closely 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. When in the game, the player is able to select from 3 different items, a sword, dagger and defender, which can either be used to attack or placed on the map in the case of the defender. The defender now helps the player by providing a barrier to incoming enemies. Finally, if the player hits an enemy enough times, they will be defeated and removed from the screen. The win condition for the game changed as we were not able to implement the desired tower due to time constraints. Now, the player must defeat all the enemies within the allotted time for the level in order to win the game.

We believe our functionality was very ambitious, however, we believe we came very close to our specification within the allotted time and created a fun and easily accessible game.

## Open Questions

Although we are now at the end of the project, we still have a few open questions that we may address should we continue to work on the game.

- How can we better manage the visibility of methods/attributes to make testing easier?
- Is there any code we can refactor/abstract to make it more simple to an outside editor?
- How can we improve the accessibility of our game and make it more offhand/keyboard only accessible as discussed in our accessibility document?

## Conclusions

**Key Learnings:**

- There is tremendous value in communication, especially early on and between phases
- Setting specific goals and deadlines early is key to efficient development
- Github features are extremely useful for project management

**Challenges:**

- It is difficult to refactor to make things clean once written. Having a second or third set of eyes look over code for inefficiencies in pull requests is invaluable
- It was often difficult to delineate between levels of clean architecture, especially when working with code from LibGDX. We had to get creative to find ways to avoid one level bleeding into another
- We struggled to scope the project upfront which led to difficulties in meeting the specification in the end, however, we believe we made a significant effort and came close given the time constraint.