# Project Structure Document

## Part 3 - Interim Demo

Computer Games Laboratory

**Summer Term 2021**

**Meeple People**
Anastasia Pomelova
Eugene Ghanizadeh Khoub
Mert Ülker
Shyam Rangarajan

# Contents

This report will discuss our development process for Flee Fi Fo From, and an update of our status at this interim stage. We will then cover the challenges we encountered as well as our next steps.

# 1. Design Approach

For our development process, we first outlined the architecture for the project. Following this, four subsections were identified as illustrated in Figure 1 below. These sub sections or modules would allow for the compartmentalization of development efforts within a particular module to allow for independent progress across the board. Once a stable code base is ready, we will integrate these modules with each other for the complete version of the game.

The subsections are:

- **UI Design:** Presentation layer that the user interacts with.
- **Game Logic:** Defines the logic for individual actions within the game.
- **Game Manager:** Central control of the game which connects to the other elements.
- **Networking:** Provides the capability for gameplay across different clients.

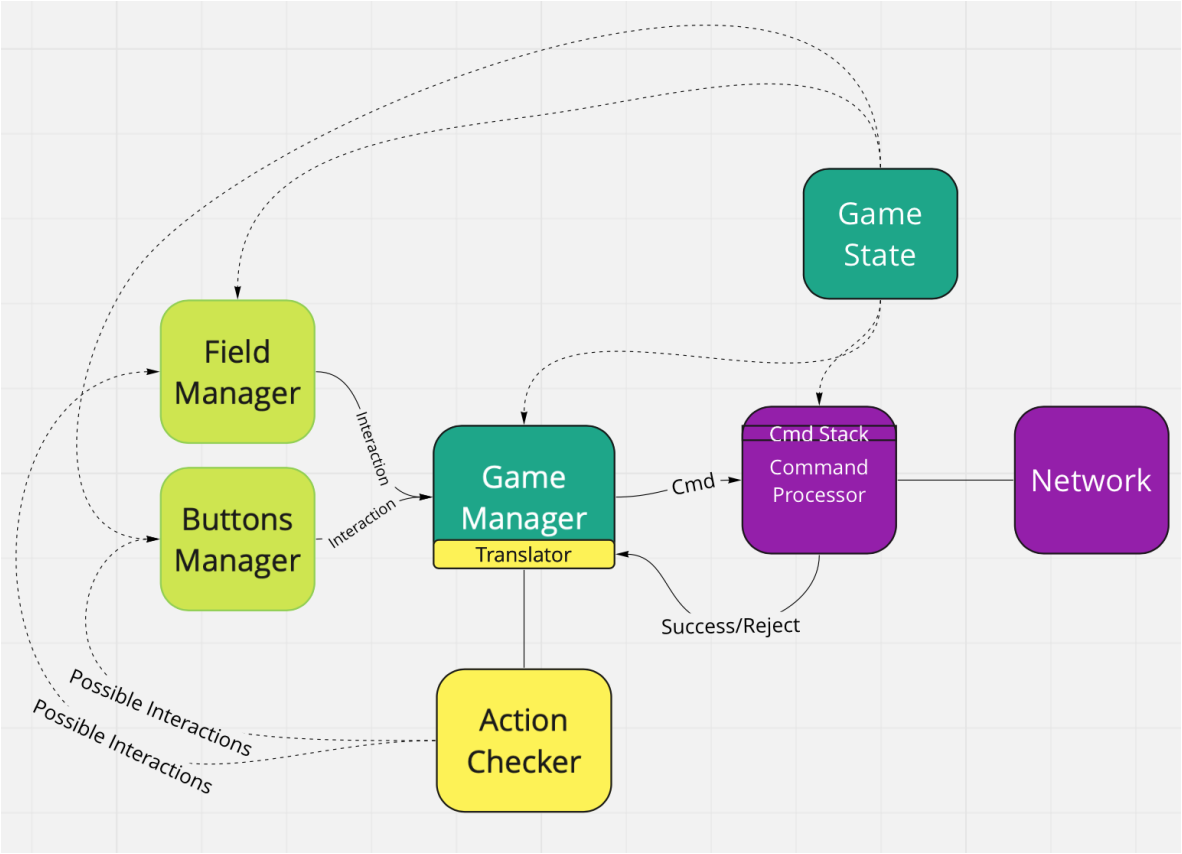



Figure 1: Design Architecture

# 2. Interim Status

In the following section, we will discuss the status update for each individual subsection.

## 2.1. UI Design

This layer consists of the visual components needed to play the game (perform actions part). In its current state, the UI already allows for a player to interact with the game board and overlay UI elements. This includes the base field commands and undo commands. For each command the interactable pieces are highlighted. A simplified version of the riot command has been implemented, allowing for a piece to be injured. This in turn allows for the testing of the Revive command. Furthermore, preliminary work on the worker resets has commenced, with the ability to depict workers per action space, as well as remove workers via the reset commands. All objective related actions are implemented as placeholder methods, until we start working on the objective mechanic in our higher goal tasks. The honor tracker above the field is already depicted in our desired UI style. An illustration is shown below in Figure 2.

The UI leverages the use of Button Manager and Field Manager classes to depict the player interactable actions and the game board respectively. These classes would interface with the Game Manager to allow for rules enforcement.

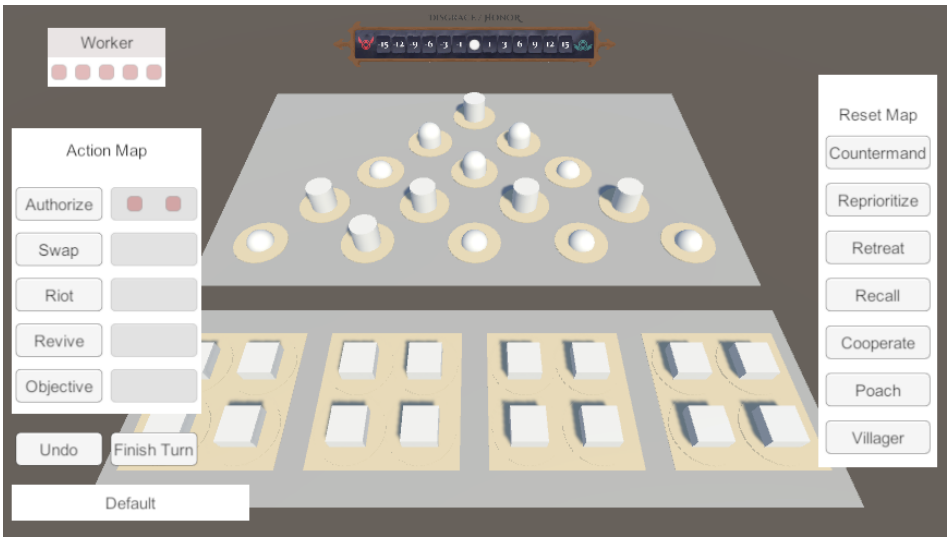



Figure 2: User Interface

## 2.2. Game Logic

This piece of the project contains the game logic, which adapts the rules of the board game into a code structure that can be processed by the game manager.

The architecture consists of a set of classes for each of the game pieces and the available actions that a player could take. The viability of these actions are evaluated via checker functions within each action (and corresponding reset). These checker functions leverage helper functionality provided by the game manager. Actions and resets selected by a player are then processed by the game manager, and added to a command stack. Further details can be found in the game manager section.

## 2.3. Game Manager

This layer includes the Game State data structure, which is a purely logical representation of the board, workers, player scores, etc. Due to the complex nature of many rules the game has, the Game State is augmented with utility helper functions that allow expression of such complex logic in an intuitive and reusable way, and allows for faster iteration on the ruleset of the game.

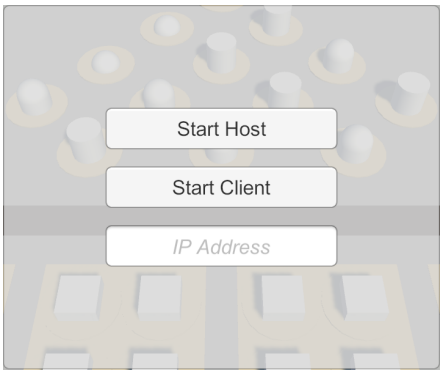
```
private void EnableRiotPath(List<Tile> path)
{
    var last = path[path.Count - 1].Position;

    GameState.Instance.TraverseBoard(p => {
        var tile = TileByPosition(p);
        var meeple = GameState.Instance.AtPosition(p);
        tile.Interactable = (
            last.CanMoveTo(p) &&
            (
                meeple == null ||
                (
                    meeple.IsHealthy() &&
                    meeple.GetType() != typeof(DKnight)
                )
            )
        );
    });
}
```

Figure 3: Game Manager Code Snippet

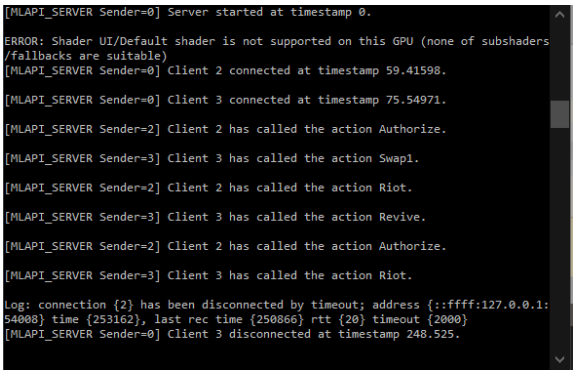
Figure 3 contains an example code for highlighting tiles that can be selected for a step of a riot action, and the expression basically boils down to a traversal of the board, and marking tiles that can be moved to from the last tile of the riot, and they are either empty or a healthy villager (not a knight) is on said tile.

The Game State also provides an Observable interface, allowing the visual layer to listen for changes to particular values and reflect them efficiently on the board (e.g. position of a villager or whether they are injured or not).

The Game Manager (still WIP) has the responsibility of translating raw interactions (clicking on a button, a piece, etc) into game commands (e.g. swap two pieces, riot, etc.) based on context and game rules. It acts as a central bus between various other modules: the visual elements communicate interactions and their feedback to the Game Manager, Action checkers communicate with the Game Manager to retrieve a list of possible interactions at current state, and passes logical commands to the command processor which in turn results in updates in the Game State and will be synced via network.




(a) Connection UI




(b) Server Logging

### 2.4. Networking

The networking layer has been fully decoupled from the remaining system, with the exception of command processor, in an effort to realize a more adaptable system, in turn also leading to an ease of parallel development of distinct layers and subsystems. This allowed for a simultaneous development of game logic/manager and the networking layer, while also rendering a smooth networking integration possible. With the current state of the networking, we are able to host a session on the local area network, which will be further extended into internet connection over the course of remaining development phase.

Client-Server model was realized by employing a dedicated server to host the session in the form of a headless server build and multiple players connecting the host as clients. In contrast to a direct communication between the clients, the model only allows a communication of clients through the host device, which is also responsible for processing commands and maintaining their history for potential undo operations. An initial version of the communication manager was implemented to publish actions through the server to other clients, utilizing remote procedure calls and executing commands locally on each client.

Furthermore, a simple UI was employed to establish connection for the clients, along with a server event listener, which proves to be especially helpful for debugging purposes by logging information such as server start and client connection/disconnection.

## 3. Challenges

We encountered a few challenges during our planning and development efforts. Firstly, the rules-heavy nature of the game meant that we had to decide our architecture and integration points before implementing any base functionality, in contrast to iterative development from a complete barebone minimum game in the typical layered approach of the course. We decided to split the architecture into subsections to allow for independent development before merging. The advantage of this approach is that we were also able to commence working on items from future layers at an earlier stage, for instance the playtesting of the gameplay and rules, as well as the networking components.

Secondly, our desire to implement an undo functionality created significant design constraints. This affected the choice for the command pattern architecture, as well as the use of a command stack for easy reversion of previous moves by a player. Finally, we are working on constantly updating the rule set of the game, modifying and trimming the rules where necessary to lower the cognitive load on the player and allow for a streamlined experience.

The process of playtesting the rules of the game in the board game form thus far has yielded feedback regarding the complexity of certain actions, specifically around the myriad options for Authorize. Furthermore, the FIFO system of tracking has proven to be challenging to track, and also has the potential for tactile mistakes during tapping. The current piece orientation also presents visual challenges when viewed from different sections of the board.

In light of this, a modified FIFO system using an externally tracked priority system across each piece type is being experimented with. Furthermore, the Authorize and Riot specific rules for the different piece types could now be removed, as they could be internally accommodated in the FIFO system. The change of these game rules will simplify the gameplay, as well as the corresponding game logic. These changes will be tested before implementation in the alpha release.

# 4. Next Steps

In terms of next steps, we will be working on integrating the independent subsections into a cohesive whole, as well as removing any duplications currently existing as placeholders. In addition, our next steps for each layer include:

- **UI Design:** We will be focusing on adding the connections for remaining worker actions, and identifying areas of duplication between UI and Game Manager. Following this, we will be performing visual improvements to the game interface and assets.
- **Game Logic:** Apart from the addition of worker based logic, we will be focusing on verifying the modified rules before integration with the code base.
- **Game Manager:** We will be focusing on decoupling overloaded classes such as Field Manager, Button Manager and Action Checker, and absorbing some of this functionality within the game manager module instead. We also need to expand the logic for the worker rules.
- **Networking:** For networking, we will be working on adjusting the system for connection over the internet. Following this, we will be adjusting the connection manager and corresponding UI to support DNS address and/or server name along with IPv4 address. Finally, we will be implementing fallback methods to handle potential server connection errors.