**Supreme Checkers**

***Software Design Document***

**Context**

*Drexel University* | **SE-181: Intro to Software Engineering**

Fall 2020

**Project Members**

Christopher Clifford

Henry Dang

Peter Mangelsdorf

Conner Pierce

Thomas Trimbur

[crc339@drexel.edu](mailto:crc339@drexel.edu)

[hd349@drexel.edu](mailto:hd349@drexel.edu)

[pjm349@drexel.edu](mailto:pjm349@drexel.edu)

[ccp49@drexel.edu](mailto:ccp49@drexel.edu)

[tft27@drexel.edu](mailto:tft27@drexel.edu)

**Faculty**

Filippos I. Vokolos

[fvokolos@drexel.edu](mailto:fvokolos@drexel.edu)

**0.0 Contents**

1. **Introduction**
2. **Overview**
3. **Networking**
4. **Requirements**
5. **Data Model**
6. **Interface**
7. **Deployment**

**1.0 Introduction**

**1.1 Purpose**

The purpose of this design document to provide design guidelines for the implementation of the Overlord-Supreme Checkers

game. This document will serve as a reference for the developers coding the game.

**1.2 Scope**

The major functionality of Overlord-Supreme Checkers is to provide an interface for two players to compete in an online match of checkers.

**1.3 Definitions**

**1.3.1 Checkers**

**Tile** A spot on the board upon which a piece could be placed or moved.

**Man** A regular checkers piece.

**King** A piece that can move diagonally backward.

**Player** A user who has connected to an opponent in the game and is about to begin the game, in the process of playing the game, or has finished the game. A player either controls the light or dark pieces. This term will be used to refer to the current player (the player whose turn it is).

**Opponent** A user who has connected to a game against the player. The opponent is also a player.

**Move** An action either player can take. This action involves moving a piece from one tile to another at least once, and may involve capturing another tile.

**Turn** A time frame in which either player makes one or more moves.

**Pile** A player's collection of captured opponent pieces.

**Software**

**Photon Unity Networking** or **PUN** A C# library for creating server- based networking applications.

# 2.0 Design Overview

## 2.1 Description of Problem

The problem is creating a way to play American checkers over the internet with automatic rules enforcement.

## 2.2 Technologies Used

This checkers game will use Unity3D as the core engine for rendering models and taking in user input. We will be utilizing Photon's PUN unity asset in order to handle the networking for the game.

The target platform will be Microsoft Windows and Linux, and the development environment will be Visual Studio Code and IntelliJ.

## 2.3 System Architecture

This system will be constructed from the following components:

* Game Model - All of the classes related to creating our checkers game, such as the Board, Piece, Player, etc. All of the game data during a checkers game is stored in-memory inside of the game model and updated/synchronized using Photon PUN.
* Photon PUN - A networking cloud service that integrates with the Unity engine through a marketplace asset and is used to handle data synchronization between clients.
* Game Interface - The UI that the player will interact with to play checkers.
* Local Storage - Storage for data across sessions, like names.

Diagram

Description automatically generated

Figure 1 above shows the connections between our high-level components.

## 2.4 System Operation

Figure 2 shows the sequence of events that occur during a normal game of checkers.

A screenshot of a computer screen

Description automatically generated

# 3.0 Networking Design Overview

The figure below shows the UML diagram of our networking interface.

Diagram

Description automatically generated

Referenced this for network structure: <https://www.raywenderlich.com/1142814-introduction-to-multiplayer-games-with-unity-and-photon>

# 4.0 Requirements Traceability

| **Requirement** | **Description** | **Design Reference** |
| --- | --- | --- |
| R4.2.\* | Matchmaking | 6 - Photon & Networking Interfaces |
| R4.3.\* | Environment | 6 - Board & Piece Interfaces |
| R4.4.\* | Start of Game | 6 - Board Interface |
| R4.5.\* | Gameplay | 6 - Board Interface |
| R4.6.\* | End of Match | 6 - Board Interface |
| R5.1.\* | Network Performance | 3, 7 |
| R5.2.\* | Operating System Requirements | 6 |
| R5.3.\* | Availability | 6 |
| R5.4.\* | Security | 7 |
| R5.5.\* | Usability | 6 |
| R5.6.\* | Maintainability | 6 |

# 5.0 Data Model and Storage

### Data Model

The figure below depicts the UML for the Profile data model.

A picture containing graphical user interface, text, application

Description automatically generated

The figure below depicts the UML for the data storage.

Diagram

Description automatically generated

### 5.1 DataModel

DataModel provides a facade for the data storage library.

#### 5.1.1 Attributes

| **Name** | **Type** | **Description** |
| --- | --- | --- |
| ProfileStore | IProfileStore | Used to access the player's profile. |

#### 5.1.2 Methods

| **SetName(name: String)** |  |
| --- | --- |
| Input | The player's new username. |
| Output | Void |
| Description | Modifies the user's current username and saves it. |

| **GetName(): String** |  |
| --- | --- |
| Input | Void |
| Output | The player's current username. |
| Description | Retrieves the player's current username. |

### 5.2 IProfileStore

IProfileStore is an interface that provides resources for accessing the user's profile.

#### 5.2.1 Methods

| **GetProfile(): Profile** |  |
| --- | --- |
| Input | Void |
| Output | The player's current profile. |
| Description | Retrieves the player's current profile from memory. |

| **SetProfile(profile: Profile)** |  |
| --- | --- |
| Input | The player's new profile. |
| Output | Void |
| Description | Modifies the player's current profile in memory. |

| **Save()** |  |
| --- | --- |
| Input | Void |
| Output | Void |
| Description | Saves the state of the player's current profile in memory. |

| **Load()** |  |
| --- | --- |
| Input | Void |
| Output | Void |
| Description | Loads the state of the player's current profile into memory. |

### 5.3 JSONProfileStore

JSONProfileStore is an implementation of IProfileStore that stores the player's profile in a JSON file.

#### 5.3.1 Attributes

| **Name** | **Type** | **Description** |
| --- | --- | --- |
| Profile | Profile | An in-memory state of the player's current profile. |

#### 5.3.2 Methods

| **GetProfile(): Profile** |  |
| --- | --- |
| Input | Void |
| Output | The player's current profile. |
| Description | Retrieves the player's current profile from memory. |

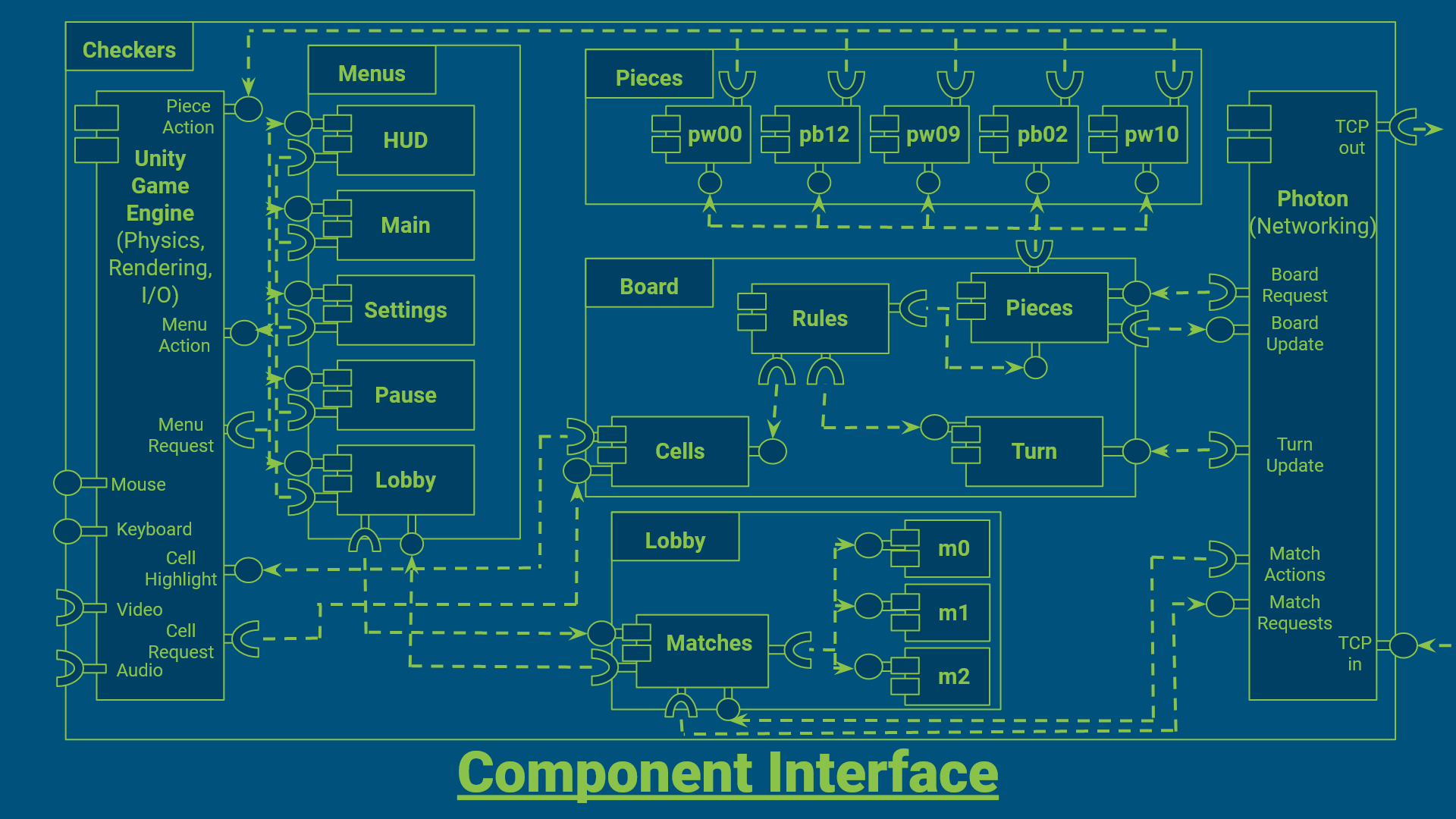
| **SetProfile(profile: Profile)** |  |
| --- | --- |
| Input | The player's new profile. |
| Output | Void |
| Description | Modifies the player's current profile in memory. |

| **Save()** |  |
| --- | --- |
| Input | Void |
| Output | Void |
| Description | Saves the state of the player's current profile to a JSON file. |

| **Load()** |  |
| --- | --- |
| Input | Void |
| Output | Void |
| Description | Loads the state of the player's current profile into memory from a JSON file. |

# 6.0 Interface

## 6.1 Diagram



## 6.2 Unity's Architecture

* The Unity Game Engine makes heavy use of the Component design pattern
* It does so in support of an Object Oriented Entity Component System
* A game is composed of Scenes
* Each Scene has a collection of GameObjects
* Each GameObject is a 3D Primitive with attached Scripts (called MonoBehaviors)
* All MonoBehaviors are extended from the base MonoBehavior and concern a set of Lifetime Events and Lifetime Methods
* These Lifetime Methods are a way of embedding side-effects within every GameObject, such as 2D Image Rendering, Physics Rendering (position), and Input/Output
* These GameObjects differentiate between themselves by which MonoBehaviors they happen to have attached to themselves
* The Scene organized GameObjects under a tree of Parent-Child relationships
* Any GameObject can be serialized to a Prefab, similarly to the Prototype software pattern
* This allows multiple instances of the same GameObject to exist within a scene
* A Common organizational pattern in Unity is the use of Empty GameObjects, that have no MonoBehaviors, and simply hold other GameObjects
* Photon, our Networking Solution, also embeds itself within GameObjects

## 6.3 Representing Unity's Architecture

* We can generally represent GameObjects in Unity as Components
* Empty GameObjects can be represented as Projects (holding Components)
* Since Unity handles 2D Image Rendering (show piece positions), Physics Rendering (move the pieces to new positions), and Input/Output Management (mouse, keyboard), we include it as a Component in the diagram, despite it actually being a capability embedded in every GameObject
* Likewise, Networking is shown as a separate Component, despite also being included in every GameObject
* Instead of focusing on all Network, Physics, and Image Rendering interactions, we show particular interactions, such as Cell Highlighting, Piece Movement, and Cell Selection

## 6.4 Major Concerns

### Menus (Project)

* each Menu has a very similar interface it shares with the Unity Game Engine
* each Menu adjusts a broad range of things, such as Settings, but otherwise has few strong dependencies with other Components
* These largely communicate with on-screen, flat (2D) images and the Unity Game Engine itself to adjust variables

### Unity Game Engine (Component)

* Unity Game Engine is largely a broad collection of side-effectful computation
* It handles User Input, User Output, Menu Interaction, and Board Interaction

### Pieces (Project)

* this is simply to demonstrate that the Pieces (Component) of the Board (Project) must be able to reference many Piece-s (Component-s) and set their Physical Position given their Logical Position

### Board (Project)

* This is the largest part of the project we are developing
* It encompasses the most responsibilities, and is shown as separate Components to illustrate this
* The Rules (Component) includes number of hops, hop distances, promotion, capturing, rule delegation
* The Turn (Component) simply tracks whose turn it is separate from the Piece Positions
* The Cells (Component) is how players interact with the board, and highlights particular cells, checks what pieces are in particular cells, and communicates those changes through the Rules (Component) to the Pieces (Component)
* as such, the majority of the Relevant State (Logical Piece Positions), carried across Turns, is contained to the Pieces (Component)
* The Pieces (Component) and Turn (Component) are therefore the only parts of the Board (Project) that needs to communicate with the Photon (Component) to issue and receive updates with the other Player

### Lobby (Project)

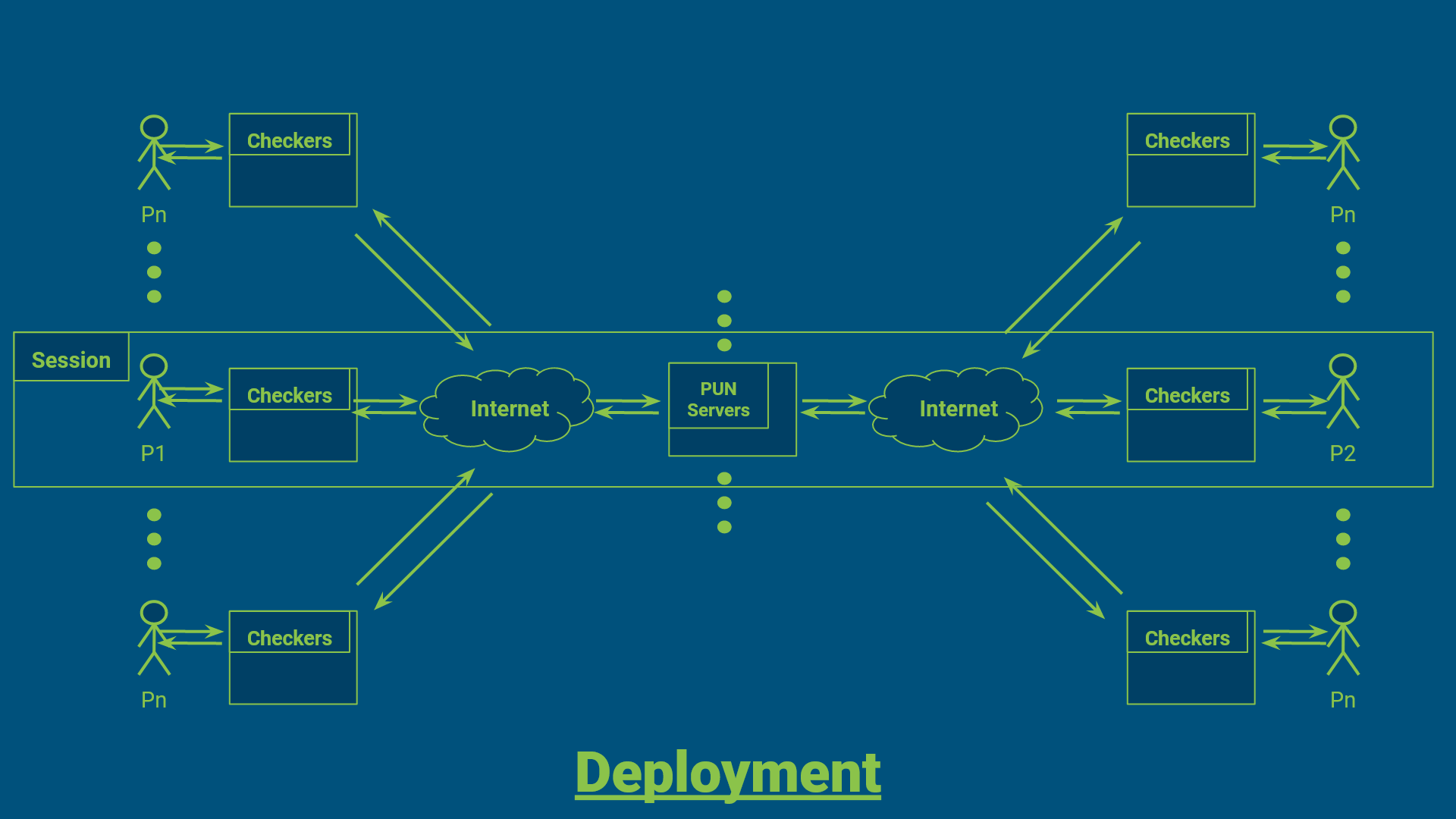
* This tracks what Matches other Players have created and allows Unmatched Players to join those Players who have opened a Match to play
* The Matches (Component) issues and receives updates using Photon

### Photon (Component)

* This enables the Game to work, by exchanging Board State in generic TCP Game Status Packets
* The Messages include Board State (positions, turns), Match State (username), and Match Actions (join, leave, create)

# 7.0 Deployment

## 7.1 Diagram



## 7.2 Notes

* Many Sessions can occur
* Many PUN Servers (Photon Unity Networking) can exist
* Many Players can exist
* All communications are done through the Internet