Chess Application

Distributed System

Aivaras Zevzikovas (315228)

Mikolaj Morawski (315229)

Wiktor Ciolkowski (315187)

Selina Ceban (315235)

Supervisors:

Jakob Knop Rasmussen

Joseph Chukwudi Okika

VIA University College

Number of characters: 98258

Software Technology Engineering

Semester III

16/12/2022

Table of content

Abstract

This project aims to provide students at a university with the opportunity to connect and participate in chess tournaments, regardless of their location. To fulfill the client's request, the system ended up consisting of a Blazor Server providing access to GUI through a browser, a C# Communication Server handling client connection using SignalR and REST, a "Stockfish" Server that returns moves imitating chess opponents with adjustable difficulty through gRPC, and Java Database Access Server for storing user data and REST API. The result is a responsive and expandable application where users can play between themselves or practice against an AI and documentation that explores how to further expand system functionality that is relevant to the client.

1 Introduction

The project development was based on the input and guidance of the client. To understand the project requirements and how to proceed with the analysis, the client provided a background description that was essential to the understanding of the project's purpose and relevance.

1.1 Background Description

One of the university conglomerates seeks to bring all benefits playing chess has to their students. Professors noticed that some students show interest in chess and created several small chess circles. The university conglomerate wishes to connect these groups in hopes of attracting more students to play chess and to connect their community of students between university campuses. Unfortunately, the university conglomerate finds it troublesome to allow all students that would be interested in participating.

As of right now, students play chess with each other on each university campus. Some campuses have very few people playing chess, so the students have few opportunities to socialize with fellow chess players and practice their skills against real opponents. Every set period, the university conglomerate organizes tournaments between campuses to promote chess playing and determine the student best at playing chess so that the university has a representative for inter-university tournaments. While these tournaments are good at selecting the best person at playing chess, they do not provide an equal level of entry. Because skill distribution in classes is not even, some students have a harder time getting to the tournament than others. The optimal solution would be to have every student play with random students from all campuses during qualification but to do that the traditional way is unfeasible.

Setting up the tournament takes a considerable amount of effort administration - wise. The goal for the university is to allow all students from each campus to play and interact with each other and streamline the process of organizing a tournament as much as possible so that it could happen at least once a semester. Additionally, to further enable students to play chess, the university conglomerate would like to provide QoL (quality of life) improvements in playing chess. One of the professors suggested for example that students would begin more chess games if the setup and resuming of the game was quicker or automated. Another suggestion was that students could improve their gaming strategy much faster if they had access to all the games and mistakes they made. Finally, in the absence of other opponents, a student should still be able to practice their tactics. Unfortunately, such options are difficult to implement having only traditional chess boards at their disposal.

The university conglomerate board considered a few solutions that would be able to fulfill the requirements but found them insufficient. Several free-to-play online solutions were considered but the blocked features, subscription requirements, and lack of control over the platform made them unsatisfactory. Simpler solutions, such as playing over zoom, would be impossible to manage on a tournament scale and wouldn't implement any suggestions given by the staff.

An opportunity to create a software solution was recognized, one that would provide a better way to play chess with student peers, organize chess tournaments, and manage student players, thereby saving time and providing fairer competition. Such a solution would help both the university and the world by aiding in UN Global Goals.

1.2 UN global goal - education and networking

Chess is one of the oldest and without a doubt most popular board games of all time. It has numerous benefits for those who engage in the game. As shown in many research papers (Aciego, García, and Betancort, 2012), it enhances one's ability to deeply focus, increases creativity, helps to develop a perspective, and much more. It is particularly beneficial for the youth as it can greatly improve their mental capabilities as well as performance in school. 2019 study (Poston and Vandenkieboom, 2019) has shown that kids who play chess – be that in a chess club or the

1. Quality Education Global Goal

official FIDE chess tournaments – gained substantially (5%-50%) in subjects such as Math, and (10%-20%) in Reading. This proves that playing chess can help young people, who are aiming at studying STEM in the future, develop an early intuition of critical thinking, planning, developing a strategy, and higher-order thinking in general. Considering all of the above, enabling more people to play chess would certainly help fulfill the quality education UN global goal (Homepage - The Global Goals, n.d.)

1.3 Aim and objectives

The project aims to develop a heterogeneous chess application that helps in fulfilling global goals and adheres to principles of developing software, therefore scalable, efficient, and with readable source code. Objectives of the project include:

1. To analyze the client's needs and create artifacts needed to meet the client's needs.
2. To design a chess application with a distributed architecture that will ensure scalability and stable connection between users.
3. To design a friendly and pragmatic UI that will encourage students to engage with the system.
4. To implement a heterogeneous system using lean principles and methodically extend it with value-adding features, so that the system may be continuously developed while in use.
5. To test and optimize created system and ensure that the system meets requirements set by a client and overall reliability
6. To create documentation, such as a user guide and installation guide, so that users will have an easier time adjusting from a physical chess game.

1.4 Delimitations

Some delimitations were set to limit the scope of the project, increasing focus on deliverable solutions and elevating chances of success.

1. The application will not support chess game analysis. Users may be able to use third-party tools for chess analysis but running automated chess analysis tools is a resource-intensive endeavor.
2. Chat will not be moderated. Chat moderation is a challenging task that requires resources and moderators that will be able to continuously supervise each conversation. It is expensive, and therefore the only way to restrict offensive content is by reporting users to administrators. The risk of harmful content is significantly reduced thanks to accounts not being anonymous. (Omernick, 2013)

In the section following, a description of the first steps that were taken during development will be presented. The Analysis part of the report outlines what requirements were created and explains the domain model and use cases that became a foundation of the system’s design and implementation.

2 Analysis

While a game of chess is thousands of years old and its rules are known to most, it is very important to analyze precisely what the client needs and expects from such a chess application, so the analysis relied on the background description and client input.

2.1 Requirements

A set of requirements have been created at the inception of the project. Their contents and importance changed as the client's needs shifted. While in the beginning university thought of tournaments as indispensable from the system, after some time they decided to create a robust chess game instead to encourage students to use the application. Therefore, tournament functionality was moved to low importance. The importance of a requirement influenced in which direction the application was developed and features were implemented first.

2.2 Functional Requirements

#	Requirements	Importance	Status
2	As a user, I want to be able to play chess so that I can play the game	CRITICAL	Done
4	As an administrator, I want to be able to create new users, so that they can play chess.	CRITICAL	Done
21	As a user, I want to join the game someone invited me to so that I can play a game with them	CRITICAL	Done
1	As a user, I want to be able to play chess with a computer so that I can practice chess with an opponent with customizable difficulty.	HIGH	Done
6	As a user, I want to be able to spectate over shared chess games.	HIGH	Done

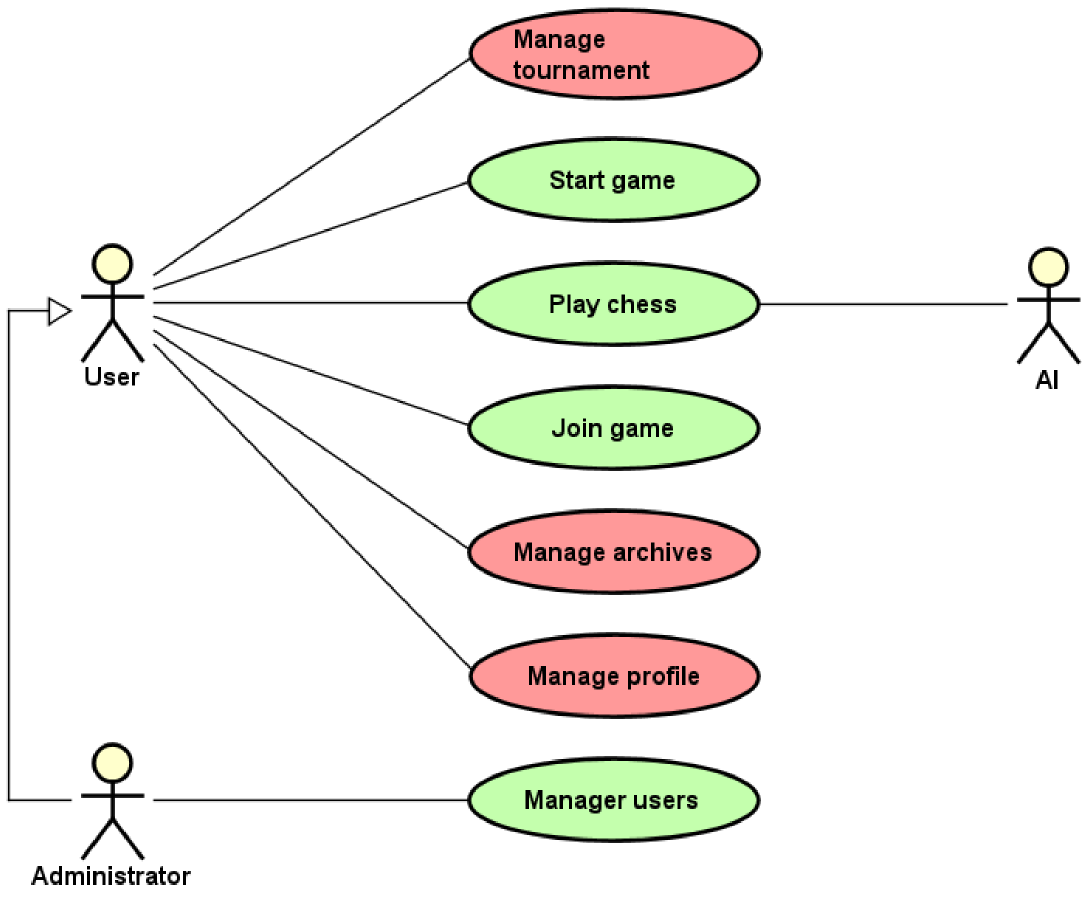
7	As a user, I want to be able to message another player, so that I can keep in touch with new connections and discuss game-related subjects.	HIGH	Done
8	As a user, I want to be able to choose a side, assign sides, and for the game to start.	HIGH	Done
9	As a user, I want to choose a time control for the game so that I can play the game under time constraints.	HIGH	Done
11	As a user, I want to change the visibility of my game, so that I could change who can participate.	HIGH	To Do
17	As a user, I want to be able to play with a random player, so that I can play a game.	HIGH	Done
18	As a user, I want to be able to invite people to play by their username, so that I can play with friends.	HIGH	Done
22	As a user, I want to see all open games, so that I can join or spectate them.	HIGH	Done
12	As a user, I want to be able to archive games after they have been played, so that I can analyze them later.	LOW	Done
15	As a user, I want to be able to resign, so that the game doesn't drag out inconveniently.	LOW	Done
16	As a user, I want to be able to offer a draw, so that in case of dead-end situations, both players can come out of the game without losing.	LOW	Done
19	As a user, I want to see my statistics so I can tell how good of a player I am.	LOW	To Do
3	As a user, I want to join a tournament I was invited to, so I can compete with other people	LOW	To Do

5	As a user, I want to be able to see my game history, so that I can keep track of the games and tournaments I have participated in.	LOW	To Do
14	As a user, I want to access an archived game to analyze my moves so that I can improve my chess skills	LOW	To Do
10	As a user, I want to create a new tournament with a name, that can be opened or invite-only so that I can compete with other users	LOW	To Do

2.3 Non-Functional Requirements

#	**Requirement**	**Status**
1	The system should only allow authenticated users with an email and a password to use the functionality of the system.	Done
2	The system should have a responsive GUI, which is usable on phones	Done
3	The primary interface language should be English	Done

2.4 Use Case Diagram



2. Use Case Diagram

Requirements were combined into several use cases. This diagram shows the actors and their possible interactions with the system. Users can start games against other players or AI, join games, and play chess. The AI actor is a chess AI opponent that users can play against. Administrators can manage the users of the system. Note that the use cases marked red: manage profile, manage archives and manage tournament use cases, haven't been designed and implemented and only have a brief analysis.

2.5 Use Case Description - Play Chess

Play chess is the first and core use case analyzed while developing the distributed chess application. Without a doubt, being able to play chess is the most important part of this system, and was developed closely with a product owner. Below is a description that is an artifact of these consultations.

All use case descriptions can be found in Appendix A

Brief Description

A user plays a game of chess against another user or AI. During the game, the users can communicate with each other via live chat. The users can offer a draw or resign if they choose to. Once the game finishes, the user is presented with options to create a new game, request a rematch, quit playing, or find a new game.

Detailed Description

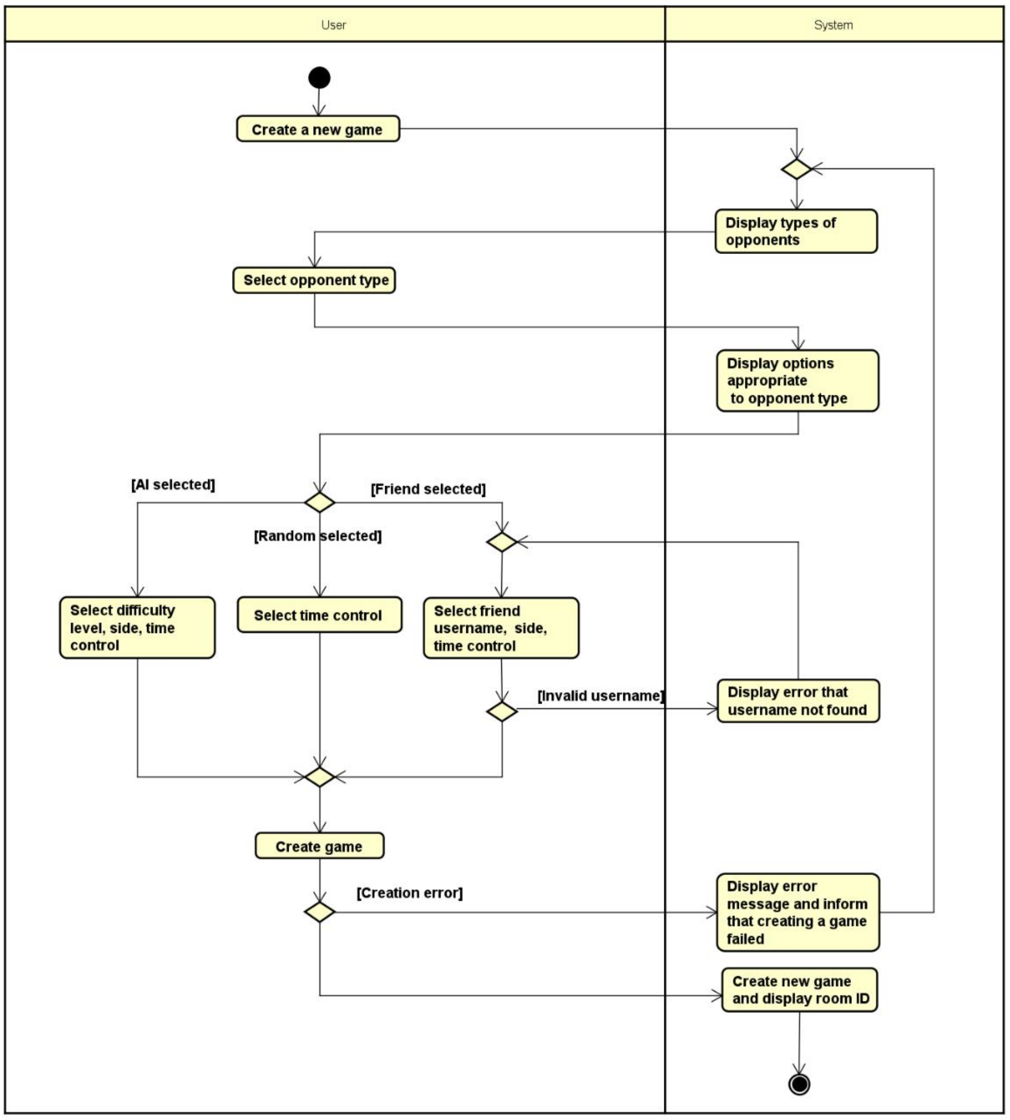
U2	**Play chess**	
Summary	The user plays a game of chess.	
Actors	User	
Precondition	The user has already logged in and joined the game.	
Postcondition	The user has finished the game and is offered further options by the system.	
Base Sequence	Actor	System
	Play a move:	
	1. User chooses a piece to move.	2. System shows available moves for the piece.
	3. User makes a move.	4. System shows the move has been played.
		5. System displays the opponent's move.

	Go to: Play a move	
Extensions	Actor	System
	*.a. User offers a draw:	
		1. System notifies the offer has been sent.
		2. System informs the users about the result of the draw offer.
	*.b. The user receives a draw offer:	
		1. System shows the draw offer.
	2. Player accepts, declines, or doesn’t respond to the draw offer.	3. System notifies about the result of the draw offer.
	*.c. User resigns:	
		Go to: End of the game
	*.d. Timer runs out:	
		Go to: End of the game
	*.e. User makes a move that ends the game:	
		Go to: End of the game
	*.f. User writes a message in the chat:	
		1. System displays the new message in the chat.
	End of the game:	
		1. System informs about the end of the game.

			2. System shows available options the player can choose from (rematch, new game, quit, join a new game).
Exception Sequence	Actor	System	
	Play a move - Invalid move:		
		4.g. System discards the move.	

2.6 Activity Diagram - Start Game

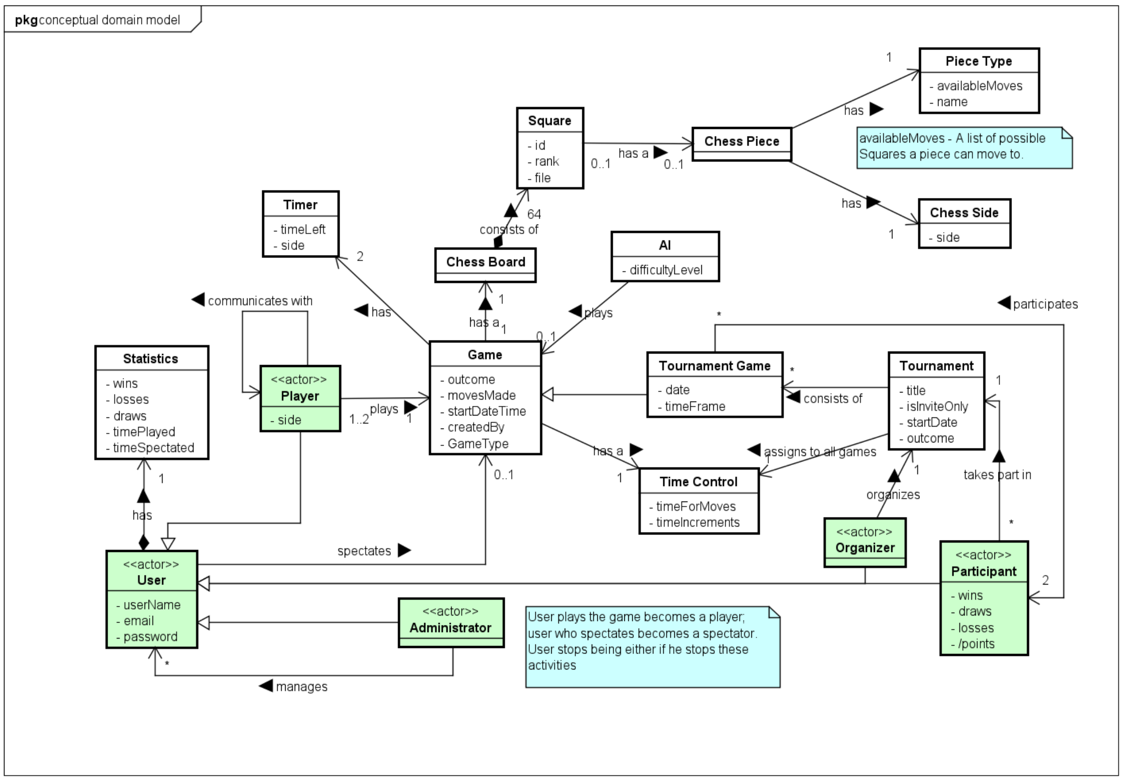
Start Game is a core use case that was just as important to analyze thoroughly. To accommodate the varying preferences and playing styles of users, the ability to customize game options was included in the requirements. The options are available for customization depending on the opponent chosen for the created game. To clarify under which conditions certain options are customizable, an activity diagram was created. This diagram outlines the steps involved in customizing game options, ensuring that the developed system allows users to tailor their game experience to their preferences.


User
System
Create a new game
Display types of opponents
Select opponent type
Display options appropriate to opponent type
[AI selected]
[Friend selected]
[Random selected]
Select difficulty level, side, time control
Select time control
Select friend username, side, time control
[Invalid username]
Display error that username not found
Create game
[Creation error]
Display error message and inform that creating a game failed
Create new game and display room ID

3. Start Game Activity Diagram

When the user chooses to start a new game, the user must first choose an opponent type, which can be AI, friend, or random. If the user is playing against AI, then it is possible to choose a difficulty, if the opponent type is a friend, then the user can enter the friend's username that the user would play against, it is only possible to choose a friend who is part of the system. If the user is playing a random opponent the side is chosen randomly, otherwise, it is possible to choose a side. Time control can be selected for any opponent type. After the game options are set, the user can choose to start the game and the system will create the game with the selected options.

2.7 Domain Model



4. Domain Model

The domain model that resulted from analyzing the problem domain consists of several actors. At the center of the model is the game, which is the main focus of the system. Each game has its chessboard and timers and is played by one or two players. In the case of a game with an artificial intelligence (AI) game type, an AI player will participate in the game alongside a player.

Users are the base actors in the domain model, from which all actors inherit from. They can create and participate in games, as well as spectate games of their choosing. Users can also create tournaments, becoming this way organizers.

A tournament consists of multiple tournament games, which have predefined time controls and time frames for play. Users who participate in a tournament are known as participants.

The chessboard is a vital part of the game and consists of squares on which game pieces can be placed. Pieces have to be moved and placed according to the rules and mechanics of a chess game.

2.8 Security analysis

The purpose of this section is to provide a security assessment of a heterogeneous chess system. The main focus was put on analyzing possible threats, stating objectives, and presenting a risk assessment model that will allow to act and enforce relevant security measures.

Overview

First, it’s important to analyze the system at hand. In short, the system consists of 4 servers:

- Blazor server, which provides users with WASM that allows them to interact with the system through a GUI in a browser
- Communication server, which handles connections between users and acts as a gateway to other system services
- “Stockfish” Server, which acts as a microservice for generating AI moves for the chess game
- Database Access Server (DAS in short), which stores critical user information for authentication and game statistics in databases.

Objectives

- Keep User data (email, passwords) confidential and unavailable to unauthorized individuals
- Keep ser statistics and data regarding application usage private according to the user’s wishes.
- Ensure the integrity of information sent between a user and a server controlling the game and authenticate all actions appropriately. (Only a player in a game can make a move, resign or offer a draw in a game)
- Maintain integrity and confidentiality of information transferred between communication and database access servers

- Protect the availability of the System’s services, such as AI moves provider and communication handler, and allow access to authorized users
- The system must adhere to EU GDPR privacy rules and regulations. Read more at source: (Privacy Shield, 2016)

The threat model was created using system analysis and design artifacts, such as use cases, architecture diagrams, and data schemas. STRIDE and EINOO methods aided in the creation of a threat model. STRIDE is a commonly used approach for identifying potential threats to a system, and EINOO is a method for assessing the potential impact of these threats on the organization.

The risks to the chess web application were assessed using a combination of threat frequency and incident consequence. Threat frequency refers to the likelihood that a particular threat will be realized, while incident consequence refers to the potential impact of the threat on the organization. By combining these two factors, the team was able to determine the overall level of risk for each threat.

Formula:

Criticality (C) = Frequency (F) * Effect (E)

Threats and Risk assessment

Goal	**Attacker**	**Means**	**Description**	**E**	**F**	**C**
Spoofing identity	External, Network	Eavesdropping	An attacker intercepts the login credentials of a user and uses them to access the system, possibly between the user and comm server connection, as well as between the comm and DAS servers.	3	5	15
	External, Offline	Unsecure endpoint access	An attacker connects directly to the DAS server through REST controllers and requests the user’s data. Requires in-depth knowledge of a system's inner workings.	5	2	10
	External, Network	Phishing	An attacker tricks users with phishing tactics, such as creating a mock version of the web application into giving their login credentials and gaining access to the system.	1	3	3
Tampering	External, Network	Replay attack	An attacker captures a request of a user to act on the system and modifies it, violating the integrity of system data	2	3	6
	External, Network	Message Modification	An attacker modifies messages coming between the comm and das server, altering game outcomes, resulting in integrity violations and improved statistics	2	2	4
	External, Offline	Malicious code injection	An attacker injects malicious code into an improperly secured Blazor server, which will serve it	5	1	5

			to users. This results in an array of security threats for the users, which might not impact the system integrity itself.			
	Internal	Illegal Inputs	User exploits unchecked inputs to alter the system to their favor, for example by moving the opponent’s pieces	1	5	5
	Internal	Overflow of inputs	A user creates credentials too long for the system to handle which might cause unforeseen consequences and bugs in the system	1	4	4
Repudiation	Internal	Denial of action	The user denies acting, which resulted in unfavorable results for the user, for example by blaming the action on external attacks. (ex. Lost game, impolite message sent to other users)	1	2	2
Information disclosure	External, Network	Eavesdropping	An attacker listens to messages sent between the communication server and the DAS server, which contain user-sensitive data, such as login, passwords, and usage patterns	4	4	16
	External, Online	Masquerade	An attacker connects to the DAS server and sends messages masked as ones coming from the communication server, gaining access to all data within the system	5	1	5
	External, Network	Eavesdropping	An attacker listens to information sent to the Stockfish server, which potentially exposes private, but not sensitive data such as game moves and overall game performance	1	3	3
	External, Network	CPA	Due to the known format of messages, (ex. using FEN and UCI notation for sending moves), the attacker breaks the	5	1	5

			encryption key by capturing an encrypted message and guessing a plain text equivalent (ex. Many chess games start with e2e4 move). Breaking the encryption is a serious threat.			
Denial of Service	External, Network	Message overflow	An attacker overflows the network with irrelevant messages that eventually block access to systems services. Can be performed on any server, with varying degrees of impact. DoS attack on Stockfish service will impact only AI games, but an attack on DAS, comm and Blazor servers would render the whole system inoperable.	4	1	4
Elevation of privileges	Internal, Online	Message Modification	An attacker modifies messages sent by a system administrator for creating users and modifies their role in the system.	4	3	12

Attacks that expose user data have higher Criticality ratings due to legal and image consequences of the leaks for the company. Attacks such as malicious code injection and masquerade are of grave danger to the system functionality but require inside knowledge of a system. DoS attacks are difficult to shield against, but the system's inoperability is of little value for the attackers.

Preventive Measures

To prevent potential security incidents, implementing the following measures is recommended:

- Secure connections between critical servers to prevent eavesdropping, which exposes to a multitude of network threats

- Authenticate each user that has access to the system. Strengthen authentication mechanisms to prevent unauthorized access to the system and its data.
- Validate user input server side to prevent illegal inputs and unstable states of the application, which could throw the system offline.
- Secure REST and gRPC endpoints in all servers to prevent unauthorized access to information. gRPC connection in this system carries technically insensitive data but still carries a privacy risk as a user might not want to share their games with external entities.
- Limit privileges of creating an account to administrators only. The system is used only by a limited number of users (students on a university campus) and users already have university-assigned emails. That would prevent the creation of false accounts and prevent internal attacks on the system with the goal of repudiation.
- Educate users of phishing schemes that could expose their sensitive data
- Log every action in the system to detect suspicious behavior

Corrective Measures

In the event of a security incident, following corrective measures are advised:

- Incident system shut down: In case of a data breach, plan a response plan that will involve searching through logs for unauthorized access and shutting down the system to prevent a further data breach
- Notification and communication: Notify users, as soon as possible after the incident has been detected. Keep them up to date about what data was exposed and alarm the law enforcement
- Implement a data recovery plan to minimize downtime.

- Identify the cause of the security breach identify previously not analyzed **T**hreats that emerged, correct security **P**olicies, and update security **M**echanisms that were circumvented. (TPM)

Security analysis conclusion

Overall, the chess web application faces a range of security risks that must be addressed due to its distributed nature. It is important to implement preventive and corrective measures, as mishandling user data is a grave threat. In the next chapter methods of implementing these measures are presented.

2.9 Analysis Summary

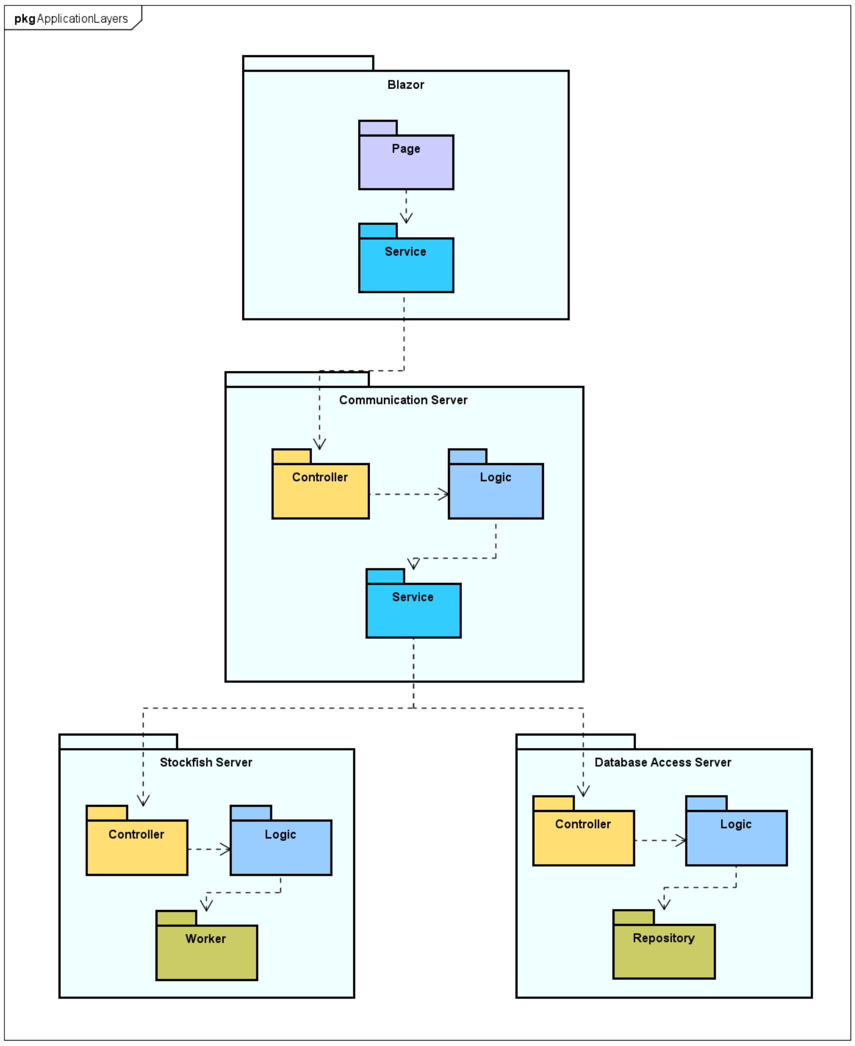
Good analysis is a promise of client satisfaction. Of course, the analysis that was created initially, changed and evolved as the client's actual needs became more apparent. A big part of developing an application is not only listening to a client’s input but helping to find what the client needs. After that comes the stage of how the client's needs should be fulfilled, which is all the design part is about.

3 Design

Due to the distributed nature of the system, it is crucial to carefully design each node and connection in the system to allow the nodes to work in efficient harmony. The design stage of the development relies on artifacts created during the analysis to create blueprints that are later used for implementation. Here below is described the chosen architecture of the system, technologies used, design patterns, and examples of artifacts created such as parts of class, sequence, and persistence diagrams, and wireframes made for UI implementation.

3.1 Architecture

The system is distributed across several servers. Because of the extra responsibilities that could be split in the system (Stockfish logic) N-Tier architecture was chosen. The N-Tier architecture allows the application to be separated into multiple tiers each with its responsibilities.



5. Layers of the Application

The application was divided into four separate systems. Blazor acts as a UI layer, with a Service that allows for exchanging information with Communication Server. The communication Server handles most of the logic of the system and also has its Controller and Service layer that allows coordinating communication with other systems. Stockfish Server was built to contain the Worker Service that generates moves for the players. Database Access Server provides access to a Persistence Layer of the entire application.

3.2 Technologies

Nodes in the Chess App utilize multiple technologies. Systems were built using C# and Java, and for each connection, a different type of communication technology was used to fully realize the potential of a heterogeneous system.

gRPC

gRPC is a Remote Procedure Call framework that was used to connect the AI service server with the communication server for getting AI to move so that resource-intensive workload is detached from a server that already handles game logic and communication between clients. gRPC is perfect for server-to-server communication, especially for this given use case.

REST

Representational state transfer (REST) controllers were created in the communication server for unary calls and database access server, which ensures reliable transfer of actions made by a client and data to be stored in a database.

SignalR

SignalR is a software library built with web sockets that were used to send asynchronous group notifications between the Blazor client and a server. While SignalR does not ensure message deliverability, it streams events to clients about data that changes quickly were skipping a few messages won't impact the application functionality (chat messages, streaming move updates).

It might seem that move updates are important to transfer reliably every time, but the idea is that the client application gets a constant stream of events that update the board, either on time passed or on move made, which ensures recovery when the message gets lost.

Blazor WebAssembly

Blazor WebAssembly was used to create a single-page application to be used as a front end for the system. The application can be accessed from popular modern browsers, and it allows for using C# libraries such as Rudzoft Chess Library**.**

Rudzoft Chess Library

Rudzoft Library provides necessary logic and entities that together build a basis for a chess application. It gives access to the Game class that controls logic and Position that handles state, as well as Pieces, Moves, Squares, etc. The library was used in both the communication server and the chess client. The communication server uses the library to handle game logic and the chess client uses it to parse sent fen notation to pieces' position on the board and to generate possible moves for a piece.

MudBlazor

MudBlazor is a component library for Blazor systems. It provides buttons, dialogs, snack bars, and app bars that are used in most of the chess client pages and components. Similarly to Bootstrap, it gives a framework on which responsive web apps can be developed.

Spring Boot

Java Spring Boot is a tool that makes developing web applications and microservices with Spring Framework faster and easier by providing auto-configuration, an opinionated approach to configuration, and the ability to create standalone applications. (IBM Cloud Education, 2020)

.NET 6

.NET is an open-source development platform for building many different types of applications. The ASP.NET framework which is provided by .NET was used to help build web applications for this project.

xUnit

xUnit is a testing framework that was used to unit test the C# logic classes.

Hibernate and JPA

Hibernate is an object–relational mapping tool which was used to map domain objects to a database.

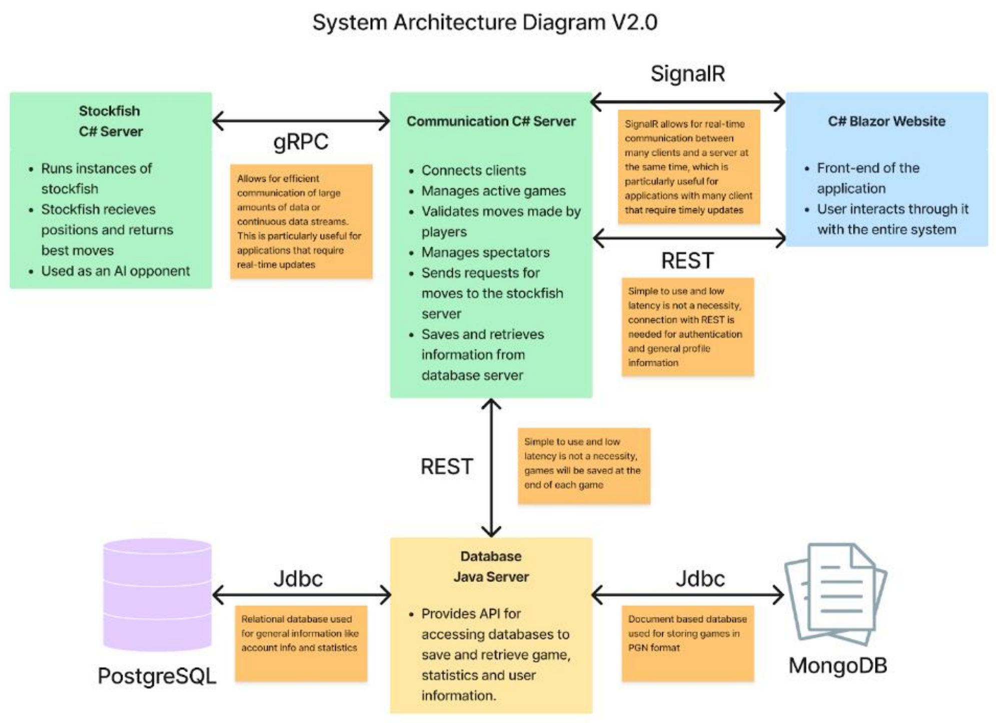
PostgreSQL

PostgreSQL is a relational database management system. It was used to store application data.

JDBC driver for PostgreSQL

The PostgreSQL Java Database Connectivity (JDBC) diver was used to connect the java application to the PostgreSQL database.

3.3 System Architecture Changes


System Architecture Diagram V2.0
Stockfish C# Server
• Runs instances of stockfish
• Stockfish recieves positions and returns best moves
• Used as an AI opponent
gRPC
Allows for efficient communication of large amounts of data or continuous data streams. This is particularly useful for applications that require real-time updates
Communication C# Server
• Connects clients
• Manages active games
• Validates moves made by players
• Manages spectators
• Sends requests for moves to the stockfish server
• Saves and retrieves information from database server
SignalR
SignalR allows for real-time communication between many clients and a server at the same time, which is particularly useful for applications with many client that require timely updates
C# Blazor Website
• Front-end of the application
• User interacts through it with the entire system
REST
Simple to use and low latency is not a necessity, connection with REST is needed for authentication and general profile information
REST
Simple to use and low latency is not a necessity, games will be saved at the end of each game
Database Java Server
• Provides API for accessing databases to save and retrieve game, statistics and user information.
Jdbc
Relational database used for general information like account info and statistics
PostgreSQL
Jdbc
Document based database used for storing games in PGN format
MongoDB

6. System Architecture Diagram

gRPC-Web -> REST and SignalR - Communication Server

In the initial phase of the collaboration phase and proof of concept, gRPC-Web was used to stream messages to clients from a communication server. This however changed in the later stages of the development, and unfortunately, gRPC-Web had to be swapped in favor of SignalR communication.

There are a few reasons why gRPC-Web was replaced with REST and SignalR as the middleware between the Blazor website and the communication server. First, SignalR is very effective when it comes to streaming data to multiple clients simultaneously. This is important in a multiplayer game where multiple players and spectators need to receive real-time updates from the server.

Additionally, REST was added to handle unary calls. REST is simple to use and understand, making it a good choice for basic CRUD operations. Furthermore, REST for unary calls was used because it ensures message delivery and will throw errors if the message was not received, unlike SignalR which does not guarantee message deliverability. This ensures that important messages are delivered and handled properly. gRPC-Web, on the other hand, is a more complex and powerful protocol that is better suited for more demanding tasks, such as streaming large amounts of data or exposing complex APIs.

In summary, gRPC-Web was replaced with REST and SignalR because SignalR is better suited to handle multiple concurrent client connections and because REST is a reliable middleware for unary calls. This combination allows us to take advantage of the strengths of both protocols to create a more efficient and effective communication system for our Blazor website and communication server.

Stockfish Server and gRPC

In the connection between the stockfish and the communication server, it was decided to use gRPC for the stockfish connection for several reasons. Firstly, gRPC is a modern, high-performance middleware technology that allows for the efficient exchange of data between client and server. It also has support for multiple programming languages and is widely used in microservices architectures. Its efficient data transfer lowers the overall latency of the system.

gRPC has several advantages when it comes to data transfer. For one, it uses HTTP/2 for transport, which allows for multiple requests and responses to be sent simultaneously over a single connection and multiplexing. This makes gRPC efficient and fast.

Additionally, gRPC uses protocol buffers for encoding data, which is more compact and efficient than JSON or XML, the data formats typically used in REST. This means that gRPC can transfer data with less overhead, resulting in faster transfer speeds and lower network usage. This suits the use case as messages are small and sent often.

Overall, gRPC in the system allows for lower latency and improves the performance of the system, and takes advantage of gRPC's efficient data transfer capabilities.

3.4 Design Patterns

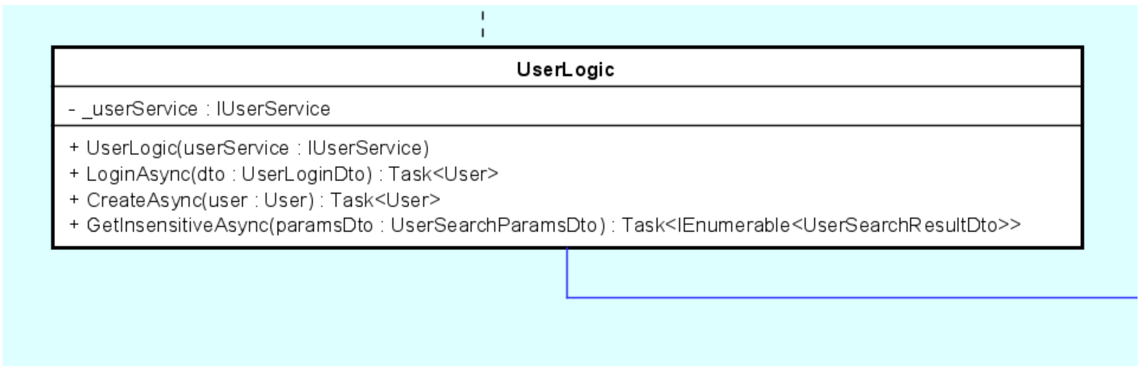
Distributed Chess Application was designed to be as maintainable, expandable, and testable as possible. Here are some examples of what design patterns were used to achieve it.

Singleton

The singleton pattern restricts the instantiation of a class to a single instance. Singletons were mainly used to avoid inconsistent data states in the client and communication server and to avoid constructing resource costly classes such as HTTP clients.

Dependency Injection

Dependency injection is a design pattern that is used to decouple the construction of classes from their usage. Using this pattern helps follow the SOLID dependency inversion principle by injecting objects into interfaces. This pattern allows one to replace the dependencies of a class without modifying it, which improves the expandability of a system. Examples of dependency injection can be found throughout the whole system in the form of constructor dependency injection. Builder injects services to classes that need them.

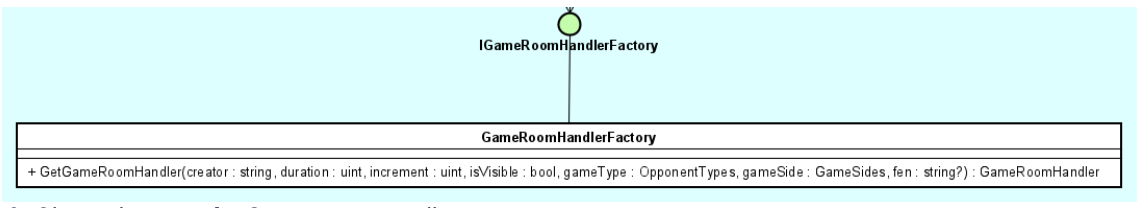

UserLogic
- _userService : IUserService
+ UserLogic(userService : IUserService)
+ LoginAsync(dto : UserLoginDto) : Task<User>
+ CreateAsync(user : User) : Task<User>
+ GetInsensitiveAsync(paramsDto : UserSearchParamsDto) : Task<IEnumerable<UserSearchResultDto>>

7. UserLogic Class Diagram

UserLogic class in Logic, Communication server has IUserService injected through a constructor.

Factory

The factory method pattern is a creational pattern that helps with creating complex objects. A class can defer creating an object to a subclass using a factory which means that the class does not need to know which object is going to implement the interface it is using. The logic that's necessary to instantiate an object is condensed into a single class.

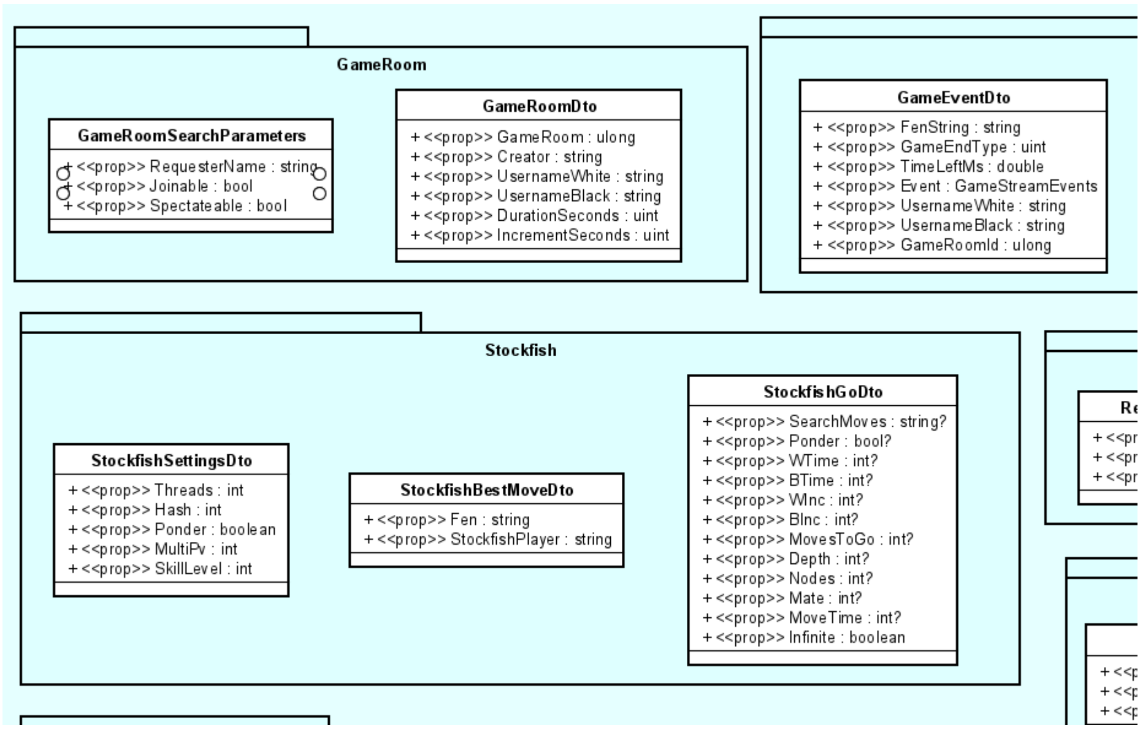

IGameRoomHandlerFactory
GameRoomHandlerFactory
+ GetGameRoomHandler(creator : string, duration : uint, increment : uint, isVisible : bool, gameType : OpponentTypes, gameSide : GameSides, fen : string?) : GameRoomHandler

8. Class Diagram of a GameRoomHandlerFactory

A notable example of a factory in the system is GameRoomHandlerFactory, which creates a GameRoomHandler that requires several dependencies. To simplify GameRoomHandler instantiation, the factory was created.

Data Transfer Object

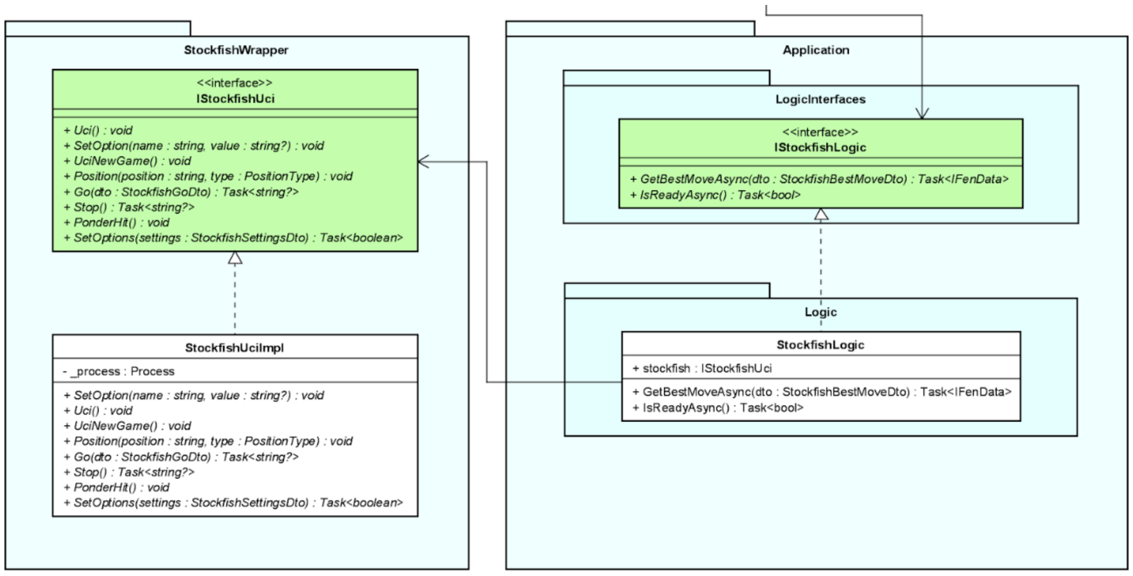
Data transfer objects improve the upgradeability of the software and allow for specific data to be sent across the app. A DTO contains no business logic and is made so that it would only have the required data for a request, which helps send only relevant information for the request. It is also used to encapsulate arguments in one class, this way removing the necessity to alter every method when there is a need to add another argument. Ex. when sending a game request, GameRequestDto is created, which stores all attributes needed to create a GameRoom. DTOs were used wherever applicable.


GameRoom
GameRoomSearchParameters
+ <<prop>> RequesterName : string
+ <<prop>> Joinable : bool
+ <<prop>> Spectateable : bool
GameRoomDto
+ <<prop>> GameRoom : ulong
+ <<prop>> Creator : string
+ <<prop>> UsernameWhite : string
+ <<prop>> UsernameBlack : string
+ <<prop>> DurationSeconds : uint
+ <<prop>> IncrementSeconds : uint
GameEventDto
+ <<prop>> FenString : string
+ <<prop>> GameEndType : uint
+ <<prop>> TimeLeftMs : double
+ <<prop>> Event : GameStreamEvents
+ <<prop>> UsernameWhite : string
+ <<prop>> UsernameBlack : string
+ <<prop>> GameRoomId : ulong
Stockfish
StockfishSettingsDto
+ <<prop>> Threads : int
+ <<prop>> Hash : int
+ <<prop>> Ponder : boolean
+ <<prop>> MultiPv : int
+ <<prop>> SkillLevel : int
StockfishBestMoveDto
+ <<prop>> Fen : string
+ <<prop>> StockfishPlayer : string
StockfishGoDto
+ <<prop>> SearchMoves : string?
+ <<prop>> Ponder : bool?
+ <<prop>> WTime : int?
+ <<prop>> BTime : int?
+ <<prop>> WInc : int?
+ <<prop>> BInc : int?
+ <<prop>> MovesToGo : int?
+ <<prop>> Depth : int?
+ <<prop>> Nodes : int?
+ <<prop>> Mate : int?
+ <<prop>> MoveTime : int?
+ <<prop>> Infinite : boolean

9. Part of the Class Diagram of multiple Data Transfer Objects

Dependency Inversion Principle

Dependency inversion is a design principle of depending on abstractions and not concretions. The higher-level objects are relying on interfaces and not objects, while the lower-level objects implement those interfaces. This inverses the flow of control and makes both high- and low-level modules depend on abstractions.

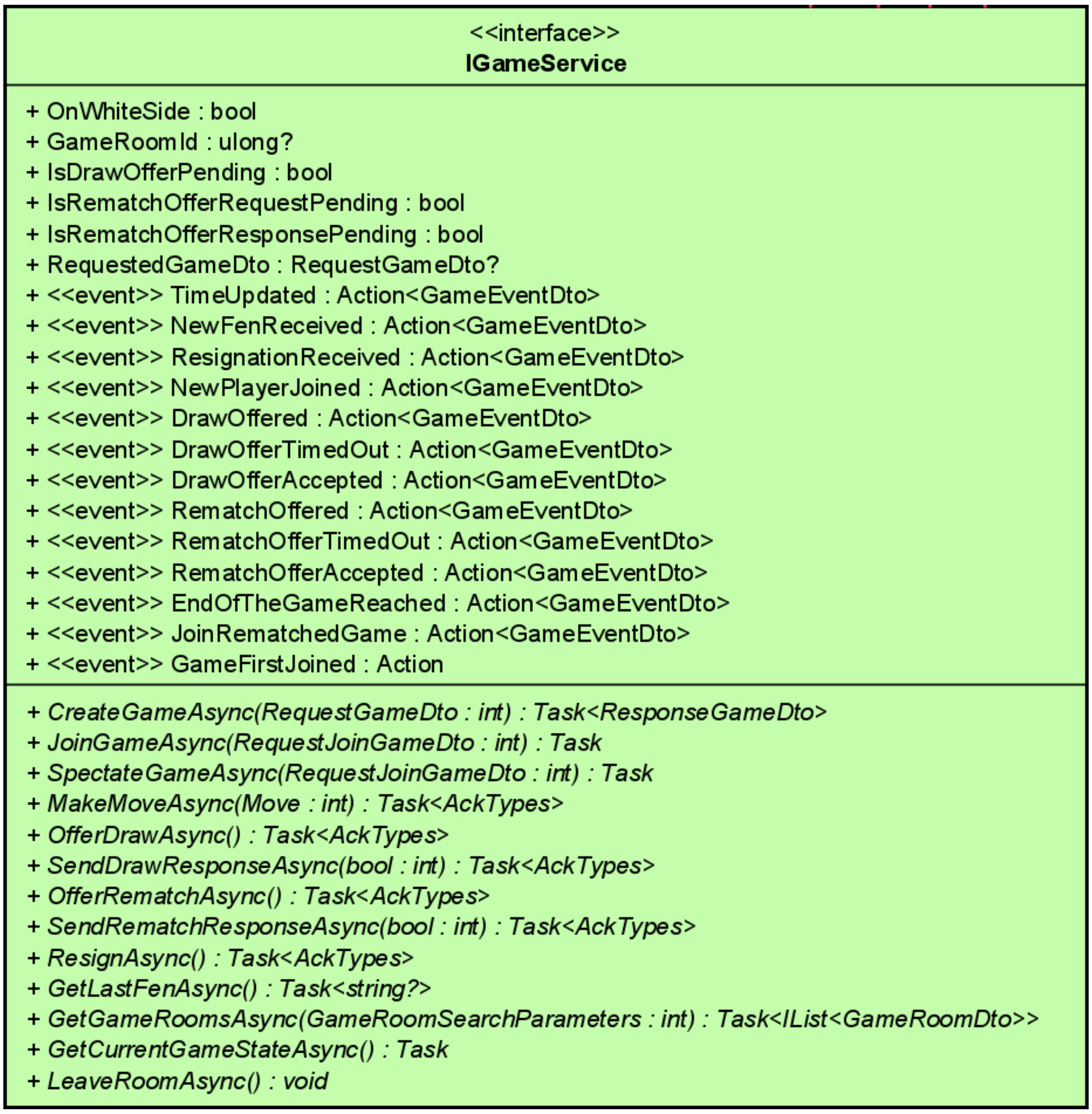

StockfishWrapper
<<interface>>
IStockfishUci
+ Uci() : void
+ SetOption(name : string, value : string?) : void
+ UciNewGame() : void
+ Position(position : string, type : PositionType) : void
+ Go(dto : StockfishGoDto) : Task<string?>
+ Stop() : Task<string?>
+ PonderHit() : void
+ SetOptions(settings : StockfishSettingsDto) : Task<boolean>
StockfishUciImpl
- _process : Process
+ SetOption(name : string, value : string?) : void
+ Uci() : void
+ UciNewGame() : void
+ Position(position : string, type : PositionType) : void
+ Go(dto : StockfishGoDto) : Task<string?>
+ Stop() : Task<string?>
+ PonderHit() : void
+ SetOptions(settings : StockfishSettingsDto) : Task<boolean>
Application
LogicInterfaces
<<interface>>
IStockfishLogic
+ GetBestMoveAsync(dto : StockfishBestMoveDto) : Task<IFenData>
+ IsReadyAsync() : Task<bool>
Logic
StockfishLogic
+ stockfish : IStockfishUci
+ GetBestMoveAsync(dto : StockfishBestMoveDto) : Task<IFenData>
+ IsReadyAsync() : Task<bool>

10. Class Diagram of Stockfish System layers illustrating DIP

Here is an example of DIP usage. StockfishLogic depends on the interface IStockfishUci (UCI stands for Universal Chess Interface). Therefore, if there was a need to replace Stockfish with another AI, the logic layer would not have to be altered. This improves flexibility and loosens the coupling of the system.

Observer

The usage of the observer design pattern means that lower-level components can notify higher-level components without depending on them. This allows for loose coupling which improves the testability and maintainability of the code.


<<interface>>
IGameService
+ OnWhiteSide : bool
+ GameRoomId : ulong?
+ IsDrawOfferPending : bool
+ IsRematchOfferRequestPending : bool
+ IsRematchOfferResponsePending : bool
+ RequestedGameDto : RequestGameDto?
+ <<event>> TimeUpdated : Action<GameEventDto>
+ <<event>> NewFenReceived : Action<GameEventDto>
+ <<event>> ResignationReceived : Action<GameEventDto>
+ <<event>> NewPlayerJoined : Action<GameEventDto>
+ <<event>> DrawOffered : Action<GameEventDto>
+ <<event>> DrawOfferTimedOut : Action<GameEventDto>
+ <<event>> DrawOfferAccepted : Action<GameEventDto>
+ <<event>> RematchOffered : Action<GameEventDto>
+ <<event>> RematchOfferTimedOut : Action<GameEventDto>
+ <<event>> RematchOfferAccepted : Action<GameEventDto>
+ <<event>> EndOfTheGameReached : Action<GameEventDto>
+ <<event>> JoinRematchedGame : Action<GameEventDto>
+ <<event>> GameFirstJoined : Action
+ CreateGameAsync(RequestGameDto : int) : Task<ResponseGameDto>
+ JoinGameAsync(RequestJoinGameDto : int) : Task
+ SpectateGameAsync(RequestJoinGameDto : int) : Task
+ MakeMoveAsync(Move : int) : Task<AckTypes>
+ OfferDrawAsync() : Task<AckTypes>
+ SendDrawResponseAsync(bool : int) : Task<AckTypes>
+ OfferRematchAsync() : Task<AckTypes>
+ SendRematchResponseAsync(bool : int) : Task<AckTypes>
+ ResignAsync() : Task<AckTypes>
+ GetLastFenAsync() : Task<string?>
+ GetGameRoomsAsync(GameRoomSearchParameters : int) : Task<IList<GameRoomDto>>
+ GetCurrentGameStateAsync() : Task
+ LeaveRoomAsync() : void

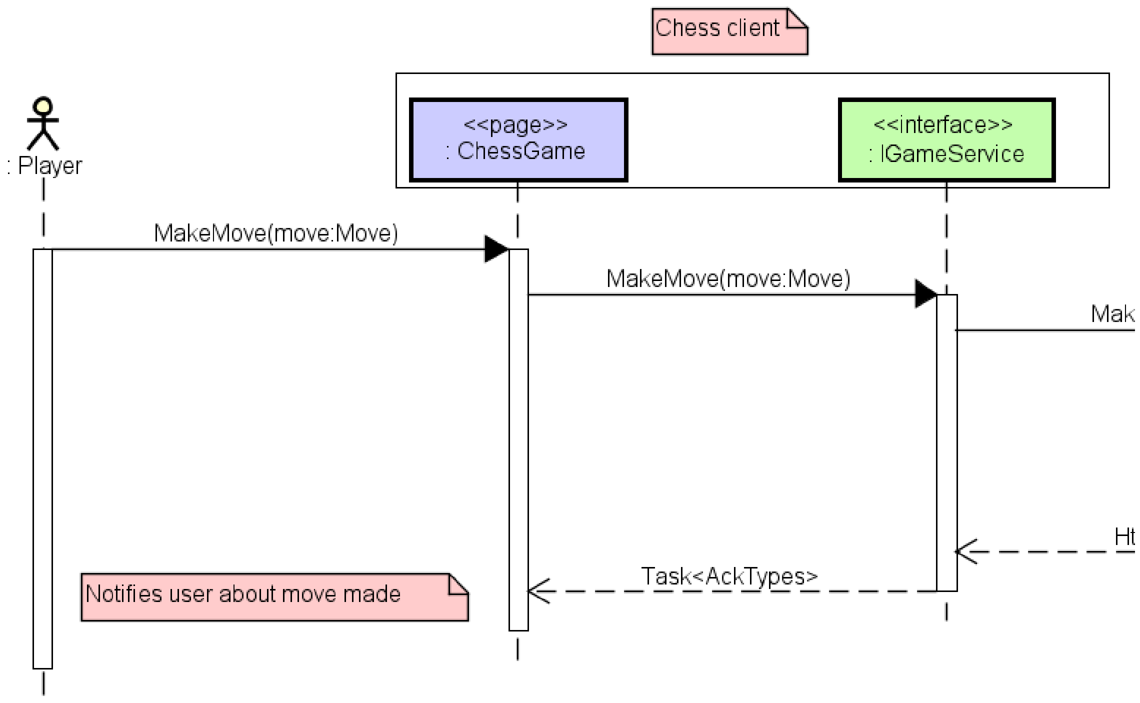
11. IGameService Interface with Multiple Events

To allow for a UI and Service to be loosely coupled, events were utilized for every event that has been sent from the communication server. There is no need for bidirectional dependency between UI and a service, which allows reusing the service in, for example, a console application.

3.5 Sequence Diagram - Make Move against AI

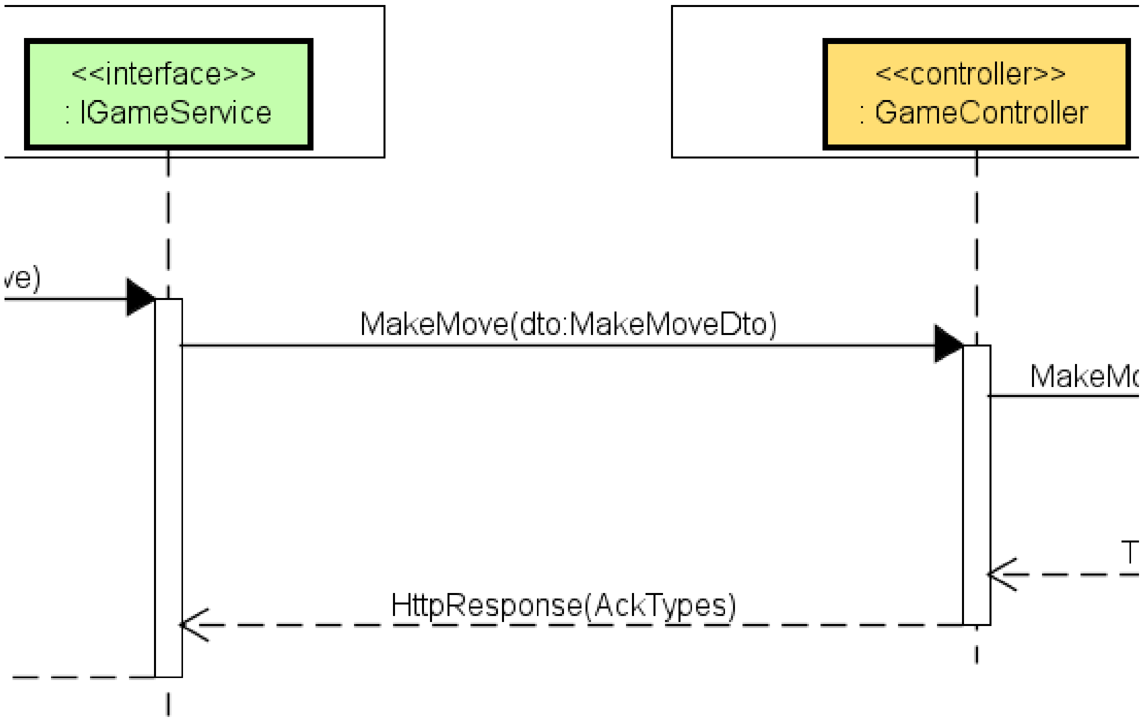
Several design tools were used to illustrate how the system should be implemented. A few sequence diagrams were created to help understand how methods between systems should interact with each other. This sequence diagram shows what parts of the system are used when a user makes a move while playing against an AI opponent.

The full sequence diagram in appendix B



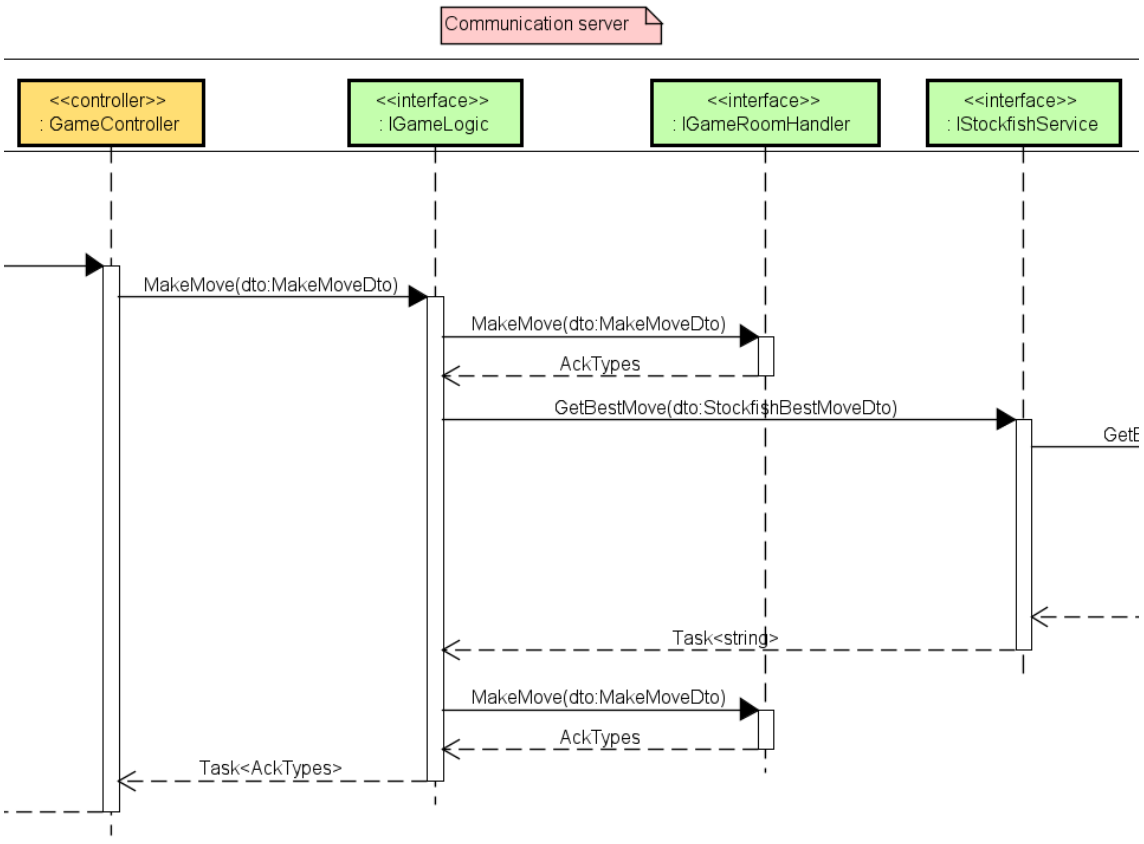
12. Chess Client part of Make Move Sequence Diagram

A player makes a move by choosing a new square for a piece on a page. Then the chess game page calls the IGameService to make the move. When the acknowledgment of the move is returned the page shows a notification about the success or failure of making the move.



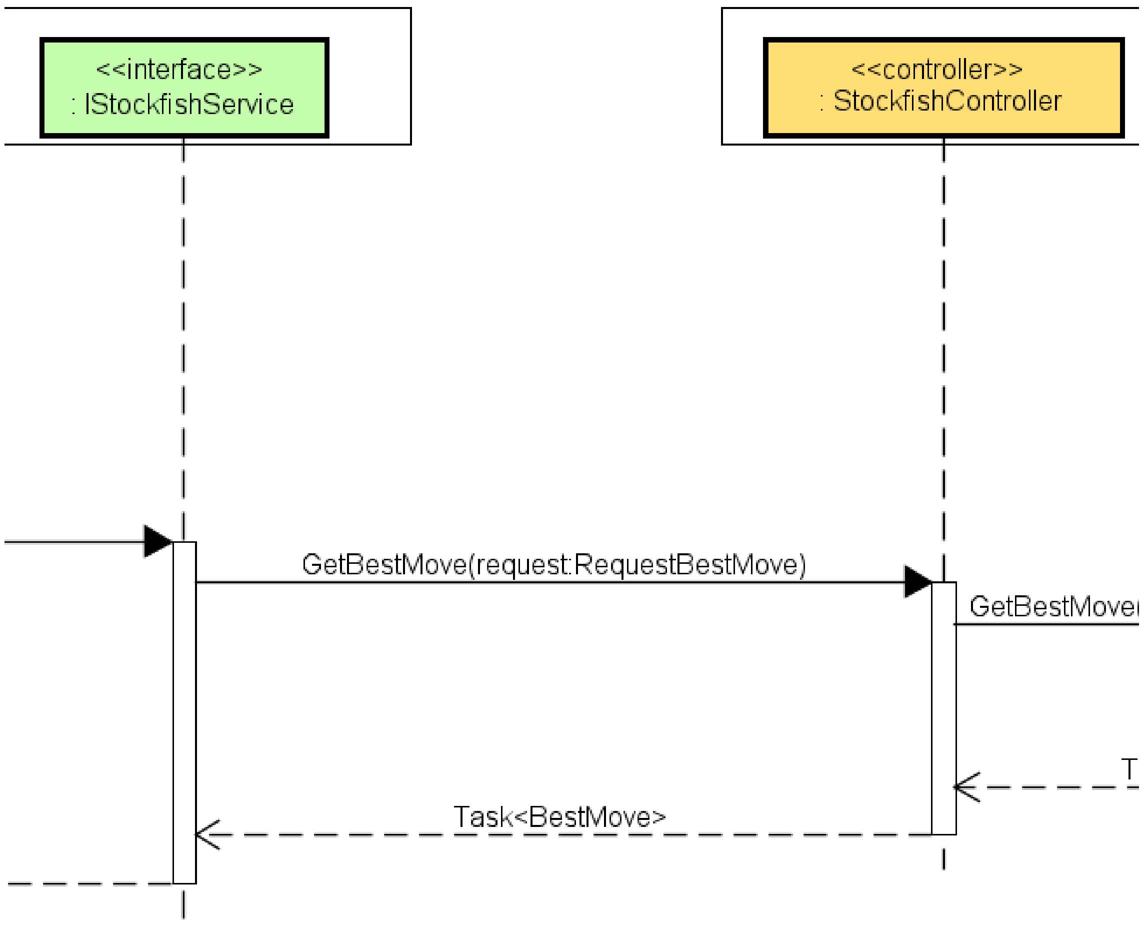
13. Chess Client and Communication Server Connection

The IGameService then takes the move data and calls the GameController on the Communication server. The connection is made using REST so the response message will take the form of HttpResponse.



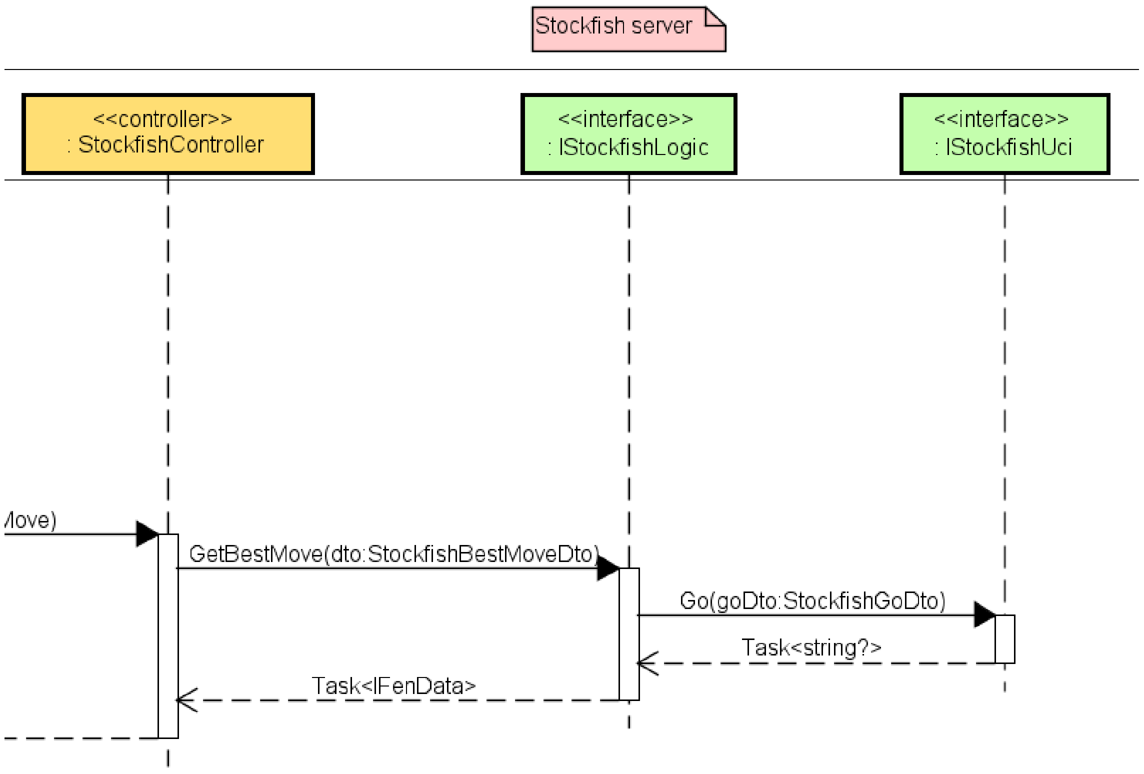
14. Communication Server part of Make Move Sequence Diagram

When the GameController receives a move, it calls the IGameLogic to process the move. In IGameLogic the move is made on the specific game room the user is playing in. Because the user is playing against an AI, IGameLogic calls IStockfishService to receive the AI opponent's move. After it receives the move from IStockfishService the move is made in the same game room and then the acknowledgment is returned to the user.



15. Communication Server and Stockfish Server Connection

IStockfishService calls the StockfishController in the stockfish server to receive the ai move. The connection is made through gRPC, so BestMove is awaited through a task.



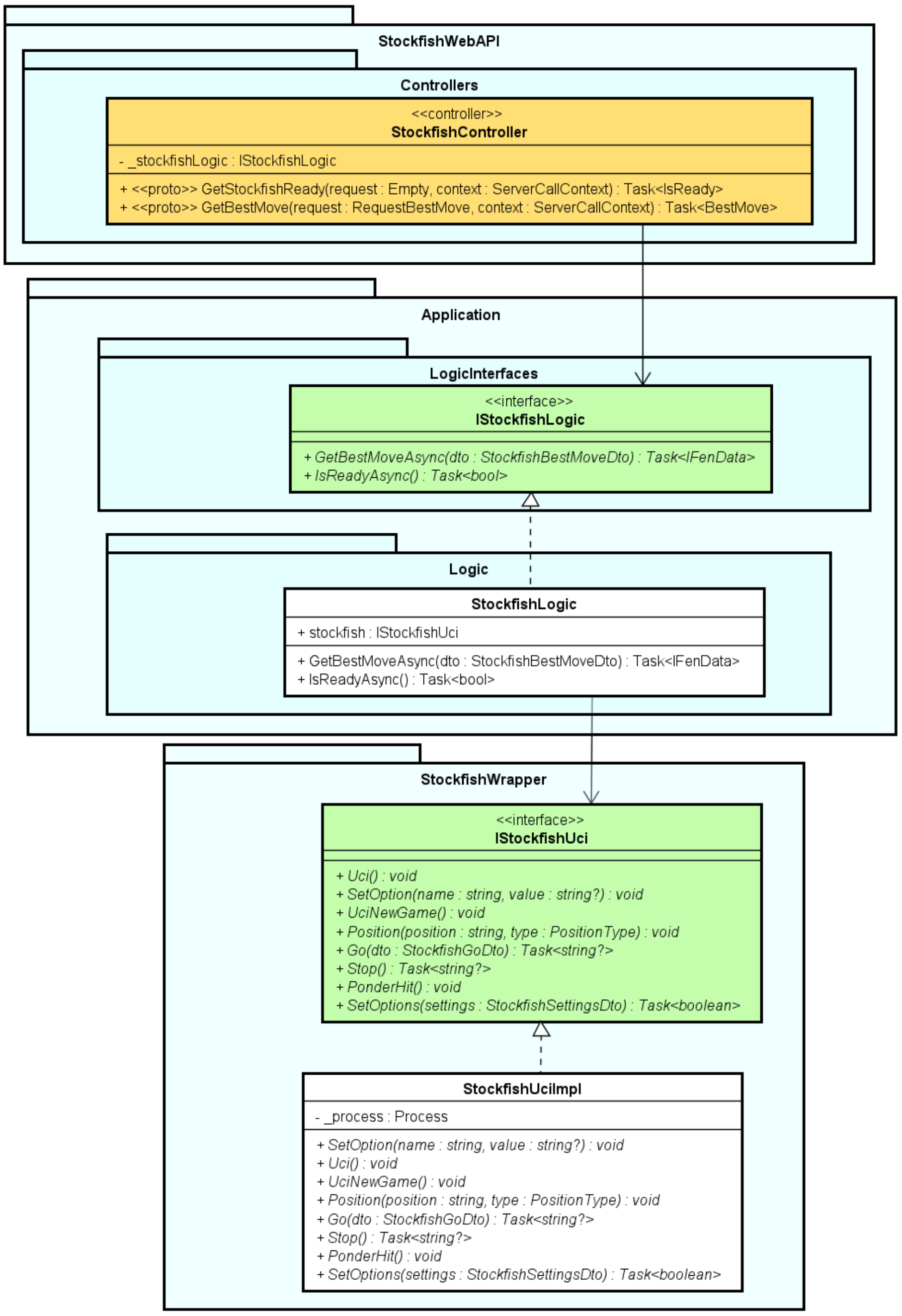
16. Stockfish Server part of Make Move Sequence Diagram

When the StockfishController receives a request for an AI move it calls IStockfishLogic. In IStockfishLogic the request is validated and then sent to IStockfishUci. IStockfishUci then generates the move using stockfish and returns it.

3.6 Class Diagrams

Most of the functionality was first designed in a global class diagram. Having an overview of the entire system was crucial, as incorrect design decisions were almost immediately noticed. Here are some parts of the class diagram. The Global Class Diagram can be found in Appendix B

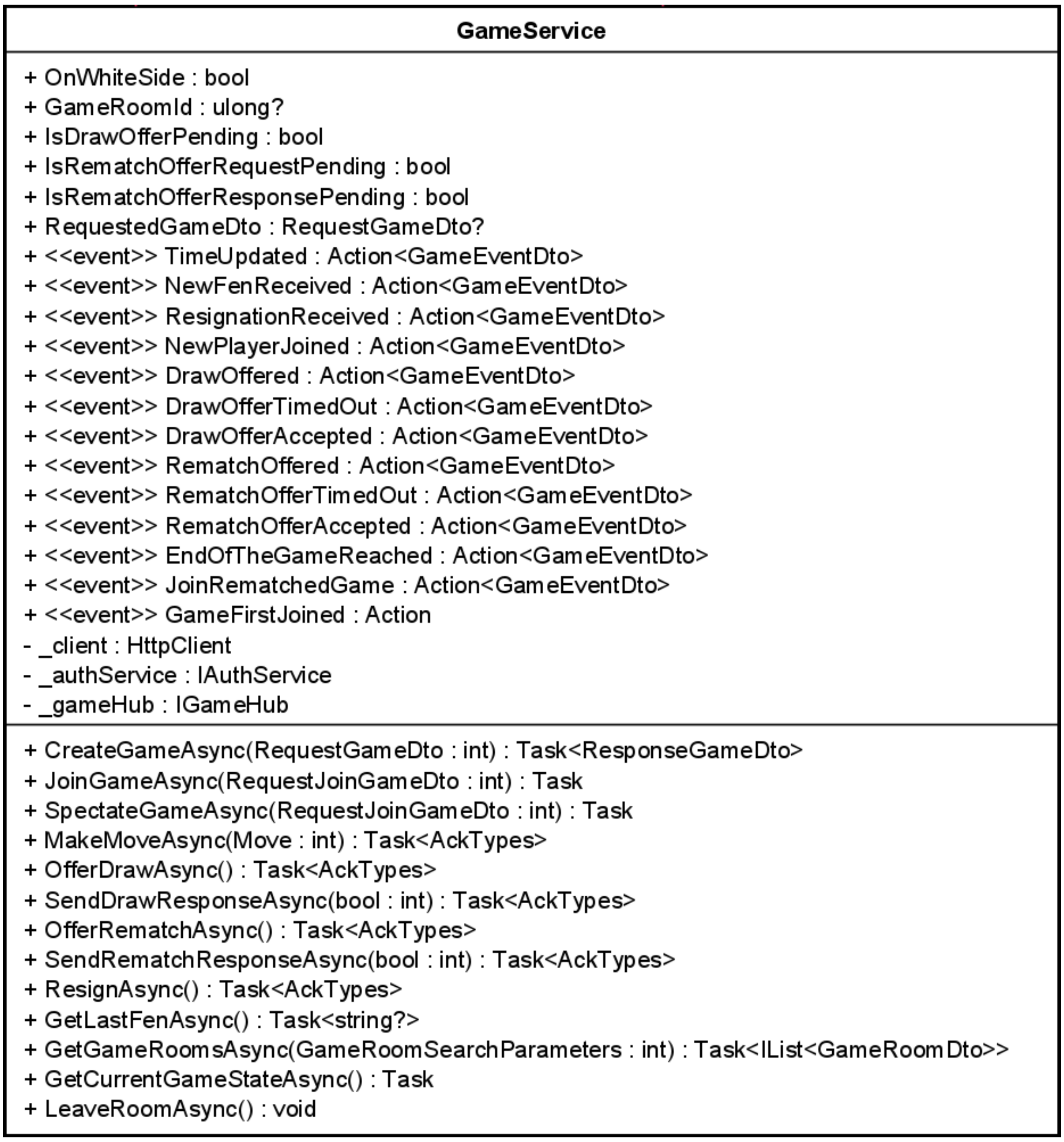
Stockfish Server


StockfishWebAPI
Controllers
<<controller>>
StockfishController
- _stockfishLogic : IStockfishLogic
+ <<proto>> GetStockfishReady(request : Empty, context : ServerCallContext) : Task<IsReady>
+ <<proto>> GetBestMove(request : RequestBestMove, context : ServerCallContext) : Task<BestMove>
Application
LogicInterfaces
<<interface>>
IStockfishLogic
+ GetBestMoveAsync(dto : StockfishBestMoveDto) : Task<IFenData>
+ IsReadyAsync() : Task<bool>
Logic
StockfishLogic
+ stockfish : IStockfishUci
+ GetBestMoveAsync(dto : StockfishBestMoveDto) : Task<IFenData>
+ IsReadyAsync() : Task<bool>
StockfishWrapper
<<interface>>
IStockfishUci
+ Uci() : void
+ SetOption(name : string, value : string?) : void
+ UciNewGame() : void
+ Position(position : string, type : PositionType) : void
+ Go(dto : StockfishGoDto) : Task<string?>
+ Stop() : Task<string?>
+ PonderHit() : void
+ SetOptions(settings : StockfishSettingsDto) : Task<boolean>
StockfishUciImpl
- _process : Process
+ SetOption(name : string, value : string?) : void
+ Uci() : void
+ UciNewGame() : void
+ Position(position : string, type : PositionType) : void
+ Go(dto : StockfishGoDto) : Task<string?>
+ Stop() : Task<string?>
+ PonderHit() : void
+ SetOptions(settings : StockfishSettingsDto) : Task<boolean>

17. Stockfish Server Class Diagram

This class diagram shows the structure of the Stockfish server. The StockfishController is responsible for receiving requests. The gRPC controller has two endpoints. One to check if the AI is running and the other one to generate a move. After the controller receives the request, it is then sent to IStockfishLogic. The IStockfishLogic is responsible for validating the request, setting the difficulty for the AI that the user specified in the request, and then requesting IStockfishUci to generate a move. IStockfishUci is responsible for communicating with the chess engine using the Universal Chess Interface (UCI) to generate moves.

GameService

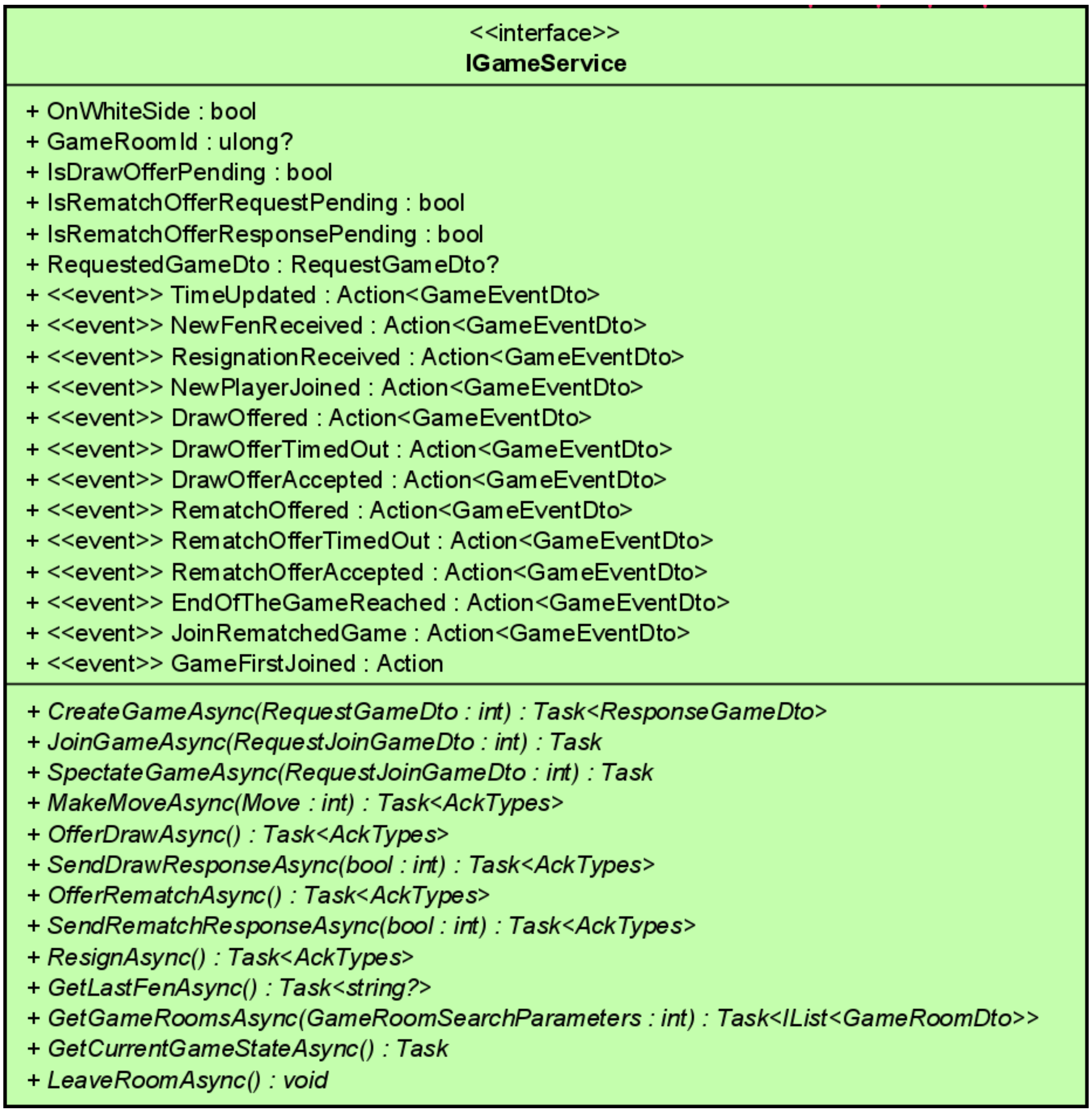


```
GameService
---
+ OnWhiteSide : bool
+ GameRoomId : ulong?
+ IsDrawOfferPending : bool
+ IsRematchOfferRequestPending : bool
+ IsRematchOfferResponsePending : bool
+ RequestedGameDto : RequestGameDto?
+ <<event>> TimeUpdated : Action<GameEventDto>
+ <<event>> NewFenReceived : Action<GameEventDto>
+ <<event>> ResignationReceived : Action<GameEventDto>
+ <<event>> NewPlayerJoined : Action<GameEventDto>
+ <<event>> DrawOffered : Action<GameEventDto>
+ <<event>> DrawOfferTimedOut : Action<GameEventDto>
+ <<event>> DrawOfferAccepted : Action<GameEventDto>
+ <<event>> RematchOffered : Action<GameEventDto>
+ <<event>> RematchOfferTimedOut : Action<GameEventDto>
+ <<event>> RematchOfferAccepted : Action<GameEventDto>
+ <<event>> EndOfTheGameReached : Action<GameEventDto>
+ <<event>> JoinRematchedGame : Action<GameEventDto>
+ <<event>> GameFirstJoined : Action
- _client : HttpClient
- _authService : IAuthService
- _gameHub : IGameHub
---
+ CreateGameAsync(RequestGameDto : int) : Task<ResponseGameDto>
+ JoinGameAsync(RequestJoinGameDto : int) : Task
+ SpectateGameAsync(RequestJoinGameDto : int) : Task
+ MakeMoveAsync(Move : int) : Task<AckTypes>
+ OfferDrawAsync() : Task<AckTypes>
+ SendDrawResponseAsync(bool : int) : Task<AckTypes>
+ OfferRematchAsync() : Task<AckTypes>
+ SendRematchResponseAsync(bool : int) : Task<AckTypes>
+ ResignAsync() : Task<AckTypes>
+ GetLastFenAsync() : Task<string?>
+ GetGameRoomsAsync(GameRoomSearchParameters : int) : Task<IList<GameRoomDto>>
+ GetCurrentGameStateAsync() : Task
+ LeaveRoomAsync() : void
```

18. GameService Class

This class diagram shows the methods and fields that the GameService class holds. GameService is responsible for everything related to the game. This class could have been divided into separate classes for each type of responsibility related to the game, but it was decided to keep it in the current state because all objects created would need to share the state. In hindsight, the class has grown to a size larger than expected and should have been divided. To make the implementation simpler and less error-prone all pages use the same implementation of the GameService.

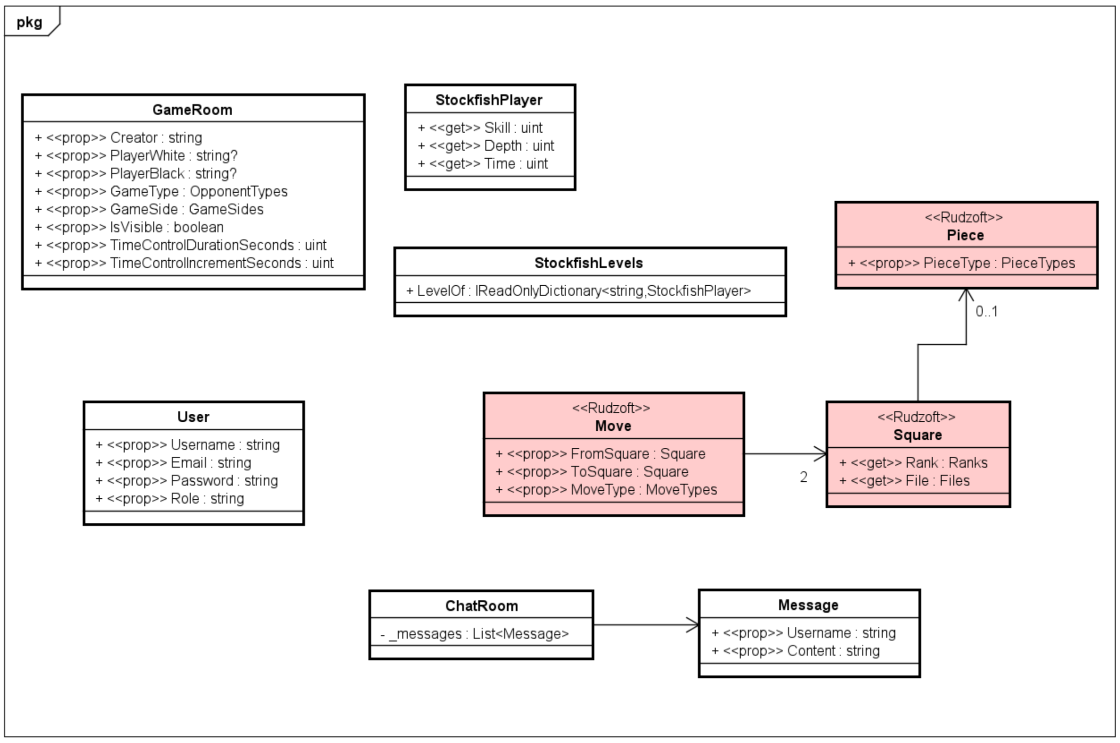
IGameService


<<interface>>
IGameService
+ OnWhiteSide : bool
+ GameRoomId : ulong?
+ IsDrawOfferPending : bool
+ IsRematchOfferRequestPending : bool
+ IsRematchOfferResponsePending : bool
+ RequestedGameDto : RequestGameDto?
+ <<event>> TimeUpdated : Action<GameEventDto>
+ <<event>> NewFenReceived : Action<GameEventDto>
+ <<event>> ResignationReceived : Action<GameEventDto>
+ <<event>> NewPlayerJoined : Action<GameEventDto>
+ <<event>> DrawOffered : Action<GameEventDto>
+ <<event>> DrawOfferTimedOut : Action<GameEventDto>
+ <<event>> DrawOfferAccepted : Action<GameEventDto>
+ <<event>> RematchOffered : Action<GameEventDto>
+ <<event>> RematchOfferTimedOut : Action<GameEventDto>
+ <<event>> RematchOfferAccepted : Action<GameEventDto>
+ <<event>> EndOfTheGameReached : Action<GameEventDto>
+ <<event>> JoinRematchedGame : Action<GameEventDto>
+ <<event>> GameFirstJoined : Action
+ CreateGameAsync(RequestGameDto : int) : Task<ResponseGameDto>
+ JoinGameAsync(RequestJoinGameDto : int) : Task
+ SpectateGameAsync(RequestJoinGameDto : int) : Task
+ MakeMoveAsync(Move : int) : Task<AckTypes>
+ OfferDrawAsync() : Task<AckTypes>
+ SendDrawResponseAsync(bool : int) : Task<AckTypes>
+ OfferRematchAsync() : Task<AckTypes>
+ SendRematchResponseAsync(bool : int) : Task<AckTypes>
+ ResignAsync() : Task<AckTypes>
+ GetLastFenAsync() : Task<string?>
+ GetGameRoomsAsync(GameRoomSearchParameters : int) : Task<IList<GameRoomDto>>
+ GetCurrentGameStateAsync() : Task
+ LeaveRoomAsync() : void

19. IGameService Interface

All pages that need a GameService class use the IGameService interface and have it injected as a singleton. While there was a possibility to apply the interface segregation principle to the GameService class, because of the added interface complexity and the time required to implement it, it was decided that the benefits of limiting the responsibilities of each component do not outweigh the downsides. This might change in the future of the project.

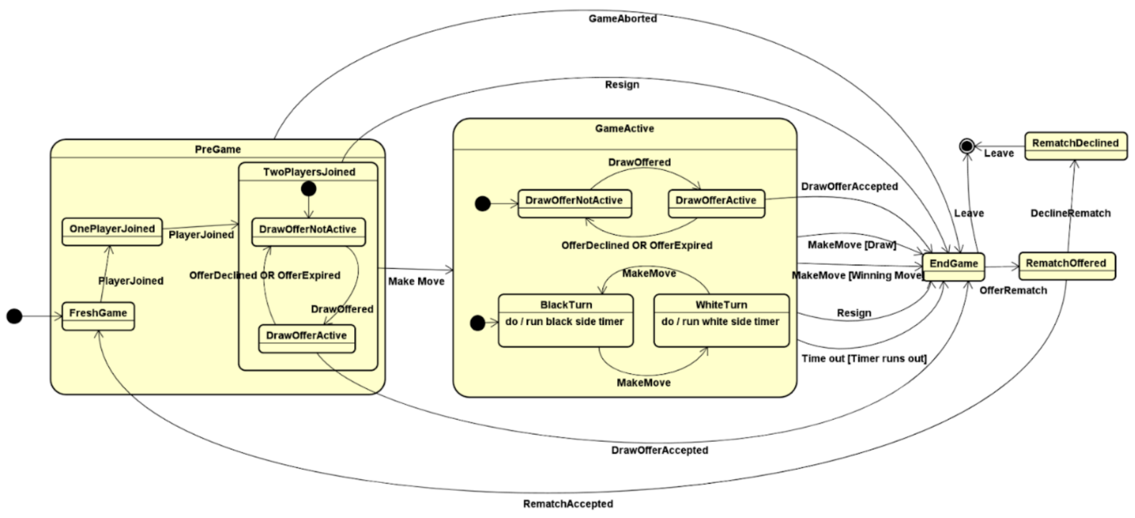
Model Class Diagram


pkg
GameRoom
+ <<prop>> Creator : string
+ <<prop>> PlayerWhite : string?
+ <<prop>> PlayerBlack : string?
+ <<prop>> GameType : OpponentTypes
+ <<prop>> GameSide : GameSides
+ <<prop>> IsVisible : boolean
+ <<prop>> TimeControlDurationSeconds : uint
+ <<prop>> TimeControlIncrementSeconds : uint
StockfishPlayer
+ <<get>> Skill : uint
+ <<get>> Depth : uint
+ <<get>> Time : uint
StockfishLevels
+ LevelOf : IReadOnlyDictionary<string,StockfishPlayer>
<<Rudzoft>>
Piece
+ <<prop>> PieceType : PieceTypes
0..1
User
+ <<prop>> Username : string
+ <<prop>> Email : string
+ <<prop>> Password : string
+ <<prop>> Role : string
<<Rudzoft>>
Move
+ <<prop>> FromSquare : Square
+ <<prop>> ToSquare : Square
+ <<prop>> MoveType : MoveTypes
2
<<Rudzoft>>
Square
+ <<get>> Rank : Ranks
+ <<get>> File : Files
ChatRoom
- _messages : List<Message>
Message
+ <<prop>> Username : string
+ <<prop>> Content : string

20. Model Class Diagram

Because the system uses the Rudzoft Chess library, core entities for the mechanics of the game were already created. User entity is used for authentication, GameRoom for handling the data stored about the game in the database and created during a game, and StockfishPlayer stores values of possible skill settings for the AI player to play against, as well as its username, so that it can be chosen for a game.

3.7 State Machine Diagram of a Game Room Handler



21. GameRoomHandler State Machine Representation

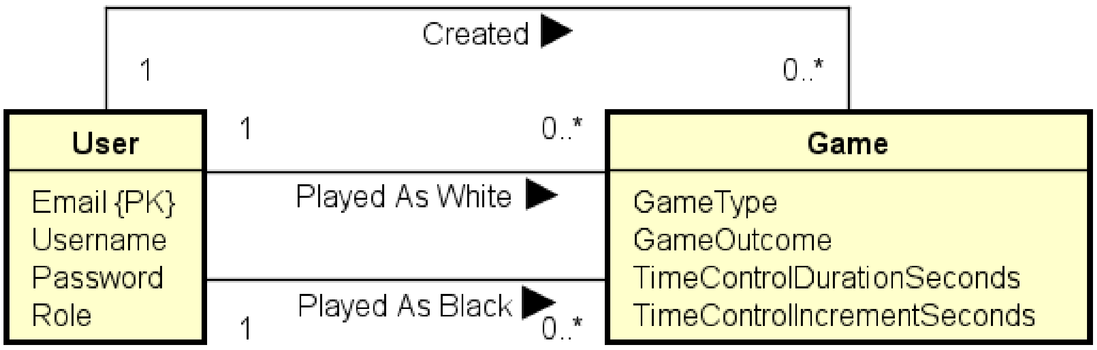
State pattern was regrettably not used within the application, but the State Machine Diagram still proved to help illustrate different constraints and possible actions within the given class.
State machine diagram in appendix B

This diagram shows how the game room handler class works by representing it as a state machine. Whenever a new game is created it starts as a fresh game. If only one player has joined, he is not allowed to make a move, resign or offer draws. Once both players have joined the player on the white side is allowed to make a move, and offering a draw and resigning is also allowed.

After the first move has been made the pre-game state is over and the timer starts ticking down. The players start making alternating moves until the game ends. The game can end in multiple ways. It could end because of the position of the chess game. If one player checkmates the opponent, the game would end. Alternatively, the game can also end if one of the players resigns or if both of the players agree to a draw.

Once the game ends the players can offer each other a rematch and play another game. If one of the players declines the offer it is not possible to offer another request.

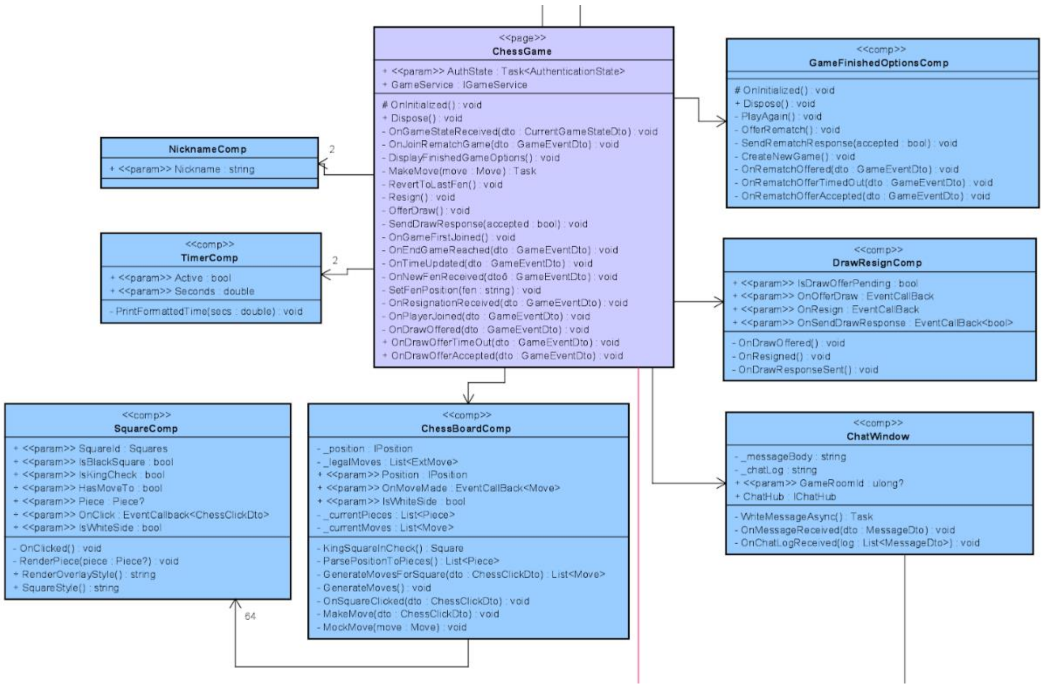
3.8 Database Design



22. EER Diagram

This diagram shows the entities that the system uses and the relations between them. The database has a user entity with an email and password to authenticate them and a role field to authorize users. Users can create and participate in games that after finishing are persisted. Games without both players are never saved because if the players weren't participating, the game didn't happen and it's not worth storing.

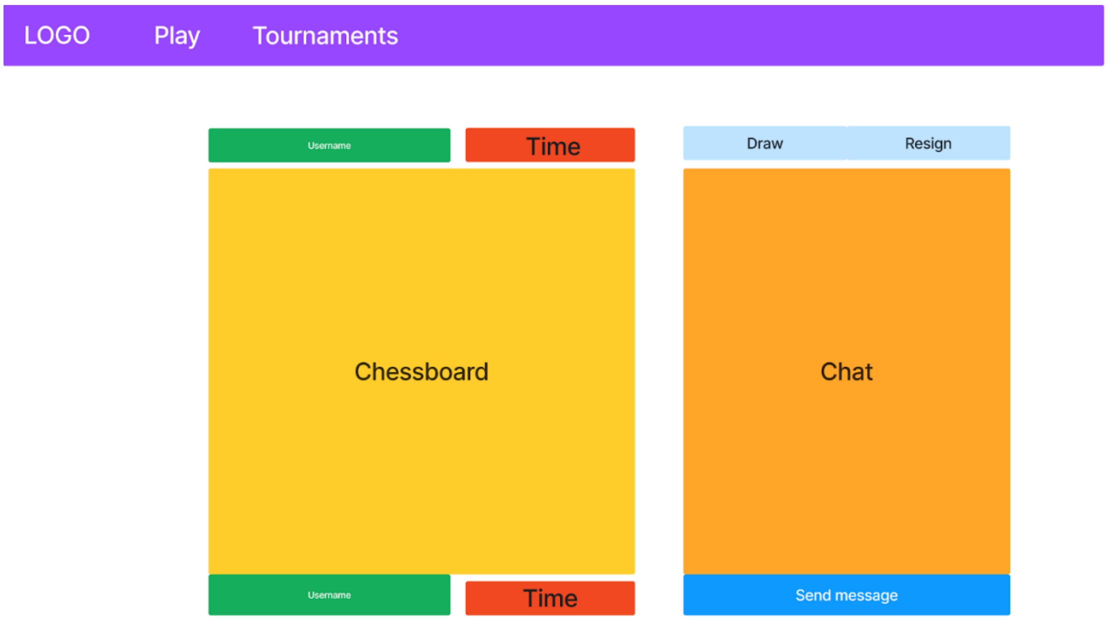
3.9 UI Design


<<page>>
ChessGame
+ <<param>> AuthState : Task<AuthenticationState>
+ GameService : IGameService
# OnInitialized() : void
+ Dispose() : void
- OnGameStateReceived(dto : CurrentGameStateDto) : void
- OnJoinRematchGame(dto : GameEventDto) : void
- DisplayFinishedGameOptions() : void
- MakeMove(move : Move) : Task
- RevertToLastFen() : void
- Resign() : void
- OfferDraw() : void
- SendDrawResponse(accepted : bool) : void
- OnGameFirstJoined() : void
- OnEndGameReached(dto : GameEventDto) : void
- OnTimeUpdated(dto : GameEventDto) : void
- OnNewFenReceived(dtoő : GameEventDto) : void
- SetFenPosition(fen : string) : void
- OnResignationReceived(dto : GameEventDto) : void
- OnPlayerJoined(dto : GameEventDto) : void
- OnDrawOffered(dto : GameEventDto) : void
+ OnDrawOfferTimeOut(dto : GameEventDto) : void
+ OnDrawOfferAccepted(dto : GameEventDto) : void
<<comp>>
GameFinishedOptionsComp
# OnInitialized() : void
+ Dispose() : void
- PlayAgain() : void
- OfferRematch() : void
- SendRematchResponse(accepted : bool) : void
- CreateNewGame() : void
- OnRematchOffered(dto : GameEventDto) : void
- OnRematchOfferTimedOut(dto : GameEventDto) : void
- OnRematchOfferAccepted(dto : GameEventDto) : void
NicknameComp
+ <<param>> Nickname : string
2
<<comp>>
TimerComp
+ <<param>> Active : bool
+ <<param>> Seconds : double
- PrintFormattedTime(secs : double) : void
2
<<comp>>
DrawResignComp
+ <<param>> IsDrawOfferPending : bool
+ <<param>> OnOfferDraw : EventCallBack
+ <<param>> OnResign : EventCallBack
+ <<param>> OnSendDrawResponse : EventCallBack<bool>
- OnDrawOffered() : void
- OnResigned() : void
- OnDrawResponseSent() : void
<<comp>>
SquareComp
+ <<param>> SquareId : Squares
+ <<param>> IsBlackSquare : bool
+ <<param>> IsKingCheck : bool
+ <<param>> HasMoveTo : bool
+ <<param>> Piece : Piece?
+ <<param>> OnClick : EventCallback<ChessClickDto>
+ <<param>> IsWhiteSide : bool
- OnClicked() : void
- RenderPiece(piece : Piece?) : void
+ RenderOverlayStyle() : string
+ SquareStyle() : string
64
<<comp>>
ChessBoardComp
- _position : IPosition
- _legalMoves : List<ExtMove>
+ <<param>> Position : IPosition
+ <<param>> OnMoveMade : EventCallBack<Move>
+ <<param>> IsWhiteSide : bool
- _currentPieces : List<Piece>
- _currentMoves : List<Move>
- KingSquareInCheck() : Square
- ParsePositionToPieces() : List<Piece>
- GenerateMovesForSquare(dto : ChessClickDto) : List<Move>
- GenerateMoves() : void
- OnSquareClicked(dto : ChessClickDto) : void
- MakeMove(dto : ChessClickDto) : void
- MockMove(move : Move) : void
<<comp>>
ChatWindow
- _messageBody : string
- _chatLog : string
+ <<param>> GameRoomId : ulong?
+ ChatHub : IChatHub
- WriteMessageAsync() : Task
- OnMessageReceived(dto : MessageDto) : void
- OnChatLogReceived(log : List<MessageDto>) : void

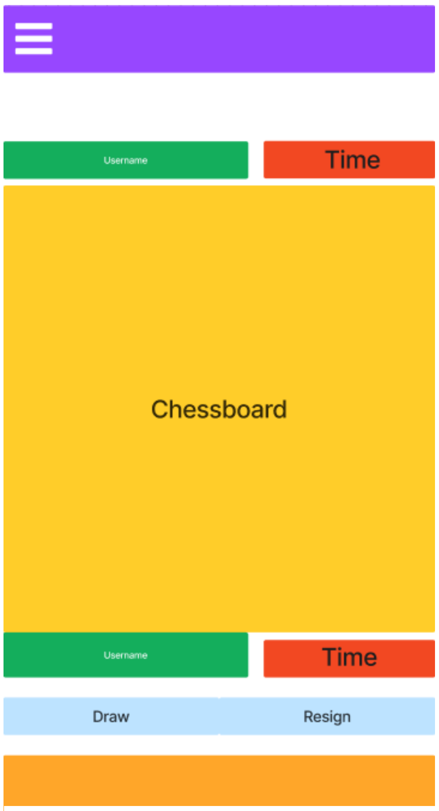
23. Play Game Page and Components in the Global Class Diagram

Blazor components are designed in a way that only a page or one main component has a service injected into it. This main component listens to all the events coming from the service and sends requests to it. Responsibilities of the remaining components are minimized to only displaying the data given through parameters and validating user input. This prevents situations where the developer has to “hunt” for logic hidden within all the components.

An Alternative option that was later discovered was to inject the service into all components and utilize interface segregation to limit the methods to which the components are exposed. This would ensure that every component is fully independent and could be used anywhere within the system.



24. Large Screen Play Game Wireframe



25. Phone wireframe of the application

The chess game is a page user interact with most. In the large screen version, all important information is displayed on the screen in a grid. Users can engage in a game and a chat conversation at the same time. The app bar at the top allows simple navigation to system functionality.

In a small-screen version of the app, the chessboard is designed to take the majority of the screen. Draw and resign buttons are still visible, but only a small part of the chat is shown, to allow users to focus on the game and the same to inform the user that there is more functionality below the chessboard. The app bar collapses to a convenient drawer to avoid clipping or overcrowding the top bar.

While simple, the chessboard page design provides GUI with all the functionality needed. Other pages in the application did not require wireframes as simple lists and input groups were sufficient to display all the necessary information in case of lobbies list and logging in.

3.10 Security design

Security Design

The purpose of this subsection is to consider security mechanisms that would be used by the online chess application to prevent threats and reduce overall risks. These include:

HTTPS protocol – Encryption and Authentication

For secure and encrypted connections between clients and communication servers, it is recommended to use HTTPS, which with the help of SSL/TLS encrypts and decrypts messages sent between actors. This would improve the trustworthiness of a Blazor server which reduces the likelihood of a mock website phishing attempt being successful.

Additionally, all messages are encrypted preventing eavesdropping, which according to analysis is a means of attack for many threats. (Cloudflare, 2019) All connections in the application should use HTTPS protocol.

JWT - Authentication and Authorization

The usage of JWT allows for secure authorization and information exchange. JWT token is given to the user after logging in and restricts the user's actions using policies and claims. Moreover, JWTs ensure that the content of the message hasn't been tampered with if the messages are signed. This way moves made by the player either get to the communication server unmodified or is disregarded if an attempt of modification occurred.

JWT could also be used to allow only authenticated server-to-server connection and prevent unauthorized access to the system resources with man-in-the-middle attacks.

S2S asymmetric encryption

To prevent eavesdropping on server-to-server communication, asymmetric encryption should be used. Each server could be associated with a specific public key statically because in the current state of the system servers providing services are known and static. This might cause issues later if the system had to expand to more servers.

Hashed passwords stored in a database

Hashing passwords would greatly mitigate damage that would result from a potential data breach. Using protocols such as SHA-256 greatly reduces password usefulness for the attacker therefore, they can't be used for breaking into user accounts, in case a user used the same password in multiple services. This additionally prevents any internal breaches of passwords, even if the administrator himself wished to retrieve passwords. Unfortunately, hashes cannot be used on all data in the databases as it's a non-reversible action.

Long Password requirements

To prevent brute force attacks, one of the security policies should be passwords that are at least 12 characters long. No need for expiring passwords, though, as the human factor of switching passwords every month is frustrating for users and may result in more security issues due to reusing similar passwords or simplifying them for the sake of easy memorization.

Firewall protecting against ping, buffer overflow, and other malicious messages

Certain firewall protections could be used to partially mitigate some of the DoS attacks. For example, unusual traffic of ping messages could lead to the automatic dropping of these messages.

Digital certificates of servers

To use the aforementioned HTTPS protocol, digital certificates are required which must be signed by a trusted third party. It is an important step in ensuring that clients connect to authenticated servers and not impersonations.

Validating user input before storing it

While this is not a threat coming from the network, it is just as important. This recommendation serves as a warning to the designers of the system to make sure to test for user input edge cases.

Security design conclusion

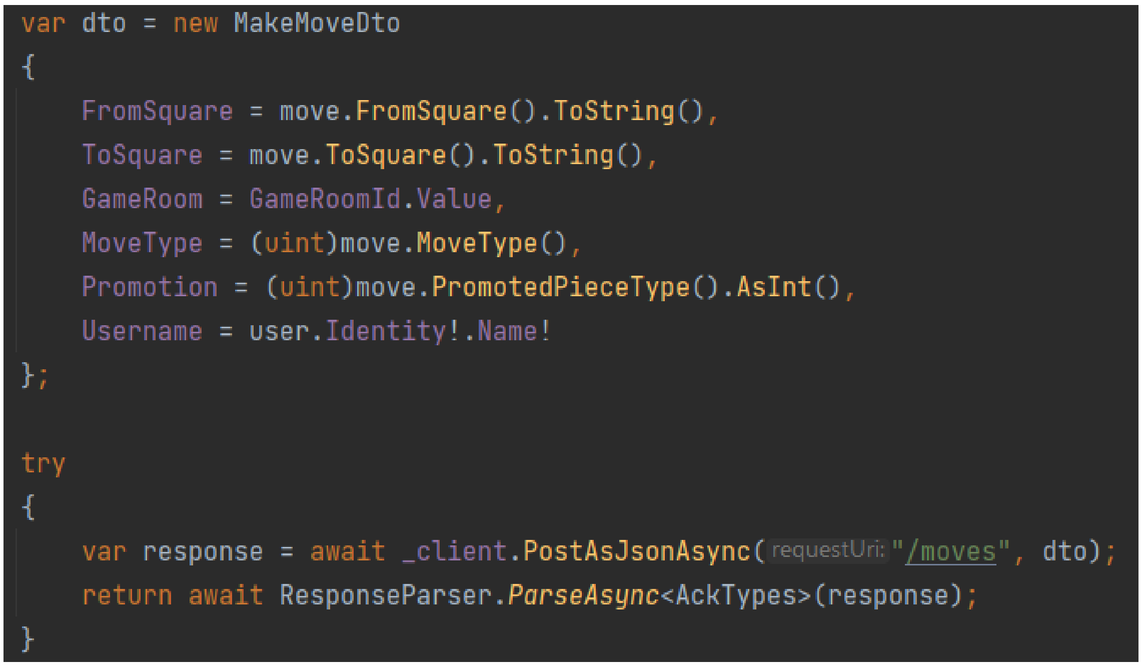
In summary, there is a multitude of countermeasures that could mitigate most of the threats presented in the analysis section. Redundancy and constant revision of the system security measures are crucial in preventing system breaches.

4 Implementation

Once the system artifacts are designed, they are also implemented in languages such as Java and C#. Based on Design Patterns, Class Diagrams, and chosen technologies, workable pieces of code are created and added to the system in each sprint. In the following subsections of the chapter, interesting snippets which show what goes on behind the system can be found.

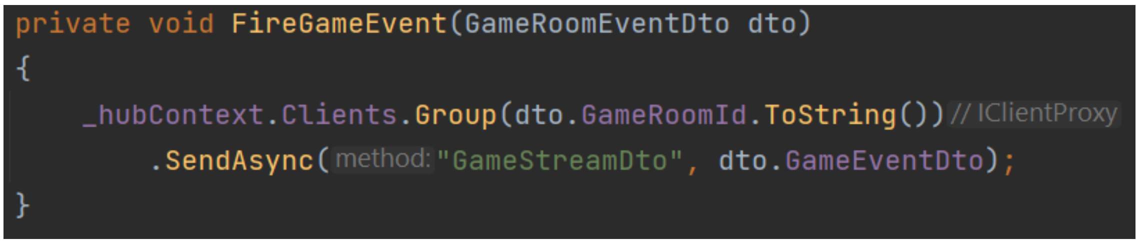
4.1 Playing Chess Implementation with SignalR and REST

The following examples illustrate how SignalR and REST are used to play chess.



```
var dto = new MakeMoveDto
{
    FromSquare = move.FromSquare().ToString(),
    ToSquare = move.ToSquare().ToString(),
    GameRoom = GameRoomId.Value,
    MoveType = (uint)move.MoveType(),
    Promotion = (uint)move.PromotedPieceType().AsInt(),
    Username = user.Identity!.Name!
};

try
{
    var response = await _client.PostAsJsonAsync(requestUri: "/moves", dto);
    return await ResponseParser.ParseAsync<AckTypes>(response);
}
```

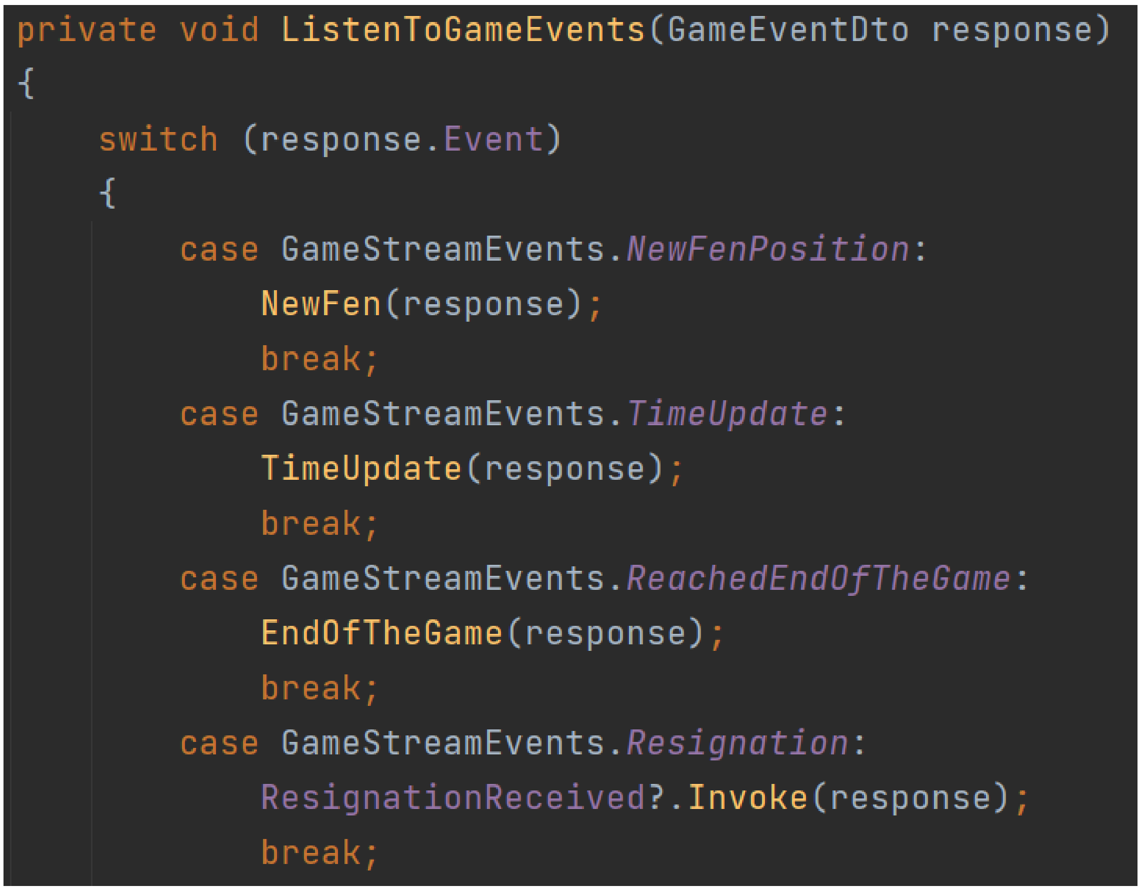
26. A part of the MakeMoveAsync method in GameService



```
private void FireGameEvent(GameRoomEventDto dto)
{
    _hubContext.Clients.Group(dto.GameRoomId.ToString()) // IClientProxy
        .SendAsync(method: "GameStreamDto", dto.GameEventDto);
}
```

27. A method which fires an Event in GroupHandler

When a user makes a move, the move data is sent to a server REST endpoint for moves. Then the request is authorized and the move is validated. After a move has been made all other users are informed about the move made using SignalR. The move that has been made is broadcasted to all other users who have joined that specific game. When users join a game they send a request to the server to join a SignalR group which notifies users about the events which are occurring in the game. All users: the players and the spectators join the same group chosen by a game room id and receive the events of that game.



```
private void ListenToGameEvents(GameEventDto response)
{
    switch (response.Event)
    {
        case GameStreamEvents.NewFenPosition:
            NewFen(response);
            break;
        case GameStreamEvents.TimeUpdate:
            TimeUpdate(response);
            break;
        case GameStreamEvents.ReachedEndOfTheGame:
            EndOfTheGame(response);
            break;
        case GameStreamEvents.Resignation:
            ResignationReceived?.Invoke(response);
            break;
```

28. Part of ListenToGameEvents method in GameService

Previously this functionality was implemented using gRPC, but because the client was in a browser it was necessary to use gRPC web which has limited functionality. Another reason SignalR is used is because of the broadcasting to groups functionality. Sending data to a group of people using gRPC was complicated. Before moving to SignalR each client had one gRPC server streaming connection open to listening to game events and because using gRPC it was difficult to differentiate users all events had to be sent to all users which meant that a lot of validation was required to only display information that was meant for that user.

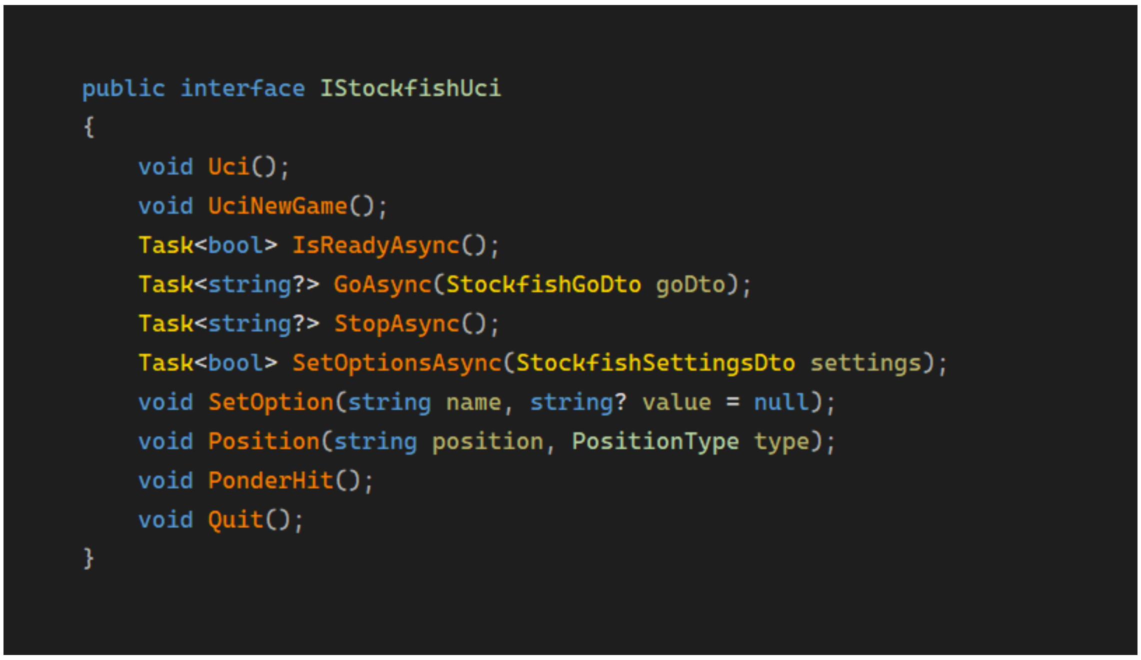
Because of lack of time SignalR was accommodated to the gRPC implementation instead of starting from scratch.

The implementation could be improved by sending events as draw or rematch offers to a specific user instead of broadcasting to all users.

Another improvement could be the splitting up of the game stream events. Using SignalR it is possible to create a listener for each type of event instead of using one listener for all types of events.

4.2 Stockfish UCI implementation

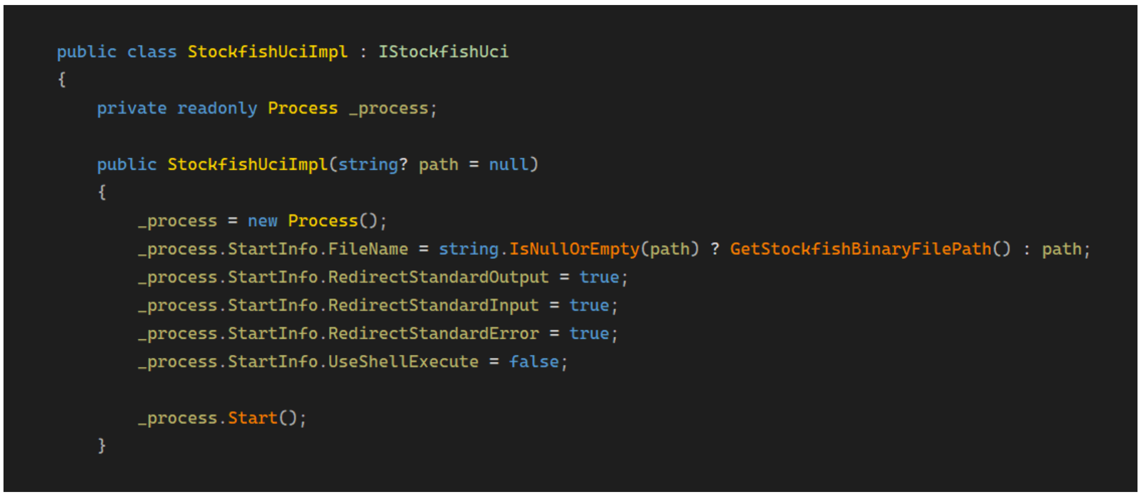
The following examples illustrate how the UCI (Universal Chess Interface) is implemented in the Stockfish Server.



```
public interface IStockfishUci
{
    void Uci();
    void UciNewGame();
    Task<bool> IsReadyAsync();
    Task<string?> GoAsync(StockfishGoDto goDto);
    Task<string?> StopAsync();
    Task<bool> SetOptionsAsync(StockfishSettingsDto settings);
    void SetOption(string name, string? value = null);
    void Position(string position, PositionType type);
    void PonderHit();
    void Quit();
}
```

29. The Interface Implemented by the Stockfish Wrapper

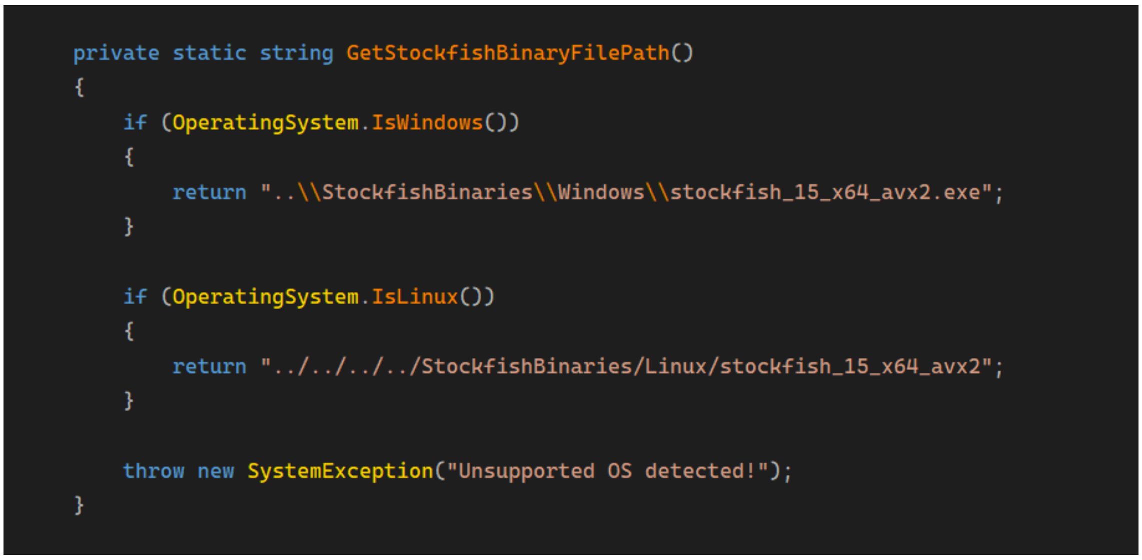
The above image shows an interface containing method prototypes that are part of a UCI communication protocol. This is a standard that is used for interacting with most chess engines. This is the API through which the system communicates with Stockfish Chess Engine and whose implementation will be explained next.



```
public class StockfishUciImpl : IStockfishUci
{
    private readonly Process _process;

    public StockfishUciImpl(string? path = null)
    {
        _process = new Process();
        _process.StartInfo.FileName = string.IsNullOrEmpty(path) ? GetStockfishBinaryFilePath() : path;
        _process.StartInfo.RedirectStandardOutput = true;
        _process.StartInfo.RedirectStandardInput = true;
        _process.StartInfo.RedirectStandardError = true;
        _process.StartInfo.UseShellExecute = false;

        _process.Start();
    }
```

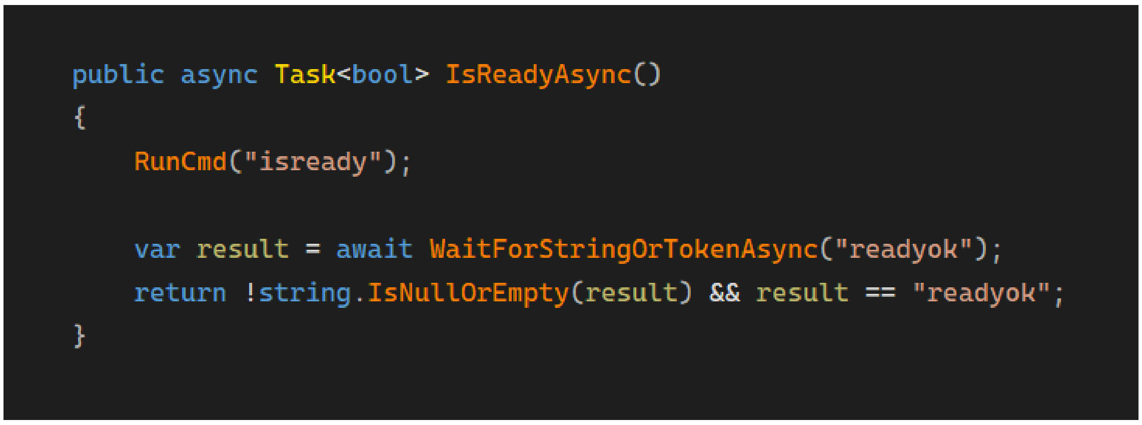
31*. Stockfish UCI Implementation Class*

This is the beginning of the class implementing the UCI interface. The way it interacts with the Stockfish program is through a *Process* object which is used to start and manage system processes. A process is an instance of a running program on a computer. The binary file for the Stockfish program is fetched with the following method:



```
private static string GetStockfishBinaryFilePath()
{
    if (OperatingSystem.IsWindows())
    {
        return "..\\StockfishBinaries\\Windows\\stockfish_15_x64_avx2.exe";
    }

    if (OperatingSystem.IsLinux())
    {
        return "../../../../StockfishBinaries/Linux/stockfish_15_x64_avx2";
    }

    throw new SystemException("Unsupported OS detected!");
}
```

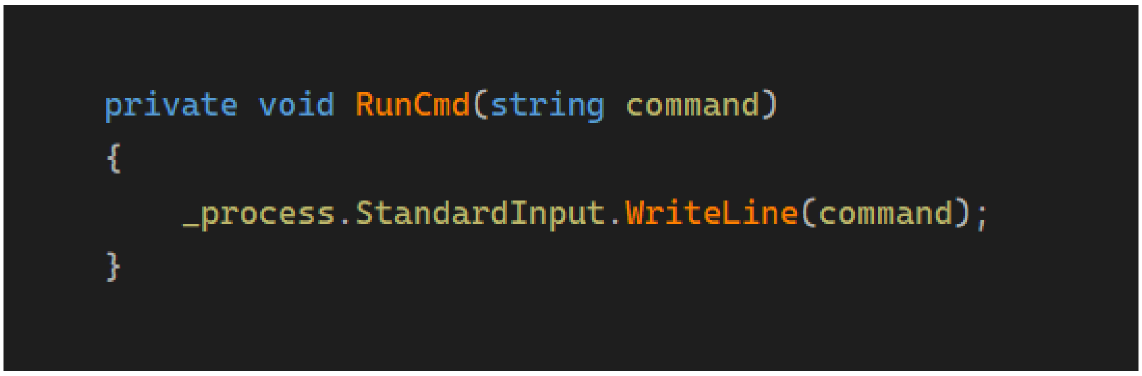
30. A method to get the path to the Stockfish binary file based on Operating System



```
public async Task<bool> IsReadyAsync()
{
    RunCmd("isready");

    var result = await WaitForStringOrTokenAsync("readyok");
    return !string.IsNullOrEmpty(result) && result == "readyok";
}
```

32. This method returns true if the Stockfish process is ready

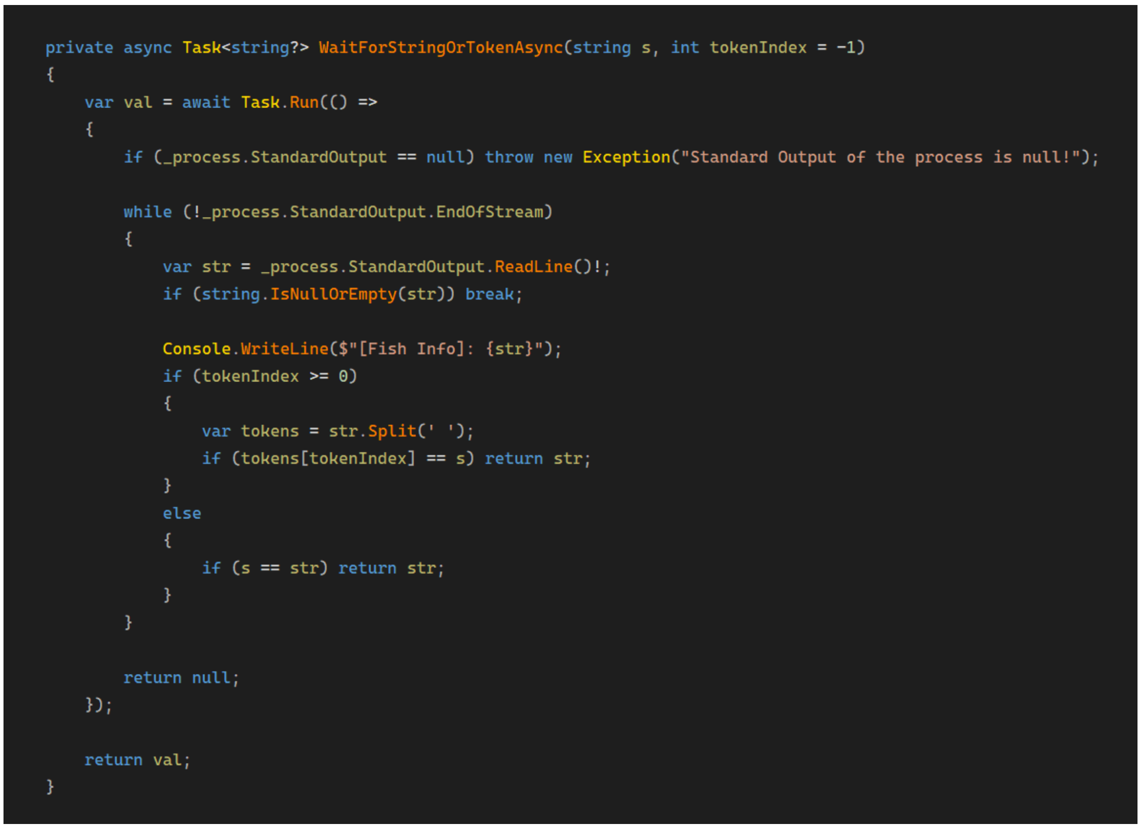
The above image shows how the process gets queried for its status. The Stockfish process, once started, expects input data. *“**isready**”* can be sent to its standard input and it will print “***readyok***” to the standard output if it has been properly initialized and is ready for further instructions. It uses the ***RunCmd***() method for printing to standard input. The following is the implementation of this method:



```
private void RunCmd(string command)
{
    _process.StandardInput.WriteLine(command);
}
```

33. Implementation of RunCmd method which prints to standard input of the process

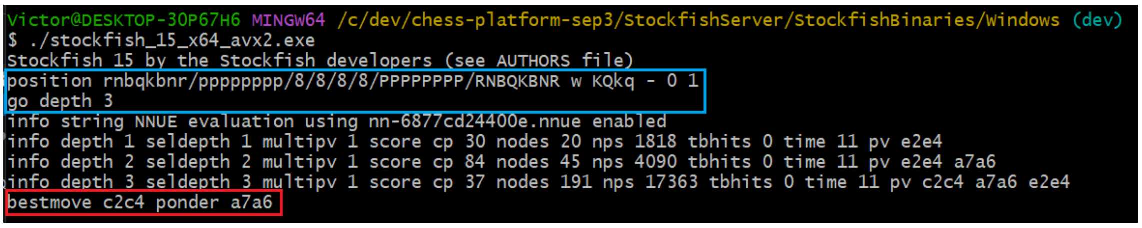
The system reads the output of the process and waits for a specific token to appear. This is what the ***WaitForStringOrTokenAsync***() method is used for. The following method shows how the standard output of the process is read:



```
private async Task<string?> WaitForStringOrTokenAsync(string s, int tokenIndex = -1)
{
    var val = await Task.Run(() =>
    {
        if (_process.StandardOutput == null) throw new Exception("Standard Output of the process is null!");

        while (!_process.StandardOutput.EndOfStream)
        {
            var str = _process.StandardOutput.ReadLine()!;
            if (string.IsNullOrEmpty(str)) break;

            Console.WriteLine($"[Fish Info]: {str}");
            if (tokenIndex >= 0)
            {
                var tokens = str.Split(' ');
                if (tokens[tokenIndex] == s) return str;
            }
            else
            {
                if (s == str) return str;
            }
        }

        return null;
    });

    return val;
}
```

34. Implementation of the WaitForStringOrTokenAsync() method which returns a string of text if it finds it in the standard output of the process

Similarly, Stockfish can be queried for the best move in a given position, by sending it a position in a FEN notation with other configurable parameters such as calculation depth and time limit for its computation.

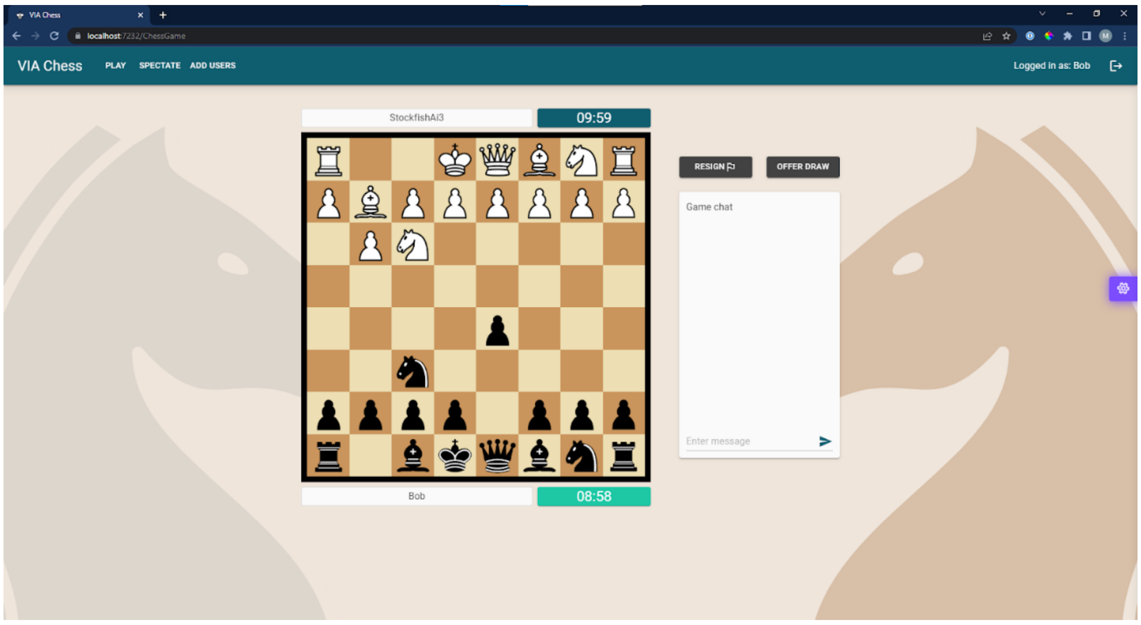


```
Victor@DESKTOP-30P67H6 MINGW64 /c/dev/chess-platform-sep3/StockfishServer/StockfishBinaries/Windows (dev)
$ ./stockfish_15_x64_avx2.exe
Stockfish 15 by the Stockfish developers (see AUTHORS file)
position rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1
go depth 3
info string NNUE evaluation using nn-6877cd24400e.nnue enabled
info depth 1 seldepth 1 multipv 1 score cp 30 nodes 20 nps 1818 tbhits 0 time 11 pv e2e4
info depth 2 seldepth 2 multipv 1 score cp 84 nodes 45 nps 4090 tbhits 0 time 11 pv e2e4 a7a6
info depth 3 seldepth 3 multipv 1 score cp 37 nodes 191 nps 17363 tbhits 0 time 11 pv c2c4 a7a6 e2e4
bestmove c2c4 ponder a7a6
```

35. Running the Stockfish process from the command-line

The above image shows what the interaction with the Stockfish process looks like in the command line. The blue rectangle contains user input, and the red rectangle shows the result of that operation. In this case, the best move in a given position was requested with a calculation depth set to 3. The result is the move in UCI notation: c2c4.

4.3 Blazor Implementation - Chessboard

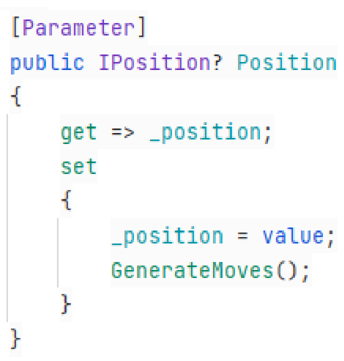


36. Game Page Application View

The Chess Game page roughly resembles the wireframe created during the design, if not more elegant.

Chess pieces were drawn by Cburnett (Cburnett, 2006)

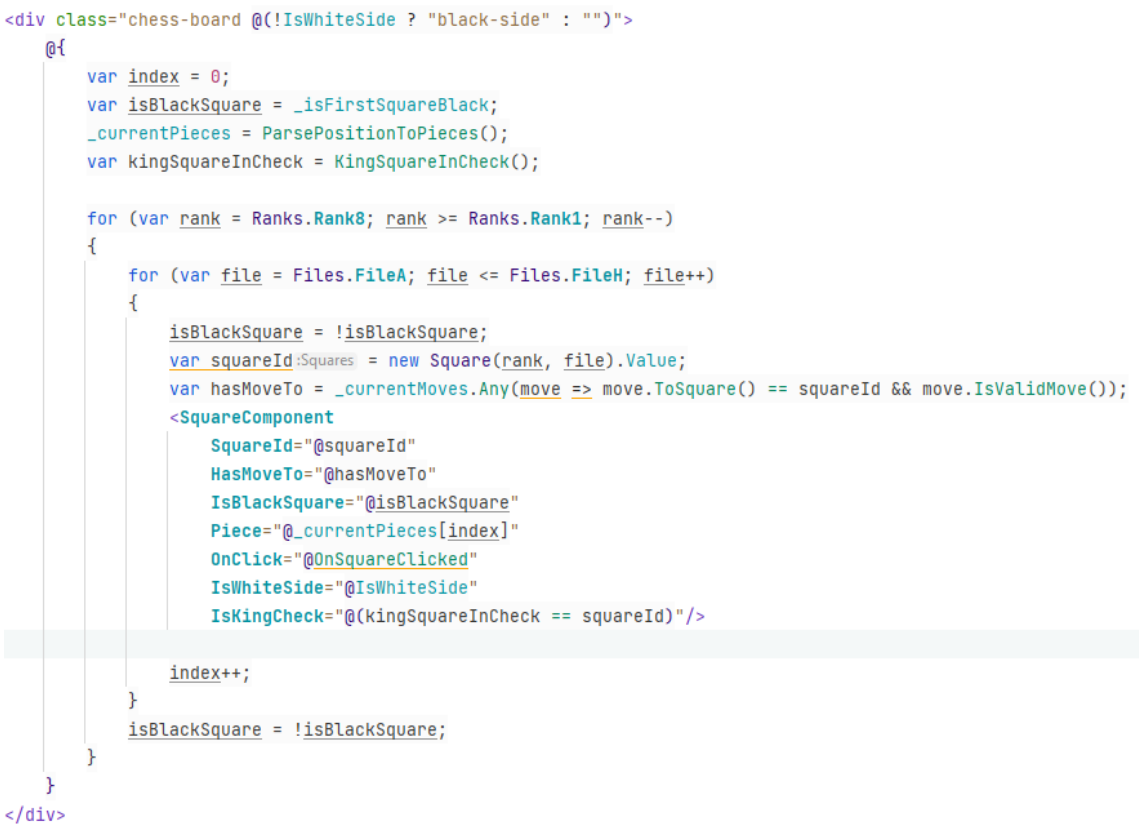
Mudblazor components, such as buttons, text fields, inputs, papers, and elevations create a clean and unified UI, that provides necessary information. The chessboard is made to always fit the viewport, even when screen dimensions are abnormal. The colors of the chessboard draw attention to the component and expose the most important piece of information on the page, the game itself.



```
[Parameter]
public IPosition? Position
{
    get => _position;
    set
    {
        _position = value;
        GenerateMoves();
    }
}
```

37. Position Setter

The Rudzoft Position object which is created by parsing FEN notation sent by the communication server is passed as a parameter to the chessboard. Every time it's set Moves for all the pieces are generated so that the chessboard appears more responsive to the user. This might be seen as a deviation from a rule where logic should all be kept in the logic layer. This was done because the message size of game updates had to be kept to the minimum and adding generated moves to each message would greatly increase its size. Additionally, Rudzoft handles move generation on both a comm server and a client, which ensures consistent behavior.



```
<div class="chess-board @(!IsWhiteSide ? "black-side" : "")">
    @{
        var index = 0;
        var isBlackSquare = _isFirstSquareBlack;
        _currentPieces = ParsePositionToPieces();
        var kingSquareInCheck = KingSquareInCheck();

        for (var rank = Ranks.Rank8; rank >= Ranks.Rank1; rank--)
        {
            for (var file = Files.FileA; file <= Files.FileH; file++)
            {
                isBlackSquare = !isBlackSquare;
                var squareId :Squares  = new Square(rank, file).Value;
                var hasMoveTo = _currentMoves.Any(move => move.ToSquare() == squareId && move.IsValidMove());
                <SquareComponent
                    SquareId="@squareId"
                    HasMoveTo="@hasMoveTo"
                    IsBlackSquare="@isBlackSquare"
                    Piece="@_currentPieces[index]"
                    OnClick="@OnSquareClicked"
                    IsWhiteSide="@IsWhiteSide"
                    IsKingCheck="@(kingSquareInCheck == squareId)"/>

                index++;
            }
            isBlackSquare = !isBlackSquare;
        }
    }
</div>
```

38. Code responsible for generating a Chessboard

Using the Rudzoft Position, necessary data is parsed and displayed in the form of 64 SquareComponents. _currentMoves is generated after the user chooses a piece, by selecting all the moves with FromSquare matching the Square that was clicked. Initially, moves used to be generated on each user click, which was unresponsive for the user.

5 Test

5.1 Unit Testing

After implementing each piece of logic within the system, unit tests were created to cover the functionality. For the good of the future of the project, comprehensive testing covering all ZOMBIES cases should always be created with each new feature. Current unit tests cover most of the code within the logic of the application.

5.2 Acceptance testing

Fortunately, when acceptance testing was performed during the transition phase of the project, most test cases for use cases that were designed and implemented were successful. Here is a sample of some of these tests.
All test cases and results can be found in Appendix C

5.3 Acceptance test sample – play chess use case

Base sequence

Action no.	**Action**	**Reaction**
1)	Player chooses a chess piece to move.	System shows available moves for the piece.
2)	Player makes a move.	System shows the move has been played.

Extensions

Offer a draw

Action no.	**Action**	**Reaction**
1)	Player offers a draw.	System shows that the offer has been sent to the opponent.

Receive a draw

Action no.	**Action**	**Reaction**
1)	System shows that there is a pending draw offer from the opponent.	Player accepts, declines, or ignores the offer which will expire after 10 seconds.

Resign the game

Action no.	Action	Reaction
1)	Player resigns from the game.	System informs about the end of the game and shows available post-game options.

Time runs out

Action no.	Action	Reaction
1)	Player doesn't make a move and their time runs out.	System informs about the end of the game and shows available post-game options.

Player makes a move that ends the game [checkmate, move repetition, insufficient material, stalemate]

Action no.	Action	Reaction
1)	Player makes a move that results at the end of the game.	System informs about the end of the game and shows available post-game options.

Player writes a message in the chat

Action no.	Action	Reaction
1)	Player writes a message in the chat and chooses to send it.	System displays the message in the chat window.

Exceptions

Invalid move

Action no.	Action	Reaction
1)	Player makes an invalid move.	System discards the move and reverts the position.

Connection error

Action no.	Action	Reaction
1)	Player does an action.	System notifies that the operation failed to complete.

Play chess test case results

Case	Result
Base sequence	Success
Offer a draw	Success
Receive a draw	Success
Resign the game	Success
Time runs out	Success
Player makes a move that ends the game (checkmate)	Success
Player makes a move that ends the game	Failure

(move repetition)	
Player makes a move that ends the game (insufficient material)	**Failure**
Player makes a move that ends the game (stalemate)	**Success**
Player writes a message in the chat	**Success**
Invalid move	**Success**
Connection error	**Success**

6 Results and Discussion

Introduction

The final product is a distributed system that allows users to play chess online with friends. The application has two roles: users which use the system to play chess and administrators who manage the users using the system.

Chess Client

A single-page application that allows users to play chess. The authentication that is implemented in the application only allows administrators to add new users and only logged-in users are allowed to play chess. Users have the choice to play against a friend, a random opponent, or an AI.

Communication Server

A server that is used as the main hub for connecting users. This server also accesses the database and stockfish server to provide data and AI moves to users. Because of the flexible design, it is easy to change what servers the communication server connects to. The logic of the server is also thoroughly tested to make sure that as little as possible unexpected behavior occurs during games.

Database Server

A server that is used for accessing and storing data. The server has validation and constraints which prevent users from saving invalid information.

Stockfish Server

A server that is used to handle requests for chess moves from an AI. The server allows the requester to modify the difficulty of the moves that are generated.

Conclusion

The result is a system that is well designed and implemented with extensibility and flexibility in mind to make modifying and adding new features to the system an easy task while fulfilling the needs of the customer.

7 Conclusions

The introduction of the report illustrated the previous situation, wants, and needs of a client. It outlined the purpose. The introduction presented the case that projects aid in the fulfillment of UN Global Goals. The chapter presented aims and objectives and set in place the delimitations. The introduction concluded with potential technical issues that might arise and a transition to the analysis of the project.

Through the analysis of the background description and close collaboration with the client 21 requirements were drafted along with importance grading that helped in focusing on the critical features of the system. Encompassing domain model was created. The analysis resulted in 7 use cases that would provide useful features to the client. The system implements 4 of them. The multitude of use case descriptions and artifacts, such as activity diagrams supports project analysis.

Security analysis of the application was conducted. Using methods such as STRIDE and EINOO analysis recognized threats for the system and recommendations of how to prevent or limit their effect. Security design section-specific technologies could potentially aid in countering security threats.

The design of the system resulted in a testable application. A combination of N-Tier architecture, modern technologies, such as SignalR and Blazor, and design patterns fulfilled the criteria of a scalable system. Sequence diagrams illustrated interactions between classes and a Global Class Diagram provided a great overview of the application. Explanations of parts of the class diagram and a state machine diagram further allow developers to understand how the implementation should be performed. Coherent design of the UI and wireframes for the crucial parts of the applications are shown to share a vision designers had for the system**.** A combination of acceptance and unit testing assured system functionality. Most of the test cases returned a positive outcome

Using communication technologies and design blueprints a heterogeneous system was formed. Hybrid communication between the communication server and a client reaps the benefits of both SignalR and REST. Stockfish UCI implementation gives adjustable difficulty to computer opponents for single players and provides a way to use a different chess engine if there is ever a need for that. Every component is elegantly detachable from the system. The UI implements the idea given in the design with a component library, giving it a consistent look.

The result of the project is a heterogenous system that provides access to an online chess game, supported by documentation with artifacts that sets the scene for a continuation of the project and expanding it with functions that the client needs.

8 Project future

In the current state of the project, it is not possible to safely deploy to the outside world, but the architecture and design patterns provided a foundation for easily expanding the systems to match the deployment needs.

Dividing the application into separate systems and these systems into layers made it simple to modify and expand. Even a big structural change of the application such as replacing web-gRPC with SignalR didn't mandate any changes, nor created unexpected behavior within systems. System distribution allows adding new nodes whenever traffic becomes an issue. Persistence and communication logic is kept separate, so there is no risk of inconsistent data states if there is a need for more communication servers.

System redundancy should be added to the application, as a failure of any of the nodes will result in complete interoperability. Right now, users could easily overwhelm a communication server, so a good idea would be to implement a few communication servers and add load balancers that would keep latency in check.

The system is not ready for deployment, as the security measures currently implemented are not enough to keep user data secure in a meaningful way. Implementing technologies listed in the security design chapter would certainly help in approaching a production-ready state.

Additionally using WASM Blazor caused the application to be unresponsive and sometimes unusable on low-end devices. Moving to server-side Blazor would improve the user experience on older devices and increase customer satisfaction.

Streamed game events should be refactored so that each event in the game is a separate event object. Single objects for all events in the game are a residue from web-gRPC, where only a few connections could be open at once and they had to be repurposed. SignalR does not have any limitations like that. tailor cutting the events would improve debugging and clarity of the logic. Measures such as Continuous

Integrity Testing should also have been developed together with the application to add to robustness and assure system integrity

Moreover, not all use cases needed to moderate the system are implemented. Manage Users use case has only creating users implemented, and for the system to be usable without developers' constant intervention to remove or block users, the whole use case functionality is required.

Some use cases were identified in the analysis and of value to the client were listed in the analysis, such as tournaments and analyzing games. Implementing these could greatly benefit users and the system.

In the end, the system lacks a few crucial elements for production, but it is feasible to approach such a state, provided extra moderation functionality is added and security measures are put in place.

9 Sources of information

IBM Cloud Education, I.B.M.C.E., 2020. *What is Java spring boot?* [online] IBM. Available at: <https://www.ibm.com/cloud/learn/java-spring-boot> [Accessed 15 Dec. 2022].

Cloudflare 2019 *What is HTTPS? | Cloudflare, What is HTTPS?* [online] Available at: https://www.cloudflare.com/learning/ssl/what-is-https/ [Accessed 15 December 2022].

Aciego, R., García, L. and Betancort, M., 2012. *The Benefits of Chess for the Intellectual and Social-Emotional Enrichment in Schoolchildren.* [online] Available at:

<https://pubmed.ncbi.nlm.nih.gov/22774429/> [Accessed 24 September 2022].

Poston, D. and Vandenkieboom, K., 2019. *The Effect of Chess on Standardized Test Score Gains.* [online] Available at: <https://journals.sagepub.com/doi/10.1177/2158244019870787> [Accessed 24 September 2022].

The Global Goals. n.d*. The Global Goals.* [online] Available at: <https://www.globalgoals.org/> [Accessed 24 September 2022].

E. Omernick and S. O. Sood, 2013 The Impact of Anonymity in Online Communities, International Conference on Social Computing, pp. 526-535,].

Privacy Shield, 2016. *European Union - Data Privacy and ProtectionEuropean Union - Data Privacy*. [online] European Union - Data Privacy and Protection | Privacy Shield. Available at: <https://www.privacyshield.gov/article?id=European-Union-Data-Privatization-and-Protection> [Accessed 15 Dec. 2022].

Kurose, J.F. and Ross, K.W., 2021. *Computer networking: A top-down approach*. Hoboken: Pearson.

Coulouris, G.F., 2012. *Distributed Systems Concepts and Design*. Boston, Mass. u.a: Addison-Wesley.

Cburnett, 2006 Chess pieces. [online] Wikimedia Commons. Available at: <https://commons.wikimedia.org/w/index.php?curid=1496738> [Accessed 16 Dec. 2022].

10 Appendices

Appendix A – Use Case Descriptions

Appendix B – SVG diagrams

Appendix C – Test cases and results

Appendix D – User Guide

Appendix E – Project-related terminology

Appendix F – Installation guide

Appendix G – Source code

Appendix H – Astah files

Appendix I – Project description

Appendix J – GitHub link

Appendix K – Video demonstration

Bring ideas to life
VIA University College

Chess Application

Distributed System

Aivaras Zevzikovas (315228)

Mikolaj Morawski (315229)

Wiktor Ciolkowski (315187)

Selina Ceban (315235)

Supervisors:

Jakob Knop Rasmussen

Joseph Chukwudi Okika

VIA University College

Number of characters: 36956

Software Technology Engineering

Semester III

16/12/2022

Table of content

1 Introduction

The scope of the Third Semester Project for Software Technology Engineering is to be able to create a heterogeneous system in Java and C# with persistence means. If semester two was concentrated around creating a client-server system, progressing in studies means also advancing in knowledge, so this time around we add one or more technologies and include server-to-server communication.

Methodology-wise, we had more freedom to choose this time around, and if documented accordingly, we even had the opportunity to switch from one to another or combine them. For this reason, our group chose to follow Scrum. At first, we wanted to follow Kanban entirely, but have realized that, for a student, and for the time constraints we had, it is wiser to have steady deadlines and checks of progress.

Overall, our group had a total of twelve documented meetings related to SEP3, where certain goals and subjects were discussed and noted down. At first, the meeting minutes were concise, and well-fitting for the start of the semester when everything is fairly new and unknown. Later on in the semester, however, they became an important part of the process by helping us save progress and review what we have already done and discussed in the following meetings.

Here we have an example of our first meeting, where we wanted to try a different approach from the previous semester. Considering we are using Jira to plan out and execute our project, we made use of its multiple templates and documentation suggestions, Meeting Minutes being one of them that proved to be useful by the end of the semester.

Meeting Minute 1

DATE: 03-10-2022

TIME: 13:20 - 16:20

PARTICIPANTS:

Selina Ceban

Mikolaj Morawski

Wiktor Ciolkowski

Aivaras Zevzikovas

GOALS:

- Establish a list of requirements;
- Assign importance.

ACTIONS:

Get started on the SEP3 Assignment, Part One.

It is to be noted that at this point in the semester, we were just starting, so there was not a lot of discussing and deciding as much as just working on our tasks to complete assignments. We were also experimenting with Jira a lot, trying out everything it has to offer. By our second and third meetings, however, we understood that for the logbook to be useful, we had to be more detailed with our goals, topics, and decisions.

Meeting Minute 2

DATE: 05-10-2022

TIME: 12:00 – 15:15

PARTICIPANTS:
Selina Ceban
Mikolaj Morawski
Wiktor Ciolkowski
Aivaras Zevzikovas

GOAL:
- Revise requirements;
- Establish use cases;
- Briefly explain use cases;
- Explain in detail the main use case;
- Establish conceptual domain model;
- Create a system architecture overview diagram;
- Create an SSD for the main use case.

DISCUSSION TOPICS:

Time	Item	Presenter	Notes
10:00 14:00	All of the above	Mikolaj Morawski	Request for assistance sent to the supervisor to elaborate on actor roles in requirements.

DECISIONS:

- 3 servers - stockfish C#, database Java, and main connection C#
- Communication technology - Web socket for low latency communication between user and comms server
- Communication technology - gRPC for reliable and relatively quick communication between Stockfish and Comms server
- Communication technology - REST for ease of use between the database server and comms server

Compared to our first documented meeting, the second one contained more information, which shows a line of progress when tracking from now to the beginning. This way, we can pinpoint exactly when and why we switched directions, as well as have factual information on the decisions we made and subjects we discussed.

Besides meetings with the group, we have also scheduled a few meetings with our main supervisor when reaching certain deadlines or milestones. Every three meetings or so, it was needed to check in with them, to not lose track of what we were supposed to be doing, especially since our project idea was a game, and game projects are known to have a high chance of losing focus and not following the semester goal.

Taking all of these factors into consideration, we ended up with a lot of new knowledge, as well as a project that was documented from start to finish. Next, we are going to reflect on how the formation of our group was decided, and how we got to the end.

2 Group Description

The third-semester project, just like the second one, presents the students with the opportunity of choosing their teammates. Thus, Group 6 was formed with 4 students that have already worked together before in a similar group project in SEP2. As a team of 4, Group 6 is culturally diverse, with the members finding their roots in 3 different countries and cultures.

Selina

Selina comes from the Republic of Moldova, a small, landlocked country that finds itself between Ukraine and Romania. Culturally and education-wise, Moldova differs greatly from Denmark, that's why upon starting her university studies here, she found herself in a new environment. It didn't help much that before coming to VIA, Selina had taken a Humanitarian Arts major, which means little to no prior experience in Software Engineering. That, however, is not an issue considering the ever-present need for knowledge one has. Having passed two semesters already, Selina gained a lot of new knowledge and experience that has left her intrigued and hopefully left its mark on SEP3. In semester one she got the basics of programming down, and more than that - she completed her first real group project, where everyone is needed to succeed. In the second one, she managed to learn more about what she is studying by creating a client-server system with her group, which helped her understand what Software Engineering is about, and what she should focus on from then on.

Aivaras

Aivaras comes from Lithuania, a country in Eastern Europe. Because Denmark has a vastly different work culture so it was necessary to acclimate to it, which thankfully was not very difficult. After having already worked in a group environment for the first and second semesters it has become easier to tackle new challenges. Continuously improving his programming skills while working on the project also gave him a helping hand in learning new ways to create software.

Mikolaj

Mikolaj comes from Poland. He has a technical background as he is coming from IT Technical School and working for an IT company for a short while. Mikolaj started programming in middle school and it became his big passion. Mikolaj participated in several EU programs for improving IT skills. He was taught web development and basic programming in technical school and came to VIA to further advance his skills. Mikolaj participated in several other projects, and he has experience managing a small team of junior developers, although Poland is a country where the loudest opinions win and Mikolaj had to adjust his ways to fit the welcoming culture of Denmark.

Wiktor

Wiktor comes from Poland. He learned to program in high school and has been doing personal projects ever since. He’s mainly interested in embedded systems and low-level programming. He moved to Denmark to experience studying software engineering abroad and explore a new culture. He’s already had experience working in a team from the two previous semesters.

Working together for two semesters in a row has let the group know one another better, as well as overcome certain aspects that concern visions and principles. Discipline and conflict solving have also been improved on and are continuously addressed where necessary so that everyone achieves their personal goals as well as the common goal of reaching the end of SEP3.

3 Project Initiation

The project initiation of the third semester for software engineering took place in a much shorter period than the previous two semesters. Although the three classes were merged and mixed into two, due to the possibility of submitting wishes for the classmates we wanted, a lot of the students in the class were familiar. Thus, group forming for a lot of the students happened as soon as the class composition was announced, and some of them formed it during the first lecture of SEP3, which took place on the 7th of September.

For Group 6, the group forming took place on the lecture day, and although we were a team in the second semester as well, it came to be a surprise that we found ourselves working together again. This time around, the criteria for working together changed a bit, mostly by taking into account the group dynamics and chemistry. Factors such as good performance and common goals influenced the decision of staying together, knowing each other’s potential, and work discipline.

After forming a group, it was time to think of project ideas that would fit within the scope of the third semester, it being a heterogeneous system with server-to-server communication and at least one database. We started brainstorming ideas to give us more options to choose from and to let our supervisor give their own opinion on the choice. Everyone gave out their suggestions, and proposals such as an instant messaging application, marketplace, peer-to-peer system, and rental system were given, but it was one particular idea that caught everyone’s attention.

At one moment, Wiktor suggested that we create a chess game, and at first, everyone dismissed the idea, because games have an 80% failure rate, but the more we thought about it, the more it grew on us. First of all, we noticed a lot of our classmates enjoy playing chess, and even within the group we had Aivaras and Wiktor play against each other from time to time. Second of all, we thought that creating a chess game would improve collaboration and competitive spirit within the school if it was to work.

With a hopeful mind, we included “Chess Game” in our list of project proposals and went to the supervisor meeting to discuss the options. The supervisor asked us what we would prefer to work on, and we told him that the game does sound intriguing and exciting, however knowing the factual information of them not always working, we are unsure. He encouraged us to go on with it if we wanted to because as long as we do not lose track of the actual scope, which is a distributed system, it will be a fun project.

This way, we decided on the idea of having a chess application as our project for the third semester, and after revising and rewriting our group contract, we could proceed with the project description.

4 Project Description

Once the project proposal was decided upon and approved, we could start diving into our subject, its background, and the problem present. After the first few lectures in SEP where we discussed important topics such as sustainability and technologies that aid global world goals, it was suggested that our project should have at least some tangency to them. This is where research on our topic started.

We had to find a reason for doing it, and we eventually got our answer. As it was described in the project report, chess as a game has existed for a long time but only recently has the world as it has been known replicated into a virtual one. With this, the opportunity to connect with people at a distance arises, and thus, we thought that creating a system that is both fun and intellectually challenging would be an original idea.

For the background description, we had to create a setup that would explain the reason behind needing a chess platform, so we gathered up a lot of sources that would support the idea of a virtual chess game, and came up with a fictional client in need of help. Once that was properly analyzed and written, we could try and move forward with understanding and describing the problem statement and definition of purpose.

Both of these parts of the project description are needed to be able to create a goal and understand what the delimitations of the project would be. It is important to stay realistic and precise when it comes to delimiting the system, in order not to lose focus and aim for an achievable end product.

Because in this semester we had the freedom of choosing our methodology and we didn't have to stick to one method only, we had to inform ourselves about the different kinds that are already available, and what would work best for us. At first, we didn't want to follow Scrum because we thought it might be a good idea to experiment with other ways of executing a project(which will be further touched upon in the next chapter), so we chose Kanban and found reasons why it would be a good idea to use it.

Reflecting, although we later changed our methodology, it was a good practice to research other methods and try them out. This way, we gained experience firsthand of what works for whom and why certain methodologies are better for us in the current state we are in.

We had two weeks to submit the first draft of our project description, followed by feedback from our main supervisor. Once that was discussed, we could proceed forward with preparing an architecture overview for our project.

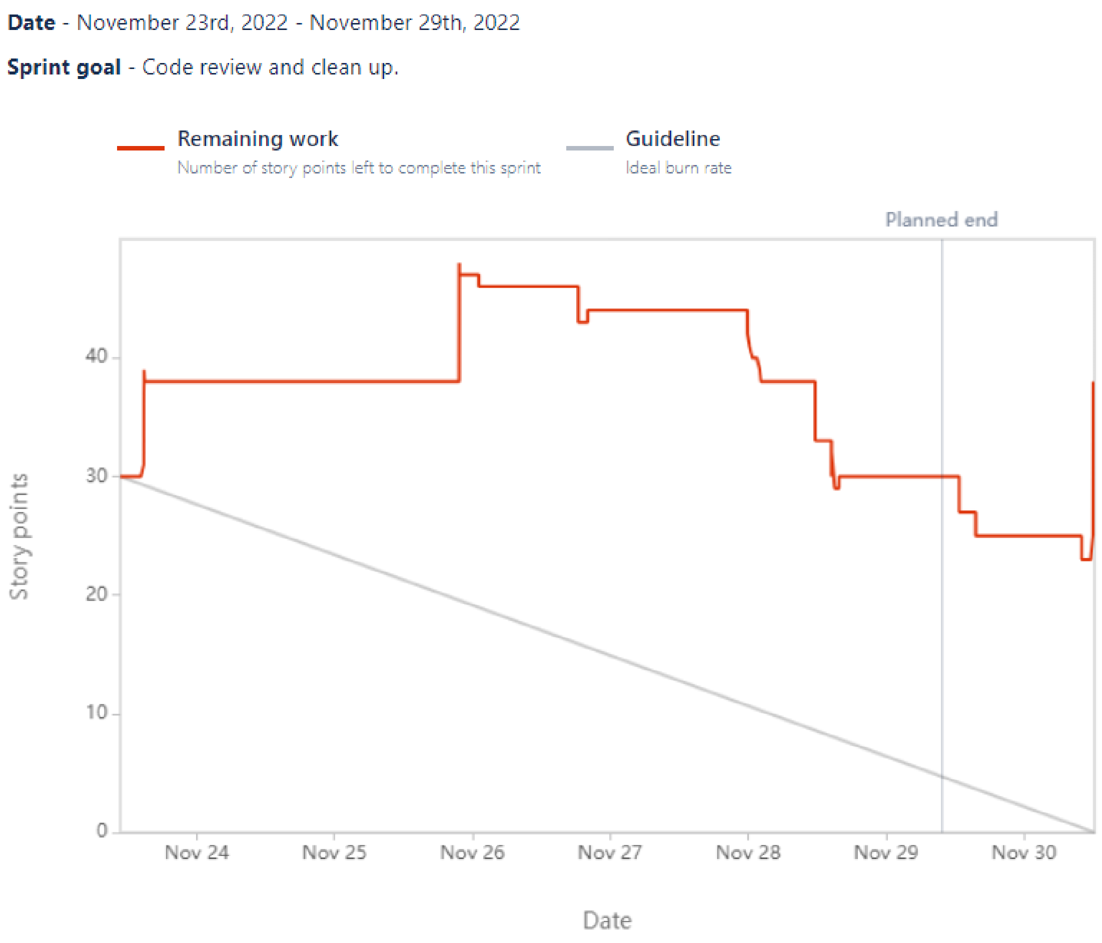
5 Project Execution

Reflecting on the rest of the duration of the project helps us notice certain patterns, especially when combined and compared to the previous reports. It is to be noted that this semester, we experimented around with the methodology used, so there are a few differences from week to week. We thought that we could try following Kanban, however shortly after entering the project execution it became slightly inconvenient because of the lack of deadlines and restraints.

We came to realize that something like Scrum, which "employs an iterative, incremental approach to optimize predictability and to control risks" (Schwaber & Sutherland, 2020) would work better for us as students, where we have so little time to create a product. Self-discipline is sometimes difficult to manage, so extrinsic motivation such as our weekly sprints or daily standups worked great for creating the system that we now have. The fact that we worked in Jira this semester also helped organize our progress and keep us up to date with the board and its tasks, user stories, reports, and diagrams.

An issue that we have picked over time (taking our second-semester project experience as well), was that Scrum is not meant for such short sprints as we executed them. We allocated 20 hours for each sprint, to keep both an adequate amount of sprints and also balance it with our other courses, and yet oftentimes we would overestimate ourselves and be left with a lot of tasks by the time we were supposed to finish a sprint.

For example, in Sprint 7(Appendix M), from a total of 36 issues and 76 story points, we only completed 15 issues worth 29 points, which was not even half of the planned amount. Here we have the example shown in our burndown chart:



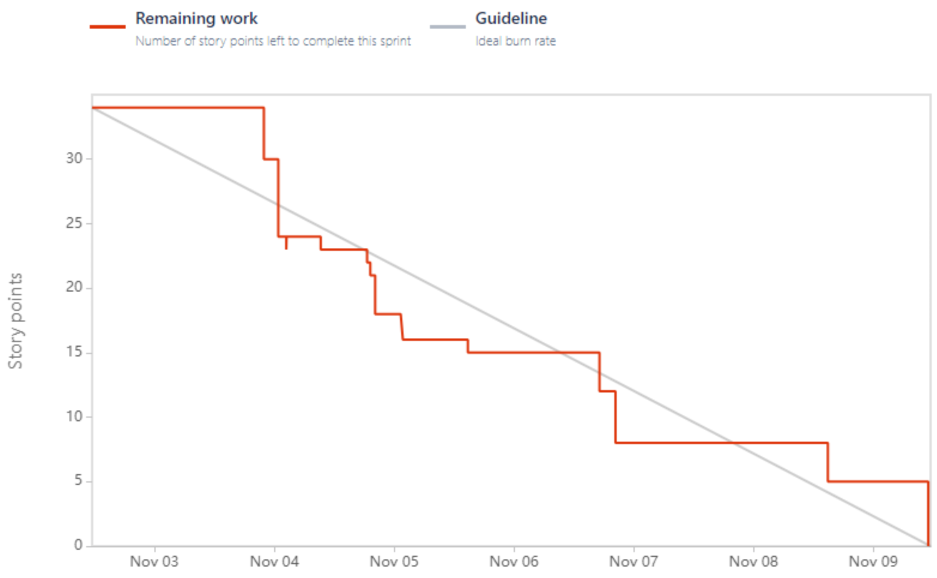
1. Burndown Chart of Sprint 7

A factor that might have contributed to the sprint looking like this was the fact that we were approaching the end of our classes and the start of our project period, which meant that assignments had to be finished and handed in.

Thankfully, we had weeks where we planned it accordingly, such as Sprint 4(figure 2) and Sprint 8(figure 3), where we completed the majority of the planned tasks:

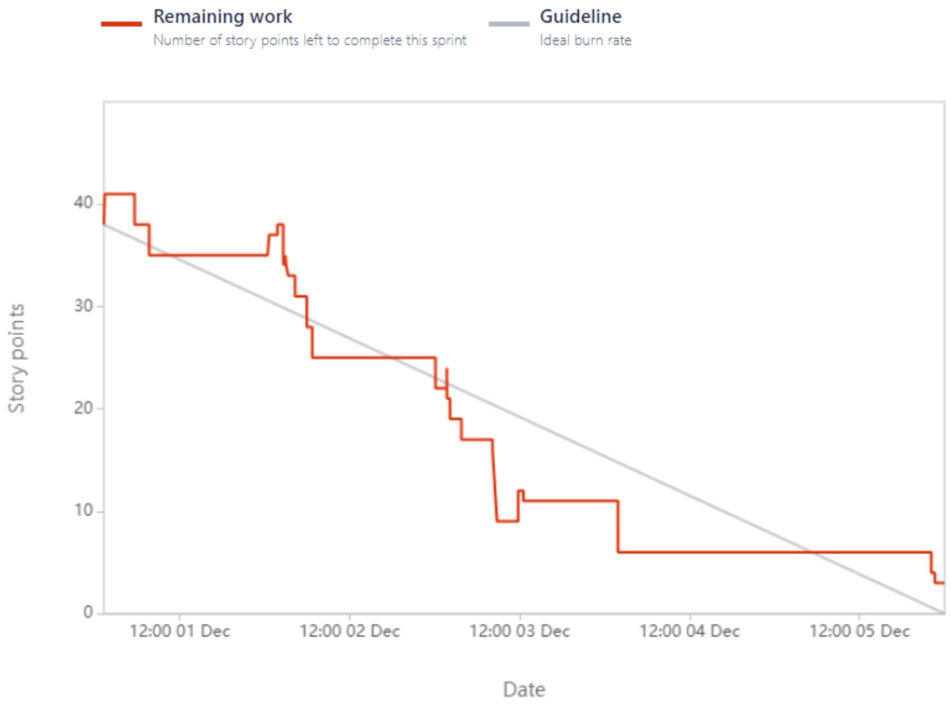
Date - November 2nd, 2022 - November 9th, 2022

Sprint goal - Polish Proof Of Concept



2. Burndown Chart of Sprint 4

Date - November 30th, 2022 - December 5th, 2022



3. Burndown Chart of Sprint 8

(the rest of the sprints can be found in Appendix M)

Reflecting, starting sprinting as soon as we did, came with great help: because of the complexity and uncertainty of our system, we had to start developing it as soon as possible, to avoid getting overwhelmed when the project period starts. We finished a total of 9 sprints and developed the majority of the features the product owner wanted to have in the system.

We made important changes along the way upon learning new information about certain technologies, such as switching from one technology to another towards the end of the project period or changing the UI design midway at the product owner's request. These changes helped us gain insight into the creation and constant change of a system before it is finished, and even made us conclude that it could always be improved and it is never really finished, and that's why new versions are always being worked on and released when it comes to applications and software.

To conclude, the execution of the project this semester proved itself more successful than the previous two semesters, possibly since it is now becoming a habit and we are familiar with the way of doing most things and are always excited to add onto it by combining the other courses into it with the newly accumulated knowledge.

6 Personal Reflections

Mikołaj Morawski

Project Topic

The group project was a significant undertaking this semester, and it was very exciting to work on. Each group member contributed to different parts of the project. We analyzed and designed the system together. In a way, every person in the group was a product owner, as everyone finds chess very entertaining. I enjoyed working on the Blazor system the most, as it is an interesting and new technology for me, but I expanded upon every project in the application. Engaging in project topics contributed significantly to overall group motivation.

Ambitions vs Timeframe

Due to all the excitement, it was easy to go off the rails and start planning way too ambitious features for the timeframe we had, such as system stress testing using bots that imitate user behavior. Fortunately, most of the time there was someone in the group to cool the expectations and focus on delivering features at the time. Our enthusiasm for the project didn’t die with the project hand in and we are already planning on expanding upon the project after the system is handed in, just for personal satisfaction.

Group Work

As a group, we are introverted and don't always discuss our ideas and opinions openly. I took on a leadership role and encouraged everyone to share their thoughts and ideas during meetings. I also made sure that each member had the opportunity to work in their areas of expertise, as well as try to develop new skills.

Problem-based learning and Supervision

Working on a project of our choosing, as well as support from supervisors whenever we had any issues allowed us to learn much more than what we would be able to with a predefined topic and requirements.

Supervisors were always available and eager to help whenever an issue arose. We also received a lot of positive feedback that resulted in greater group motivation and an even bigger drive to work on the project.

Yet full freedom might not be best for everyone and our group encountered several times where we almost drifted from the course of pursuing knowledge relevant to the current semester. It might be wise to set a few more obligatory meetings, where a group would share their progress and plans with the supervisors, just to make sure that current goals and objectives are correct.

Aivaras Zevzikovas

Project Topic

This semester our objective was to create a distributed system. Our group was quite ambitious and decided to create a distributed chess application. The scope of the application that we wanted to create was very wide, but our group was concerned about that because it was an idea that everyone in the group was excited about. Our group for this semester consisted of the same four group members as last time which meant that we already knew each other and could cooperate easily.

Methodology

The methodology we used this semester was SCRUM with UP (Unified Process). The methodology used was the same but this semester we improved upon our previous experiences. We also started using the Jira planning software to help with organizing our group. Using Jira helped us manage our group meeting notes and scrum board and the built-in tools for visualizing sprints meant that our group could better understand what issues we were facing and how to alleviate them for the next sprints. While we did improve our process, there were also still some issues left unresolved. Sometimes, the tasks that we created for our sprints were unclear and did not have a description explaining them. Other times the tasks were too broad and vague which meant that sometimes that task set to be done still had some parts that needed to be worked on.

Group Work

This semester we followed the same two meetings a-week schedule. Meeting up twice a week meant that we could revise our project aims often which were always changing. The part of the process which we improved upon was having daily stand-up meetings where we notified other group members what work was finished the previous day and what will be worked on the day of the meeting.

Problem-based learning and Supervision

While working on this project our group learned how to apply a wide variety of technologies. Because there were many new technologies that we needed to learn, it was necessary to divide the work in the group. Dividing these responsibilities and then teaching and informing the other group members about them helped the group to use these technologies.

Our group's supervisors were very helpful and answered our questions whenever we had them. Because sometimes we were getting sidetracked, our supervisors helped us focus on the important issues of our project.

Conclusion

To sum up, while we could not finish all our ideas for this system, it was still an interesting and engaging experience where I learned many new technologies and made another step in my journey as a software developer.

Wiktor Ciolkowski

Project topic

The third-semester project was about making a chess platform application. We knew that it would take a lot of effort to properly implement such a system, but we were very excited to work on it at the same time. Throughout the development of the project, we had to slightly shift the objective of the project as there was not enough time to implement everything we initially thought of doing. In the end, we managed to successfully create a working application with the main objectives finished. Even though there was not enough time to finish all the initial features we planned, since we're very engaged in the project, we plan on continuing its development.

Group work

This semester we continued working with the SCRUM methodology as it proved very efficient and effective last semester. In the beginning, we thought of switching to KANBAN, but in the end, we decided to stick to SCRUM as we already had quite a lot of experience with it. All our group members were from the previous semester, so we already had a feeling of what it would be like to work with each other again. We followed our last semester's schedule of meeting twice a week, but we also established daily stand-up meetings which allowed us to be on the same page with what was going on with the project and each group member.

Problem-based learning and Supervision

This semester we learned a lot of new technologies and we were able to put them into practice with assignment work and the semester project. This greatly improved my understanding of software development.

Throughout the project period, we were getting support and advice from our supervisors. It was very helpful, whenever we needed guidance in the right direction with the project. We were able to get feedback on our questions very quickly which saved us a lot of time.

Conclusion

Overall, I feel like I’ve gained a lot of new knowledge about software development this semester. The project turned out to be quite well done in the end, even though we couldn’t finish everything we planned at the beginning. In the end, I think I managed to improve a lot in both technical and non-technical skills.

Selina Ceban

The third semester in terms of SEP flowed differently, starting with the topic selection to the supervision. Our group selected a different idea this time around, and instead of falling within ‘safe’ options, after giving it some thought, we decided on a chess game. Even though the word “game” does sound like a red flag, in retrospect it all went better than expected.

The group was formed just like last semester, keeping its members and principles. I feel like it was more comfortable to work together on this project since we already went through this together once before. We wanted to change some ways of working, such as our planning tool or methodology, so we planned to choose it together. This way, for methodology we chose Kanban but later on switched to Scrum and Unified Process, and for our planning tool, we chose to use Jira instead of Trello, thanks to its multiple features made specifically for working with Scrum.

Shortly after forming our group and discussing some details, we were ready to proceed to the Project Description. In my opinion, a lot of what we initially aimed for and planned changed by the time we were done with our system. For example, the goal we set felt unrealistic, to begin with, considering it was a high-effort type of project. We wanted to achieve as much as possible, so we thought of all the things we could add to our project, and even though some of it wasn’t obtained, I think that with more time, it all would have come anyways.

To reflect on our Project Execution, there are a lot of things I would have liked to do differently, at least on my part. For example, I wish I were as passionate as my team is, or I wish I had as much discipline to go out of my way for a project. While enjoying the process of working on a project, there have been quite a few difficulties I have stumbled upon that might have held the group back, one of them being my slower-than-average pace.

The methods we have worked with have changed a bit throughout the execution. As mentioned above, at first we thought that using a different methodology would improve and diversify our knowledge in working with multiple methods. Quite early in the project, however, we switched back to Scrum, for reasons explained in other chapters. To reflect, I think that what we used from then till now (Scrum & UP) worked even better than the last semester thanks to our experience and Jira as well. If we were to start the project today, I feel like I would still choose Scrum unless shown as a better option for students as we now are.

In retrospect, although the goals were big, my opinion is that the results achieved are bigger than what we have accomplished until now. I am extremely satisfied with the results, if not even delighted. It is a product my teammates and I have placed a lot of effort into, so, naturally, I would be thankful and proud of it. There were a few risks established from the beginning, like the fact that it might not work at all, or the possibility of us getting sidetracked and forgetting the actual goal of SEP3. We however made a conscious effort to not lose sight of the main goal and accomplish something that both fits the criteria and is to our liking.

If I were to reflect on our group work, just like last semester I am very thankful to work with the team I am in. Everyone tried their best to live up to the group contract that was set at the beginning of the semester and everyone felt responsible for putting in as much effort as they could. Considering we have been working together for 8 months now, I think that there is a common consensus of trusting each other and holding each other accountable where necessary. There is always room for improvement, and constructive criticism as well, - two of the things that are present and present some of the main reasons this group works well together.

Retrospecting the way we have been provided supervision this semester leaves me with a few comments in mind. First of all, I am now understanding how important it is to reach out to your teachers for supervision. When I first heard about the way supervision is handled at VIA as the semesters progress, if I am being honest, I got intimidated. However, I have realized that it is no shame to ask and talk and show the progress to your supervisor, for they provide a greater result than not cooperating and bothering them at all. Although I still feel like supervision can mostly be used for what the word stands for, it is a step ahead of doing everything yourself and not obtaining insight from someone who is technically out of the picture.

The way my group and I handled the cooperation with our main supervisor, Jakob, proved to be satisfactory and even enjoyable. It has been both a rocky and a smooth process and taking our teacher's opinion into account only benefited us all through it.

In conclusion, most things from group work to supervision have left me with a hopeful heart for the future. I am excited to try to give my best once again in the following semesters and become as much of a valuable asset to the team as my group mates are.

7 Supervision

Supervision in SEP3 took place similar to the one in the second semester, with more freedom to self-regulate the work amount and deadlines. This time, however, we learned how to better make use of this opportunity of having someone to guide us in certain areas, compared to SEP2, where we were too unsure of talking with our supervisor and contacting him a mere two times.

As we advance in semesters, the difficulty of the project also increases, so to keep us on the right track, we had quite a few pre-planned meetings with our main supervisor. The meetings consisted of checking in on the progress, from the project proposal, where we presented ideas we would like to work on this semester, to the project description review, system architecture feedback, and the proof of concept.

Our supervisor plays a great role in the final choice of subject for our project, for his approval and encouraging words gave us the green light to proceed in creating something that we were excited for. His showing support without much hesitation let us become hopeful and motivated to succeed at something that so rarely works within the university's restraints.

Besides the mandatory deadlines, oftentimes we would communicate with our supervisor through emails. The group would gather and list down questions related to our system, then discuss them to make sure the supervisor would get a good picture of our problem. The feedback was always straight to the point and reassuring where it was seen fit. If not by email, we could find them after class either in the classroom or in the office, always ready to answer our questions.

Guidance was really important this semester, due to the newly gained project complexity and the new programming language we had to learn. Thankfully, supervision was executed effectively and that was strongly appreciated. And so, we have reached the end of SEP3.

In general, the warm and encouraging attitude of the supervisors and mentors helped us keep going. Through trial and error, and accepting that we do not know some things yet.

An own learning process is just that: testing study techniques until one of them works for you, reflecting on your motivators and finding out how you can move towards intrinsic motivation for greater self-fulfillment and sufficiency, and providing feedback both for yourself and for people you are working with, for better teamwork.

8 Conclusions

Working on SEP for the third time around led us to find out more about what to do and not to do in group work and helped us gain even more skills that will hopefully come useful for the following semester and eventually for our internships. This time, the semester project became a pastime, because now we found ourselves to be more familiar with the way of doing things and the general process of it.

Besides that, we also gained more freedom in directing the flow of work, choosing the desired methodology and software that would keep track of our progress. Analyzing the past 15 weeks from the beginning of the semester up until now left us with a few postscripts, which will be split into 3 points:

The main recommendation is,- don't be afraid to challenge yourself. When working in a group it is important to take account of everyone's potential, however, it is always welcome to try doing a bit more than you are comfortable doing at the moment. Whether it's taking a leadership role, involving yourself more in the process of implementation and coding, or documenting areas you are not an expert in, learning is always a valuable experience and can turn into an asset eventually. Success is, of course, not linear, so failure and downs are expected at times, but rain comes with the sun after.

The second piece of advice is, to try using all the resources that are available to you: libraries, the internet, supervisors, tutorials, and other friends. We are not all-knowing creatures, so, naturally, we won't know everything possible. A recommendation that is not taken enough into account, or at least wasn't by us for a long time, is asking for help and contacting your supervisor when stuck. In our second semester, and for a while in our third, we would often feel confused about certain subjects or details, and sometimes even about the whole project and direction, and yet, we wouldn't reach out. That however has changed midway, as we have come to realize that consulting yourself with someone from the outside offers valuable insight.

Lastly, you don't have to form close friendships, but it is nice to form amicable relations with the people you are working with daily. Even though after the project is finished you might not keep in touch like you would when doing daily stand-ups, it always benefits everyone to keep a warm, collegial atmosphere in the group.

Reflecting, we have gained more insight into the process of being able to create a system by ourselves, and we gained the possibility of choosing from a wider range of ideas, which in turn provoked us to study outside of the curriculum. Reflecting forward, although SEP is becoming more and more complex and of proportions, so to say, it is also turning into an exercise to see what you can challenge yourself within this new upcoming semester, which keeps the studies both relevant and practical.

9 Sources of information

Schwaber, K. and Sutherland, J., 2020. *The Scrum Guide*. [online] The Scrum Guide The Definitive Guide to Scrum: The Rules of the Game. Available at: <https://billlewistraining.com/wp-content/uploads/2017/02/PMP-Agile-Study-Materials.pdf> [Accessed 15 Dec. 2022].

10 Appendices

Appendix L - Meeting Minutes

Appendix M - Sprint Book