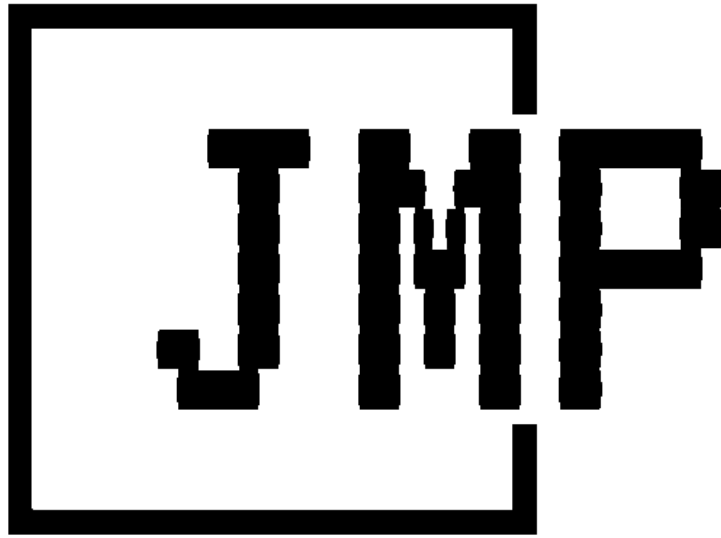


JMP Designdocument



Made by:

Mitchel Meskes

Joshua de Bruijn

Pedro Eduardo Cardoso

Date: 11 February 2024

Table of contents

-	Table of contents	
-	Intro	3
-	Important roles	4
-	Plan of requirements	5
-	Corporate Identity	6
-	User stories	7
-	Wireframe	12
-	Mockup	15
-	Database model	17
-	ERD	18
-	Design patterns	19
-	UML Diagrams	24
-	Collaborations	31
-	Unit Tests	32
-	Security	33

Intro

Welcome to our C# Pong coding project!

One of the first and most iconic video games, it has entertained generations of gamers and has become a cornerstone in the history of the video game industry. In this project, we will recreate the classic Pong game using the C# programming language.

Pong is known for its simple but addictive gameplay, where two players each operate a paddle to bounce a virtual ball back and forth. The goal is simple: prevent the ball from hitting your side of the screen, while trying to hit the ball back to your opponent to score points.

Through this project we will not only explore the basics of C# programming, but also understand fundamental concepts of game development, such as managing game state, handling user input, and rendering on-screen graphics.

Let's dive into the fascinating world of Pong development together and bring our own version of this timeless game to life!

Important roles

The Product Owners

Jan Zuur

Ron van Zuilichem

Tom Sievers

SCRUM Master

Joshua de Bruijn

DEV Team

Pedro Eduardo Cardoso

Joshua de Bruijn

Mitchel Meskes

Plan of requirements

Key Features

The application will use object-oriented programming in C# to structure the code.

The main functionalities of the application include:

Gamemode selection, score display, winner display.

General goals

Building a user-friendly Pong Game.

Testing and validating the application to ensure that it meets the set requirements and standards.

Corporate identity

Color pallet/ Font


The color palette for Pong will consist of original colors:

Used color:

Player 1 bad: #0000FF (blauw)

Player 2 bad: #FF0000 (rood)

Pong Ball: #FFFFFF (wit)

Background:  #000000

Details: #FFFFFF

Font: Default font.

This combination creates a quiet and netting environment for the application, and it evokes nostalgia for the pre-existing game.

Buttons

The buttons will be black and white.

background buttons:  #000000

Background button/ text: #FFFFFF

User stories

General Stories

The user stories were created using Azure DevOps/Github Projects.

How we create our User stories is:

We choose an EPIC then divide it into a number of Sprints and in one of those sprints we then pick up parts of that EPIC so that each one works on a part of the project.

(See the following example)

Step 1

Choose an EPIC. (User registration and authentication).

Step 2

Then we divide those into parts and each picks a story.

Step 3

Then each will choose one of the following and work with it.

Every EPIC we handle we run all the roles one through so that everyone has worked on all areas both backend, frontend and databases.

User stories

Epic 1: Maak Pong

User Stories for Frontend 1.1:

User Story 1.1.1: Design Game Board Interface

As a player, I want to see a well-designed game board interface, so that I can easily understand the game layout.

Acceptance Criteria:

- Design a visually appealing game board with clear boundaries and markings for the playing area.
- Ensure the game board layout is intuitive and easy to understand.

User Story 1.1.2: Implement Paddle Display

As a player, I want to see the paddles displayed on the game board, so that I can understand my position.

Acceptance Criteria:

- Display paddles as rectangular shapes on both sides of the game board.
- Ensure paddles are positioned correctly and respond to player input for movement.

User Story 1.1.3: Display Ball

As a player, I want to see the ball moving on the game board, so that I can understand the game's dynamics.

Acceptance Criteria:

- Display the ball as a circular shape moving across the game board.
- Ensure the ball moves smoothly and bounces off walls and paddles accurately.

User story's

User Story 1.1.4: Score Display

As a player, I want to see my score displayed on the screen, so that I can track my progress.

Acceptance Criteria:

- Display the player's score prominently on the screen.
- Update the score when a player scores a point by hitting the opponent's wall.

Backlog Items for Frontend 1.1:

Backlog Item 1.1.1: Implement Game Rendering Logic

As a front-end developer, I want to write code to render the game elements on the screen, so that players can see the game environment.

Acceptance Criteria:

- Use a graphics library or framework to render game elements such as the game board, paddles, ball, and score.
- Ensure the game renders correctly on different screen sizes and resolutions.

User story's

User Stories for Backend 1.2:

User Story 1.2.1: Manage Paddle Movement

As a developer, I want to enable paddle movement based on player input, so that players can control their paddles.

Acceptance Criteria:

- Capture player input from keyboard or controller to move paddles up and down.
- Limit paddle movement within the boundaries of the game board.

User Story 1.2.2: Implement Ball Physics

As a developer, I want to define the behavior of the ball, including its movement and collisions, so that the game feels realistic.

Acceptance Criteria:

- Define ball movement using physics principles such as velocity, acceleration, and collision detection.
- Ensure the ball reflects off walls and paddles accurately according to the laws of physics.

User Story 1.2.3: Detect Collisions

As a developer, I want to detect collisions between the ball and paddles, as well as with the game boundaries, so that I can handle game interactions.

Acceptance Criteria:

- Implement collision detection algorithms to detect when the ball collides with paddles or game boundaries.
- Trigger appropriate actions such as bouncing the ball off surfaces or updating player scores.

User story's

Backlog Items for Backend 1.2:

Backlog Item 1.2.1: Implement Game Mechanics

As a backend developer, I want to code the core game mechanics for Pong, including ball movement, collision detection, and scoring rules.

Acceptance Criteria:

- Write code to manage the game loop, including updating the game state and rendering frames.
- Implement scoring logic to award points when the ball hits the opponent's wall.
- Ensure game mechanics are implemented accurately and efficiently.

Wireframe

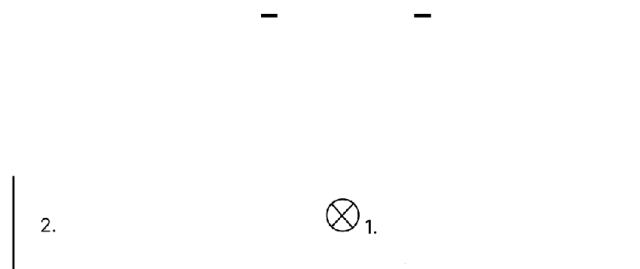
The wireframe consists of the following pages:

Start Page, Show the Home page with a play button.



1. Is the box with the gamemode choice and start.

SinglePlayer Page, Shows the Singleplayer page with the two bats and a ping pong ball and the current score.



1. Starting position of the ball after score and the start of the game.

2. Number two are the bats and their starting position

Wireframe

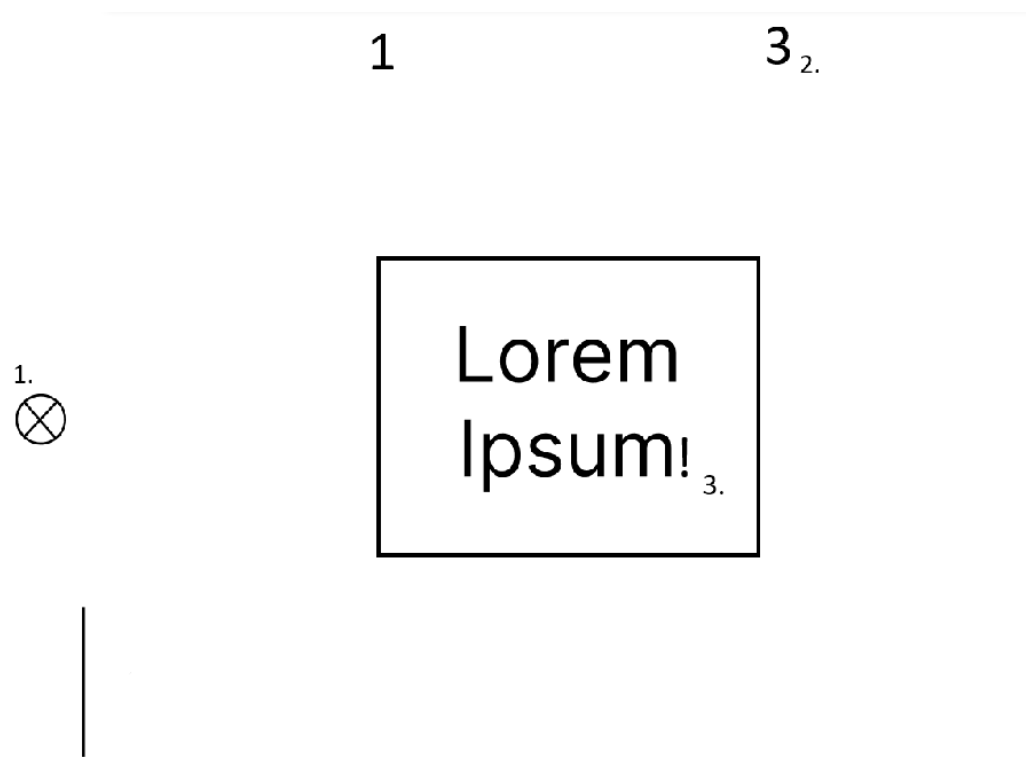
SinglePlayer (after Score) Page, Shows the Singleplayer page with the points scored after hitting the opponents goal.



1. After collision with the back wall, the the pop up will occur with who scored.
2. Moveable bat.
3. A text pop-up showing the player who scored.

Wireframe

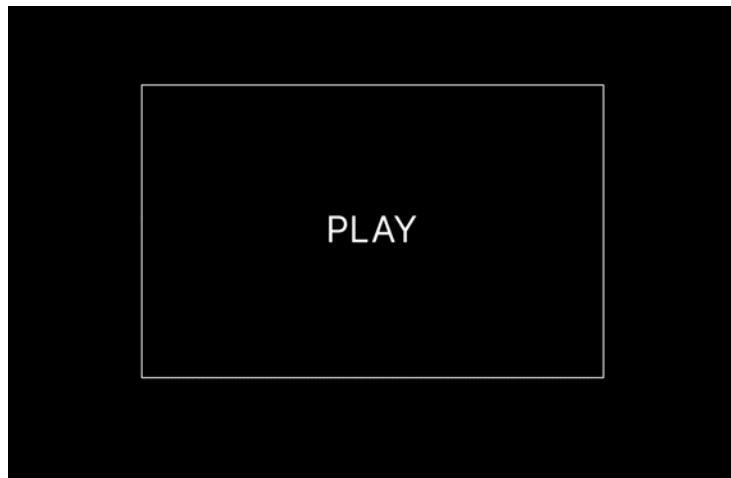
SinglePlayer (after Win) Page, Shows the Singleplayer page with the points scored and the win screen with the player who won.



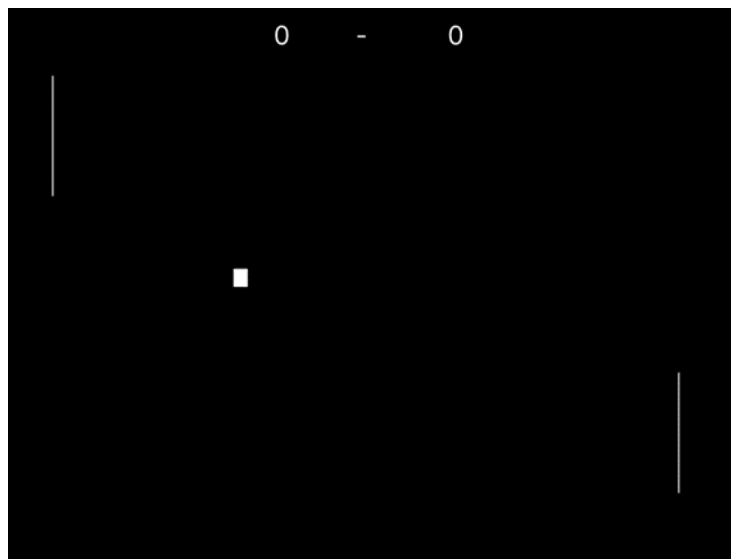
1. after the third collision with the back wall, one of the players won
2. place of scores
3. Pop up in a box with the text with who won.

Mockup

Start Page

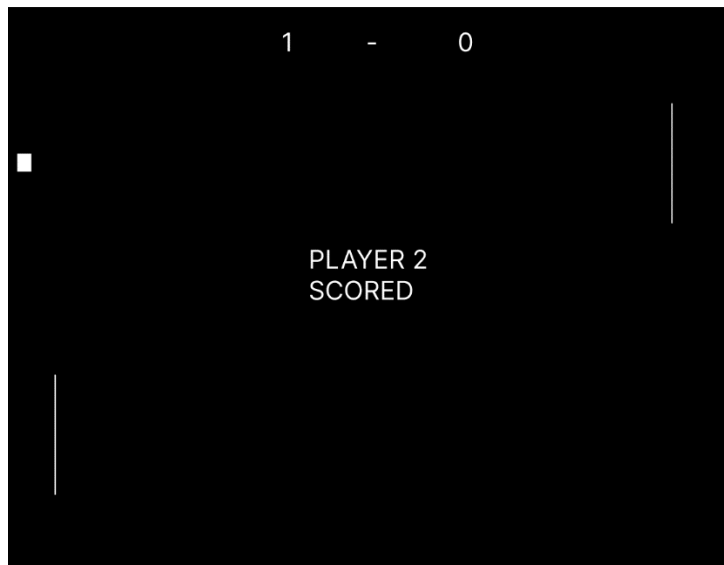


SinglePlayer Page

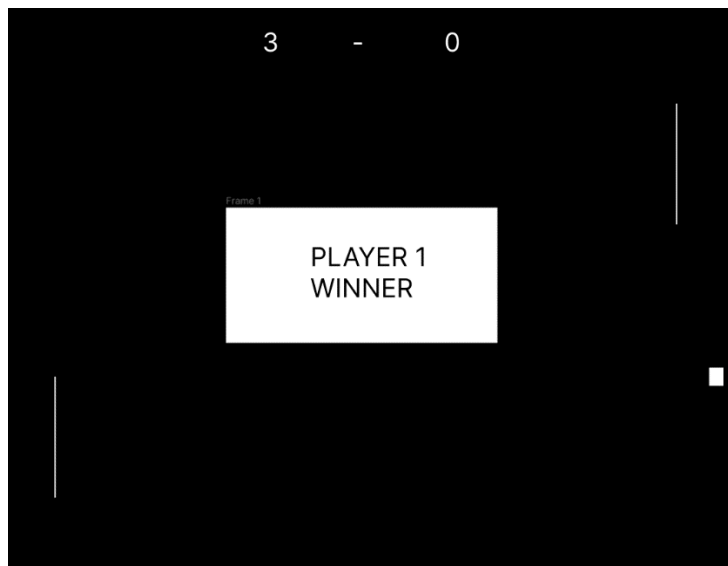


Mockup

SinglePlayer (after scoring) Page



SinglePlayer (after Win) Page



Database model

Entities and Fields

Users

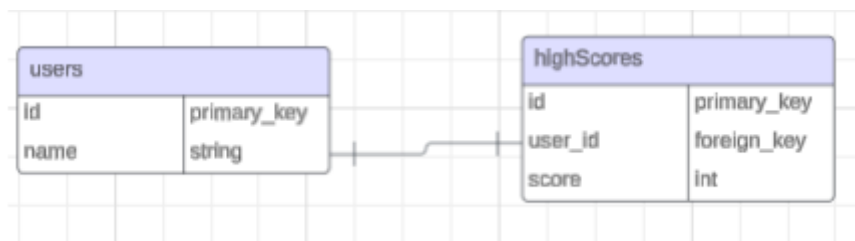
- ID (PK)
- name

Users

- id (PK)
- user_id (FK)
- score

ERD

ERD: For the database design.



The user gets registered with a name, scores are assigned to users.

Design Patterns

Creational Patterns

Creational patterns in software development are design patterns that deal with object creation mechanisms, trying to create objects in a manner suitable to the situation. Creational patterns aim to provide flexibility in the instantiation process while promoting the reuse of existing code and adhering to principles like encapsulation and separation of concerns.

1. **Singleton Pattern:** Ensures that a class has only one instance and provides a global point of access to that instance. It's useful when exactly one object is needed to coordinate actions across the system.
2. **Factory Method Pattern:** Defines an interface for creating an object, but allows subclasses to alter the type of objects that will be created. It provides a way for a class to delegate the instantiation logic to subclasses.
3. **Abstract Factory Pattern:** Provides an interface for creating families of related or dependent objects without specifying their concrete classes. It's useful when a system should be independent of how its objects are created, composed, and represented.
4. **Builder Pattern:** Separates the construction of a complex object from its representation so that the same construction process can create different representations. It's useful when the construction process must allow for different representations of the object that's being built.
5. **Prototype Pattern:** Creates new objects by copying an existing object, known as the prototype. This pattern allows for reducing the cost of creating objects compared to creating them from scratch.
6. **Object Pool Pattern:** Maintains a pool of reusable objects for use by multiple clients. It's useful when the cost of initializing a class instance is high or when instantiation and destruction of a large number of instances would degrade system performance.
7. **Dependency Injection Pattern:** Provides a technique to inject dependencies into an object rather than having the object create its dependencies. It promotes loose coupling and allows for easier testing and configuration management.

These creational patterns provide solutions to various problems encountered during object creation and initialization, helping developers to write more maintainable, flexible, and efficient software systems. Choosing the appropriate creational pattern depends on the specific requirements and constraints of the system being developed.

Behavioural Patterns

Behavioral patterns in software development are design patterns that focus on how objects interact and communicate with each other. These patterns define patterns of communication between classes and objects, emphasizing the assignment of responsibilities between objects and how they interact to achieve desired behaviors. Behavioral patterns help in making the system more flexible and easy to understand by promoting loose coupling and high cohesion.

1. **Observer Pattern:** Defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically. It's useful when there is a need for a consistent state between multiple objects.
2. **Strategy Pattern:** Defines a family of algorithms, encapsulates each one, and makes them interchangeable. It allows the algorithm to vary independently from clients that use it. It's useful when different variations of an algorithm are required.
3. **Command Pattern:** Encapsulates a request as an object, thereby allowing for parameterization of clients with queues, requests, and operations. It's useful for implementing undo functionality, auditing, and logging.
4. **Chain of Responsibility Pattern:** Allows a set of objects to handle a request one by one, providing a flexible way to pass a request along a chain of handlers. It's useful when the system should process a request through multiple handlers, and the specific handler to process the request is not known a priori.
5. **Iterator Pattern:** Provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation. It's useful when there is a need to traverse a collection of objects without knowing its internal structure.
6. **State Pattern:** Allows an object to alter its behavior when its internal state changes. The object appears to change its class. It's useful when an object's behavior depends on its state, and it must change its behavior dynamically based on that state.
7. **Visitor Pattern:** Defines a new operation to a set of objects without changing their class. It's useful when there is a need to perform operations on elements of a complex structure without adding new methods to their classes.
8. **Interpreter Pattern:** Defines a grammar for interpreting sentences in a language and provides a way to evaluate sentences in a language. It's useful when there is a need to interpret sentences or commands in a language.
9. **Mediator Pattern:** Defines an object that encapsulates how a set of objects interact. It promotes loose coupling by keeping objects from referring to each other explicitly and allows for a more flexible and reusable design.

These behavioral patterns provide solutions to various challenges encountered in designing systems where objects need to communicate and collaborate effectively. Choosing the appropriate behavioral pattern depends on the specific requirements and constraints of the system being developed.

Concurrency Patterns

Concurrency patterns in software development are design patterns that address the challenges related to managing concurrent execution and synchronization in multi-threaded or distributed systems. These patterns help in designing systems that efficiently handle multiple tasks running simultaneously, ensuring proper synchronization and coordination between concurrent operations.

1. **Thread Pool Pattern:** Pre-creates a pool of threads to execute tasks concurrently, avoiding the overhead of creating and destroying threads for each task. It's useful when there's a need to limit the number of concurrent threads and manage resources efficiently.
2. **Producer-Consumer Pattern:** Decouples the production of tasks (produced by one or more threads) from their consumption (consumed by one or more threads), using a shared data structure such as a queue. It's useful when there's a need to balance the workload between producers and consumers, preventing resource contention.
3. **Readers-Writers Pattern:** Manages access to a shared resource that can be read by multiple threads simultaneously but only written to by one thread at a time. It's useful when there are multiple readers and occasional writers accessing a shared resource, and it's necessary to prioritize access to ensure data consistency.
4. **Barrier Pattern:** Synchronizes a group of threads at a specific point in their execution, forcing them to wait until all threads reach the barrier before proceeding. It's useful when there's a need to coordinate multiple threads to perform a task in stages.
5. **Semaphore Pattern:** Controls access to a shared resource using a counter that limits the number of threads allowed to access the resource concurrently. It's useful when there's a need to limit access to a shared resource or control the number of threads accessing it.
6. **Mutex Pattern:** Provides mutual exclusion to shared resources by allowing only one thread to access the resource at a time. It's useful when there's a need to prevent multiple threads from concurrently modifying shared data, ensuring data integrity.
7. **Monitor Pattern:** Uses synchronization primitives such as locks to protect critical sections of code and coordinate access to shared resources. It's useful when there's a need to ensure exclusive access to shared resources while minimizing contention and deadlock.
8. **Actor Pattern:** Models concurrent computations as independent actors that communicate through message passing. Each actor has its own state and processes messages asynchronously, enabling scalable and fault-tolerant systems.
9. **Futures and Promises Pattern:** Represents the result of an asynchronous operation that may not yet be available, allowing the program to continue executing other tasks while waiting for the result. It's useful for asynchronous programming and parallel computation.

These concurrency patterns provide solutions to various challenges encountered in designing concurrent and parallel systems, helping developers to write scalable, responsive, and efficient software. Choosing the appropriate concurrency pattern depends on the specific requirements and constraints of the system being developed, such as the level of parallelism required, the nature of shared resources, and the desired synchronization mechanisms.

Structural Patterns

Structural patterns in software development are design patterns that focus on organizing classes and objects to form larger structures, making it easier to manage complex systems and relationships between components. These patterns help in defining how classes and objects are composed to create larger structures while keeping them flexible and efficient.

1. **Adapter Pattern:** Allows incompatible interfaces to work together by providing a bridge between them. It's like using an adapter to plug a European device into a North American outlet.
2. **Bridge Pattern:** Separates an abstraction from its implementation so that the two can vary independently. It's like building a bridge that connects two different islands but can be built using different materials.
3. **Composite Pattern:** Composes objects into tree structures to represent part-whole hierarchies. It's like organizing files and folders on a computer where folders can contain both files and other folders.
4. **Decorator Pattern:** Dynamically adds new functionality to objects by wrapping them with other objects. It's like adding toppings to a pizza without changing its base ingredients.
5. **Facade Pattern:** Provides a simplified interface to a complex subsystem, making it easier to use. It's like using a remote control to operate various devices instead of dealing with each device's controls individually.
6. **Flyweight Pattern:** Shares common objects to reduce memory usage, especially when dealing with a large number of similar objects. It's like sharing resources such as pens and paper in a classroom rather than giving each student their own set.
7. **Proxy Pattern:** Provides a placeholder for another object to control access to it. It's like using a security guard to control access to a building, allowing or denying entry based on certain criteria.
8. **Mixin Pattern:** Allows objects to inherit functionality from multiple sources without requiring multiple inheritance. It's like adding specific traits to a character in a game to enhance their abilities.
9. **Module Pattern:** Organizes code into modules or namespaces to improve maintainability and encapsulation. It's like dividing a large book into chapters and sections to make it easier to navigate and understand.

These structural patterns provide solutions to various challenges encountered in designing software systems, helping developers to create well-organized, scalable, and maintainable code. Choosing the appropriate structural pattern depends on the specific requirements and constraints of the system being developed, such as the need for flexibility, scalability, or performance optimization.

Our only used patterns are due to a lack of code to apply the other patterns to.

The configuration class is our singleton; we chose this design pattern so that the configuration variables are accessible everywhere.

Configuration	
FieldHeight	static int
FieldWidth	static int
Velocity	static int
currentTime	DateTime
lastTime	DateTime
YStart	static int
XStart	static int
Score01	int
Score02	int
BatSize	static int

Ball	
_ball	string
_diraction	int
_yPartial	double
_ballPosition	int[]
_config	Configuration
Ball()	
RollBall()	

Bat	
GenerateBats()	
MoveUp	
MoveDown()	
MovingLeftBat()	
MovingRightBat()	

The bat and field classes are builder classes that are certain ones that take care of generating the playing field objects.

UML Diagrams

Unified Modeling Language (UML) diagrams are graphical representations used in software engineering to visualize, specify, construct, and document software-intensive systems. UML provides a standard way to model software systems, making it easier for developers, analysts, and stakeholders to communicate ideas, understand system behavior, and design software effectively.

There are several types of UML diagrams, each serving a specific purpose:

1. **Use Case Diagrams:** These describe the interactions between users (actors) and the system, showing the various ways the system can be used.
2. **Class Diagrams:** These illustrate the structure of the system by depicting classes, their attributes, methods, and relationships between them.
3. **Sequence Diagrams:** These depict interactions between objects in a sequential order, showing how messages are exchanged over time.
4. **Activity Diagrams:** These represent workflows or processes within the system, showing the flow of control from one activity to another.
5. **State Diagrams:** Also known as state machines, these model the states of objects and transitions between states based on events.
6. **Component Diagrams:** These show the physical components of the system and their relationships.
7. **Deployment Diagrams:** These depict the physical deployment of artifacts on nodes, such as hardware or software components.

UML Diagrams

UML diagrams are used in software development for several reasons:

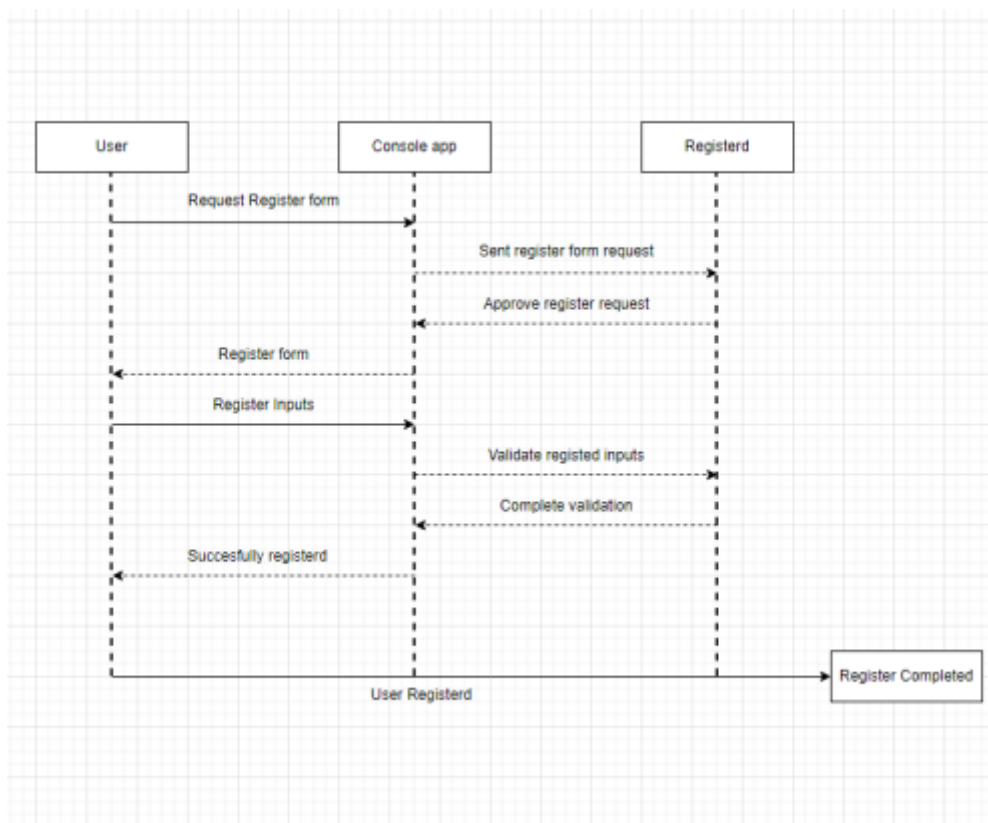
1. **Visualization:** They provide a clear and concise way to represent complex systems visually, making it easier for stakeholders to understand and discuss system requirements and designs.
2. **Specification:** UML diagrams serve as blueprints for software systems, documenting the system's structure, behavior, and interactions.
3. **Design:** They aid in the design process by helping developers to plan, organize, and refine the architecture of the software system.
4. **Communication:** UML diagrams facilitate communication among team members, allowing them to share ideas, clarify requirements, and collaborate effectively.
5. **Analysis:** They support analysis activities, such as identifying potential design flaws, performance bottlenecks, or inconsistencies in requirements.
6. **Documentation:** UML diagrams serve as valuable documentation for future reference, providing insights into the system's design and rationale behind design decisions.

Overall, UML diagrams play a crucial role in software development by improving communication, fostering collaboration, and aiding in the design and documentation of software systems.

Sequence Diagram

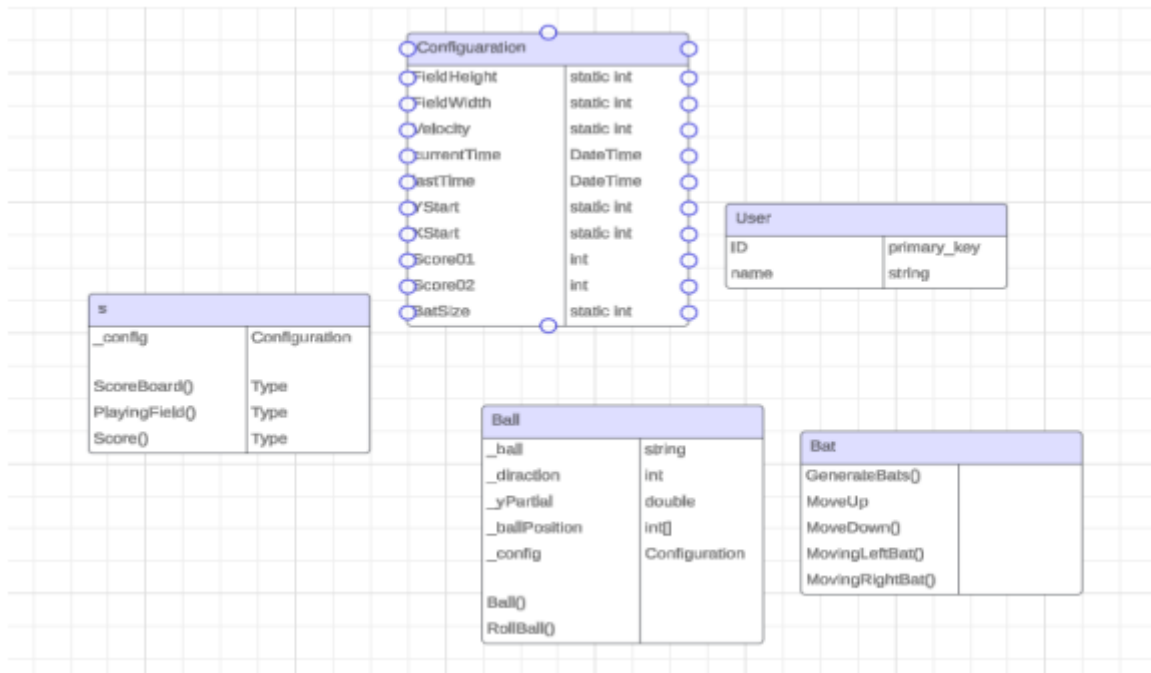
Sequence Diagram: Describes the interaction between objects in a sequential order.

User Registration Sequence: Shows steps encountered to be recorded.



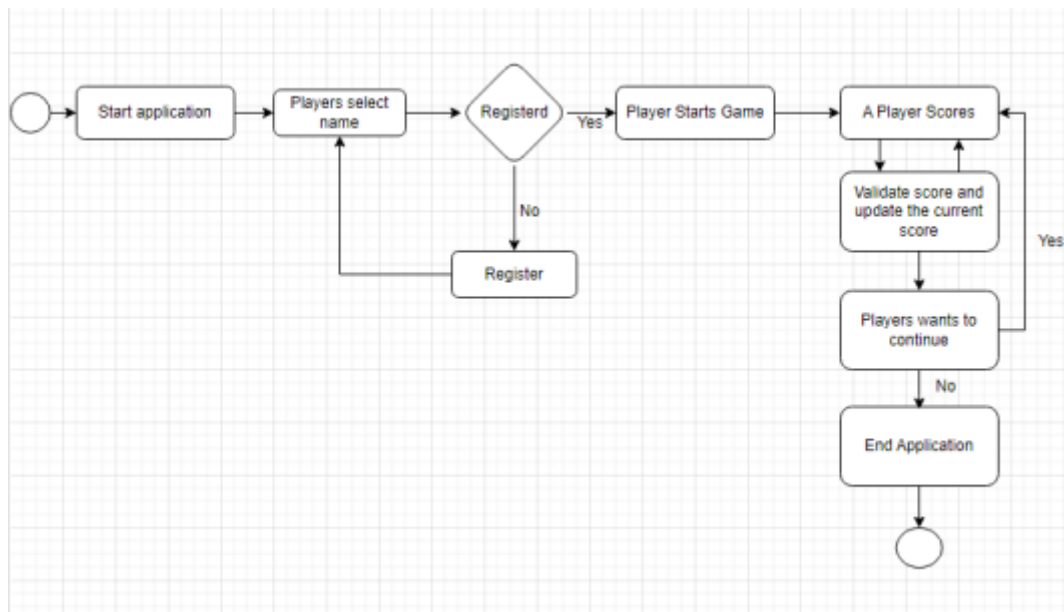
Class Diagram

Class Diagram: For visualizing classes, their attributes, methods, and relationships.



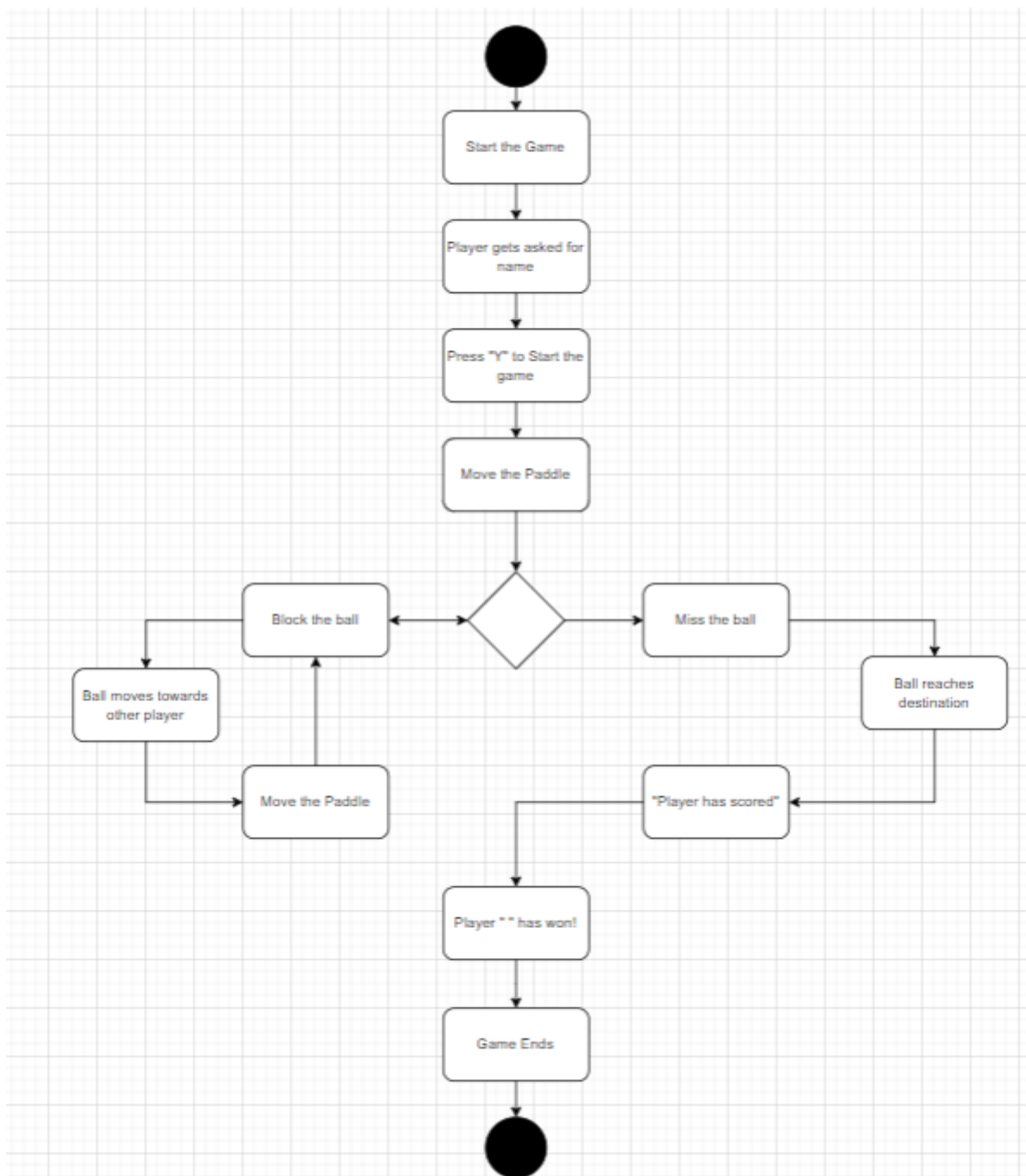
Flow Diagram

Flow Diagram: For visualizing processes



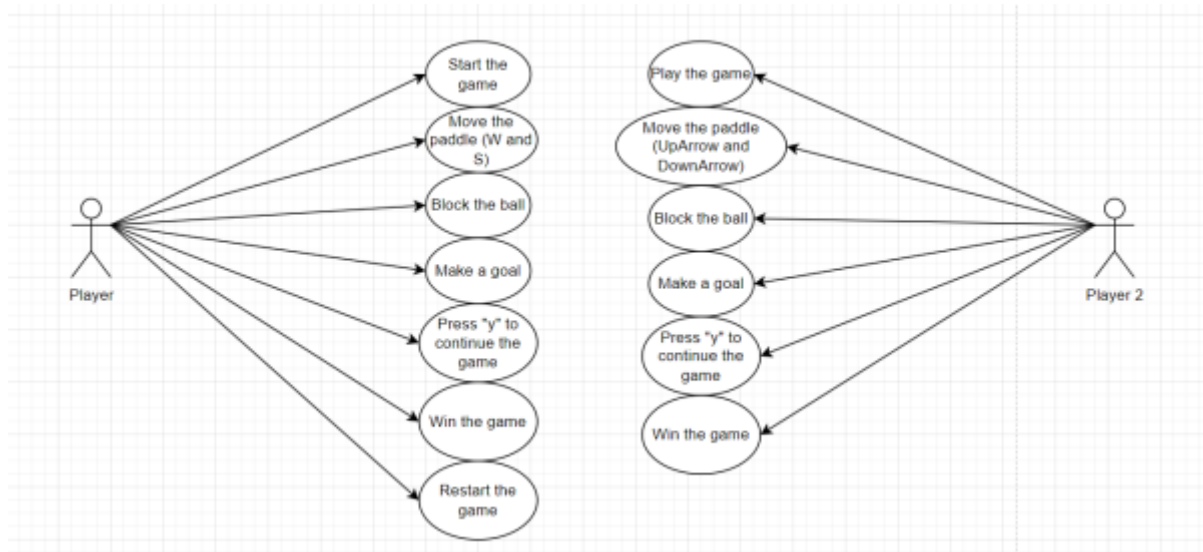
Activity diagram

Activity Diagram: Een activiteitendiagram is een van de methoden en technieken in de informatica om stapsgewijze bedrijfsprocessen in een diagram weer te geven, als onderdeel van een systeemanalyse.



Use Case Diagram

Use Case Diagram: Laat zien hoe verschillende gebruikers (actoren) interactie hebben met het systeem.



Collaboration

Pair coding

Because we had no experience programming in C#, we chose to build the application with a pair coding approach. The downside of this is that it is way more time consuming than just picking up user stories independently, however we learned a lot more this way.

Approach

At thursdays we free from school so we choose to meet up in Joshua his appartment so that we had the time and space to work on the project. We coded on one computer and because of that we could fix the errors we got together.

Scrum

For this project we had no azure DevOps environment, this however did not matter. When we were working on the project, we were together. This is why we just wrote down the user stories to finish the application gradually.

Unit Test

We have chosen to do unit testing during our production. For example, pressing non valid keys and testing the performance on different pcs.

Security

Since this is a local app security wasn't needed.