SL2A & ADP document

Studenten: Joel, Rik & Emma

Docenten: Jan zuur & Tom sievers

Datum: 14/06/2024

Inhoudsopgave

Sprint documentatie:

Wij hebben als groepje 4 (0-3) sprints gedaan om tot het eindproduct te komen. Hier hebben wij per sprint een aantal “issues” in de sprintplanning mee gemaakt.

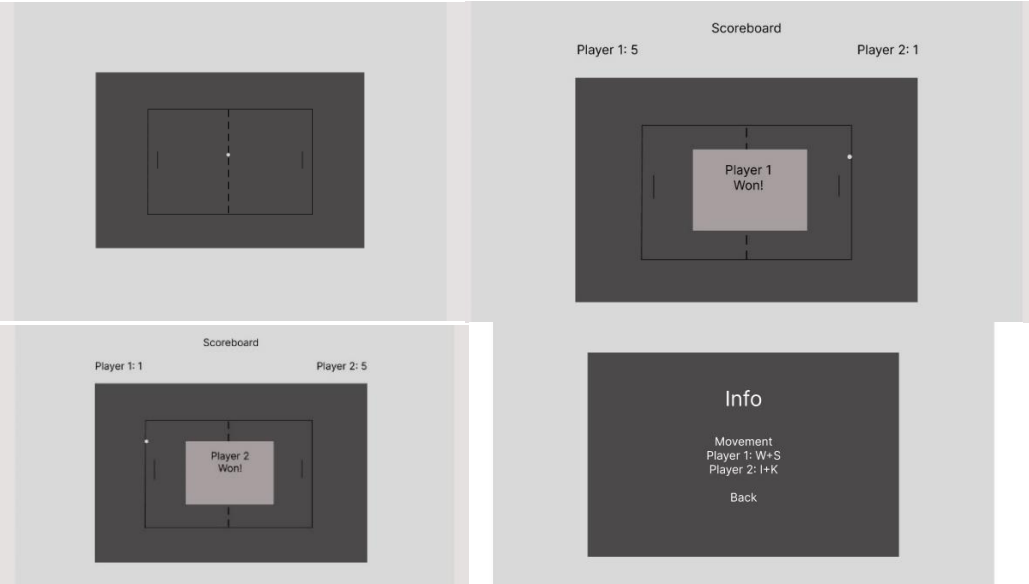
Sprint 0:

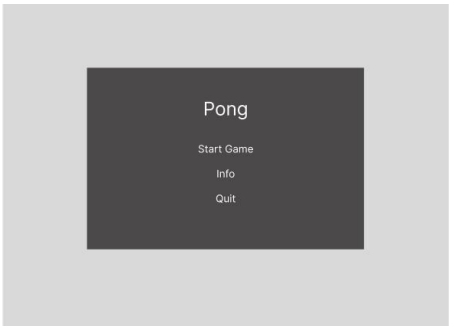
In de eerste sprint zijn wij begonnen met brainstormen. Hieruit is gekomen dat wij de game PONG willen maken, welke toetsen gebruikt zullen worden als je de game speelt.
Hiervoor was het nodig om de volgende opdrachten te maken:
User stories, Product backlog, Sprint backlog, Design, Wireframes verder uitwerken.

In deze sprint waren weinig “issues” ontstaan aangezien het enkel het bedenken van een plan was.

Sprint 1:

In deze sprint zijn de opdrachten die wij hadden gepland in de vorige sprint uitgevoerd. Zo hebben wij de volgende dingen uitgewerkt:
User stories, Product backlog, Sprint backlog, Design, Wireframes verder uitwerken.





In deze sprint was er ook nog geen spraken van veel “issues”. De enige “issue” was dat we niet een hele strakke planning hadden gemaakt voor de eerstvolgende sprint.

Sprint 2:

Bij deze sprint is de sprint reviews met de andere groepjes gemaakt.
En de rollen verdeeld binnen ons groepje, wie wat gaat oppakken.

De “issue” van deze sprint is voornamelijk dat we ons niet volledig hebben gefocust op dit project aangezien wij hiernaast nog een project hebben lopen.
Doordat de deadline van onze game PONG ook in de buurt komt is dit niet ideaal en zouden wij dat in de laatste sprint moeten oplossen.

Sprint3:

Deze sprint is belangrijk omdat het einde inzicht komt en er nog een aantal stappen ondernomen moeten worden.
In deze sprint hebben we gefocust op het daadwerkelijk coderen van het spel, en de hiervoor nodige taken onder elkaar verder te verdelen. Op deze manier zullen we het eindproduct volledig afmaken en inleveren.

“Issues” die bij deze sprint zijn voorgekomen waren dat wij niet precies wisten hoe groot de opdracht was en dus ook niet precies wisten hoeveel tijd we hiervoor kwijt zouden zijn.

Teamreflectie Document:

Onze reis door het C#-project was een leerzaam avontuur. Gelukkig verliep onze samenwerking helemaal goed, waarbij we open communiceerden en elkaar konden helpen waar nodig was.

In de eerste sprints lag de focus op het opstellen van een plan en documentatie. Hoewel we weinig problemen ondervonden, beseften we achteraf dat we wel te weinig tijd hadden besteed aan het maken van ons plan voor de volgende sprints en het daadwerkelijk beginnen met de game.

Later werden we afgeleid door andere projecten, waardoor onze focus op de game iets achterliep. Dit zorgde voor dat we wat minder werk gingen leveren aan dit project.

Ondanks deze opstakels hebben wij alsnog ons eindproduct optijd kunnen afmaken. We hebben geleerd dat een goede verdeling over de sprints belangrijk is voor succes en zullen dit meenemen naar mogelijke volgende projecten.

ADP Document

Auteurs:

- Joël Magalhães
- Rik Bakker
- Emma Koster

Recensent:

- Jan Zuur

Introductie

Dit Software Design Document (SDD) beschrijft de architectuur, het ontwerp en de implementatie van de Pong-game. Dit document is bedoeld voor de programmeurs en de docenten.

1. Functionele Beschrijving

De Pong game is ontworpen om een eenvoudige game-ervaring te bieden voor gebruikers. Het lijkt op het echte pingpongspel waarbij spelers peddels kunnen gebruiken om een bal heen en weer over de tafel te slaan.

Functionele Vereisten

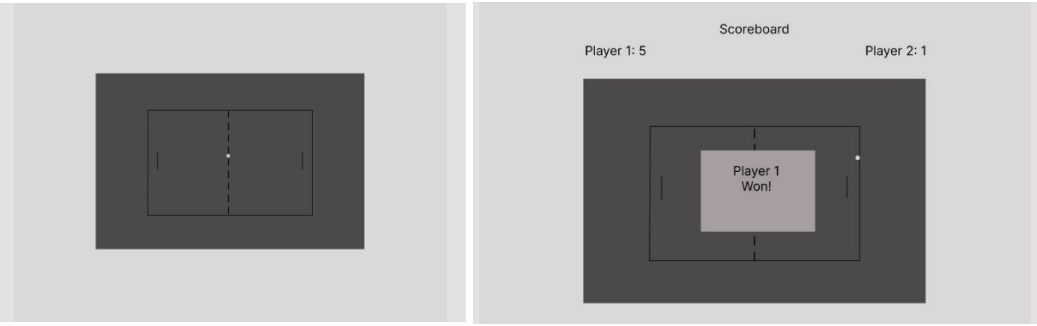
- **Spel:**
    - Spelers kunnen peddels besturen met de toetsen W+S en I+K om een bal heen en weer over het speelveld te slaan.
    - De bal zal na een aantal keer een peddel te raken versnellen om een uitdaging in het spel toe te voegen.
    - Het spel eindigt als een speler de bal 5 keer mist en de tegenstander punten scoort.
    - Bij het bereiken van 5 punten wint een speler het spel.

- **Error Handling:**
  - Als er fouten optreden tijdens het spel, zal het spel opnieuw starten zonder gegevensverlies.
- **Opstart:**
  - Bij het opstarten van het spel kiezen spelers wie welke toetsenbordknoppen zal gebruiken om de peddels te bedienen.
  - Spelers kunnen het spel starten zodra ze klaar zijn met de keuze.
- B**eperkingen:**
  - Het spel heeft twee spelers om gelijktijdig deel te nemen.
  - Elke speler kan individueel hun eigen toetsen gebruiken om de peddel te besturen.

2. Gebruikersinterface

De gebruikersinterface bestaat uit grafische elementen die het pong speelveld, peddels, bal en scorebord voorstellen(deze komt boven in het scherm te staan). De interface zal worden gemaakt om te lijken op een normale pingpongtafel om ping pong na te bootsen.

Wireframe Diagram:



3. Lean Methodologie

Om low-fidelity wireframes voor de gebruikersinterface te maken, zullen het maken volgens de opdracht.

4. Doelen en Mijlpalen

Doel: Bied een simpele multiplayer game-ervaring die lijkt op het klassieke pingpong spel.

Mijlpalen:

- Maken van grafische elementen (peddels, bal, scoreboard).
- Laten lijken op klassiek pingpong (balbeweging, peddelbediening).
- Maken van multiplayer-functionaliteit.
- Laatste test- en debugfase voor het einde.

5. Prioritisering

Het belangrijkste van het spel is dat het gebruiksvriendelijk is, en duidelijk is. Dit wordt bereikt doordat alles binnen het spel soepel verloopt en werkt zoals is aangegeven.

6. Huidige en Voorgestelde Oplossingen

Huidige Oplossing:
Klassiek pingpongspel met fysieke materialen.

Voorgestelde Oplossing:
Virtueel pingpongspel dat toegankelijkheid en gemak biedt voor spelers. De rechtvaardiging omvat gemakkelijke toegang, mogelijkheid om op afstand te spelen en potentieel voor aanvullende functies zoals aanpasbare instellingen en online multiplayer.

7. Tijdlijn

De tijdlijn zal specifieke taken, deadlines en toegewezen teamleden voor elke fase van ontwikkeling bevatten, inclusief codering, testen en implementatie.

8. Teststrategie

Wij zullen de game runnen om te kijken of alles werkt en de code hiervan nakijken. Ook als het werkt zullen we andere aspecten ook nog een keer testen om zeker ervan te zijn dat de code werkend is.

9. Risicobeheer

Mogelijke risico's binnen het project kunnen zijn:

- Tijdsdruk
- Samenwerking
- Onvoldoende kennis
- Apparatuur problemen

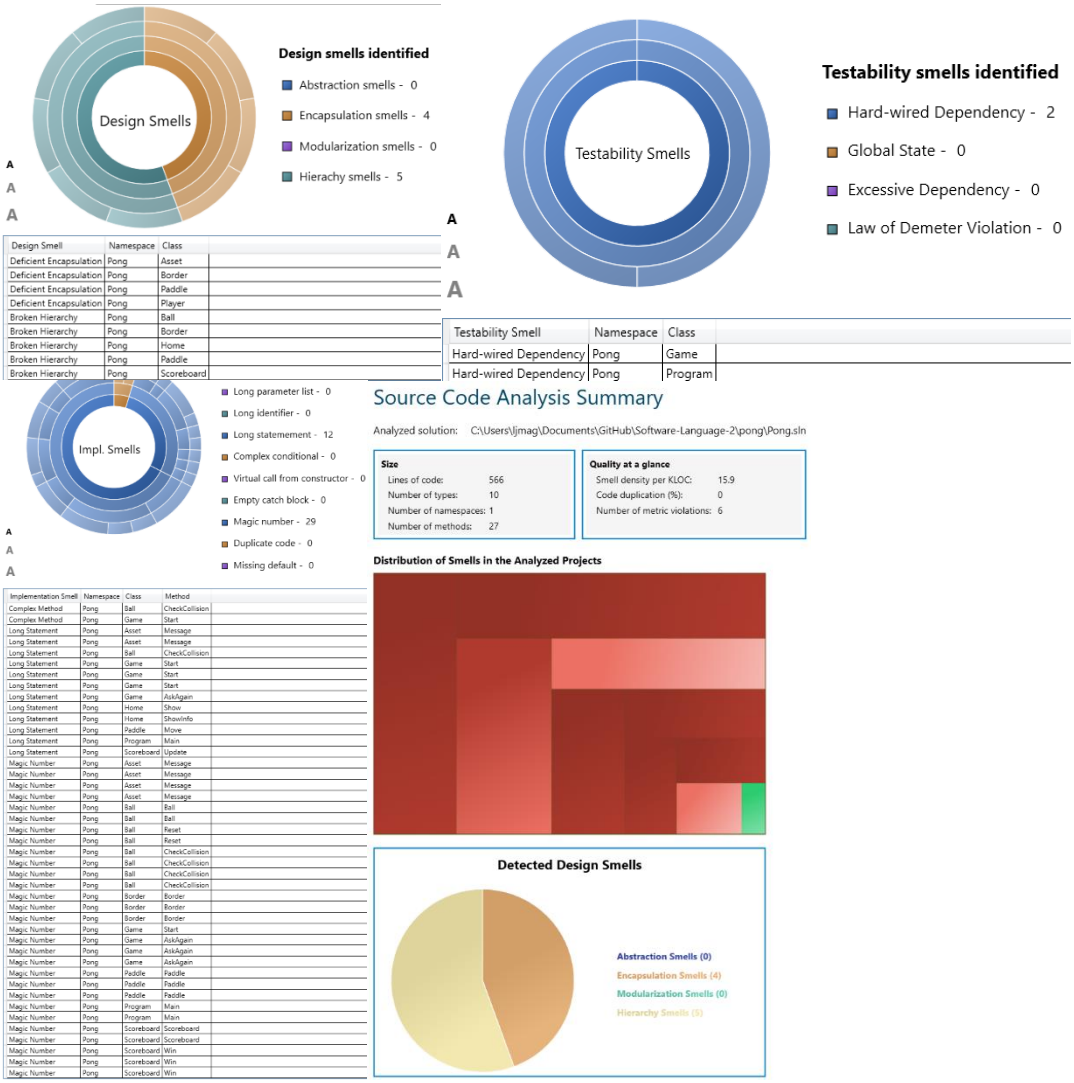
Design Smells

Bij ons komen nog de volgende design smells voor:

- Complex method: 2
- Long statement: 12
- Magic number: 29
- Deficient encapsulation: 4
- Broken hierarchy: 5
- Hard-wired dependency: 2

Dit zijn allemaal fouten die nog verholpen moeten worden om CleanCode te hebben.

Om een groot deel hiervan te verhelpen zal alles herordert moeten worden. Hier is alleen geen tijd meer voor geweest.



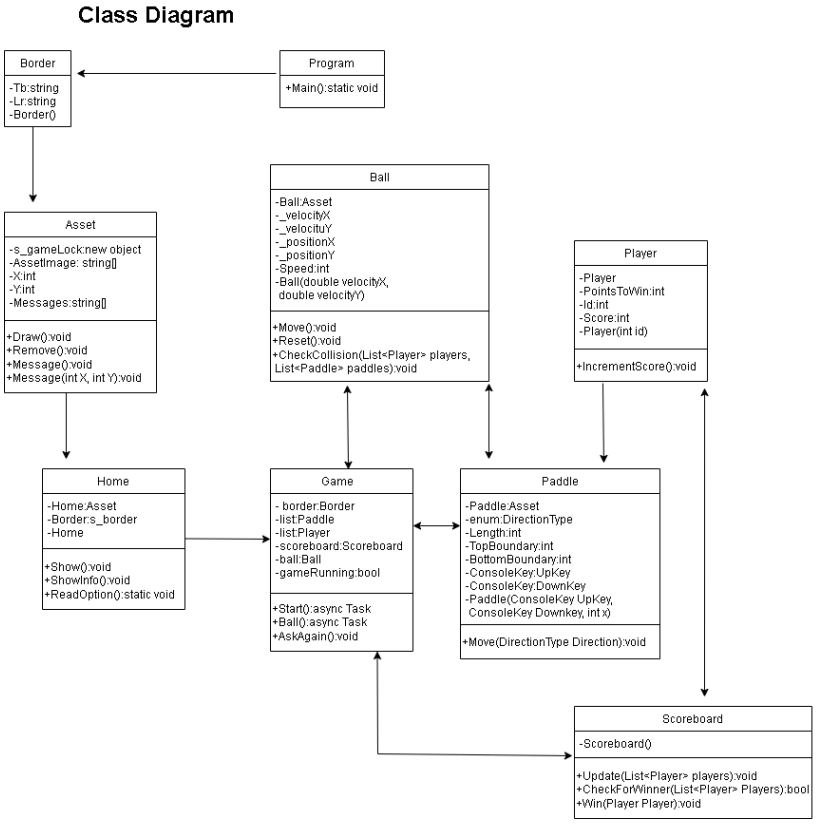
UML

Een methode genaamd UML wordt gebruikt om ingewikkelde systemen te ontleden en duidelijk te beschrijven. Met UML kunnen we de structuur en het gedrag van software begrijpen en inzichtelijk maken door middel van visuele modellen.

Klassendiagrammen en Ontwerppatronen

Klassendiagrammen onthullen hoe klassen binnen een softwareprogramma met elkaar zijn verbonden en hoe ze samenwerken. Hier zijn enkele veelgebruikte ontwerppatronen die je kunt integreren in klassendiagrammen:

- **Singleton Pattern:** Gebruik om te laten zien dat een klasse slechts één instantie kan hebben. Dit kan worden weergegeven door een statische methode te markeren die de enige instantie van de klasse retourneert.
- **Factory Method Pattern:** Toon hoe een klasse verantwoordelijk is voor het creëren van objecten. Dit kan worden weergegeven door een abstracte creatie methode in een superclass en concrete implementaties in subclasses.
- **Observer Pattern:** Laat zien hoe objecten worden geïnformeerd over veranderingen in een ander object. Dit kan worden weergegeven door een subject-klasse met een lijst van observer-objecten en een update-methode die wordt aangeroepen wanneer de staat van het subject verandert.



Use Case Diagrammen en Architectuurpatronen

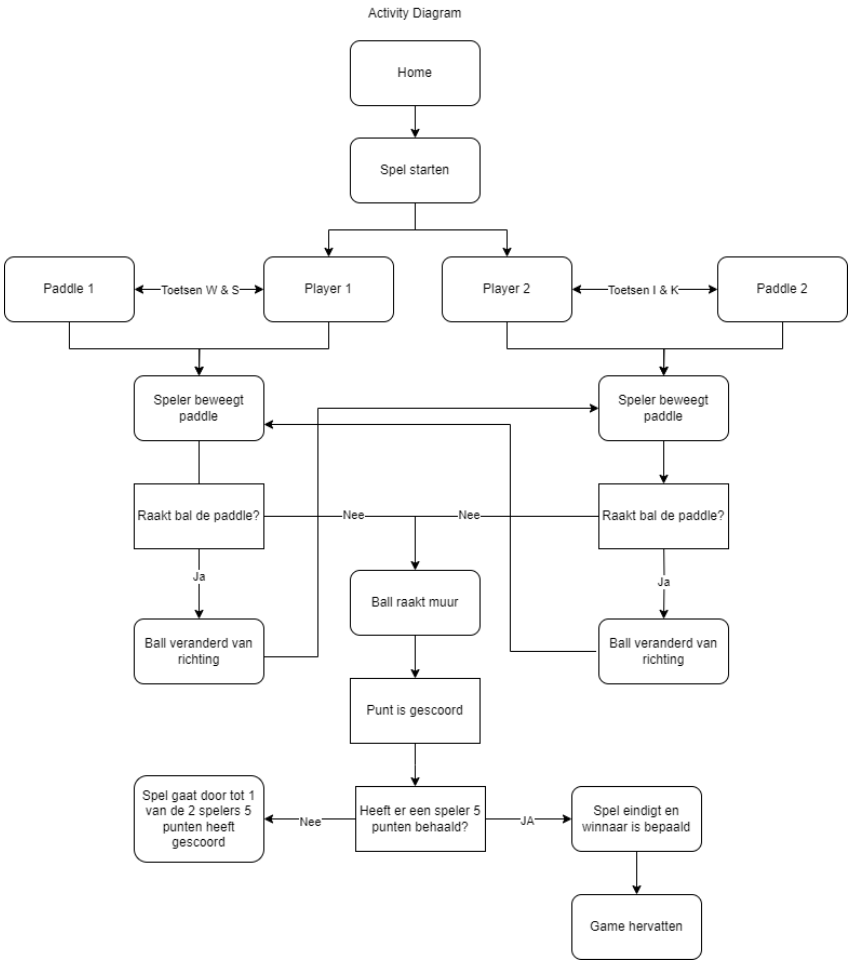
Use case diagrammen bieden een visuele representatie van hoe gebruikers met een systeem kunnen omgaan. Hier zijn enkele patronen die je kunt integreren:

- **Model-View-Controller (MVC) Pattern:** Gebruik dit patroon om de interactie tussen de gebruiker (view), de logica (controller) en de data (model) te illustreren. Gebruik cases kunnen aangeven welke functionaliteiten door welke componenten worden afgehandeld.
- **Facade Pattern:** Toon een vereenvoudigde interface voor een complex subsysteem. Use cases kunnen de interactie met de -klassen illustreren die als interfaces voor gebruikers dienen.

Activity Diagrammen en Gedrags- of Procespatronen

Activity diagrammen illustreren de stappen en taken in een proces. Hier zijn enkele patronen die je kunt gebruiken:

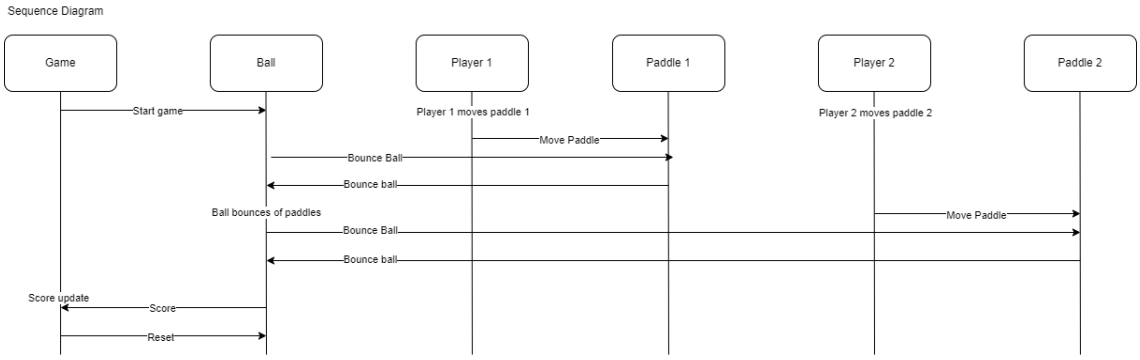
- **Template Method Pattern:** Toon een reeks stappen waarbij bepaalde stappen kunnen worden overschreven door subclasses. Activiteiten in het diagram kunnen methoden vertegenwoordigen die door subclasses worden geïmplementeerd.
- **State Pattern:** Gebruik dit patroon om verschillende toestanden in een proces te modelleren, waarbij elke toestand specifieke activiteiten heeft. Activiteiten kunnen worden gekoppeld aan de methoden van toestandsobjecten.



Sequence Diagrammen en Communicatiepatronen

Sequence diagrammen visualiseren hoe objecten binnen een systeem met elkaar communiceren. Hier zijn enkele patronen die je kunt gebruiken:

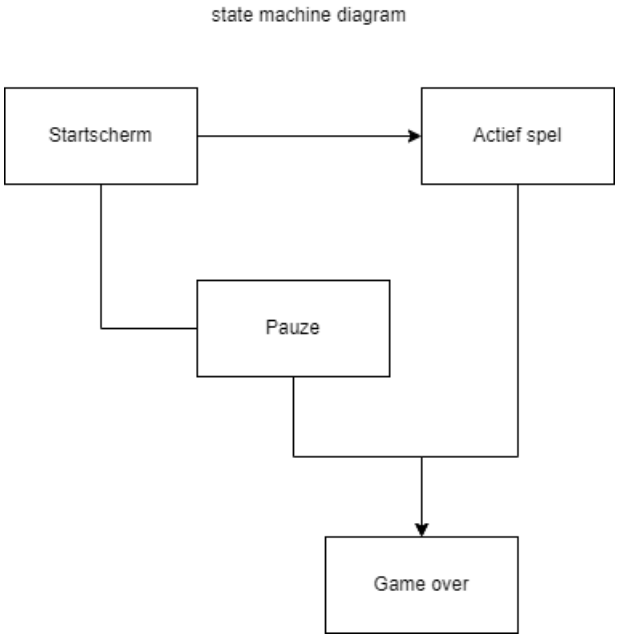
- **Chain of Responsibility Pattern:** Toon hoe een verzoek langs een keten van handlers wordt doorgegeven. Dit kan worden weergegeven door een reeks berichten tussen objecten die elk het verzoek kunnen afhandelen of doorgeven.
- **Mediator Pattern:** Gebruik dit patroon om de interactie tussen verschillende objecten te coördineren. De mediator ontvangt berichten van verschillende objecten en stuurt ze door naar andere betrokken objecten.



State Machine Diagrammen en Gedragspatronen

State machine diagrammen laten de verschillende toestanden en overgangen zien waarin objecten zich kunnen bevinden binnen een systeem. Hier zijn enkele patronen die je kunt gebruiken:

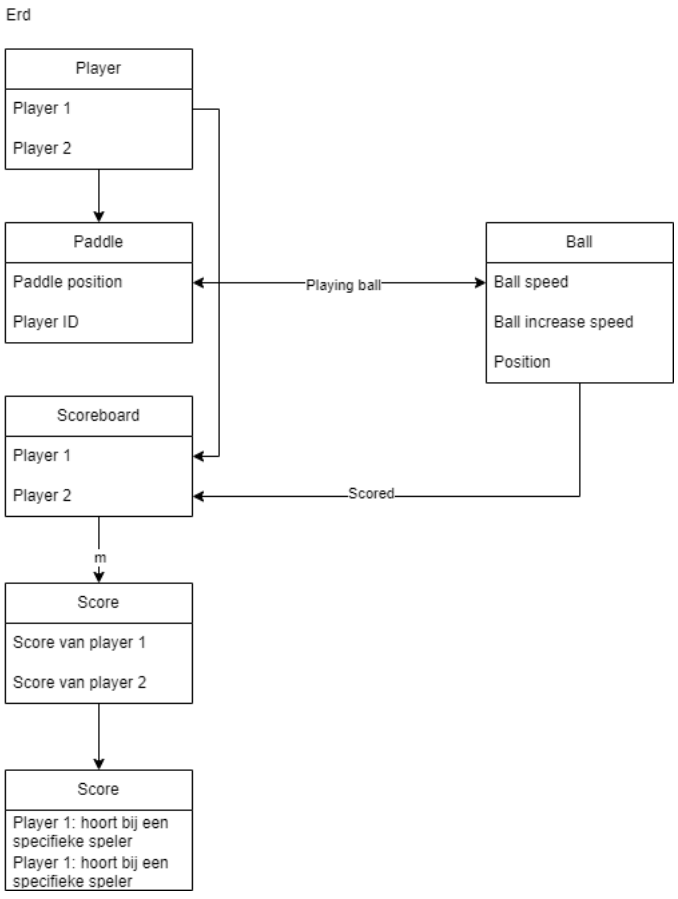
- **State Pattern:** Toon hoe een object zijn gedrag kan wijzigen wanneer zijn interne toestand verandert. Toestanden kunnen klassen zijn die specifieke gedragingen implementeren.
- **Strategy Pattern:** Gebruik dit patroon om verschillende algoritmen te modelleren die kunnen worden geselecteerd op basis van de toestand van het object. Elke strategie kan een specifieke toestand in het diagram vertegenwoordigen.



UML is een krachtig hulpmiddel dat wordt gebruikt voor het verbeteren van het ontwerp en de ontwikkeling van software. Het stelt ons in staat om complexe systemen helder en gestructureerd te begrijpen en te communiceren. Door middel van verschillende diagrammen kan UML de relaties tussen objecten en klassen visualiseren, wat essentieel is voor een effectieve ontwikkeling en communicatie binnen teams, evenals tussen klanten en ontwikkelaars.

UML wordt gebruikt omdat het:

1. **Visuele Representatie:** Complexe systemen duidelijk visualiseert.
2. **Standaardisatie:** Consistente en duidelijke documentatie biedt.
3. **Analyse en Ontwerp:** Helpt bij het analyseren van vereisten en ontwerpen van oplossingen.
4. **Documentatie:** Zorgt voor uitgebreide en nuttige systeemdocumentatie.
5. **Communicatie:** Vergemakkelijkt de communicatie tussen teamleden en stakeholders.
6. **Flexibiliteit:** Breed toepasbaar is op verschillende soorten systemen.
7. **Tool Ondersteuning:** Ondersteund wordt door diverse softwaretools voor efficiënt modelleren.



SOLID-Principe & Patterns

Om de SOLID-Principes rond te maken missen wij de Interface Segragtion Principle(ISP). Deze konden wij zelf niet terugvinden.

Scoreboard.cs

Solid:

Single Responsibility Principle (SRP):

De klasse scoreboard heeft de verantwoordelijkheid om score van speler bij te houden en te updaten, en om de winnaar te controleren en vermelden als het spel klaar is.

Patterns:

- Rik Bakker
- Joël Magalhães

Template Method Pattern:

Creational Patterns

De updatevan heckForWinner en de win methoden definiëren een vaste structuur voor het updaten van de scores en het controleren van een winnaar en het tonen van melding wie er heeft gewonnen. Het volgt de algemene structuur van het template method pattern waarbij de methoden een vaste volgorde van stappen uitvoert.

Observer Pattern:

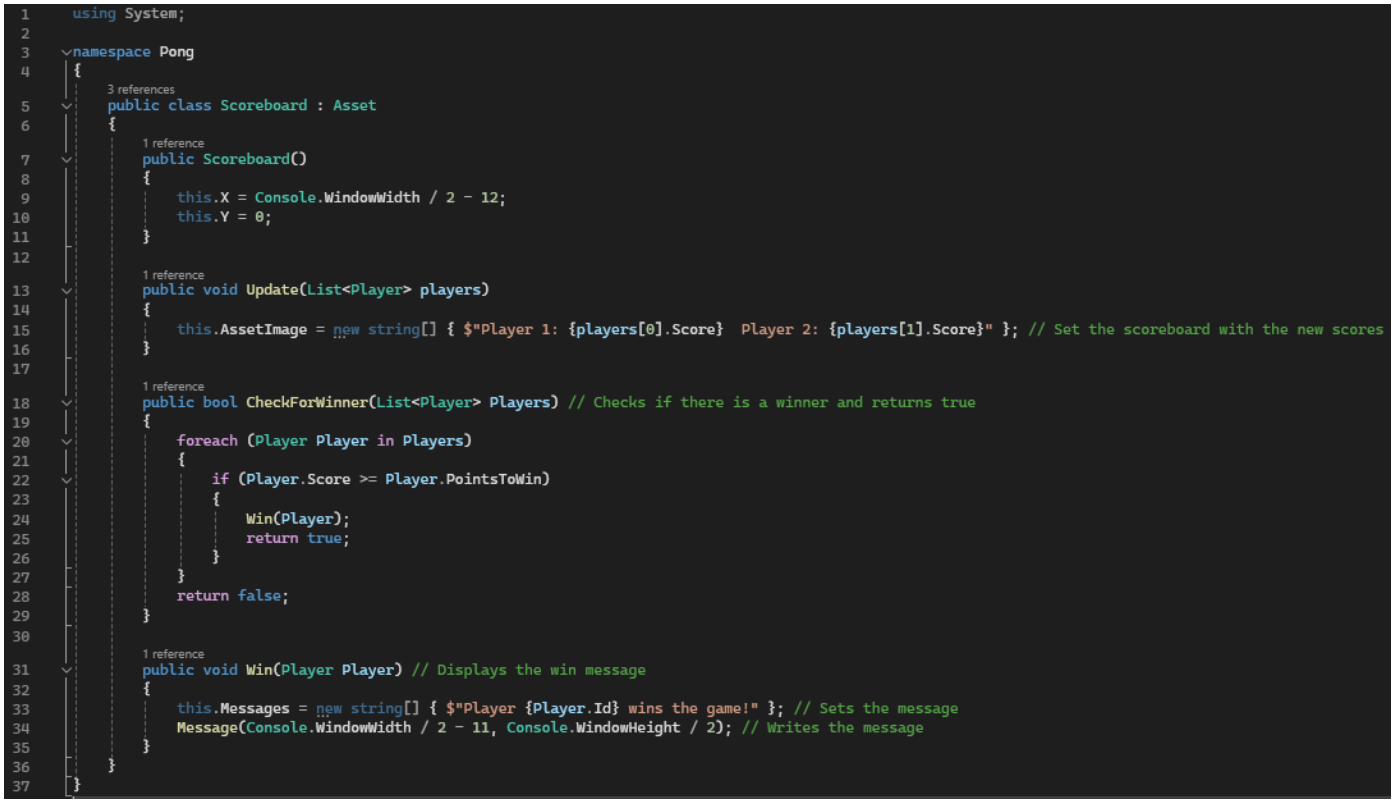
Behavioral Patterns

Er is een implicatie van het observer pattern in hoe de Scoreboard klasse omgaat met de player objecten. De scoreboard klasse kijkt naar de scores van de spelers om te bepalen wanneer er een winnaar is en om de scores bij te werken. Dit is niet een volledig geïmplementeerd observer pattern omdat er geen mechanismen zijn voor het registreren en melden van alle veranderingen.

Composite Pattern:

Structural Patterns

Dit patroon maakt het mogelijk om individuele objecten en samenstellingen van objecten op dezelfde manier te behandelen.



```
using System;

namespace Pong
{
    3 references
    public class Scoreboard : Asset
    {
        1 reference
        public Scoreboard()
        {
            this.X = Console.WindowWidth / 2 - 12;
            this.Y = 0;
        }

        1 reference
        public void Update(List<Player> players)
        {
            this.AssetImage = new string[] { $"Player 1: {players[0].Score}  Player 2: {players[1].Score}" }; // Set the scoreboard with the new scores
        }

        1 reference
        public bool CheckForWinner(List<Player> Players) // Checks if there is a winner and returns true
        {
            foreach (Player Player in Players)
            {
                if (Player.Score >= Player.PointsToWin)
                {
                    Win(Player);
                    return true;
                }
            }
            return false;
        }

        1 reference
        public void Win(Player Player) // Displays the win message
        {
            this.Messages = new string[] { $"Player {Player.Id} wins the game!" }; // Sets the message
            Message(Console.WindowWidth / 2 - 11, Console.WindowHeight / 2); // Writes the message
        }
    }
}
```

Player.cs

Solid:

Single Responsibility Principle(SRP) & Open/Closed Principle(OCP):

Het beheren van spelersinformatie zoals ID en score. (SRP)
Nieuwe functionaliteiten toevoegen. (OCP)

Patterns:

- Rik Bakker
- Joël Magalhães

Encapsulation pattern:

structural patterns

De score eigenschap maakt gebruik van een private setter waardoor alleen methods binnen de player klasse de score kunnen veranderen. Dit zorgt ervoor dat de score alleen kan worden veranderd via de IncrementScore methode wat encapsulatie bevordert.

Constant Pattern:

creational patterns

De pointstowin variabele is gemarkeerd als readonly wat betekent dat deze waarde alleen kan worden ingesteld tijdens de inzet van het object en daarna niet meer kan worden veranderd. Dit patroon zorgt ervoor dat constante waarden binnen de klasse blijven en niet per ongeluk worden aangepast.

```
1   using System;
2   using System.Collections.Generic;
3   using System.Linq;
4   using System.Text;
5   using System.Threading.Tasks;
6
7   namespace Pong
8   {
        10 references
9       public class Player
10      {
11          public readonly int PointsToWin = 5;
12          public int Id;
            4 references
13          public int Score {  get; private set; }
14
            2 references
15          public Player(int id)
16          {
17              this.Id = id;
18          }
            2 references
19          public void IncrementScore()
20          {
21              Score++;
22          }
23      }
24  }
```

Ball.cs

Solid:

Single Responsibility Principle(SRP), Liskov Substitution Principle(LSP) & Open/Closed Principle(OCP):

Beheren van de beweging en positie van de bal in het spel.(SRP)
Ondersteuning van de structuur van de bal.(LSP)
Mogelijkheden om de klasse “Ball” uit te breiden.(OCP)

Patterns:

- Rik Bakker
- Emma Koster
- Joël Magalhães

Encapsulation pattern:

structural patterns
Veel velden in deze code zijn allemaal privé, wat de implementatie details verbergt en toegang tot deze velden beperkt tot de methoden van de ball. Dit beschermt de binnenste staat van het object tegen niet gewilde veranderingen vanuit buiten.

State Pattern:

Behavioural Patterns

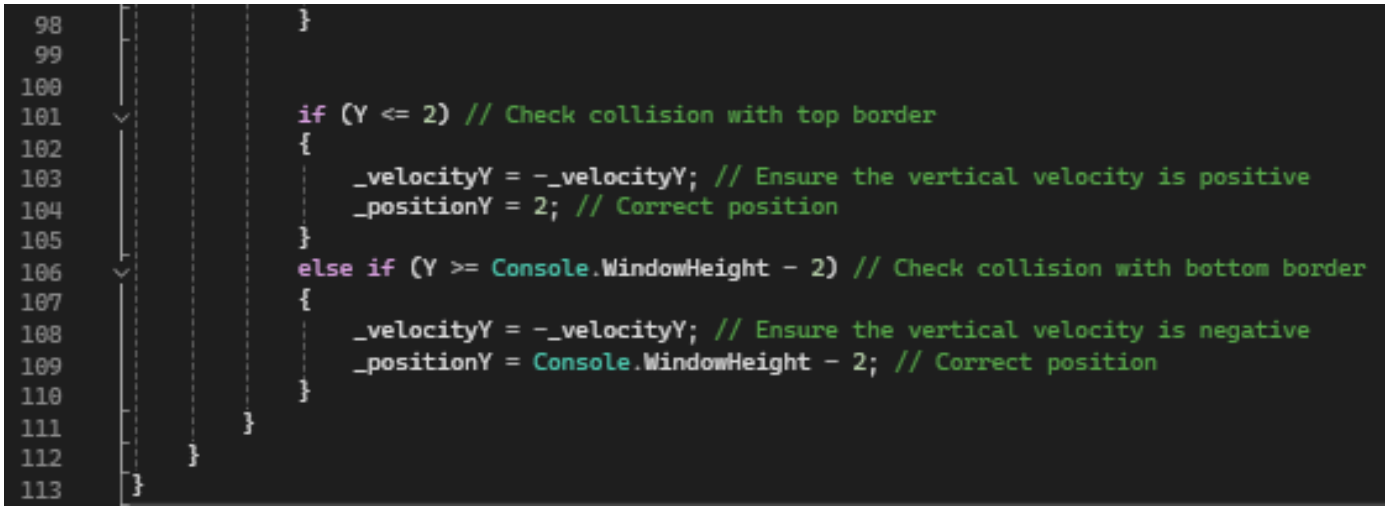
De checkcollision methode onderhoud de stand van de bal op basis van de botsingen met peddels of de muren. Afhankelijk van de botsing verandert de bal van richting en soms van snelheid na een aantal keer. Dit impliceert dat de bal verschillende toestanden kan hebben, zoals bewegen naar links of rechts, of omhoog en omlaag en van snelheid.

Strategy Pattern:

Behavioural Patterns

De manier waarop de snelheid van de bal wordt aangepast bij botsingen met de peddels zou een strategy pattern kunnen zijn. Het wijzigen van de snelheid kan worden veranderd zonder de rest van de ball aan te passen.

```
using System;
using System.Collections.Generic;

namespace Pong
{
    3 references
    public class Ball : Asset
    {
        private double _velocityX;
        private double _velocityY;
        private double _positionX;
        private double _positionY;
        4 references
        public int Speed { get; set; } = 150; // Speed of the ball

        // Constructor with correct parameter types
        1 reference
        public Ball(double velocityX, double velocityY)
        {
            // Sets the ball's start location
            _positionX = Console.WindowWidth / 2;
            _positionY = Console.WindowHeight / 2;

            _velocityX = velocityX;
            _velocityY = velocityY;

            this.AssetImage = new string[] { "O" };
        }

        1 reference
        public void Move()
        {
            Remove(); // Remove the ball from the old location

            // Set the new coordinates based on the velocity of the ball
            _positionX += _velocityX;
            _positionY += _velocityY;

            // Update integer positions for rendering
            X = (int)_positionX;
            Y = (int)_positionY;

            // Draw(); // Draw the ball in the new position
        }

        1 reference
        public void Reset()
        {
            Remove(); // Remove the ball from the old location

            // Change the ball coordinates to default
            // Default is the middle of the screen
            _positionX = Console.WindowWidth / 2;
            _positionY = Console.WindowHeight / 2;

            // Update integer positions for rendering
            X = (int)_positionX;
            Y = (int)_positionY;

            Draw(); // Draw the ball in the default position
        }

        1 reference
        public void CheckCollision(List<Player> players, List<Paddle> paddles)
        {
            bool isPaddleCollision = false; // Boolean for paddle collisions
            foreach (Paddle paddle in paddles) // Checks if the ball collides with a paddle and if so sets the isPaddleCollision to true
            {
                if (X == paddle.X && Y >= paddle.Y && Y <= paddle.Y + paddle.Length)
                {
                    isPaddleCollision = true;
                    break;
                }
            }

            if (isPaddleCollision) // If there is a paddle collision, change the way the ball is going
            {
                _velocityX = -_velocityX;
                if (Speed > 50)
                {
                    Speed -= 10; // Decrease the speed by 10 if it is higher than 50
                }
            }
            else // If there is no paddle collision, check for border x collisions
            {
                bool isLeftBorderCollision = (X <= 1); // Check collision with left border
                bool isRightBorderCollision = (X >= Console.WindowWidth - 2); // Check collision with right border

                if (isLeftBorderCollision || isRightBorderCollision)
                {
                    Speed = 150; // Reset the speed
                    _velocityX = -_velocityX; // Reverse the horizontal velocity
                    Reset();

                    if (isLeftBorderCollision)
                    {
                        players[1].IncrementScore(); // Increment the score of player 2
                    }
                    else if (isRightBorderCollision)
                    {
                        players[0].IncrementScore(); // Increment the score of player 1
                    }
                }
            }
```



```
98              }
99
100
101             if (Y <= 2) // Check collision with top border
102             {
103                 _velocityY = -_velocityY; // Ensure the vertical velocity is positive
104                 _positionY = 2; // Correct position
105             }
106             else if (Y >= Console.WindowHeight - 2) // Check collision with bottom border
107             {
108                 _velocityY = -_velocityY; // Ensure the vertical velocity is negative
109                 _positionY = Console.WindowHeight - 2; // Correct position
110             }
111         }
112     }
113 }
```

Paddle.cs

Solid:

Single Responsibility Principle(SRP), Liskov Substitution Principle(LSP) & Open/Closed Principle(OCP):

Alleen verantwoordelijk voor het bewegen van de paddles.(SRP)
Ondersteuning van Asset.(LSP)
Klasse “Paddle” heeft mogelijkheden om code te veranderen.(OCP)

Patterns:

- Rik Bakker
- Joël Magalhães

Encapsulation pattern:

structural patterns

De eigenschappen length, topboundary, en bottomboundary hebben privé setters wat betekent dat ze alleen binnen de paddle klasse kunnen worden veranderd. Dit beschermt de srtand binnen de klasse tegen niet gewilde veranderingen vanuit buiten.

Enum Pattern:

creational patterns

De directiontype enum wordt gebruikt om de richting van de beweging van de peddel aan te geven. Dit patroon helpt om de leesbaarheid en het onderhouden van de code te verbeteren door handige namen voor de richtingen te gebruiken in plaats van gekke getallen.

Factory Method Pattern:

creational patterns

De constructor van de paddle klasse kan als een eenvoudige versie van het factory method pattern worden gezien. De constructor ziet de peddel met specifieke eigenschappen zoals de positie en de bedieningsknoppen. Hoewel het niet precies een Factory Method is, volgt het de basis van het creëren van objecten met specifieke configuraties.

```
using System;
using System.Threading.Tasks;

namespace Pong
{
    11 references
    public class Paddle : Asset
    {
        5 references
        public enum DirectionType
        {
            up,
            down
        };
        // Define properties for the paddle
        6 references
        public int Length { get; private set; }
        3 references
        public int TopBoundary { get; private set; }
        3 references
        public int BottomBoundary { get; private set; }

        // Input keys for controlling the paddle
        public ConsoleKey UpKey;
        public ConsoleKey DownKey;

        2 references
        public Paddle(ConsoleKey UpKey, ConsoleKey DownKey, int x)
        {
            this.X = x; // The x position the paddle will be in
            this.Y = 12; // Set the initial Y position

            Length = 7; // Length of the paddle
            TopBoundary = 2; // Top boundary
            BottomBoundary = Console.WindowHeight - 1; // Default bottom boundary

            this.UpKey = UpKey; // The key which the player uses to move the paddle up
            this.DownKey = DownKey; // The key which the player uses to move the paddle down

            // Array with the paddle
            this.AssetImage = new string[Length];
            for (int i = 0; i < Length; i++) { this.AssetImage[i] = "|"; } // Makes the array with the paddle character the length of the paddle
        }

        2 references
        public void Move(DirectionType Direction)
        {
            this.Remove(); // Removes the paddle
            switch (Direction)
            {
                case DirectionType.up:
                    if (Y > TopBoundary) { Y--; } // Moves the paddle up if it is lower than the top
                    break;
                case DirectionType.down:
                    if (Y + Length /* Y coordinate + the length of the paddle */ < BottomBoundary) { Y++; } // Moves the paddle down if it is higher than the bottom of the screen
                    break;
                default:
                    throw new ArgumentException("Invalid direction. Expected 'up' or 'down'.", nameof(Direction));
            }

            this.Draw(); // Draws the paddle in the new location
        }
    }
}
```

Game.cs

Solid:

Single Responsibility Principle(SRP), Open/Closed Principle(OCP) & Dependency Inversion Principles(DIP):

Beheren van spelloop en coördineren van verschillende spelobjecten.(SRP)
Klasse "Game" ontworpen met mogelijkheden voor uitbreiding.(OCP)
Klasse "Game" maakt gebruik van andere klasses.(DIP)

Patterns:

- Rik Bakker
- Emma Koster

Observer Pattern:

Behavioural Patterns

De methode start checkt constant de toestand van de Console.KeyAvailable om te reageren op gebruikersinvoer. Hoewel dit een simpele vorm van observatie is, lijkt het op het observer pattern waar de game klasse fungeert als een observer voor gebruikersinvoer.

Strategy Pattern:

Behavioural Patterns

De ball klasse en de paddle klasse gebruiken een vorm van de strategy pattern voor hun bewegingen. De move methode in de paddle klasse gebruikt de directiontype enum om verschillende strategieën voor beweging te implementeren.

Task-based Asynchronous Pattern:

concurrency patterns

De start en ball methoden zijn asynchroon en maken gebruik van async en await om het spel soepel te laten verlopen zonder de UI te blokkeren.

Factory Method Pattern:

Creational Patterns

De constructor van de Game klasse creëert en initialiseert alle noodzakelijke spelobjecten zoals border, paddle, player, scoreboard, en de ball.

```
using System;
using System.Threading.Tasks;

namespace Pong
{
    5 references
    public class Game
    {
        // Sets all the game assets into variables
        private Border _border = new Border();
        private List<Paddle> _paddles = new List<Paddle>() // List with the 2 paddles
        {
            new Paddle( ConsoleKey.W, ConsoleKey.S, 1 ),
            new Paddle(ConsoleKey.I, ConsoleKey.K, Console.WindowWidth - 2)
        };
        private List<Player> _players = new List<Player>()
        {
            new Player(1),
            new Player(2),
        };
        private Scoreboard _scoreboard = new Scoreboard();
        private Ball _ball = new Ball(1, 1); // Initial position and velocity of the ball
        private bool _gameRunning;

        2 references
        public Game()
        {
            _gameRunning = true; // Starts the game
        }

        2 references
        public async Task Start()
        {
            Console.Clear(); // Clears the console

            // Draws the assets onto the screen
            _border.Draw(); // Draw the border
            _ball.Draw();
            foreach (Paddle Paddle in _paddles) { Paddle.Draw(); }

            // Start the game loop
            while (_gameRunning)
            {
                if (Console.KeyAvailable) // If there is a key pressed, check if it is to move a paddle and then move a paddle. This is being done in this function so you dont have to press the key twice
                {
                    ConsoleKey Key = Console.ReadKey(true).Key; // Puts the pressed key into the variable key

                    foreach (Paddle Paddle in _paddles)
                    {
                        if (Key == Paddle.UpKey && Paddle.Y > Paddle.TopBoundary) // Checks if it is the paddle upKey and if paddle is not on the top of the screen
                        {
                            Paddle.Move(Paddle.DirectionType.up); // Move paddle up
                        }
                        if (Key == Paddle.DownKey && Paddle.Y + Paddle.Length < Paddle.BottomBoundary) // Checks if it is the paddle downKey and if paddle is not on the bottom of the screen
                        {
                            Paddle.Move(Paddle.DirectionType.down); // Move paddle down
                        }
                    }
                }
                    }

                    // Draw the scoreboard above the game area and centered
                    _scoreboard.Update(_players);
                    _scoreboard.Draw();

                    // Check for game end conditions
                    if (_scoreboard.CheckForWinner(_players)) { _gameRunning = false; } // If someone won, stop the games

                    await Task.Delay(2); // Task delay to control the speed
                }
            }
            2 references
            public async Task Ball() // This task makes the ball run seperate from the rest of the game so it wont interupt the moving of the paddles
            {
                while (_gameRunning)
                {
                    _ball.Move(); // Update and draw the
                    _ball.CheckCollision(_players, _paddles); // Check for collisions between the ball and the border or paddles
                    _ball.Draw(); // Draw the ball in the new position
                    await Task.Delay(_ball.Speed); // Task delay to control the speed
                }
            }

            1 reference
            public void AskAgain()
            {
                while (true)
                {
                    Console.SetCursorPosition(Console.WindowWidth / 2 - 18, Console.WindowHeight / 2 + 1); // Writes the message on the specified line
                    Console.Write("Do you wanna play again? Press enter"); // Sets the message

                    ConsoleKey Key = Console.ReadKey(true).Key; // Check the pressed key
                    if (Key == ConsoleKey.Enter)
                    {
                        // If key = enter, start the game again
                        Game game = new Game();
                        Task.WaitAny(game.Start(), game.Ball());
                    }
                    else
                    {
                        // If key is not enter, exit
                        Environment.Exit(0);
                    }
                }
            }
        }
    }
```

Asset.cs

Solid:

Single Responsibility Principle(SRP) & Liskov Substitution Principle(LSP):

> Beheren en weergeven van spelobjecten.(SRP)
> Door de klasse "Asset" kunnen andere klasses ter werk zonder onverwachts gedrag te vertonen.(LSP)

Patterns:

- Rik Bakker
- Emma Koster

Template Method Pattern:

> behavioral patterns
>
> De methoden `draw()` en `remove()` bevatten een algoritme met stappen die in een bepaalde volgorde moeten worden uitgevoerd (`Draw` om iets op het scherm te tekenen en `Remove` om het te verwijderen).

Command Pattern:

> Behavioural Patterns
>
> De methoden `draw()` en `remove()` kunnen worden beschouwd als commando's die een actie uitvoeren (tekenen of verwijderen van een asset).

State Pattern:

> Behavioural Patterns
>
> De asset klasse heeft methoden om een asset te tekenen (`draw`) en te verwijderen (`remove`), die kunnen worden gezien als overgangen tussen verschillende standen van de asset.

Exception Handling Pattern:

behavioural patterns

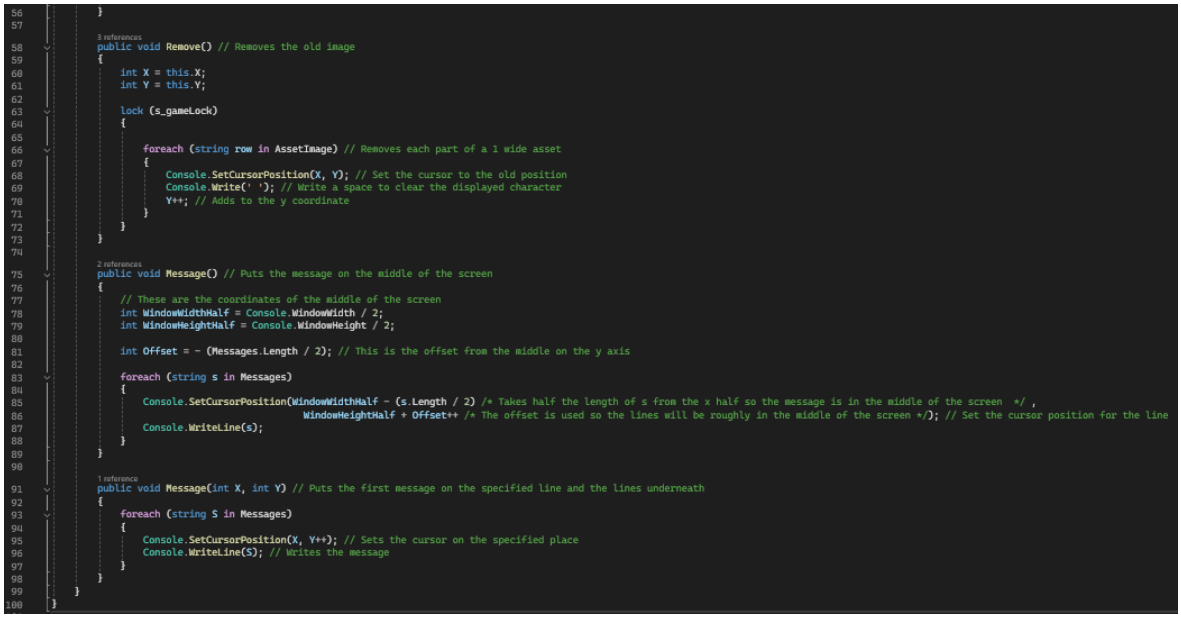
De draw() methode omvat een try-catch blok om ArgumentOutOfRangeException te hanteren die komt als er iets buiten het scherm is. Dit patroon zorgt ervoor dat onverwachte fouten worden afgehandeld en er feedback wordt gegeven aan de spelers.

Observer Pattern:

Behavioural Patterns

De asset klasse kan gezien worden als observable in combinatie met de draw() methode. Andere klassen kunnen zich aansluiten op wijzigingen in de assetimage array om te reageren op updates en hun eigen weergave bij te werken.

```
using System;
using System.Collections.Generic;
using System.Data;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Pong
{
    5 references
    public class Asset
    {
        private readonly static object s_gameLock = new object(); // Makes a lock so you cant draw two things at once
        public string[] AssetImage = {""}; // The array for the to be drawn image
        public int X; // The X coordinate on the screen
        public int Y; // The Y coordinate on the screen

        public string[] Messages = { "" }; // The array for the messages

        0 references
        public void Draw(String Input)
        {
            Console.WriteLine(Input);
        }

        9 references
        public void Draw()
        {
            int X = this.X;
            int Y = this.Y;

            lock (s_gameLock)
            {
                foreach (string Row in AssetImage) // Foreach loop to draw what is in the assetImage array
                {
                    try
                    {
                        // Check if coordinates are within range of the screen
                        if (Y >= 0 && Y < Console.WindowHeight - 1)
                        {
                            Console.SetCursorPosition(X, Y); // Set cursor position
                            Console.WriteLine(Row); // Writes the row onto the screen
                            Y++;
                        }
                        else
                        {
                            // On the last row, dont change the cursor position and write the last row
                            Console.Write(Row);
                        }
                    }
                    catch (ArgumentOutOfRangeException ex)
                    {
                        // If there is an error with writing to the screen, display it
                        Console.WriteLine($"Error: {ex.Message}");
                        throw new Exception(ex.Message);
                    }
                }
            }
        }
```



```
        }

        3 references
        public void Remove() // Removes the old image
        {
            int X = this.X;
            int Y = this.Y;

            lock (s_gameLock)
            {

                foreach (string row in AssetImage) // Removes each part of a 1 wide asset
                {
                    Console.SetCursorPosition(X, Y); // Set the cursor to the old position
                    Console.Write(' '); // Write a space to clear the displayed character
                    Y++; // Adds to the y coordinate
                }
            }
        }

        2 references
        public void Message() // Puts the message on the middle of the screen
        {
            // These are the coordinates of the middle of the screen
            int WindowWidthHalf = Console.WindowWidth / 2;
            int WindowHeightHalf = Console.WindowHeight / 2;

            int Offset = - (Messages.Length / 2); // This is the offset from the middle on the y axis

            foreach (string s in Messages)
            {
                Console.SetCursorPosition(WindowWidthHalf - (s.Length / 2) /* Takes half the length of s from the x half so the message is in the middle of the screen  */ ,
                                          WindowHeightHalf + Offset++ /* The offset is used so the lines will be roughly in the middle of the screen */); // Set the cursor position for the line
                Console.WriteLine(s);
            }
        }

        1 reference
        public void Message(int X, int Y) // Puts the first message on the specified line and the lines underneath
        {
            foreach (string S in Messages)
            {
                Console.SetCursorPosition(X, Y++); // Sets the cursor on the specified place
                Console.WriteLine(S); // Writes the message
            }
        }
    }
}
```

Home.cs

Solid:

Single Responsibility Principle(SRP) & Liskov Substitution Principle(LSP):

> Klasse "Home" heeft als primaire verantwoordelijkheid de start en informatiescherm te weergeven.(SRP)
> Klasse "Home" maakt gebruik van de "Asset" klasse.(LSP)

Patterns:

- Rik Bakker
- Joël Magalhães

Singleton Pattern:

> Creational Patterns
>
> De s_border variabele is als static gezet wat betekent dat er 1 instantie van border wordt gemaakt en gedeeld door alle instanties van home.

Command Pattern:

Behavioural Patterns

De readoption() methode kan worden gezien als een commando dat bij een bepaalde ingedrukte toets een bepaalde actie uitvoert. Verschillende uitvoerbare opdrachten zoals zie informatie, ga terug naar start, of stop.

Template Method Pattern:

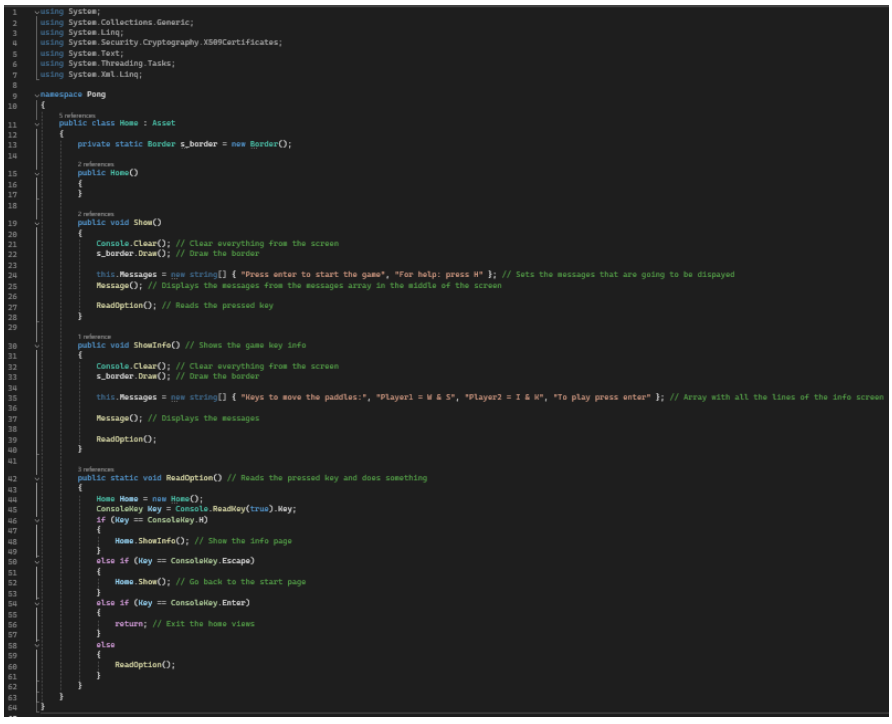
Behavioural Patterns

De show() en showinfo() methoden horen bij deze pattern Beide methoden volgen een vergelijkbaar algoritme (wis scherm, teken grens, stel berichten in, toon berichten, lees optie).

Factory Method Pattern:

Creational Patterns

ReadOption() kan als een soort factory method worden gezien die afhankelijk van de getikte toets een nieuw object (een andere weergave of actie) kan maken en terugsturen.



```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Security.Cryptography.X509Certificates;
using System.Text;
using System.Threading.Tasks;
using System.Xml.Linq;

namespace Pong
{
    5 references
    public class Home : Asset
    {
        private static Border s_border = new Border();

        2 references
        public Home()
        {
        }

        2 references
        public void Show()
        {
            Console.Clear(); // Clear everything from the screen
            s_border.Draw(); // Draw the border

            this.Messages = new string[] { "Press enter to start the game", "For help: press H" }; // Sets the messages that are going to be dispayed
            Message(); // Displays the messages from the messages array in the middle of the screen

            ReadOption(); // Reads the pressed key
        }

        1 reference
        public void ShowInfo() // Shows the game key info
        {
            Console.Clear(); // Clear everything from the screen
            s_border.Draw(); // Draw the border

            this.Messages = new string[] { "Keys to move the paddles:", "Player1 = W & S", "Player2 = I & K", "To play press enter" }; // Array with all the lines of the info screen

            Message(); // Displays the messages

            ReadOption();
        }

        2 references
        public static void ReadOption() // Reads the pressed key and does something
        {
            Home Home = new Home();
            ConsoleKey Key = Console.ReadKey(true).Key;
            if (Key == ConsoleKey.H)
            {
                Home.ShowInfo(); // Show the info page
            }
            else if (Key == ConsoleKey.Escape)
            {
                Home.Show(); // Go back to the start page
            }
            else if (Key == ConsoleKey.Enter)
            {
                return; // Exit the home views
            }
            else
            {
                ReadOption();
            }
        }
    }
}
```

Program.cs

Solid:

Single Responsibility Principle(SRP) & Liskov Substitution Principle(LSP):

> Klasse "Program" heeft als primaire verantwoordelijkheid het initialiseren en starten van het spel.(SRP)
> Klasse "Program" defineert het startpunt van de applicatie.(LSP)

Patterns:

- Rik Bakker

Singleton Pattern:

> Creational Patterns
>
> Het Singleton Pattern zie je door het gebruik van de Home en Game objecten, die maar 1 keer gebruikt wordt.

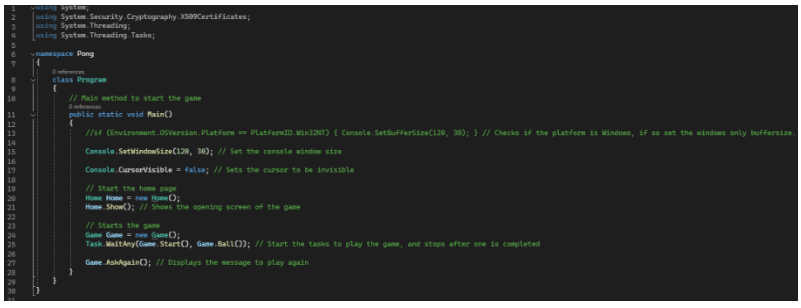
Facade Pattern:

> Structural Patterns
>
> De program klasse is door een simpele interface te zien voor het starten van het spel. Het verbergt de lastige dingen bij het opzetten en uitvoeren van het spel achter een enkele main methode, waardoor het gebruik van de Home en Game klassen gezien wordt voor de gebruiker.

Task-based Asynchronous Pattern:

concurrency patterns

Het gebruik van Task.WaitAny(Game.Start(), Game.Ball()) toont aan dat het spel gebruikmaakt van asynchrone taken om het spel en de balbeweging los van elkaar te besturen. Dit past goed bij het task based patern.



Border.cs

Solid:

Single Responsibility Principle(SRP):

Klasse "Border" concentreert zich alleen op het maken vam de visuele grens rondom het speelveld.(SRP)

Patterns:

- Rik Bakker
- Joël Magalhães

Factory Method Pattern:

Creational Patterns

De border maakt gebruik van de factory Method patroon om een instantie te maken en te checken met standaardwaarden zoals Tb en Lr, die de visuele grens van het Pong spel laten zien.

Template Method Pattern:

Creational Patterns

De border klasse past het Template Method patroon toe door de draw() methode van asset te gebruiken waarmee subclasses zoals de border de tekenlogica kunnen implementeren via de assetimage array voor het showen van visuele dingen zoals de randen.

```
using System;

namespace Pong
{
    5 references
    public class Border : Asset
    {
        public string Tb = ""; // String for the top and bottom of the border
        public string Lr = ""; // String for the left and right sides of the border

        2 references
        public Border()
        {
            // Setting the default start location of the border
            this.X = 0;
            this.Y = 1;

            for (int i = 0; i < Console.WindowWidth; i++) { Tb += "+"; } // Makes a row with '+' for the entire screen width
            for (int i = 0; i < Console.WindowWidth - 2; i++) { Lr += " "; } // Makes a string that start with '+' and end with '+'. And makes it the width of the screen
            Lr = "+" + Lr + "+"; // Set lr correctly

            this.AssetImage = new String[Console.WindowHeight]; // Initialize the array with correct size

            this.AssetImage[0] = Tb; // Assign the top border
            for (int i = 1; i < Console.WindowHeight - 2; i++) // Runs for each row of the border after the top border
            {
                this.AssetImage[i] = Lr; // Assign the side borders
            }
            this.AssetImage[Console.WindowHeight - 2] = Tb; // Assign the bottom border
        }
    }
}
```

Bijlage

- https://github.com/JoelMagalhaes/Software-Language-2

  Dit project is gemaakt met behulp van:

  - Microsoft Visual Studio
  - https://www.w3schools.com
  - https://stackoverflow.com
  - https://chatgpt.com
  - Hulp van andere studenten en docenten