Applied Design Patterns

Gemaakt door Team Basker

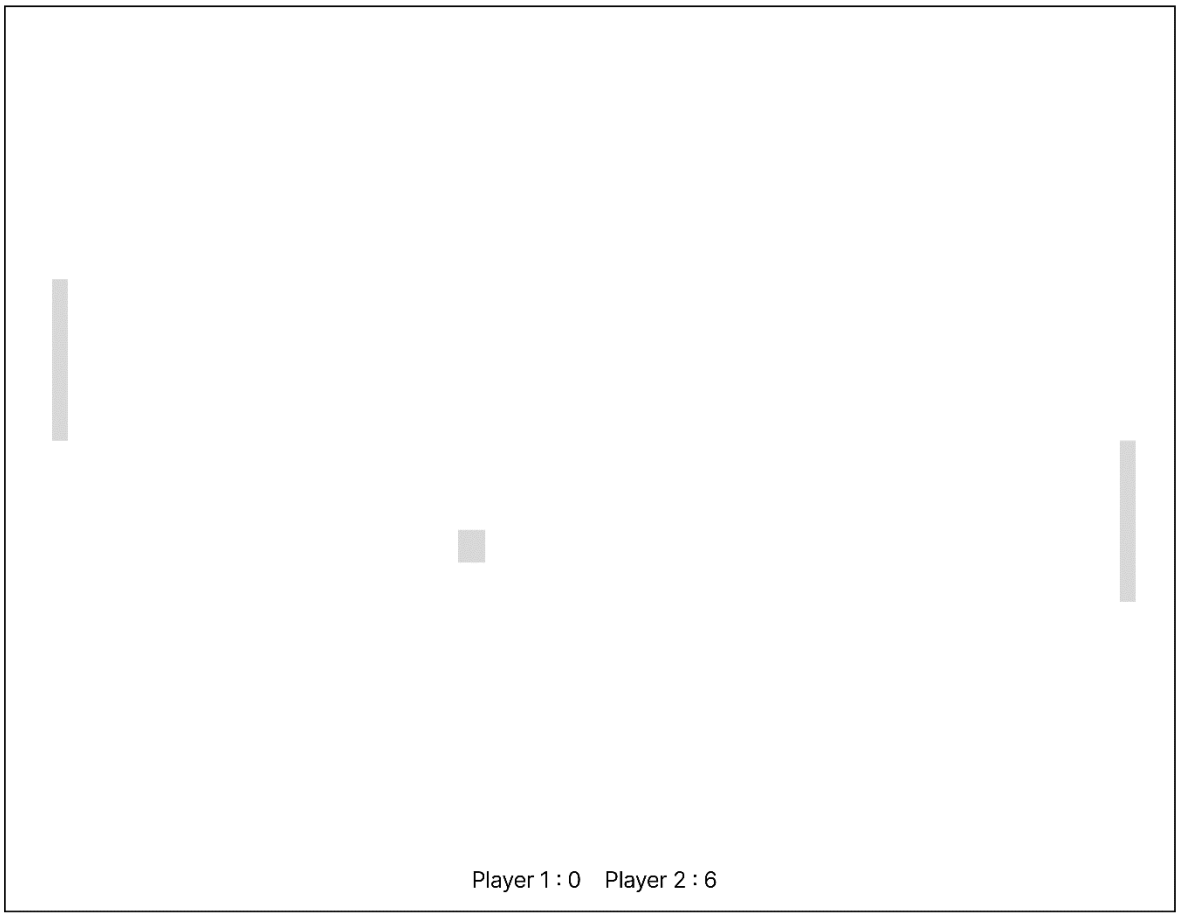
04/04/2024

Inhoudsopgave

Inhoud

Wireframes

Dit is onze wireframe. Wij hebben gekozen voor de score onderin het scherm zodat het niet te veel van het scherm gebruikt en toch makkelijk zichtbaar is.



User Stories

Als eindgebruiker wil ik een paddle hebben die omhoog en omlaag kan bewegen, zodat ik de bal terug naar de tegenstander kan slaan.

Als eindgebruiker wil ik een balletje hebben die heen en weer gaat, zodat ik de bal achter de paddle van de speler kan slaan om een punt te scoren.

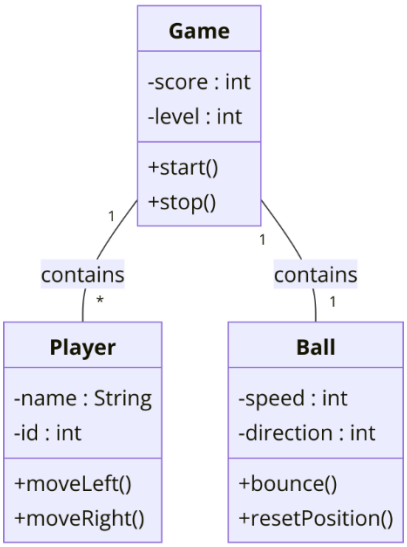
Als eindgebruiker wil ik dat de bal verdwijnt als de bal de muur achter de paddle van mijn tegenstander belandt, zodat ik een punt krijg.

Als eindgebruiker wil ik een scoreboard hebben die werkt voor beiden partijen zodat wij beide de score bij kunnen houden.

Diagrammen

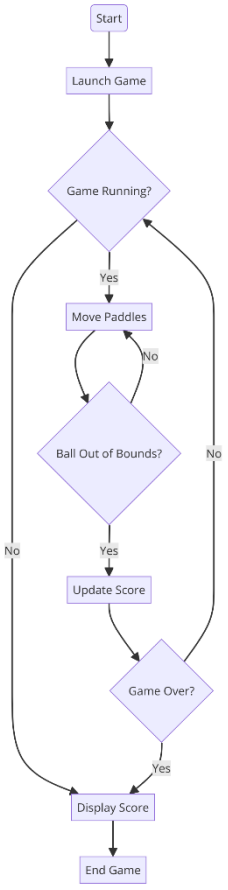
Class Diagram

Dit is onze class diagram
Game heeft 1 of meer Spelers
Spelers heeft één Game
Game heeft één bal
Bal heeft één Game



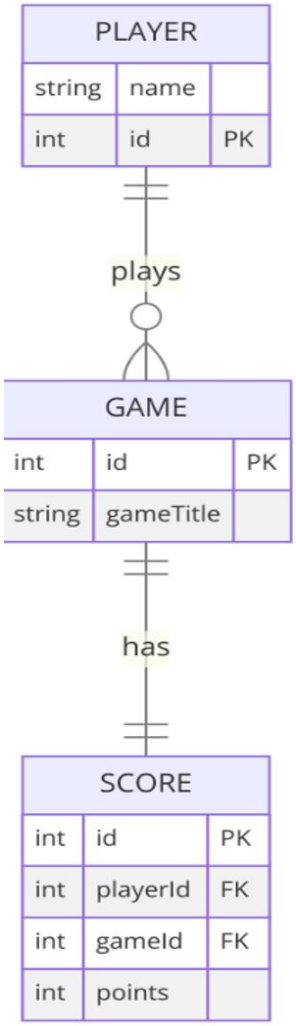
Activity Diagram
Dit is de activity diagram van onze PONG game.

Het begint met “Launch Game” en het eindigt met “display score”

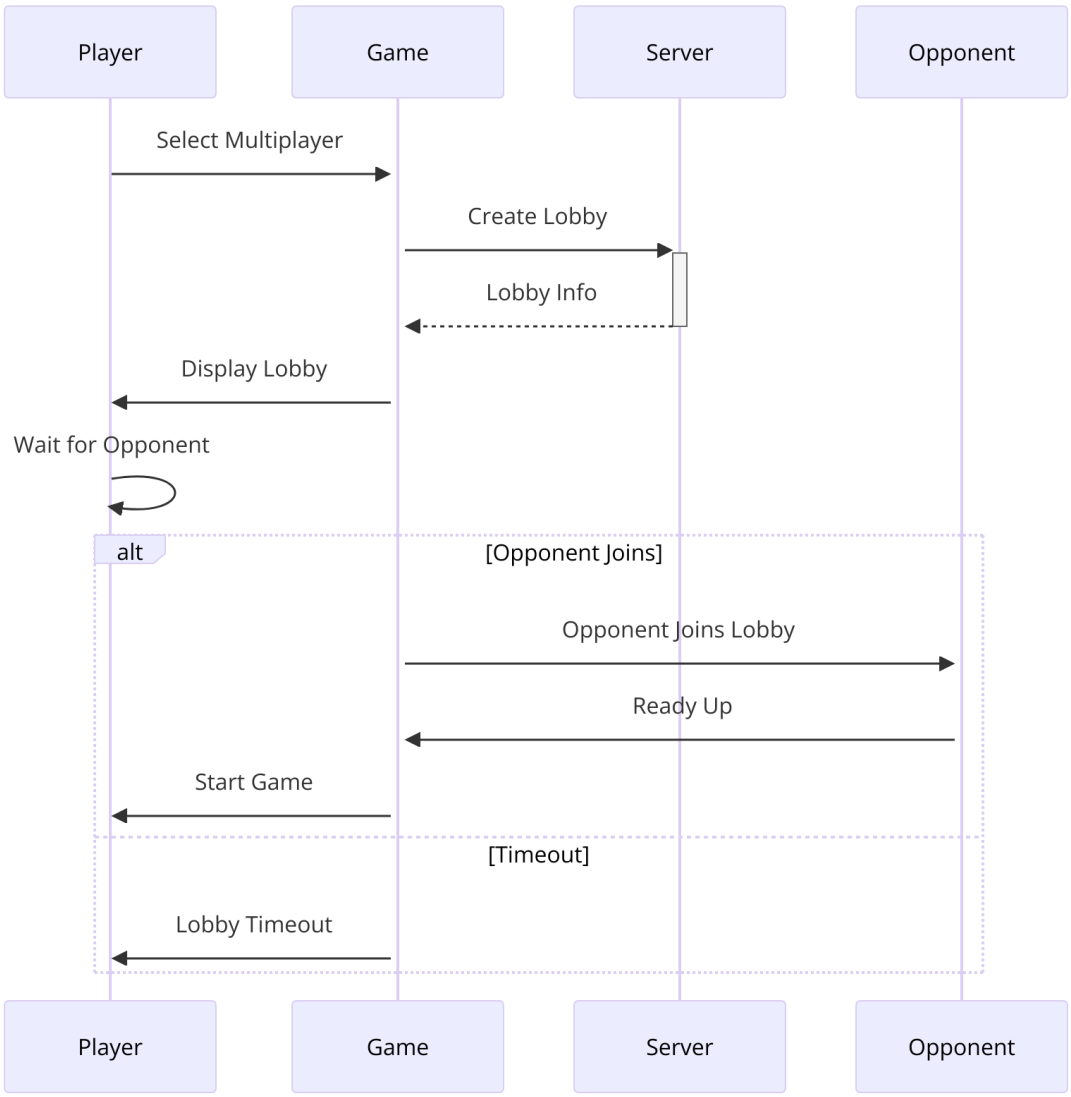


ERD

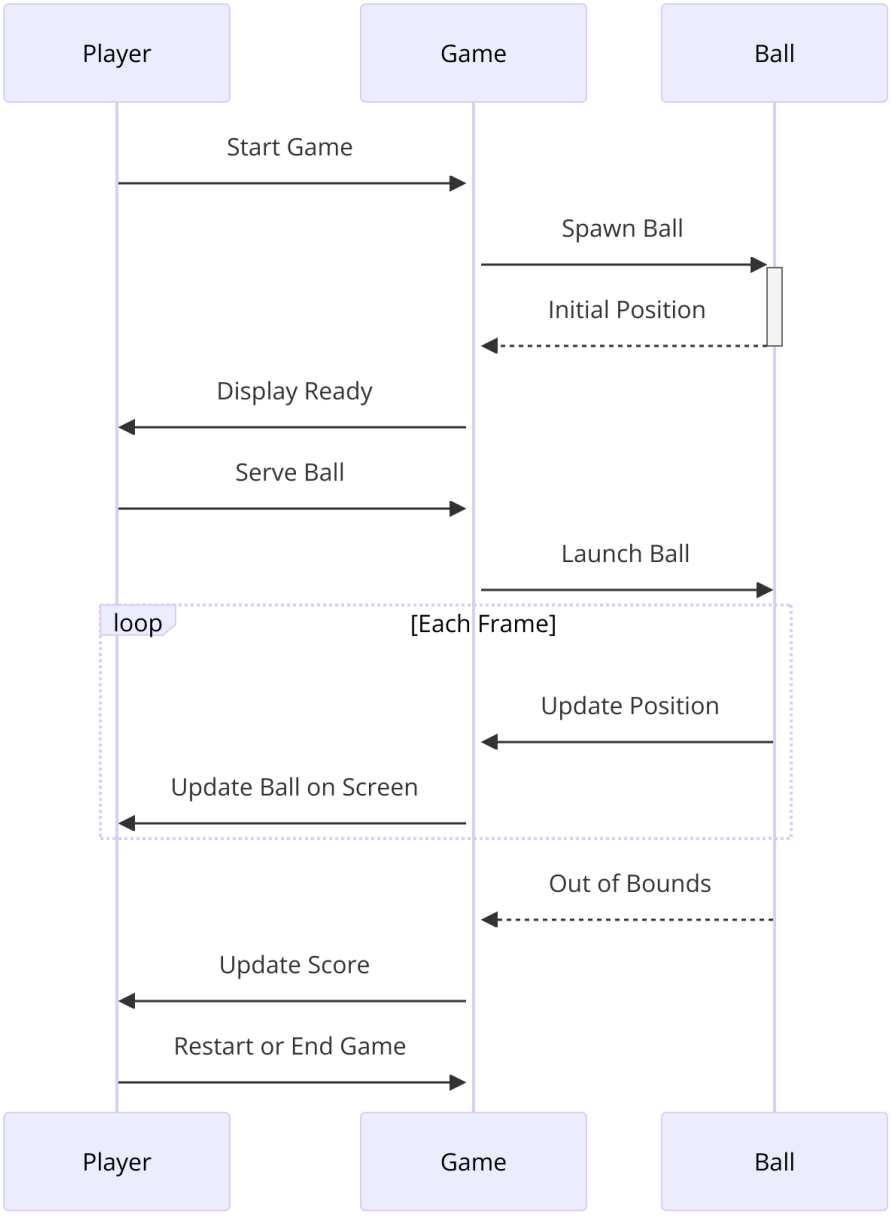
Dit is de ERD van onze PONG game.



Sequence diagram
Dit is de sequence diagram van de PONG game.



Dit is onze sequence diagram



Creational Patterns

Wij gebruiken hier het **Singleton-patroon** omdat het er voor zorgt dat er van een klasse maar één instantie bestaat. Dit helpt bij het verminderen van geheugengebruik.

```
private PongGame()
{
    Console.CursorVisible = false;
    ResetGame();
}

public static PongGame GetInstance()
{
    if (instance == null)
    {
        instance = new PongGame();
    }
    return instance;
}
```

De code zorgt ervoor dat er maar één PongGame object kan zijn. Als er nog geen is dan word er een gemaakt. Als er al een is dan geeft het hij terug. Zo heb je altijd maar één spel actief.

Om het Abstract Factory-patroon toe te passen, kunnen we een fabriek maken die verantwoordelijk is voor het maken van de inputhandler en de monitoren. Hier is hoe je dat kunt doen:

Maak een abstracte klasse genaamd GameFactory waarin methoden worden gedeclareerd voor het maken van de inputhandler en de monitoren.

Maak concrete fabriekklassen die de GameFactory-klasse implementeren en die verantwoordelijk zijn voor het maken van concrete implementaties van de inputhandler en de monitoren.

In de Main-methode gebruiken we de concrete fabriek om de inputhandler en de monitoren te maken en door te geven aan het spel.

```
using System;
using System.Collections.Concurrent;
using System.Text;
using System.Threading;
using System.Threading.Tasks;
using System.Collections.Generic;
using System.Windows.Input;

abstract class GameFactory
{
    public abstract IInputHandler CreateInputHandler();
    public abstract IScoreMonitor CreateScoreMonitor();
```

```
    public abstract IPaddlePositionMonitor CreatePaddlePositionMonitor();
    public abstract IBallPositionMonitor CreateBallPositionMonitor();
}

Concrete Factory 1
class SimpleGameFactory : GameFactory
{
    public override IInputHandler CreateInputHandler()
    {
        return new InputHandler();
    }

    public override IScoreMonitor CreateScoreMonitor()
    {
        return new ScoreMonitor();
    }

    public override IPaddlePositionMonitor CreatePaddlePositionMonitor()
    {
        return new PaddlePositionMonitor();
    }

    public override IBallPositionMonitor CreateBallPositionMonitor()
    {
        return new BallPositionMonitor();
    }
}


class LoggingInputHandlerDecorator : IInputHandler
{
    private readonly IInputHandler _inputHandler;

    public LoggingInputHandlerDecorator(IInputHandler inputHandler)
    {
        _inputHandler = inputHandler;
    }

    public void AddCommand(ConsoleKey key, ICommand command)
    {
        _inputHandler.AddCommand(key, command);
    }

    public void HandleInput(PongGame game)
    {
        Console.WriteLine("Logging: Input is being handled.");
        _inputHandler.HandleInput(game);
    }
}

class Program
{
    static void Main()
    {
        GameFactory factory = new SimpleGameFactory();
```

```
        IInputHandler inputHandler = factory.CreateInputHandler();
        IScoreMonitor scoreMonitor = factory.CreateScoreMonitor();
        IPaddlePositionMonitor paddlePositionMonitor = factory.CreatePaddlePositionMonitor();
        IBallPositionMonitor ballPositionMonitor = factory.CreateBallPositionMonitor();

        PongGame game = new PongGame(inputHandler, scoreMonitor, paddlePositionMonitor,
ballPositionMonitor);

        game.Run();
    }
}
```

Prototype Pattern

```
using System;


// Interface voor het klonen
public interface ICloneableShape
{
    ICloneableShape Clone();
}


// Concrete klasse die het klonen implementeert
public class Rectangle : ICloneableShape
{
    public int Width { get; set; }
    public int Height { get; set; }


    public Rectangle(int width, int height)
    {
        Width = width;
        Height = height;
    }


    public ICloneableShape Clone()
    {
```

```
        // Maak een nieuw exemplaar van het huidige object en kopieer de eigenschappen
        return new Rectangle(Width, Height);
    }

    public override string ToString()
    {
        return $"Rectangle: Width={Width}, Height={Height}";
    }
}

class Program
{
    static void Main()
    {
        // Maak een prototype van een rechthoek
        var rectanglePrototype = new Rectangle(10, 20);

        // Maak een kopie van het prototype
        var clonedRectangle = rectanglePrototype.Clone() as Rectangle;

        // Controleer of de kloon correct is gemaakt
        Console.WriteLine("Prototype: " + rectanglePrototype);
        Console.WriteLine("Cloned: " + clonedRectangle);
    }
}
```

Behavioral Patterns

Dit **Command Pattern** maakt het makkelijker om nieuwe acties toe te voegen en verbeterd de herbruikbaarheid van de code.

```
interface ICommand
{
    void Execute(PongGame game);
}

class MovePaddleUpCommand : ICommand
{
    private readonly int _player;

    public MovePaddleUpCommand(int player)
    {
        _player = player;
    }

    public void Execute(PongGame game)
    {
        game.MovePaddleUp(_player);
    }
}

class MovePaddleDownCommand : ICommand
{
    private readonly int _player;

    public MovePaddleDownCommand(int player)
    {
        _player = player;
    }

    public void Execute(PongGame game)
    {
        game.MovePaddleDown(_player);
    }
}

class PauseCommand : ICommand
{
    public void Execute(PongGame game)
    {
        game.Pause();
    }
}

class InputHandler
{
    private readonly Dictionary<ConsoleKey, ICommand> _commands = new Dictionary<ConsoleKey,
ICommand>();
```

```
    public void AddCommand(ConsoleKey key, ICommand command)
    {
        _commands[key] = command;
    }

    public void HandleInput(PongGame game)
    {
        if (Console.KeyAvailable)
        {
            var key = Console.ReadKey(true).Key;
            if (_commands.ContainsKey(key))
            {
                _commands[key].Execute(game);
            }
        }
    }
}

class PongGame
{
    private int paddle1Position, paddle2Position;
    private int ballX, ballY;
    private int ballVelocityX, ballVelocityY;
    private const int playfieldWidth = 80;
    private const int playfieldHeight = 24;
    private const int paddleHeight = 4;
    private int player1Score, player2Score;
    private bool gameRunning = true;
    private DateTime lastBallMoveTime;
    private bool paused = false;
    private readonly InputHandler _inputHandler;

    public PongGame()
    {
        Console.CursorVisible = false;
        _inputHandler = new InputHandler();
        _inputHandler.AddCommand(ConsoleKey.W, new MovePaddleUpCommand(1));
        _inputHandler.AddCommand(ConsoleKey.S, new MovePaddleDownCommand(1));
        _inputHandler.AddCommand(ConsoleKey.UpArrow, new MovePaddleUpCommand(2));
        _inputHandler.AddCommand(ConsoleKey.DownArrow, new MovePaddleDownCommand(2));
        _inputHandler.AddCommand(ConsoleKey.Escape, new PauseCommand());
        ResetGame();
    }

    private void ResetGame()
    {
        paddle1Position = paddle2Position = playfieldHeight / 2 - paddleHeight / 2;
        player1Score = 0;
        player2Score = 0;
        ResetBall();
        Console.Clear();
    }

    private void ResetBall()
    {
```

```
        ballX = playfieldWidth / 2;
        ballY = new Random().Next(1, playfieldHeight - 1);
        ballVelocityX = new Random().Next(0, 2) * 2 - 1
        ballVelocityY = new Random().Next(0, 2) * 2 - 1;
        lastBallMoveTime = DateTime.Now;
    }

    public void Run()
    {
        while (gameRunning)
        {
            if (!paused)
            {
                if ((DateTime.Now - lastBallMoveTime).TotalMilliseconds > 100)
                {
                    MoveBall();
                    CheckCollision();
                    lastBallMoveTime = DateTime.Now;
                }
            }

            _inputHandler.HandleInput(this);
            Draw();
            Thread.Sleep(20);
        }
    }

    private void MoveBall()
    {
        ballX += ballVelocityX;
        ballY += ballVelocityY;
    }

    private void CheckCollision()
    {
        if (ballY <= 1 || ballY >= playfieldHeight - 2)
        {
            ballVelocityY = -ballVelocityY;
        }

        if (ballX == 3 && ballY >= paddle1Position && ballY <= paddle1Position + paddleHeight)
        {
            ballVelocityX = -ballVelocityX;
        }

        if (ballX == playfieldWidth - 4 && ballY >= paddle2Position && ballY <= paddle2Position +
paddleHeight)
        {
            ballVelocityX = -ballVelocityX;
        }

        if (ballX < 1)
        {
            player2Score++;
            ResetBall();
```

```
        }
        else if (ballX > playfieldWidth - 2)
        {
            player1Score++;
            ResetBall();
        }
    }

    public void MovePaddleUp(int player)
    {
        if (player == 1)
            paddle1Position = Math.Max(1, paddle1Position - 1);
        else if (player == 2)
            paddle2Position = Math.Max(1, paddle2Position - 1);
    }

    public void MovePaddleDown(int player)
    {
        if (player == 1)
            paddle1Position = Math.Min(playfieldHeight - paddleHeight - 1, paddle1Position + 1);
        else if (player == 2)
            paddle2Position = Math.Min(playfieldHeight - paddleHeight - 1, paddle2Position + 1);
    }

    public void Pause()
    {
        paused = !paused;
    }
```

Momento pattern:
Om het Memento-patroon toe te voegen aan de PongGame, kunnen we een Memento-klasse maken die de toestand van het spel op een bepaald moment vastlegt en herstelt. Laten we de wijzigingen aanbrengen:

```
static void Main()
{
    var baseInputHandler = new InputHandler();
    var decoratedInputHandler = new LoggingInputHandlerDecorator(baseInputHandler);
    var game = new PongGame(decoratedInputHandler);
    var gameFacade = new PongGameFacade(game);

    var scoreMonitor = new ScoreMonitor();
    var paddlePositionMonitor = new PaddlePositionMonitor();
    var ballPositionMonitor = new BallPositionMonitor();

    game.Run(scoreMonitor);

    game.SaveState();
    game.MovePaddleUp(1);
```

```
    game.RestoreState();
}
```

Strategy-pattern

```
using System;


// Context
class ShoppingCart
{
   private IShippingStrategy shippingStrategy;


   public void SetShippingStrategy(IShippingStrategy shippingStrategy)
   {
      this.shippingStrategy = shippingStrategy;
   }


   public void Checkout()
   {
      Console.WriteLine("Bestelling wordt verwerkt...");
      double shippingCost = shippingStrategy.CalculateShippingCost();
      Console.WriteLine($"Verzendkosten: ${shippingCost}");
      // Andere checkout logica...
   }
}


// Strategy interface
interface IShippingStrategy
{
   double CalculateShippingCost();
}


// Concrete Strategy 1
```

```
class StandardShippingStrategy : IShippingStrategy
{
    public double CalculateShippingCost()
    {
        // Simulatie van de berekening van de verzendkosten voor standaardverzending
        return 5.00;
    }
}

// Concrete Strategy 2
class ExpressShippingStrategy : IShippingStrategy
{
    public double CalculateShippingCost()
    {
        // Simulatie van de berekening van de verzendkosten voor express-verzending
        return 10.00;
    }
}

class Program
{
    static void Main()
    {
        // Maak een winkelwagen
        var cart = new ShoppingCart();

        // Kies een verzendstrategie (Standard Shipping)
        cart.SetShippingStrategy(new StandardShippingStrategy());

        // Voltooi de aankoop
        cart.Checkout();
```

```
        // Verander de verzendstrategie naar Express Shipping en voltooi de aankoop opnieuw
        cart.SetShippingStrategy(new ExpressShippingStrategy());
        cart.Checkout();
    }
}
```

Concurrency Pattern

Het **Active Object Pattern** toegepast door een aparte inputhandler te maken die verantwoordelijk is voor het afhandelen van invoer van de speler en het uitvoeren van bijbehorende acties zonder de hoofdthread van het spel te blokkeren.

```
using System;
using System.Collections.Concurrent;
using System.Text;
using System.Threading;
using System.Threading.Tasks;
using System.Collections.Generic;

class Command
{
    public Action Execute { get; set; }
}

interface ICommand
{
    void Execute(PongGame game);
}

class MovePaddleUpCommand : ICommand
{
    private readonly int _player;

    public MovePaddleUpCommand(int player)
    {
        _player = player;
    }

    public void Execute(PongGame game)
    {
        game.MovePaddleUp(_player);
    }
}

class MovePaddleDownCommand : ICommand
{
    private readonly int _player;

    public MovePaddleDownCommand(int player)
    {
        _player = player;
    }

    public void Execute(PongGame game)
    {
        game.MovePaddleDown(_player);
    }
}

class PauseCommand : ICommand
```

```
{
    public void Execute(PongGame game)
    {
        game.Pause();
    }
}

class InputHandler
{
    private readonly Dictionary<ConsoleKey, ICommand> _commands = new Dictionary<ConsoleKey,
ICommand>();

    public void AddCommand(ConsoleKey key, ICommand command)
    {
        _commands[key] = command;
    }

    public void HandleInput(PongGame game)
    {
        if (Console.KeyAvailable)
        {
            var key = Console.ReadKey(true).Key;
            if (_commands.ContainsKey(key))
            {
                _commands[key].Execute(game);
            }
        }
    }
}

class PongGame
{
    private int paddle1Position, paddle2Position;
    private int ballX, ballY;
    private int ballVelocityX, ballVelocityY;
    private const int playfieldWidth = 80;
    private const int playfieldHeight = 24;
    private const int paddleHeight = 4;
    private int player1Score, player2Score;
    private bool gameRunning = true;
    private DateTime lastBallMoveTime;
    private bool paused = false;
    private readonly InputHandler _inputHandler;
    private readonly BlockingCollection<Command> _commandQueue = new
BlockingCollection<Command>();

    public PongGame()
    {
        Console.CursorVisible = false;
        _inputHandler = new InputHandler();
        _inputHandler.AddCommand(ConsoleKey.W, new MovePaddleUpCommand(1));
        _inputHandler.AddCommand(ConsoleKey.S, new MovePaddleDownCommand(1));
        _inputHandler.AddCommand(ConsoleKey.UpArrow, new MovePaddleUpCommand(2));
        _inputHandler.AddCommand(ConsoleKey.DownArrow, new MovePaddleDownCommand(2));
        _inputHandler.AddCommand(ConsoleKey.Escape, new PauseCommand());
```

```
    Task.Factory.StartNew(Worker);
    ResetGame();
}

private void ResetGame()
{
    paddle1Position = paddle2Position = playfieldHeight / 2 - paddleHeight / 2;
    player1Score = 0;
    player2Score = 0;
    ResetBall();
    Console.Clear();
}

private void ResetBall()
{
    ballX = playfieldWidth / 2;
    ballY = new Random().Next(1, playfieldHeight - 1);
    ballVelocityX = new Random().Next(0, 2) * 2 - 1;
    ballVelocityY = new Random().Next(0, 2) * 2 - 1;
    lastBallMoveTime = DateTime.Now;
}

public void Run()
{
    while (gameRunning)
    {
        if (!paused)
        {
            if ((DateTime.Now - lastBallMoveTime).TotalMilliseconds > 100)
            {
                MoveBall();
                CheckCollision();
                lastBallMoveTime = DateTime.Now;
            }
        }

        _inputHandler.HandleInput(this);
        Draw();
        Thread.Sleep(20);
    }
}

private void Worker()
{
    while (true)
    {
        var command = _commandQueue.Take();
        command.Execute();
    }
}

private void EnqueueCommand(Action action)
{
    _commandQueue.Add(new Command { Execute = action });
}
```

```
    public void EnqueueMovePaddleUp(int player)
    {
        EnqueueCommand(() => MovePaddleUp(player));
    }

    public void EnqueueMovePaddleDown(int player)
    {
        EnqueueCommand(() => MovePaddleDown(player));
    }

    public void EnqueuePause()
    {
        EnqueueCommand(Pause);
    }
```

Monitor pattern:

```
interface IMonitor
{
    void DisplayStatus(PongGame game);
}

class ScoreMonitor : IMonitor
{
    public void DisplayStatus(PongGame game)
    {
        Console.WriteLine($"Current Score - Player 1: {game.Player1Score}, Player 2: {game.Player2Score}");
    }
}

class PaddlePositionMonitor : IMonitor
{
    public void DisplayStatus(PongGame game)
    {
        Console.WriteLine($"Paddle Positions - Player 1: {game.Paddle1Position}, Player 2:
{game.Paddle2Position}");
    }
}

class BallPositionMonitor : IMonitor
{
    public void DisplayStatus(PongGame game)
    {
        Console.WriteLine($"Ball Position - X: {game.BallX}, Y: {game.BallY}");
    }
}

class PongGame
{
    public int Paddle1Position { get; private set; }
```

```
    public int Paddle2Position { get; private set; }
    public int BallX { get; private set; }
    public int BallY { get; private set; }
    public int Player1Score { get; private set; }
    public int Player2Score { get; private set; }


    public void Run(IMonitor monitor)
    {
        while (gameRunning)
        {
            if (!paused)
            {
                if ((DateTime.Now - lastBallMoveTime).TotalMilliseconds > 100)
                {
                    MoveBall();
                    CheckCollision();
                    lastBallMoveTime = DateTime.Now;
                }
            }

            _inputHandler.HandleInput(this);
            Draw();
            monitor.DisplayStatus(this);
            Thread.Sleep(20);
        }
    }
}

class Program
{
    static void Main()
    {
        var baseInputHandler = new InputHandler();
        var decoratedInputHandler = new LoggingInputHandlerDecorator(baseInputHandler);
        var game = new PongGame(decoratedInputHandler);

        var scoreMonitor = new ScoreMonitor();
        var paddlePositionMonitor = new PaddlePositionMonitor();
        var ballPositionMonitor = new BallPositionMonitor();
        game.Run(scoreMonitor);

    }
}
```

Producer-Consumer Pattern

```
using System;
using System.Collections.Concurrent;
using System.Threading;
using System.Threading.Tasks;

class Program
{
  static void Main()
  {
    // Maak een gedeelde buffer met een maximale capaciteit van 5 items
    var buffer = new BlockingCollection<int>(boundedCapacity: 5);

    // Start een producent en een consument
    Task.Run(() => Producer(buffer));
    Task.Run(() => Consumer(buffer));

    Console.ReadLine(); // Wacht tot de gebruiker op Enter drukt om het programma af te sluiten
  }

  static void Producer(BlockingCollection<int> buffer)
  {
    for (int i = 0; i < 10; i++)
    {
      // Voeg een item toe aan de buffer
      buffer.Add(i);
      Console.WriteLine($"Producent voegt item toe: {i}");
      Thread.Sleep(TimeSpan.FromSeconds(1)); // Simuleer wat verwerkingstijd
    }
```

```
        // Geef aan dat de producent klaar is door een negatief getal toe te voegen aan de buffer
        buffer.Add(-1);
        Console.WriteLine("Producent is klaar.");
    }

    static void Consumer(BlockingCollection<int> buffer)
    {
        // Blijf items uit de buffer halen totdat de producent klaar is
        while (true)
        {
            int item = buffer.Take();

            // Controleer of de producent klaar is
            if (item == -1)
            {
                Console.WriteLine("Consument heeft aangegeven dat de producent klaar is.");
                break;
            }

            // Verwerk het item
            Console.WriteLine($"Consument ontvangt item: {item}");
            Thread.Sleep(TimeSpan.FromSeconds(2)); // Simuleer wat verwerkingstijd
        }
    }
}
```

Structural Patterns

Wij hebben voor het **Decorator Pattern** gekozen en toegepast omdat het een flexibele manier is om functionaliteit toe te voegen aan bestaande klassen zonder de structuur daarvan te wijzigen.

In dit specifieke geval wilden we de mogelijkheid toevoegen om invoer te loggen voordat deze wordt afgehandeld door de InputHandler klasse.

```
interface IInputHandler
{
    void AddCommand(ConsoleKey key, ICommand command);
    void HandleInput(PongGame game);
}

class InputHandler : IInputHandler
{
    private readonly Dictionary<ConsoleKey, ICommand> _commands = new Dictionary<ConsoleKey,
ICommand>();

    public void AddCommand(ConsoleKey key, ICommand command)
    {
        _commands[key] = command;
    }

    public void HandleInput(PongGame game)
    {
        if (Console.KeyAvailable)
        {
            var key = Console.ReadKey(true).Key;
            if (_commands.ContainsKey(key))
            {
                _commands[key].Execute(game);
            }
        }
    }
}

class LoggingInputHandlerDecorator : IInputHandler
{
    private readonly IInputHandler _inputHandler;

    public LoggingInputHandlerDecorator(IInputHandler inputHandler)
    {
        _inputHandler = inputHandler;
    }

    public void AddCommand(ConsoleKey key, ICommand command)
    {
        _inputHandler.AddCommand(key, command);
    }

    public void HandleInput(PongGame game)
    {
        Console.WriteLine("Logging: Input is being handled.");
        _inputHandler.HandleInput(game);
    }
}
```

```
class Program
{
    static void Main()
    {
        var baseInputHandler = new InputHandler();
        var decoratedInputHandler = new LoggingInputHandlerDecorator(baseInputHandler);
        var game = new PongGame(decoratedInputHandler);
        game.Run();
    }
}
```

Facade Pattern:

```
public class PongGameFacade
{
    private readonly PongGame _game;

    public PongGameFacade(PongGame game)
    {
        _game = game;
    }

    public void StartGame()
    {
        _game.ResetGame();
        _game.Run();
    }

    public void StopGame()
    {
        _game.Stop();
    }
}

static void Main()
{
    var baseInputHandler = new InputHandler();
```

```
    var decoratedInputHandler = new LoggingInputHandlerDecorator(baseInputHandler);
    var game = new PongGame(decoratedInputHandler);
    var gameFacade = new PongGameFacade(game);

    gameFacade.StartGame();
}
```

Proxy Pattern:

```
using System;

// Interface voor het onderwerp
interface ISubject
{
    void Request();
}

// Werkelijk onderwerp
class RealSubject : ISubject
{
    public void Request()
    {
        Console.WriteLine("Het echte onderwerp voert de aanvraag uit.");
    }
}

// Proxy voor het onderwerp
class Proxy : ISubject
{
    private RealSubject realSubject;

    public void Request()
```

```
  {
    // Het echte onderwerp wordt alleen gemaakt wanneer het nodig is
    if (realSubject == null)
    {
      realSubject = new RealSubject();
    }

    // Voor of na de werkelijke aanroep van het onderwerp kunnen extra logica worden toegevoegd
    Console.WriteLine("De proxy voert extra logica uit vóór het doorsturen van de aanvraag.");

    // Doorsturen van de aanvraag naar het echte onderwerp
    realSubject.Request();

    Console.WriteLine("De proxy voert extra logica uit na het doorsturen van de aanvraag.");
  }
}

class Program
{
  static void Main()
  {
    // Maak een instantie van de Proxy
    ISubject proxy = new Proxy();

    // Voer de aanvraag uit via de proxy
    proxy.Request();
  }
}
```