

SSD Snake applicatie

Applied design pattern



Student: Sam Maijer
Studie: AD Software development
Vak: Applied Design Patterns
Datum: 10/02/2026

Inhoudsopgaven

Inhoudsopgaven.....	1
1. Inleiding.....	2
2. Design.....	3
3. UML.....	8
3.1 Wat is UML.....	8
3.2 Activity diagram.....	8
3.3 Sequence diagram.....	9
3.4 Use case diagram.....	11
3.5 Class diagram.....	12
3.6 State diagram.....	13
4. Solid principles.....	15
5. Design smells.....	16
6. Design patterns.....	17
7.1 Creational pattern.....	17
7.2 Behavioural pattern.....	21
7.3 Concurrency pattern.....	26
7.4 Structural pattern.....	29
7. Bronnenlijst.....	32

1. Inleiding

In dit document wordt het ontwerp en architectuur van de Snake game beschreven. Het doel van dit document is om te laten zien hoe het project technisch is opgebouwd en welke designprincipes en patronen zijn gebruikt.

2. Design

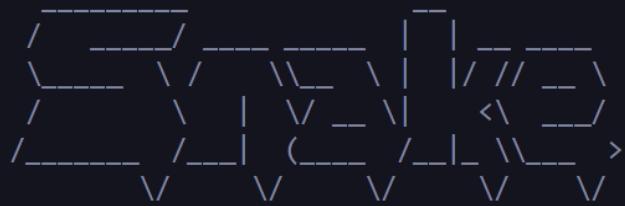
Start menu



Bij het opstarten van het programma komt de gebruiker terecht in het startmenu. Vanuit dit menu kan de gebruiker navigeren naar de verschillende onderdelen van het spel, zoals het starten van een nieuw spel of het openen van de help functie. Het startmenu vormt het centrale punt van waaruit alle andere onderdelen bereikbaar zijn.

In het oorspronkelijke ontwerp was het plan om een scoreboard toe te voegen, waardoor er in de eerste ontwerpen een score knop aanwezig was. Dit onderdeel is uiteindelijk niet geïmplementeerd en daarom ook verwijderd uit het eindproduct. In plaats daarvan is later een help knop toegevoegd, die de gebruiker extra uitleg geeft over de spelbesturing en spelregels.

Uiteindelijke design

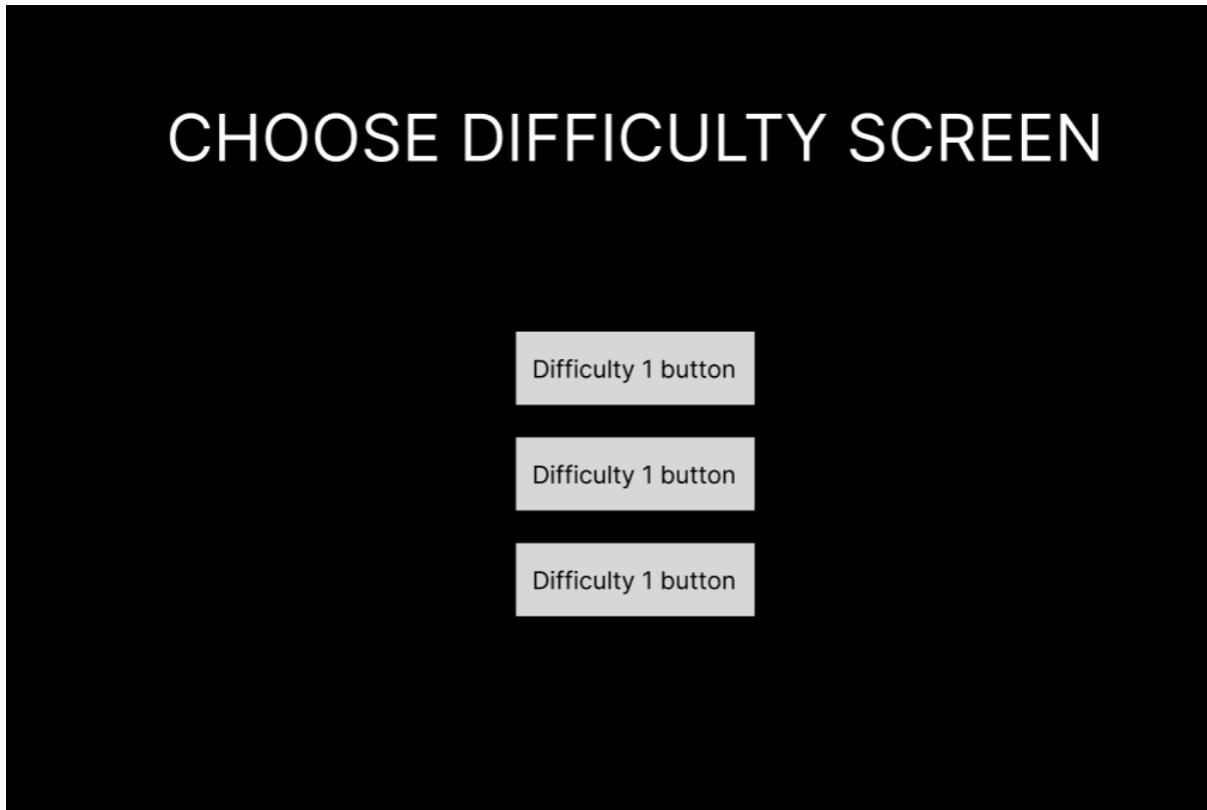


>Play
Help
Quit

Use WASD keys to navigate.

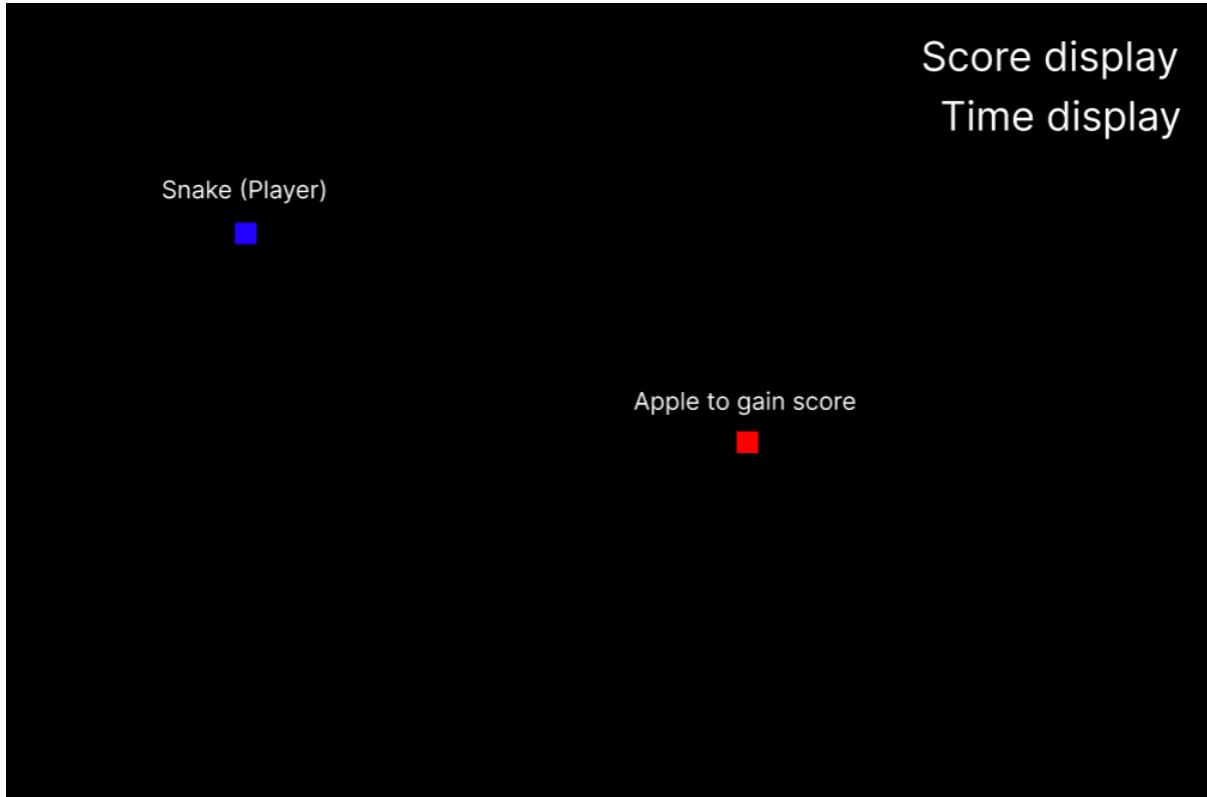
Made by: Sam Maijer
Sadek Al Mousawi
Daniel Pustjens

Niveau selectie



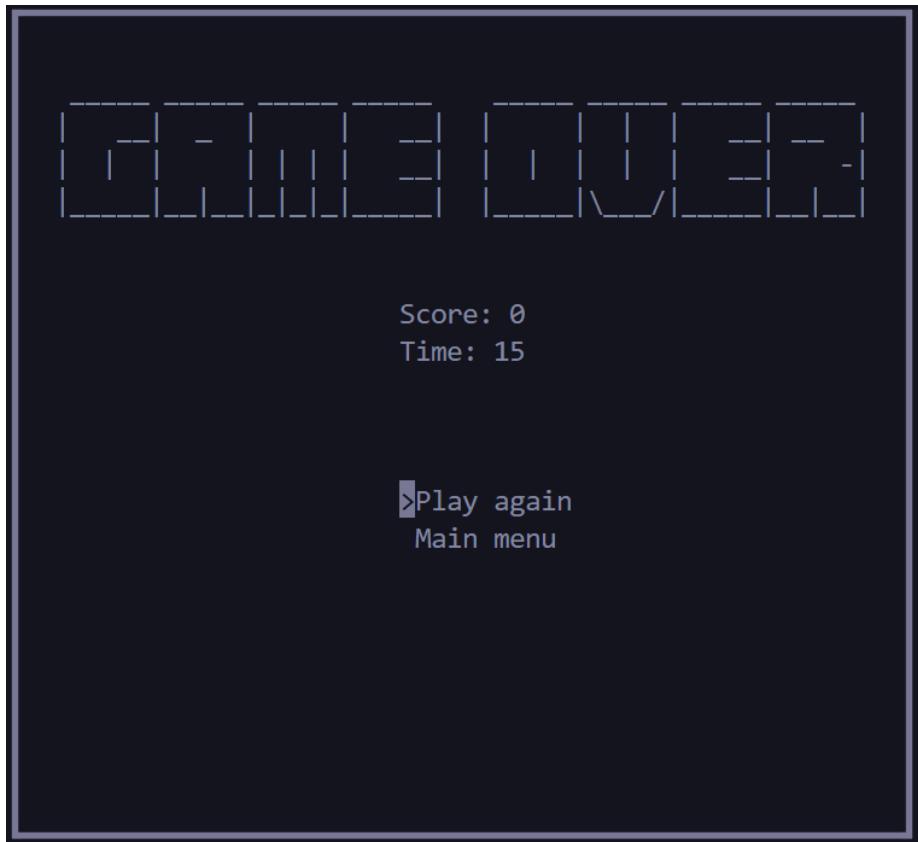
In het niveau selectiescherm kan de gebruiker kiezen op welke moeilijkheidsgraad het spel gespeeld wordt. De gekozen moeilijkheid heeft invloed op de snelheid van de slang. Hoe hoger de moeilijkheidsgraad, hoe sneller de slang beweegt en hoe moeilijker het spel wordt.

Gameplay



Het gameplay-scherm is het hoofdscherm van het spel waarin de Snake-game wordt gespeeld. Tijdens het spelen ziet de gebruiker de slang bewegen over het speelveld. Daarnaast worden de score en de speeltijd weergegeven. De score geeft aan hoeveel appels de speler heeft gegeten, terwijl de tijd bijhoudt hoe lang de speler actief is in het spel.

Game over



Wanneer het spel eindigt, bijvoorbeeld doordat de speler tegen een muur of tegen de eigen slang botst, wordt het game-over scherm weergegeven. In dit scherm ziet de gebruiker de behaalde score en de totale speeltijd. Vanuit het game-over scherm kan de speler ervoor kiezen om opnieuw te spelen of terug te keren naar het startmenu.

Omdat het scorebord-systeem uiteindelijk is geschrapt, is het oorspronkelijke ontwerp waarin de speler zijn score kon invoeren vervangen door dit game-over scherm met directe keuzemogelijkheden.

3. UML

3.1 Wat is UML

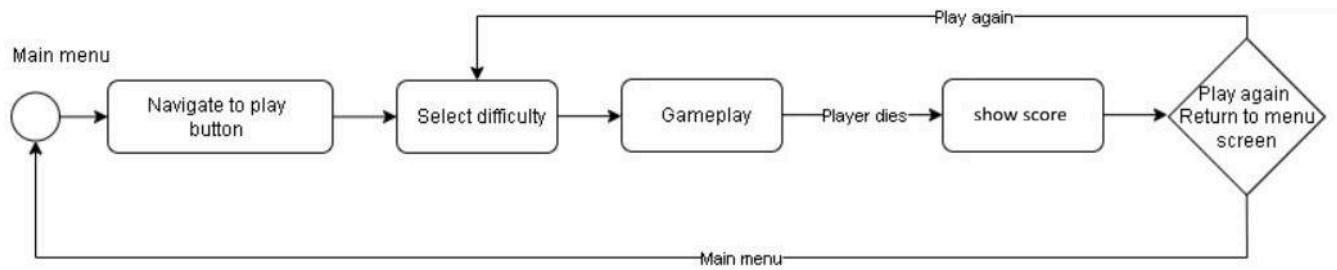
UML staat voor Unified Modeling Language en is een gestandaardiseerde modelleertaal die wordt gebruikt om software systemen te beschrijven, visualiseren en documenteren. UML maakt gebruik van grafische diagrammen om de structuur en het gedrag van een systeem overzichtelijk te laten zien.

Het gebruik van UML helpt ontwikkelteams om complexe systemen beter te begrijpen en te communiceren. Door middel van diagrammen kunnen ontwerpkeuzes duidelijk worden vastgelegd, nog voordat de echte implementatie begint. Dit maakt het makkelijker om fouten vroegtijdig te vinden, alternatieve ontwerpen te bespreken en de software beter te onderhouden.

Daarnaast ondersteunt UML verschillende diagram typen, die elk een ander perspectief op het systeem geven. Zo tonen structurele diagrammen, zoals het Class Diagram, de opbouw van het systeem, terwijl gedrags diagrammen, zoals het Use Case-, Activity- en State Machine diagram, laten zien hoe het systeem zich gedraagt en hoe gebruikers ermee interacteren.

Samengevat wordt UML gebruikt om de complexiteit van software te beheersen, de communicatie binnen een project te verbeteren en het ontwerp van een systeem duidelijk en gestructureerd vast te leggen.

3.2 Activity diagram

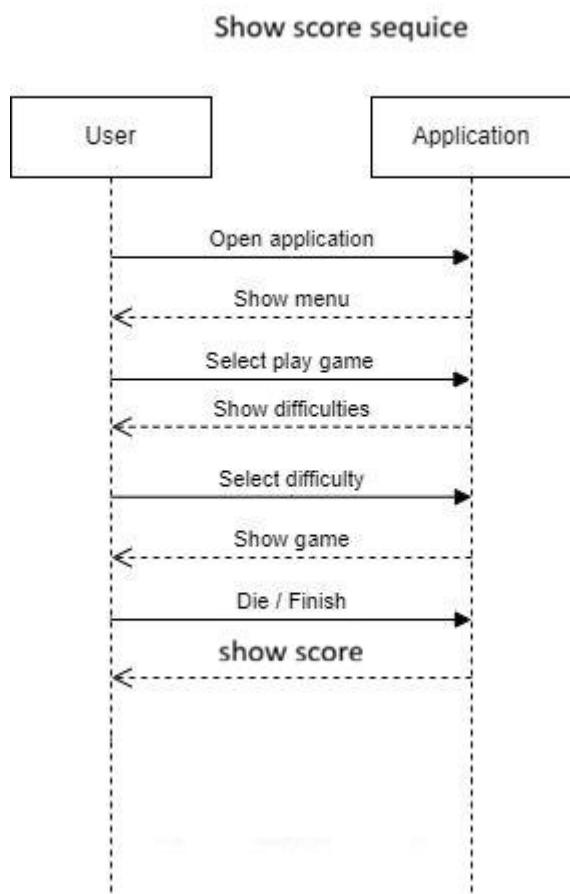


De activity diagram laat zien hoe het proces er voor de gebruiker uitziet ziet van de gehele applicatie

Het begint in het hoofdmenu waar de gebruiker op de play knop moet klikken om een spel te spelen. Voordat de gebruiker kan spelen, moet hij eerst de moeilijkheidsgraad kiezen.

Daarna start het spel en sterft de gebruiker uiteindelijk in het spel, waardoor het spel wordt beëindigd en de score wordt getoond. De gebruiker heeft dan de keuze om opnieuw te spelen (keert terug naar de moeilijkheidsgraad) of om naar het main menu te gaan.

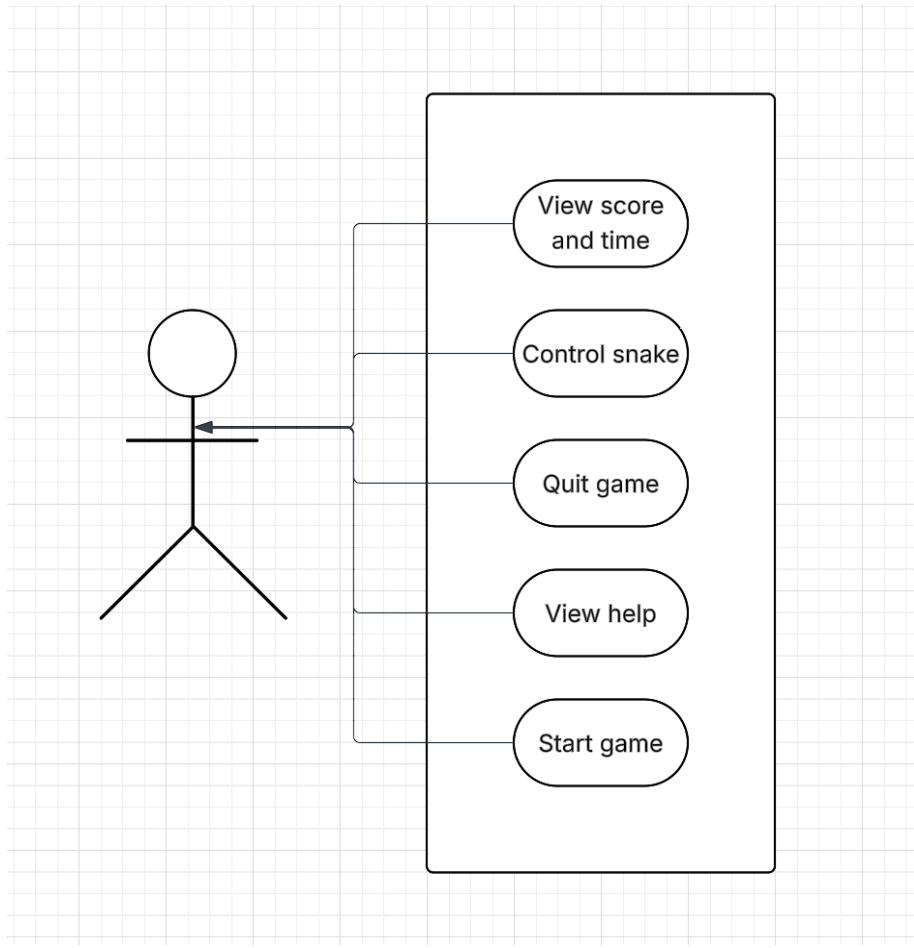
3.3 Sequence diagram



Het sequence diagram laat zien hoe het proces tussen gebruiker en de applicatie eruitziet. De 'Show Score' sequence diagram toont het proces van hoe de score wordt getoond. Het proces:

1. Het begint met de gebruiker die de game (applicatie) opstart.
2. De applicatie laat de gebruiker het hoofdmenu zien aan de gebruiker.
3. De gebruiker kiest voor 'play game'.
4. De applicatie laat de moeilijkheidsgraden (levels) zien.
5. De gebruiker kiest een moeilijkheidsgraad.
6. De applicatie laat de gameplay zien.
7. De gebruiker gaat dood in de game en het spel is voorbij.
8. De applicatie laat de gebruiker zijn score zien

3.4 Use case diagram



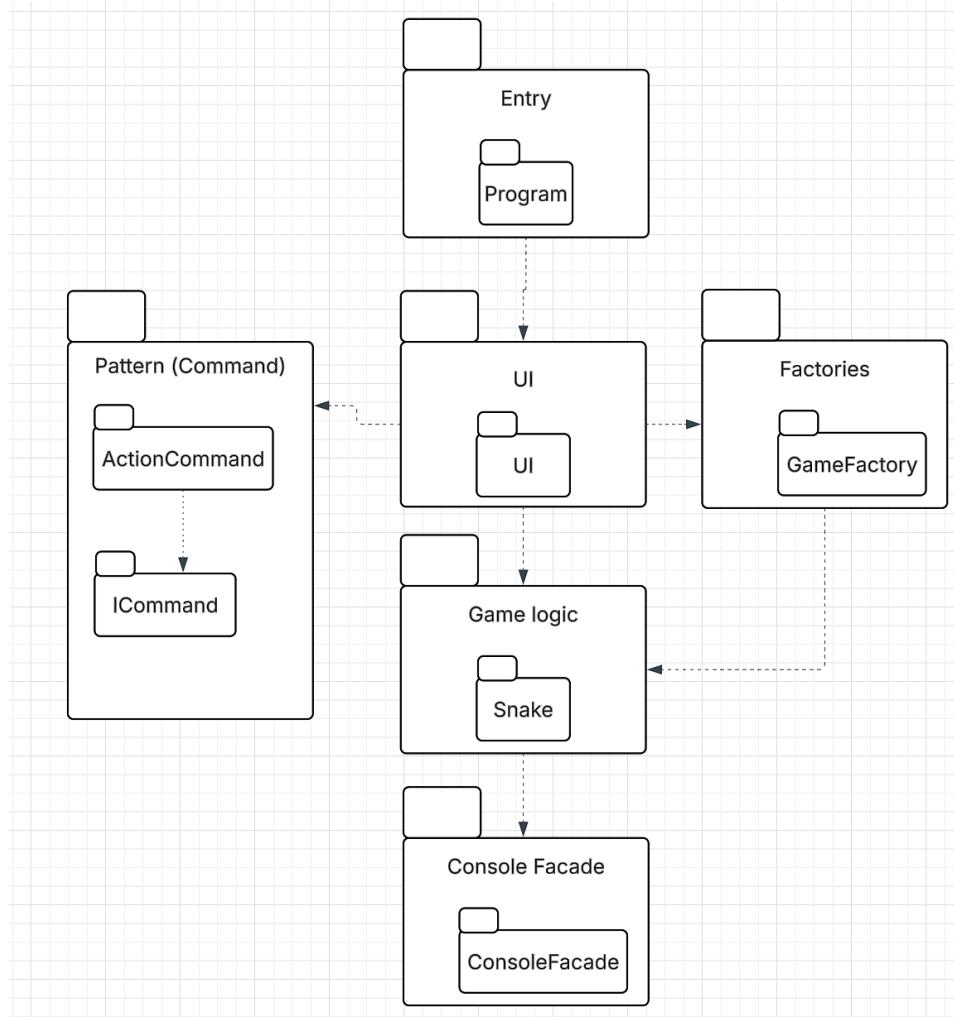
Het Use Case diagram geeft een overzicht van de functionaliteiten van de Snake-game vanuit het perspectief van de gebruiker. Het diagram laat zien welke acties de speler kan uitvoeren en hoe deze acties door het systeem worden ondersteund.

De speler is de enige actor in het systeem. Vanuit het spel kan de speler het spel starten, de slang besturen, de score en speeltijd bekijken, de help functie openen en het spel afsluiten. Deze use cases komen direct overeen met de functionaliteit die in de applicatie is geïmplementeerd, zoals het hoofdmenu, de spelbesturing en de weergave van score en tijd tijdens het spelen.

Het diagram bevat geen externe services, zoals authenticatie- of databaseservices. Dit is een bewuste keuze, omdat de Snake-game volledig lokaal draait en geen gebruik maakt van externe systemen. Er worden geen gebruikersaccounts aangemaakt en scores of tijden worden niet permanent opgeslagen. Alle logica en gegevensverwerking vinden plaats binnen de applicatie zelf.

Het Use Case diagram is bedoeld om een globaal en functioneel overzicht te geven van het systeem. De volgorde van acties en de interne werking van het spel zijn daarom niet opgenomen in dit diagram, maar worden uitgewerkt in andere UML-diagrammen zoals het Activity- en State Machine diagram.

3.5 Package diagram



Het package diagram geeft een overzicht van de architecturale opbouw van de Snake-applicatie. De applicatie is opgesplitst in meerdere packages, waarbij iedere package een duidelijke verantwoordelijkheid heeft. Door deze scheiding blijft de code overzichtelijk, onderhoudbaar en uitbreidbaar.

Entry

De Entry package bevat het startpunt van de applicatie (Program). Deze klasse is verantwoordelijk voor:

- het initialiseren van de applicatie
- het instellen van console-configuratie
- het starten van de UI via het hoofdmenu

De Entry-laag bevat geen game-logica en fungeert uitsluitend als ingang van het systeem.

UI

De UI package bevat de klasse UI en is verantwoordelijk voor:

- het tonen van menu's
- het verwerken van gebruikersinvoer
- het aansturen van game-flows (start spel, difficulty selecteren, game over menu)

De UI bevat geen directe game-logica, maar stuurt acties aan via het Command pattern en maakt objecten aan via de Factory. Hierdoor blijft de UI losgekoppeld van concrete implementaties.

Pattern (Command)

De Pattern (Command) package implementeert het Command pattern en bevat:

- ICommand (interface)
- ActionCommand (concrete command)

Elke gebruikersactie wordt verpakt in een command-object.

De UI voert alleen Execute() uit op een command en hoeft niet te weten wat de actie precies doet. Dit sluit aan bij het Open/Closed Principle, omdat nieuwe menu-acties kunnen worden toegevoegd zonder bestaande UI-code aan te passen.

Factories

De Factories package bevat de klasse GameFactory.

Deze factory is verantwoordelijk voor:

- het aanmaken van Snake-objecten
- het configureren van deze objecten op basis van de gekozen difficulty

Door object creatie te centraliseren in een factory wordt de UI ontkoppeld van de game-logica. Dit voorkomt sterke koppeling en maakt toekomstige uitbreidingen eenvoudiger.

Game logic

De Game logic package bevat de kern van het spel, met als belangrijkste klasse Snake.

Deze package is verantwoordelijk voor:

- Spelregels
- beweging van de slang
- score- en tijdsberekening
- collision detection en game-over logica

De game-logica is onafhankelijk van de UI en communiceert voor output uitsluitend via de ConsoleFacade.

Console Facade

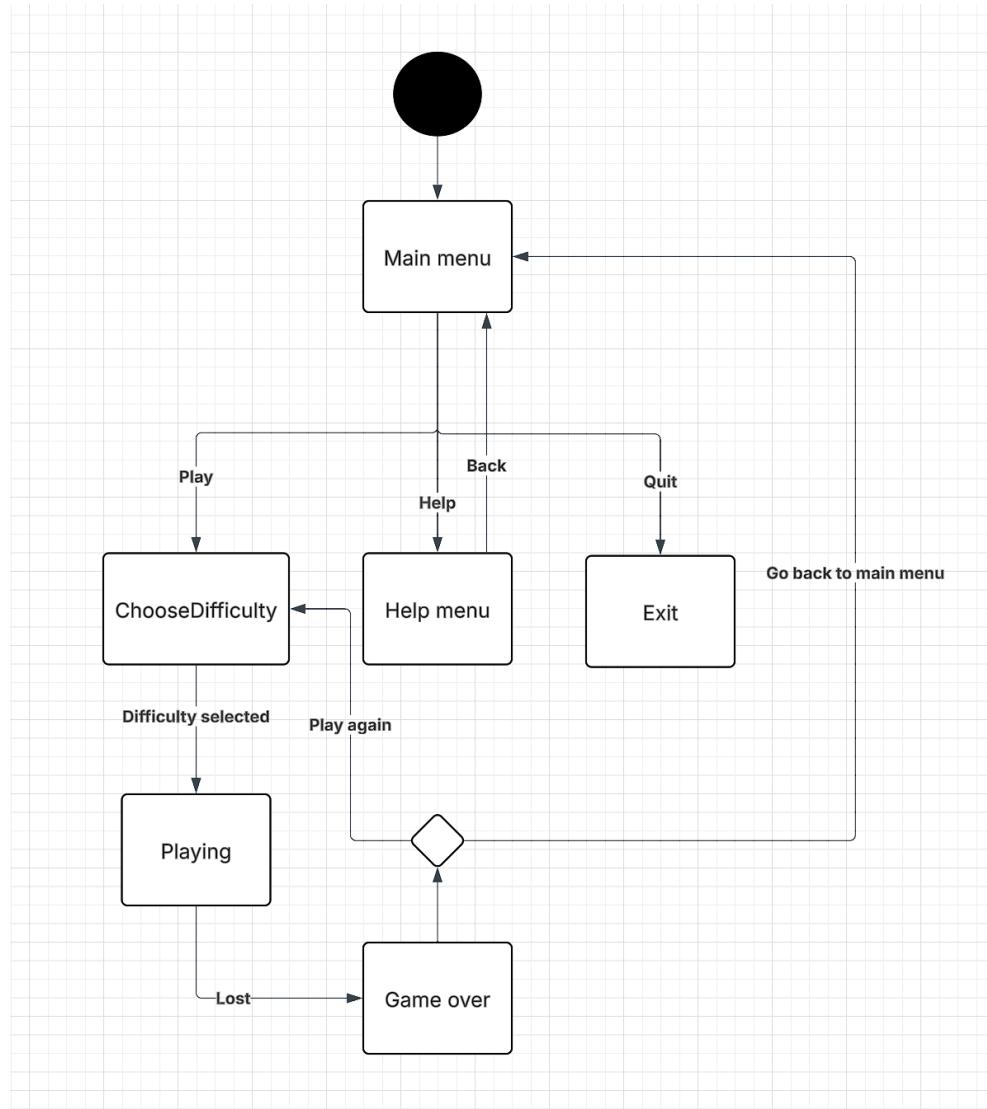
De Console Facade package bevat de `ConsoleFacade` klasse.

Deze facade:

- verbergt console-specifieke details
- regelt thread-safe output via locking
- biedt een eenvoudige interface voor schrijven naar de console

Hiermee wordt het Facade pattern toegepast en blijft de game-logica vrij van low-level console-implementatie.

3.6 State diagram



Het State Machine diagram beschrijft de verschillende toestanden waarin de Snake-game zich kan bevinden en de overgangen tussen deze toestanden. Het diagram geeft inzicht in de globale spelstroom en hoe het systeem reageert op gebruikers keuzes en spel omstandigheden.

Het spel start in de Main menu-toestand. Vanuit hier kan de speler kiezen om het spel te starten, de helpfunctie te openen of het spel af te sluiten. Wanneer de speler kiest voor "Play", gaat het systeem naar de toestand ChooseDifficulty, waar een moeilijkheidsgraad wordt geselecteerd. Na het selecteren van een moeilijkheid start het spel en komt het systeem in de toestand Playing.

Tijdens de Playing-toestand is de game actief en bestuurt de speler de slang. Wanneer de speler verliest, bijvoorbeeld door een botsing met de muur of met de eigen slang, gaat het spel over naar de toestand Game over. Vanuit deze toestand kan de speler ervoor kiezen

om opnieuw te spelen, wat leidt tot het opnieuw kiezen van een moeilijkheidsgraad, of om terug te keren naar het hoofdmenu.

De Help menu-toestand kan vanuit het hoofdmenu worden bereikt en biedt informatie over de spelbesturing. Vanuit dit menu kan de speler altijd terugkeren naar het hoofdmenu.

Wanneer de speler kiest voor “Quit”, gaat het systeem naar de Exit-toestand en wordt het spel beëindigd.

Dit diagram richt zich op de globale toestanden van het systeem en niet op de interne spel logica. Gedetailleerde acties, zoals het bewegen van de slang en het bijhouden van score en tijd, worden uitgewerkt in andere UML-diagrammen zoals het Activity- en Sequence diagram.

4. Solid principles

Single Responsibility Principle (SRP)

Het Single Responsibility Principle stelt dat een klasse slechts één verantwoordelijkheid moet hebben. In dit project is dit principe zichtbaar in de scheiding tussen verschillende klassen. De Snake-klasse is verantwoordelijk voor de spel logica, zoals bewegen, botsingen en scoreberekening. De UI-klasse houdt zich bezig met het weergeven van menu's en het verwerken van gebruikersinvoer. Daarnaast is de Console Facade verantwoordelijk voor console-output en synchronisatie. Door deze verantwoordelijkheden te scheiden, blijft elke klasse overzichtelijk en makkelijker aan te passen.

Open/Closed Principle (OCP)

Het Open/Closed Principle houdt in dat klassen open moeten zijn voor uitbreiding, maar gesloten voor wijziging. Dit principe is toegepast door gebruik te maken van het Command pattern. Het ICommand-interface en de ActionCommand-klasse maken het mogelijk om nieuwe menu-acties toe te voegen zonder bestaande code in de UI-logica aan te passen. Nieuwe functionaliteit kan worden toegevoegd door een nieuwe command te implementeren.

Interface Segregation Principle (ISP)

Het Interface Segregation Principle stelt dat interfaces klein en specifiek moeten zijn. In dit project is dit zichtbaar in het ICommand-interface, dat slechts één methode (Execute) bevat. Hierdoor zijn klassen die dit interface implementeren niet verplicht om onnodige methodes te bevatten.

Liskov Substitution Principle (LSP)

Het Liskov Substitution Principle stelt dat objecten van een subtype zonder problemen gebruikt moeten kunnen worden in plaats van hun basistype. Dit betekent dat een implementatie het verwachte gedrag van de abstractie moet behouden.

In dit project is dit principe toegepast via het ICommand-interface. De UI-klasse werkt met dit interface en is daardoor onafhankelijk van de concrete implementatie. Klassen zoals ActionCommand kunnen probleemloos worden vervangen of uitgebreid zolang zij het ICommand-contract volgen.

Dependency Inversion Principle (DIP)

Het Dependency Inversion Principle houdt in dat hogere niveaus niet afhankelijk zijn van concrete implementaties, maar van abstracties. In dit project is dit principe toegepast doordat de UI werkt met het ICommand-interface in plaats van concrete implementaties. Hierdoor is de UI minder afhankelijk van specifieke acties en kan de onderliggende logica eenvoudiger worden aangepast of uitgebreid.

Invloed van SOLID op het class diagram

In het class diagram is rekening gehouden met de SOLID-principes door verantwoordelijkheden te scheiden over meerdere klassen en gebruik te maken van interfaces en design patterns. De aanwezigheid van het ICommand-interface en de ActionCommand-klasse laat zien hoe abstracties worden gebruikt om afhankelijkheden te verminderen. Daarnaast zorgt de scheiding tussen UI, Snake, GameFactory en ConsoleFacade ervoor dat elke klasse een duidelijke rol heeft binnen het systeem.

Door SOLID toe te passen blijft het ontwerp overzichtelijk en flexibel, wat het onderhoud en eventuele uitbreidingen van de applicatie vereenvoudigt.

5. Design smells

Design smells zijn aanwijzingen in de structuur van software die kunnen wijzen op een minder goed ontwerp. Ze betekenen niet direct dat de code fout is, maar geven aan dat de code mogelijk moeilijker te onderhouden, uit te breiden of te testen is. Design smells ontstaan vaak tijdens het ontwikkelen van een applicatie en kunnen later leiden tot technische problemen.

Veel voorkomende design smells zijn God Class, waarbij één klasse te veel verantwoordelijkheden heeft, Tight Coupling, waarbij klassen sterk van elkaar afhankelijk zijn, en Duplicated Code, waarbij dezelfde logica op meerdere plekken voorkomt. Ook Long Methods en Feature Envy komen vaak voor in kleinere projecten.

In dit project zijn enkele lichte design smells herkenbaar. De UI-klasse bevat bijvoorbeeld veel verantwoordelijkheden, zoals het tekenen van menu's, het afhandelen van gebruikersinvoer en het starten van het spel. Dit kan worden gezien als een vorm van een God Class, omdat meerdere taken in één klasse zijn samengebracht. Daarnaast is er sprake van Tight Coupling tussen de UI en de Snake-klasse, omdat de UI direct de spel-logica aanroept. Voor een klein console project is dit acceptabel, maar bij een grotere applicatie zou dit kunnen worden verbeterd door verantwoordelijkheden verder te scheiden.

6. Design patterns

7.1 Creational pattern

Wat is een Creational pattern?

Een Creational design pattern richt zich op hoe objecten worden aangemaakt. In plaats van dat objecten overal in de code direct worden gemaakt, zorgt een creational pattern ervoor dat object-creatie op 1 plek wordt gemaakt voor het hele project.

Factory Method (Creational Pattern)

In dit project wordt het Factory Method pattern toegepast.

Definitie (bron: Refactoring.Guru): “*Factory Method is a creational design pattern that provides an interface for creating objects, but allows subclasses or specialized logic to alter the type of objects that will be created.*”

Het probleem (toegepast op mijn project)

In de game wordt op basis van de gekozen difficulty (Easy, Medium, Hard) een Snake-object aangemaakt met een andere snelheid.

Zonder een Factory zou dit betekenen dat:

- de UI zelf een Snake moet aanmaken
- de UI moet weten hoe difficulty wordt ingesteld
- wijzigingen aan difficulty-logica meerdere plekken in de code raken

Dit zorgt voor sterke koppeling tussen UI-logica en game-logica, wat volgens Refactoring.Guru precies het probleem is dat Factory Method probeert op te lossen.

Hoe pas ik dit toe in mijn code?

Factory (GameFactory)

```
internal static class GameFactory
{
    // Creational pattern: Factory
    internal static UI CreateUI()
    {
        return new UI();
    }

    internal static Snake CreateSnake(string difficulty)
    {
        Snake snake = new Snake();
        snake.SetDifficulty(difficulty);
        return snake;
    }
}
```

Deze factory-methode:

- maakt het Snake-object aan
- configureert het object (difficulty -> speed)
- geeft het object terug aan de cliënt code

Dit sluit aan bij Refactoring.Guru's beschrijving dat: "*The Factory Method pattern suggests that you replace direct object construction calls (using the new operator) with calls to a special factory method.*"

Product (Snake)

```
public void SetDifficulty(string difficulty)
{
    switch (difficulty)
    {
        case "Easy":
            speed = 400;
            break;
        case "Medium":
            speed = 200;
            break;
        case "Hard":
            speed = 300;
            break;
        default:
            speed = 200;
            break;
    }
}
```

De Snake-klasse bevat de logica die hoort bij het product zelf, maar niet de verantwoordelijkheid om te beslissen wanneer of waar hij wordt aangemaakt.

Client code (UI)

```
Snake snake = GameFactory.CreateSnake(Difficulty);
snake.SnakeLoop(Difficulty);
```

De UI:

- vraagt simpelweg om een Snake
- weet niets over de interne configuratie
- blijft werken, ook als de creatie logica verandert

7.2 Behavioural pattern

Wat is een Behavioural pattern?

Een Behavioural pattern richt zich op hoe objecten met elkaar communiceren en hoe verantwoordelijkheden voor acties worden verdeeld. In plaats van dat één klasse alles afhandelt, zorgen behavioural patterns ervoor dat gedrag wordt losgekoppeld en flexibel kan worden aangepast.

Volgens Refactoring.Guru richten behavioural patterns zich op:

- communicatie tussen objecten
- het loskoppelen van de aanroeper van een actie en de uitvoerder van die actie
- het flexibeler maken van gedrag zonder bestaande code te breken

Definitie (bron: Refactoring.Guru): “*Command is a behavioral design pattern that turns a request into a stand-alone object that contains all information about the request.*”

Het Command pattern verpakt een actie (request) in een object. Hierdoor:

- weet de aanroeper niet wat er precies gebeurt
- kan de actie worden uitgesteld, vervangen of uitgebred
- ontstaat er minder koppeling tussen UI en logica

Het probleem (toegepast op mijn project)

In mijn Snake-game worden menu-keuzes gemaakt zoals:

- Play
- Help
- Easy / Medium / Hard
- MainMenu

Zonder het Command pattern zou de UI deze keuzes direct afhandelen met een grote switch-statement. Hierdoor:

- wordt de UI sterk gekoppeld aan de game-logica
- moet bestaande code worden aangepast bij elke nieuwe menu-optie
- ontstaat moeilijk onderhoudbare code

Dit is vergelijkbaar met het probleem dat Refactoring.Guru beschrijft bij GUI-knoppen die allemaal verschillende acties uitvoeren.

De oplossing: Command pattern

Het Command pattern stelt voor om:

- elke actie te verpakken in een command-object
- de UI alleen het command te laten uitvoeren
- de logica van de actie los te koppelen van de UI

De UI hoeft hierdoor niet te weten wat een actie doet, alleen dat er een command uitgevoerd moet worden.

Hoe pas ik dit toe in mijn code?

Sender (Invoker): UI

```
public void Exit(string selectedOption)
{
    Console.Clear();

    var commands = BuildCommandMap();

    if (commands.TryGetValue(selectedOption, out ICommand? command))
    {
        command.Execute();
    }
}
```

De UI:

- start de actie
- maar voert de logica niet zelf uit
- roept alleen Execute() aan op een command

Volgens Refactoring.Guru:

“The sender triggers that command instead of sending the request directly to the receiver.”

Command interface

```
4 references
internal interface ICommand
{
    2 references
    void Execute();
}
```

Deze interface komt overeen met Refactoring.Guru's beschrijving:

“The Command interface usually declares just a single method for executing the command.”

Concrete Command

```
internal class ActionCommand : ICommand
{
    2 references
    private readonly Action _action;

    7 references
    public ActionCommand(Action action)
    {
        _action = action;
    }

    2 references
    public void Execute()
    {
        _action();
    }
}
```

Deze Klasse:

- verpakt een actie (Action)
- voert die actie uit wanneer Execute() wordt aangeroepen
- bevat alle informatie die nodig is om de request uit te voeren

Cliënt: het koppelen van opties aan commands

```
private Dictionary<string, ICommand> BuildCommandMap()
{
    return new Dictionary<string, ICommand>
    {
        { "Play", new ActionCommand(() => ChooseDifficulty()) },
        { "HelpMenu", new ActionCommand(() => HelpMenu()) },
        { "Quit", new ActionCommand(() => Environment.Exit(0)) },

        { "Easy", new ActionCommand(() => GamePlayMenu("Easy")) },
        { "Medium", new ActionCommand(() => GamePlayMenu("Medium")) },
        { "Hard", new ActionCommand(() => GamePlayMenu("Hard")) },

        { "MainMenu", new ActionCommand(() => MainMenu()) }
    };
}
```

Hier worden:

- commands aangemaakt
- gekoppeld aan concrete acties
- verbonden met menu-opties

Dit komt overeen met Refactoring.Guru's beschrijving van de client, die commands configureert en aan de sender koppelt.

7.3 Concurrency pattern

Wat is concurrency pattern?

Concurrency betekent dat meerdere stukken werk “tegelijk” kunnen draaien (bijvoorbeeld op verschillende threads), zodat één taak niet de hele applicatie blokkeert. In mijn Snake-project draait de game door terwijl een timer parallel de tijd bijhoudt en op het scherm schrijft.

Task Parallel Library (TPL)

Volgens Microsoft is de Task Parallel Library (TPL) een set API's in System.Threading en System.Threading.Tasks die het makkelijker maken om parallelisme en concurrentie toe te voegen aan applicaties. De TPL regelt onder andere thread scheduling via de ThreadPool en andere “low-level details”, zodat je kunt focussen op het werk van je programma.

Hoe pas ik dit toe in mijn code?

Parallel uitvoeren van de timer met Task.Run

In SnakeLoop start ik de timer in een aparte Task:

```
public void SnakeLoop(string Difficulty)
{
    //Reset start position
    pos_x = 15;
    pos_y = 15;

    // Start thread for timer
    Task.Run(() => Timer());
    Thread.Sleep(100);
    // Generate initial apple
    GenerateApple();

    while (alive)
    {
```

Task.Run “queued” werk om te draaien op de ThreadPool en geeft een Task terug die dat werk representeert.

Hierdoor loopt de timer parallel aan de main game loop, waardoor de game niet elke seconde hoeft te wachten om de tijd te updaten.

Synchronisatie met lock (voorkomen van race conditions)

Omdat de timer-task en de game-loop beide naar de console kunnen schrijven, is synchronisatie nodig om te voorkomen dat output door elkaar heen loopt.

In C# zorgt de lock statement ervoor dat maximaal één thread tegelijk de code in het lock-blok uitvoert . Andere threads wachten tot het lock is vrijgegeven.

In mijn project gebruik ik dit om console updates thread-safe te maken, zodat de timer en snake rendering niet tegelijk de cursorpositie en output kunnen veranderen.

Wat is Timer() in mijn project?

Timer() is een methode die periodiek (elke seconde) de tijd bijwerkt en dit op het scherm toont. In mijn Snake-game is dit losgekoppeld van de main game-loop, zodat:

- de game-loop kan blijven draaien (input, beweging, collisions)
- terwijl de tijd onafhankelijk wordt bijgehouden

De methode doet dit door:

- seconds++ en bij 60 seconden -> minutes++
- console-output updaten (tijd tonen)
- Thread.Sleep(1000) om ongeveer 1 seconde te wachten
- zichzelf opnieuw aanroepen zolang alive == true (dus zolang het spel loopt)

```
public void Timer()
{
    try
    {
        seconds++;
        if (seconds == 60)
        {
            minutes++;
            seconds = 0;
            lock (consoleLock)
            {
                Console.SetCursorPosition(10, 2);
                Console.WriteLine("                 ");
            }
        }
        lock (consoleLock)
        {
            console.WriteLineAt(10, 2, $"{minutes}:{seconds}");
        }
        Thread.Sleep(1000);
        if (alive) { Timer(); }
    }
    catch (Exception ex)
    {
        Console.WriteLine($"An error occurred: {ex.Message} restarting in 5 seconds...");
        Thread.Sleep(5000);
        // If an error occurs this restart the game
        System.Diagnostics.Process.Start(System.Reflection.Assembly.GetExecutingAssembly().Location);
        Environment.Exit(0);
    }
}
```

7.4 Structural pattern

Wat is een Structural pattern?

Een Structural design pattern beschrijft hoe objecten en klassen worden samengesteld tot grotere structuren, terwijl deze structuren overzichtelijk, flexibel en onderhoudbaar blijven.

Volgens Refactoring.Guru richten structural patterns zich op het vereenvoudigen van relaties tussen onderdelen van een systeem en het verbergen van complexiteit.

Definitie (bron: Refactoring.Guru): “*Facade is a structural design pattern that provides a simplified interface to a library, a framework, or any other complex set of classes.*”

Het Facade pattern verbergt de complexiteit van een subsystem achter een eenvoudige interface. De cliënt werkt alleen met de facade en hoeft de interne details van het subsysteem niet te kennen.

Het probleem (toegepast op mijn project)

In mijn Snake-game wordt veel gebruikgemaakt van de Console-API, zoals:

- Console.SetCursorPosition
- Console.Write / Console.WriteLine
- synchronisatie met lock om race conditions te voorkomen

Zonder een facade zouden deze details:

- verspreid staan door meerdere klassen (Snake, UI)
- de game-logica sterk koppelen aan low-level console-implementatie
- de code moeilijker leesbaar en onderhoudbaar maken

Dit sluit aan bij het probleem dat Refactoring.Guru beschrijft: “*the business logic of your classes would become tightly coupled to the implementation details of 3rd-party classes, making it hard to comprehend and maintain.*”

De oplossing: Facade pattern

Het Facade pattern stelt voor om:

- een aparte facade-klasse te maken
- die een eenvoudige interface aanbiedt
- en intern alle complexe subsystem-logica afhandelt

De facade biedt alleen de functionaliteit die de cliënt nodig heeft en verbergt alle overige details.

Hoe pas ik dit toe in mijn code?

Facade

```
internal class ConsoleFacade
{
    4 references
    private readonly object _consoleLock;

    1 reference
    public ConsoleFacade(object consoleLock)
    {
        _consoleLock = consoleLock;
    }

    3 references
    public void WriteAt(int x, int y, string text)
    {
        lock (_consoleLock)
        {
            Console.SetCursorPosition(x, y);
            Console.Write(text);
        }
    }

    2 references
    public void WriteLineAt(int x, int y, string text)
    {
        lock (_consoleLock)
        {
            Console.SetCursorPosition(x, y);
            Console.WriteLine(text);
        }
    }

    0 references
    public void ClearAt(int x, int y, int length)
    {
        lock (_consoleLock)
        {
            Console.SetCursorPosition(x, y);
            Console.Write(new string(' ', length));
        }
    }
}
```

Deze klasse:

- verbergt console-specifieke details
- Regelt thread-safety intern
- biedt een simpele API aan de rest van het programma

Client: Snake

```
Console.SetCursorPosition(10, 2);
Console.WriteLine(" ");
```

De Snake-klasse:

- gebruikt alleen de facade
- hoeft niets te weten over Console.SetCursorPosition
- hoeft geen synchronisatie-logica te bevatten

Volgens Refactoring.Guru: “*The Client uses the facade instead of calling the subsystem objects directly.*”

Structuur van het pattern in mijn project

Rol (Facade pattern)	Klasse in mijn project
Facade	ConsoleFacade
Subsystem	Console API + locking
Client	Snake en UI

De subsystem-klassen zijn zich niet bewust van de facade, en de client communiceert uitsluitend via de facade.

7. Bronnenlijst

UML Activity Diagram Tutorial. (2026, February 5). Lucidchart.

<https://www.lucidchart.com/pages/tutorial/uml-activity-diagram>

UML Sequence Diagram Tutorial. (2026, February 5). Lucidchart.

<https://www.lucidchart.com/pages/uml-sequence-diagram>

UML Use Case Diagram Tutorial. (2026, February 9). Lucidchart.

<https://www.lucidchart.com/pages/tutorial/uml-use-case-diagram>

What is Package Diagram? (n.d.).

<https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-package-diagram/>

State Machine Diagram Tutorial. (2026, February 5). Lucidchart.

<https://www.lucidchart.com/pages/tutorial/uml-state-machine-diagram>

Refactoring.Guru. (2026, January 1). *Factory method*.

<https://refactoring.guru/design-patterns/factory-method>

Adegeo. (n.d.). *Task Parallel Library (TPL) - .NET*. Microsoft Learn.

<https://learn.microsoft.com/en-us/dotnet/standard/parallel-programming/task-parallel-library-tpl>

Refactoring.Guru. (2026b, January 1). *Facade*.

<https://refactoring.guru/design-patterns/facade>

What is Unified Modeling Language (UML)? (n.d.).

<https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-uml/>