**Applied Design Patterns**

***Gemaakt door Team Basker 19/05/2024***

# Inhoudsopgave

Inhoud

[Inhoudsopgave 2](#_Toc30707)

[Wireframes 3](#_Toc30708)

[User Stories 4](#_Toc30709)

[Diagrammen 5](#_Toc30710)

[Creational Patterns 10](#_Toc30711)

[Behavioral Patterns 14](#_Toc30712)

[Concurrency Pattern 21](#_Toc30713)

[Structural Patterns 27](#_Toc30714)

# 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.

**Paddles:** De linker paddle word bestuurd door speler 1 met “W” en “S”. De rechter paddle word bestuurd door speler 2 met Pijltje omhoog en Pijltje omlaag.

**Ball:** De ball word via de muren boven en beneden gekaatst en de spelers moeten voorkomen dat de bal hun kant van de muur raakt.

**Score:** Als de bal de muur raakt van de rechter speler, dan krijgt speler 1 één punt. Als een speler 10 punten heeft, Word het spel beëindigt.

**Veld:** Het veld is 511 pixels breed bij 294 pixels lang.

A black screen with white text

Description automatically generated

# Huisstijl

Wij gebruiken als huisstijl een zwart-wit thema, wij hebben hiervoor gekozen omdat het retro lijkt en PONG is een retro game. Dit zijn de kleuren:

zwart

rgba(12,12,12,255) #0c0c0c

wit

rgba(204,204,204,255) #cccccc

# 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.

# UML

Waarom gebruiken wij UML’s en wat zijn UML’s? Wij gebruiken UML’s om een beter zicht te krijgen van de applicatie die gemaakt moet worden. Dit geeft ons (en de klant) een beter zicht om het idee te realiseren. UML is een gestandaardiseerde modelleertaal die wordt gebruikt in software-engineering om software systemen te visualiseren, specificeren, construeren en documenteren. Het biedt grafische technieken om visuele modellen van softwaresystemen te maken.

Bron: <https://www.gain.nl/kennisbank/wat-is-uml/>

Hieronder staan de diagrammen die wij gebruiken voor onze PONG applicatie:

Activity Diagram

Sequence Diagram

Klasse Diagram

Use Case Diagram

# Activity Diagram

Wat is een Activity Diagram? Een activity diagram is een soort diagram dat wordt gebruikt in softwareontwikkeling om de stroom van activiteiten of stappen in een proces weer te geven. Het helpt om te visualiseren hoe een systeem werkt door de volgorde van acties en beslissingen te tonen.

**Activity diagram:** Deze diagram laat zien hoe een game word gestart.

Dit activiteitendiagram beschrijft de stroom van een Pong-spel. Hier is een korte uitleg:

1. Initialiseer Spel: Het spel wordt gestart.

2. Toon Speelveld: Het speelveld wordt weergegeven.

3. Plaats Rackets en Bal: De rackets en bal worden op hun startposities geplaatst.

4. Beweeg Bal: De bal wordt bewogen.

- Bal raakt Racket?: Controleert of de bal een racket raakt. Zo ja, verander de richting van de bal.

- Bal raakt Rand?: Controleert of de bal de rand raakt.

- Score Punten: Punt wordt toegekend.

- Punten Behaald?: Controleert of de vereiste punten zijn behaald om het spel te eindigen. Zo ja, eindig het spel.

5. Beweeg Rackets: De speler beweegt de rackets.

6. Spel Actief: Het spel blijft actief zolang de eindconditie niet is bereikt.

Deze diagram toont de stappen en beslissingen die tijdens het spel worden genomen.

A diagram of a computer

Description automatically generated with medium confidence

# Sequence Diagram

Wat is een Sequence Diagram? Een sequence diagram is een type diagram dat wordt gebruikt in software-engineering om de interacties tussen objecten in een systeem gedurende een bepaalde periode te visualiseren. Het toont hoe objecten samenwerken door middel van uitwisselingen van berichten en de volgorde waarin deze berichten worden verzonden.

**Sequence Diagram:**

Deze sequence diagram toont de interactie tussen de speler en verschillende onderdelen van een Pong-spel:

1. Speler start het programma.

2. Program maakt een game-instantie aan.

3. Game initialiseert het speelveld, de rackets en de bal.

4. Spel begint:

- Field tekent het veld.

- Racket tekent de linker en rechter rackets.

- Ball tekent de bal.

5. Speler beweegt de rackets:

- Racket beweegt omhoog of omlaag.

- Ball beweegt en controleert botsingen en grenzen.

6. Program toont de score aan de speler.

Het diagram visualiseert de volgorde van deze stappen en de communicatie tussen de verschillende componenten van het spel.

A screenshot of a computer

Description automatically generated

# Klasse Diagram

Wat is een klasse diagram? Een klasse diagram toont de structuur van een systeem in de softwareontwikkeling. Het bevat:

- Klassen: Rechthoeken met naam, attributen (eigenschappen), en methoden (functies).

- Relaties: Lijnen die verbanden tussen klassen aangeven, zoals:

- Associatie: Basisrelatie tussen klassen.

- Aggregatie: 'Heeft-een' relatie.

- Compositie: Sterkere vorm van aggregatie.

- Inheritantie: Overerving van eigenschappen en methoden.

Klasse diagrammen helpen bij het plannen en visualiseren van de objectgerichte structuur van een systeem.

**Klasse Diagram:**

Deze klasse diagram beschrijft een Pong-spel:

1. Program: Start het spel met `Main()`.

2. Game: Beheert het spel met `Start()`, `HandleInput()`, en `EndGame()`.

3. Field: Singleton klasse die het speelveld tekent met `Draw()`.

4. Racket: Beweegt en tekent de rackets.

5. Ball: Beweegt, tekent, en houdt bij of het een racket raakt. Beheert observators met `AddObserver()`, `RemoveObserver()`, en `NotifyObservers()`.

6. IObserver: Interface voor objecten die de bal observeren, met de methode `Update(ball)`.

Belangrijkste relaties:

- Game bevat Field, Racket, en Ball.

- Field is een singleton.

- Ball gebruikt het Observer patroon met IObserver.

A diagram of a computer code

Description automatically generated with medium confidence

# Use Case Diagram

Wat is een Use Case Diagram? Een use case diagram toont de interacties tussen gebruikers (actoren) en een systeem, met de doelen (use cases) die gebruikers willen bereiken. Het bevat:

- Actoren: Gebruikers of systemen die interageren.

- Use cases: Specifieke functies of doelen.

- Associaties: Lijnen die de interacties aangeven.

- Systeemgrens: De rechthoek die het systeem afbakent.

Het helpt bij het definiëren van systeemvereisten en gebruikersinteracties.

**Use Case Diagram:**

Dit use case diagram toont de interacties tussen de speler en het Pong-spel. Het bevat de volgende elementen:

- Speler: De gebruiker die het spel speelt, weergegeven door het stokfiguur.

- PongGame: Het systeem dat het spel beheert, weergegeven door de rechthoek.

De use cases (ovalen) die de speler kan uitvoeren zijn:

1. Start Spel: De speler kan het spel starten.

2. Beweeg Racket: De speler kan het racket bewegen.

3. Score Weergave: De speler kan de huidige score bekijken.

4. Pauzeer Spel: De speler kan het spel pauzeren.

5. Reset Spel: De speler kan het spel resetten.

6. Einde Spel: De speler kan het spel beëindigen.

Deze diagram helpt bij het visualiseren van de functies die beschikbaar zijn voor de speler en hoe de speler met het systeem (PongGame) interageert.

A diagram of a person's game

Description automatically generated

# State Diagram

Een state diagram toont de verschillende toestanden (staten) van een object en hoe het object van de ene naar de andere toestand overgaat. Het beschrijft het gedrag van een systeem door de verschillende toestanden en de gebeurtenissen die deze overgangen veroorzaken.

Belangrijke elementen:

- Staten: Verschillende toestanden waarin een object kan verkeren, weergegeven als rechthoeken met afgeronde hoeken.

- Overgangen: Lijnen met pijlen die de verandering van de ene toestand naar de andere aangeven.

- Gebeurtenissen: Triggers die een overgang veroorzaken.

- Starttoestand: De initiële toestand van het object, weergegeven door een zwarte stip.

- Eindtoestand: De toestand waarin het object zich bevindt na het voltooien van alle processen, weergegeven door een zwarte stip met een cirkel eromheen.

State diagrams helpen bij het modelleren van de dynamische aspecten van een systeem, zoals de reacties op gebeurtenissen.

**State Diagram:**

Dit state diagram toont de verschillende toestanden van een object:

1. Idle: Starttoestand, object is in rust.

2. Moving: Object beweegt.

3. Collision: Object heeft een botsing.

4. Goal: Object bereikt een doel.

Overgangen:

- Idle → Moving

- Moving → Collision of Goal

- Collision → Moving

- Goal → Idle

Het diagram toont hoe het object tussen deze toestanden beweegt.

A diagram of a moving process

Description automatically generated

# Solid Principes

Bij het ontwerpen van onze classdiagrammen hebben we de SOLID-principes toegepast. Deze principes helpen om de code beter gestructureerd, onderhoudbaar en uitbreidbaar te maken. Hieronder volgt een korte uitleg van hoe elk SOLID-principe invloed heeft op de klassen in onze diagrammen:

**Single Responsibility Principle (SRP):** Elke klasse heeft slechts één verantwoordelijkheid of reden om te veranderen. Dit betekent dat een klasse slechts één taak of functie heeft. In de diagrammen zie je dat klassen eenvoudig blijven en slechts één aspect van de functionaliteit afhandelen.

**Open/Closed Principle (OCP):** Klassen zijn open voor uitbreiding maar gesloten voor modificatie. Dit betekent dat de functionaliteit van een klasse kan worden uitgebreid zonder de bestaande code te wijzigen. Dit wordt in de diagrammen bereikt door gebruik te maken van abstracte klassen of interfaces die kunnen worden uitgebreid of geïmplementeerd door andere klassen.

**Liskov Substitution Principle (LSP):** Subklassen moeten kunnen worden gebruikt als vervanging voor hun basisklasse zonder dat de correctheid van het programma in gevaar komt. De erfrelaties in de diagrammen zijn zodanig opgezet dat objecten van een subklasse overal waar objecten van de basisklasse worden gebruikt, kunnen worden toegepast zonder problemen.

**Interface Segregation Principle (ISP):** Klanten mogen niet gedwongen worden om interfaces te implementeren die ze niet gebruiken. Dit betekent dat er meerdere, specifieke interfaces moeten zijn in plaats van één algemene interface. In de diagrammen worden interfaces opgesplitst in kleinere, gerichtere sets van methoden, waardoor klassen alleen de methoden implementeren die ze daadwerkelijk nodig hebben.

**Dependency Inversion Principle (DIP):** Hoge-level modules mogen niet afhankelijk zijn van lage-level modules. Beide moeten afhankelijk zijn van abstracties. Dit wordt in de diagrammen bereikt door het gebruik van interfaces of abstracte klassen, zodat afhankelijkheden worden geïnverteerd en hoge-level modules kunnen werken met abstracties in plaats van concrete implementaties.

Door deze SOLID-principes toe te passen in onze classdiagrammen, wordt het ontwerp van het systeem modulairder, beter onderhoudbaar en uitbreidbaar. De klassen en hun relaties zijn zodanig gestructureerd dat ze gemakkelijker te begrijpen, testen en aanpassen zijn zonder grote risico's op fouten.

# Design Smells

Design smells zijn aanwijzingen dat er iets mis kan zijn met hoe je software is ontworpen. Ze zijn niet meteen bugs, maar wijzen erop dat je code later moeilijker te onderhouden en te begrijpen kan worden. Het zijn een soort van waarschuwingen dat je je ontwerp beter kan maken, zodat je later minder problemen hebt.

In onze Pong applicatie zorgen we ervoor dat elke klasse maar één taak heeft. Zo wordt het niet te ingewikkeld. Lange stukken code maken we korter en overzichtelijker door ze op te breken in kleinere stukjes. Dat maakt het makkelijker om te begrijpen en aan te passen.

We proberen ook te voorkomen dat klassen te veel aan elkaar vastzitten. Als ze te veel van elkaar afhankelijk zijn, wordt het lastiger om dingen te veranderen. Daarom proberen we de verbindingen tussen klassen zo klein mogelijk te houden.

Om alles netjes te houden, doen we regelmatig een soort controle op elkaars code. Zo zorgen we ervoor dat alles goed blijft werken en dat we niet in de war raken. Ook schrijven we duidelijke opmerkingen bij de code, zodat iedereen begrijpt wat er gebeurt.

Dat is hoe we ervoor zorgen dat onze Pong applicatie goed blijft werken en makkelijk te begrijpen is.

# Design Patterns

**Ontwerppatronen**

Ontwerppatronen zijn herbruikbare oplossingen voor veelvoorkomende problemen in softwareontwerp. Hier zijn de hoofdtypen ontwerppatronen, eenvoudiger uitgelegd:

**Creational Patterns**

Deze patronen helpen bij het maken van objecten op een manier die geschikt is voor de situatie. Ze scheiden het maken van objecten van hun gebruik.

- Voorbeelden: Abstract Factory, Builder, Singleton, Prototype, Factory Method.

**Behavioral Patterns**

Gedragspatronen richten zich op hoe objecten met elkaar communiceren en verantwoordelijkheden toewijzen. Ze helpen bij het definiëren van hoe objecten samenwerken.

- Voorbeelden: Command, Observer, Visitor, Strategy, State, Iterator.

**Concurrency Patterns**

Concurrency-patronen helpen bij het ontwerpen van applicaties die meerdere taken tegelijkertijd uitvoeren en omgaan met de uitdagingen van gelijktijdige uitvoering.

- Voorbeelden: Leader/Followers, Half-Sync/Half-Async, Active Object, Monitor Object.

**Structural Patterns**

Structurele patronen richten zich op hoe klassen en objecten worden gevormd of samengesteld om grotere structuren te vormen. Ze zorgen ervoor dat veranderingen in één deel van het systeem niet het hele systeem beïnvloeden.

- Voorbeelden: Bridge, Composite, Decorator, Flyweight, Adapter, Proxy.

Elk van deze patronen speelt een cruciale rol bij het oplossen van specifieke ontwerpproblemen en het verbeteren van de flexibiliteit van de code.