

SDD Dierentuin Applicatie

Applied Design Patterns



Studenten: Daniel Pustjens en Sadek al Mousawi

Studie: AD Software Development

Vak: Applied Design Patterns WFSDAD.DP.23

School: Windesheim

Datum: 07-04-2025

Docenten: Jan Zuur en Tom Sievers

Inhoudsopgave

1. Inleiding	3
2. Designs	4
3. Kleurenpalet	9
4. Design Smells	12
5. UML	13
5.1 Activity diagram (Create Animal).....	14
5.2 Sequence diagram	16
5.3 Use case diagram.....	18
5.4 Class diagram	20
6. Solid Principles	23
7. Entity Relationship Diagram (ERD).....	26
8. Design Patterns.....	30
8.1 Creational pattern	30
8.2 Behavioral pattern	38
8.3 Concurrency Pattern	45
8.4 Structural Pattern.....	51
9. Bronnenlijst	57

1. Inleiding

In dit Software Design Document (SDD) wordt het ontwerp en de architectuur van de Dierentuin Applicatie uitgelegd. De applicatie is ontwikkeld als onderdeel van het vak Applied Design Patterns (ADP). Het doel van dit document is om te laten zien hoe de applicatie technisch is opgebouwd, welke ontwerpprincipes en tot slot welke patronen zijn gebruikt.

2. Designs

Dierentuin Homepagina

Dierentuin App Home Create Animal Create Stall Animals index Stall index About us

Welcome to the Zoo application created by Group 3

Zoo application Guideline:

Create animal/stall

Step 1: Create a stall by going to [stall](#).
Step 2: Create an animal by going to [animal](#) and assign it to the newly created stall.
Step 3: To view the stall and the assigned animal, go to [stall index](#) and click on the name of the stall.

Day/Night:

The application has a 10 minute day/night cycle which displays "animal is awake" or "animal is sleeping" based on the animal activity pattern.



Feeding animals:



Once an animal is hungry, it will display "Animal has not eaten". Click on the "Feed all animals" button to feed all animals.

It is currently Day ✨

Dierentuin App

Zoo application Designed
by team Groep 3

 [Github Sam](#)
 [Github Renas](#)

 [Github Daniel](#)
 [Github Sadek](#)

Na het opstarten komt de gebruiker op de homepagina van de dierentuinapp terecht. Hier wordt uitleg gegeven over het gebruik van de applicatie.

Belangrijke elementen:

- Dynamische dag-/nacht-uitleg afhankelijk van de systeemklok
- Uitleg over het aanmaken van dieren en stallen
- Toelichting op het voeden van dieren via de "Feed all animals"-knop
- Navigatielinks naar de dieren- en stallenpagina

Create/Edit Animal/Stall


[Dierentuin App](#) [Home](#) [Create Animal](#) [Create Stall](#) [Animals index](#) [Stall index](#) [About us](#)



Create Animal

Name	<input type="text"/>
Species	<input type="text"/>
Category	<input type="text"/>
Size	<input type="text" value="Choose size"/>
DietaryClass	<input type="text" value="Choose dietary class"/>
ActivityPattern	<input type="text" value="Choose activity pattern"/>
Prey	<input type="text"/>
Enclosure	<input type="text"/>
SpaceRequirement	<input type="text"/>
SecurityRequirement	<input type="text" value="Choose security requirement"/>
Stall	<input type="text" value="Lake weston"/>
<input type="button" value="Create"/>	
Back to list	

Dierentuin App

Zoo application Designed
by team Groep 3

 [Github Sam](#)
 [Github Renas](#)

 [Github Daniel](#)
 [Github Sadek](#)

De gebruiker komt via een formulier op een scherm om een dier of stal aan te maken of te bewerken.

Belangrijke elementen:

- Tekstvelden voor gegevens zoals naam, soort, grootte
- Dropdowns voor onder andere dieet, activiteit, beveiligingsniveau en stal
- Eénzelfde formulierstructuur wordt gebruikt voor zowel aanmaken als aanpassen

Indexpagina's voor Animals & Stalls

[Dierentuin App](#) [Home](#) [Create Animal](#) [Create Stall](#) [Animals index](#) [Stall index](#) [About us](#)

Animals

Create new animal

Size:

Dietary Class:

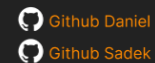
Activity Pattern:

Security Requirement:

Name	Species	Category	Size	DietaryClass	ActivityPattern	Prey	Enclosure	SpaceRequirement	SecurityRequirement	Name
Horacio	Elephant	Fish	Microscopic	Omnivore	Nocturnal	Fish	Fish Pond	1.20m2	Medium	Lake
Eryn	Frog	Reptile	Microscopic	Herbivore	Nocturnal	Fish	Fish Pond	1.20m2	Medium	Lake
Horacio	Elephant	Fish	Microscopic	Omnivore	Nocturnal	Fish	Fish Pond	1.20m2	Medium	Lake
Eryn	Frog	Reptile	Microscopic	Herbivore	Nocturnal	Fish	Fish Pond	1.20m2	Medium	Lake

Dierentuin App

Zoo application Designed
by team Groep 3



Een overzichtelijke tabel met filtermogelijkheden toont alle aangemaakte dieren of stallen.

Belangrijke elementen:

- Zoekveld en dropdownfilters voor verfijning
- Tabel met eigenschappen van dieren of stallen
- Knoppen om details te bekijken, te bewerken of te verwijderen



Details Animal/Stall



[Dierentuin App](#) [Home](#) [Create Animal](#) [Create Stall](#) [Animals index](#) [Stall index](#) [About us](#)

Details of Jacob	
Name	Jacob
Species	Elephant
Category	Mamalia
Size	VeryLarge
DietaryClass	Herbivore
ActivityPattern	Diurnal
Prey	None
Enclosure	Enclosure 1
SpaceRequirement	12.3
SecurityRequirement	Medium
Stall	Lake weston
Edit Back to list	

Dierentuin App

Zoo application Designed
by team Groep 3

 [Github Sam](#)
 [Github Renas](#)

 [Github Daniel](#)
 [Github Sadek](#)

Toont uitgebreide informatie over een geselecteerd dier of stal.

Belangrijke elementen:

- Alle eigenschappen worden overzichtelijk in een lijst getoond
- Voor een stal worden ook gekoppelde dieren weergegeven en een knop op dieren te voeren
- Alleen-weergave (read-only)

About Us pagina

[Dierentuin App](#) [Home](#) [Create Animal](#) [Create Stall](#) [Animals index](#) [Stall index](#) [About us](#)

About us

Opdracht:

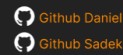
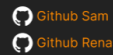
Dit is een virtuele dierentuin waarbij de gebruiker de dierentuin kan beheren. Het moet mogelijk zijn om dieren aan te maken, hokken voor de dieren en om de dieren in categorieën te verdelen voor het makkelijk opzoeken aan de hand van de categorieën. Daarnaast kun je als gebruiker acties doen zoals: nieuwe indeling maken, bestaande indeling afmaken, etenstijd, zonsopgang en zonsondergang triggeren, en checken of aan alle voorwaarden wordt voldaan.

Gemaakt door de studenten van projectgroep 3:

- Student: Renas Khalil (S1149006)
- Student: Sam Maijer (S1199941)
- Student: Daniel Pustjens (S1173932)
- Student: Sadek Al Mousawi (S1192769)

Dierentuin App

Zoo application Designed
by team Groep 3



De 'About Us'-pagina geeft achtergrondinformatie over het project en het ontwikkelteam.

Belangrijke elementen:

- Beschrijving van de opdracht
- Namen en studentnummers van het projectteam
- Beschrijving van de doelen van de applicatie

3. Kleurenpalet

In de **Dierentuin App** wordt gebruikgemaakt van een zorgvuldig samengesteld kleurenpalet om een duidelijke en toegankelijke gebruikersinterface te creëren. De kleuren zorgen voor visuele consistentie en helpen gebruikers om acties sneller te herkennen, zoals bewerken, aanmaken of verwijderen van dieren en stallen.

Algemeen kleurgebruik

De kleuren zijn gekozen op basis van functionaliteit, sfeer en toegankelijkheid. Elke kleur ondersteunt een specifiek doel in de applicatie.

Dag- en Nachtmodus

De applicatie wisselt tussen dag- en nachtmodus, en de achtergrondkleuren passen zich automatisch aan op basis van de tijd:

Modus Dag:

Achtergrondkleur: Lichtblauw

Hex-code: #87cefa



Modus Nacht:

Achtergrondkleur: Donkerblauw/grijs

Hex-code: #2a2a35



UI-Kleurenoverzicht

Edit-knop op indexpagina's

Achtergrondkleur: Primair blauw (bewerken)

Hex-code: #0d6efd



Delete-knop op indexpagina's

Achtergrondkleur: Rood (verwijderen)

Hex-code: #dc3545



Footer achtergrond

Achtergrondkleur: Donkergrijs

Hex-code: #212121



Hyperlinks in footer

Tekstkleur: Oranje (val op tegen donker)

Hex-code: #ff9800



Toepassing in de UI

- **Knoppen:** Actieknoppen zoals 'Edit' en 'Delete' zijn visueel direct herkenbaar door hun unieke kleur.
- **Thema's:** De achtergrond wisselt dynamisch tussen dag- en nachtmodus zonder herladen van de pagina, voor een natuurlijke gebruikerservaring.
- **Footer:** De donkere footer met oranje links zorgt voor goede leesbaarheid en sluit visueel mooi af bij het scherm.

Voordelen van dit palet

- **Gebruiksvriendelijkheid:** Gebruikers herkennen acties sneller dankzij kleurcodering.
- **Consistentie:** Elke actie heeft een vaste kleur, wat de navigatie intuïtiever maakt.
- **Uiterlijk:** Het kleurenpalet zorgt voor een professionele uitstraling passend bij een interactieve dierentuinomgeving.

4. Design Smells

Wat zijn Design Smells?

Design smells zijn signalen van slechte ontwerpkeuzes in de code. Hierdoor wordt de applicatie lastiger te onderhouden, uit te breiden en te begrijpen.

Veelvoorkomende design smells

- **God Classes:** klassen die te veel taken uitvoeren en logica bevatten.
- **Te veel afhankelijkheden tussen klassen:** Klassen die te sterk afhankelijk van elkaar zijn, waardoor wijzigingen moeilijk zijn.
- **Slechte naamgeving:** Klassen, methoden of variabelen met onduidelijke namen, wat de leesbaarheid van de code vermindert.
- **Duplicatie van code:** Het herhalen van dezelfde code op meerdere plaatsen.
- **Fragiele code:** Code breekt gemakkelijk bij kleine wijzigingen.

Hoe is er gelet op design smells binnen de applicatie

- **Design patterns geïmplementeerd:** Er zijn o.a. Factory en Strategy patterns gebruikt zodat code niet opnieuw geschreven hoeft te worden.
- **Scheiding van verantwoordelijkheden:** De controller verwerkt alleen HTTP-verzoeken en laat het uitvoeren van de logica over aan andere klassen of methoden.
- **Interfaces:** Taken die door meerdere klassen moeten worden uitgevoerd, worden duidelijk gedefinieerd in de interfaces.
- **Herbruikbare Code:** Er worden methods gemaakt (bijv. UpdateStallSize methods) die op meerdere plaatsen gebruikt kunnen worden om dezelfde functionaliteit te bieden en code-duplicatie te verminderen.
- **Caching:** Statistieken worden tijdelijk opgeslagen zodat ze niet opnieuw berekend hoeven te worden.

5. UML

Begrijpen wat UML is

UML-Diagrammen (Unified Modeling Language) is een visuele modelleertaal die gebruikt wordt in softwareontwikkeling om complexe systemen duidelijk te maken en begrijpen. Het helpt ontwikkelaars, ontwerpers en stakeholders om een duidelijk inzicht te krijgen over het systeem. UML biedt verschillende diagrammen om verschillende aspecten van het systeem weer te geven.

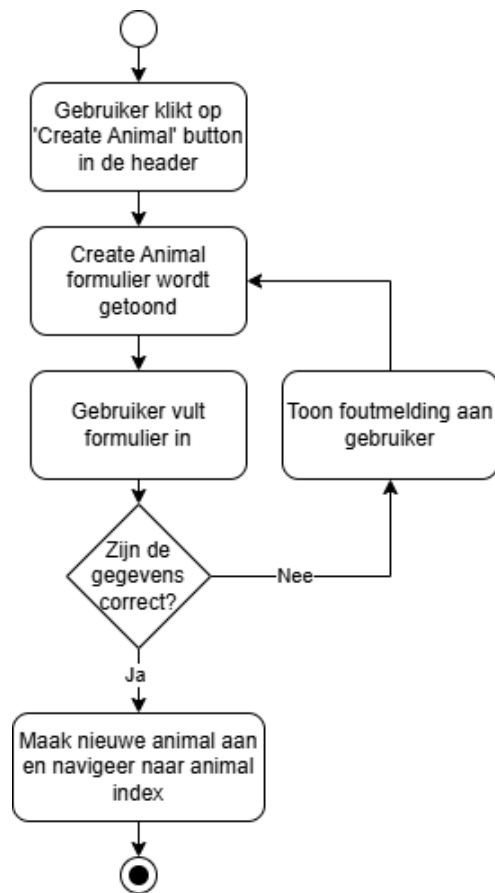
Waarom UML

- UML-Diagrammen maakt het technisch gedeelte van het applicatie duidelijk, voor andere ontwikkelaars, ontwerpers en stakeholders.
- Ondersteunt met het opsporen waar je fouten hebt gemaakt.
- Helpt met het ontwerpen van complexe systemen door uitgewerkte UML-Diagram(en) te hebben.

Hoe kun je UML toepassen

UML-Diagrammen zijn belangrijke onderdelen van de voorbereidingsfase in het ontwikkelproces van een applicatie. Hier kunnen de relaties tussen processen van een systeem duidelijk visueel worden gemaakt. Tijdens het ADP-vak heeft het team de eisen duidelijk gemaakt door de eisen op te splitsen in user story's in de backlog en de complexere stappen uitgewerkt in UML-Diagrammen. Dit hielp ons om het proces duidelijk te maken en vast te leggen hoe het zou werken in de ontwikkelfase.

5.1 Activity diagram (Create Animal)



Activity Diagram – Stapsgewijze Uitleg

1. Startpunt

Het proces start wanneer de gebruiker de website van de dierentuin opent en vervolgens op de knop '**Create Animal**' klikt in de header van de pagina.

2. Animal aanmaken

De gebruiker wordt doorgestuurd naar het **animal aanmaakformulier**. Hier vult de gebruiker alle vereiste gegevens in, zoals naam, soort, categorie, enzovoort.

3. Validatie van ingevoerde gegevens

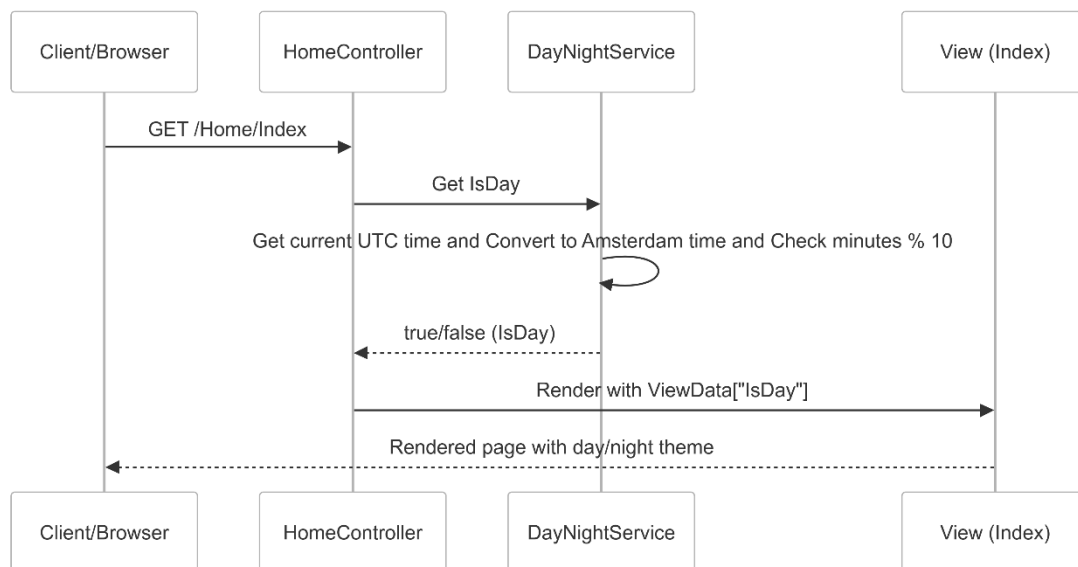
Zodra het formulier is ingevuld en verzonden, voert het systeem een validatie uit van de ingevoerde gegevens.

- **Gegevens zijn correct:**
 - Het systeem maakt het nieuwe dier (animal) aan.
 - De gebruiker wordt automatisch doorgestuurd naar de **animal index pagina**, waar een overzicht te zien is van alle aangemaakte dieren.
- **Gegevens zijn onjuist:**
 - Het systeem toont een **foutmelding** die aangeeft welke velden incorrect of leeg zijn.
 - De gebruiker blijft op het formulier en krijgt de mogelijkheid om de invoer te corrigeren en opnieuw te proberen.

4. Eindpunt

Bij een succesvolle creatie van een animal eindigt het proces op de **animal index pagina**, waar de gebruiker het nieuw aangemaakte dier terug kan zien tussen de bestaande records.

5.2 Sequence diagram (Day/night proces)



1. Pagina-aanvraag door gebruiker

- De **gebruiker** (Client/Browser) navigeert naar de hoofdpagina van de Dierentuin App.
- Er wordt een HTTP-request verstuurd: GET /Home/Index.

2. Afhandeling in HomeController

- De HomeController ontvangt het verzoek en moet bepalen of het **dag of nacht** is.
- Daarvoor roept de controller de methode IsDay aan in de DayNightService.

3. Dag/nacht-bepaling in DayNightService

- De DayNightService haalt de **huidige UTC-tijd** op.
- Deze UTC-tijd wordt **omgezet naar Amsterdam-tijd**.
- De service controleert het **laatste cijfer van de minuten** van deze tijd:
 - Als het laatste cijfer 0-4 is het **dag**.
 - Als het laatste cijfer 5-9 is het **nacht**.
- De service retourneert true (dag) of false (nacht) aan de controller.

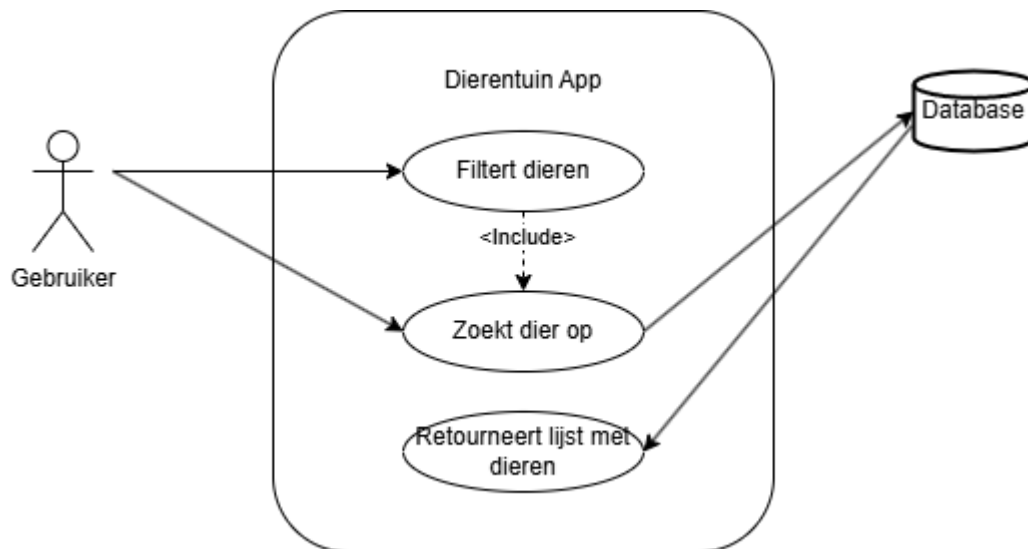
4. View rendering met IsDay

- De HomeController slaat de waarde van IsDay op in ViewData["IsDay"].
- Deze ViewData wordt doorgestuurd naar de bijbehorende View (Index).

5. Weergave aan de gebruiker

- De View gebruikt de waarde van `IsDay` om de pagina in een **dag- of nacht-thema** weer te geven.
- De gegenereerde HTML wordt teruggestuurd naar de browser.

5.3 Use case diagram (Dieren zoekfunctionaliteit)



Use Case Diagram Uitleg – Dierentuin App (Zoekfunctionaliteit)

Het onderstaande Use Case Diagram beschrijft de interacties tussen de primaire actor (**Gebruiker**), het systeem (**Dierentuin App**), en het externe subsysteem (**Database**).

Het diagram laat zien hoe een gebruiker een zoekopdracht uitvoert naar dieren binnen de dierentuin, eventueel met filters, en hoe het systeem hierop reageert door relevante resultaten weer te geven.

Actoren

1. Gebruiker

- De eindgebruiker van de Dierentuin App (bijv. een bezoeker van de dierentuin).
- Deze persoon voert zoekopdrachten uit, past filters toe en bekijkt zoekresultaten.
- Er is geen login nodig; de app is openbaar toegankelijk.

2. Database

- Externe gegevensopslag waarin alle diereninformatie is opgeslagen.
- De database wordt bevraagd op basis van zoek- en filtercriteria en retourneert de resultaten aan de applicatie.
- Bevat tabellen zoals Animals (diereninformatie) en Stalls (verblijven), maar de zoekopdracht richt zich enkel op de Animals-tabel.

Use Cases

De belangrijkste stappen en interacties in de zoekfunctionaliteit zijn als volgt:

1. Filtert dieren

- De gebruiker past optionele filters toe op basis van kenmerken zoals:
 - Grootte
 - Dieet
 - Activiteitspatroon
 - Beveiligingsniveau
- Deze filters zijn bedoeld om de zoekresultaten te verfijnen.

2. Zoekt dier op

- De gebruiker voert een vrije zoekopdracht in (zoals naam, soort, categorie).
- De zoekopdracht vormt de kernactie en maakt gebruik van de gekozen filters (zie <include> in het diagram).
- De applicatie stuurt een query naar de database met de ingevoerde parameters.

3. Retourneert lijst met dieren

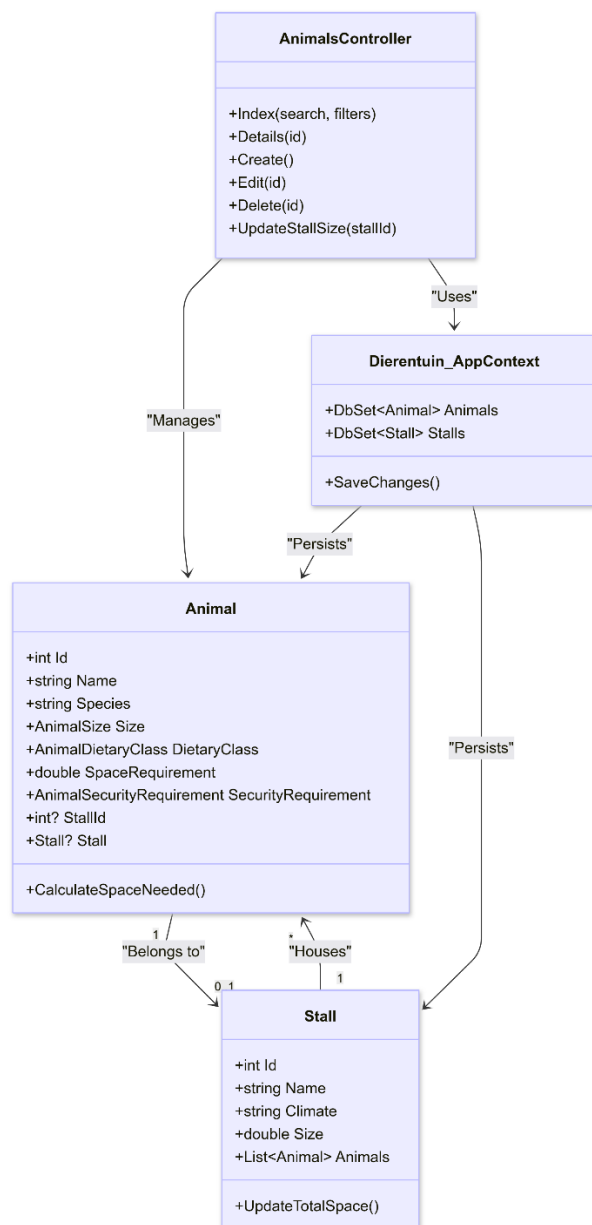
- De database stuurt de bijpassende dieren terug naar het systeem.
- De applicatie toont de resultaten in een overzicht met paginering (bijv. 25 dieren per pagina).
- De gebruiker kan eventueel doorklikken naar detailpagina's van specifieke dieren.

Externe Systemen

Database

- Verwerkt queries met zoek- en filtercriteria.
- Bevat de tabel Animals, waarop de zoekfunctie gebaseerd is.
- Retourneert gegevens zoals naam, soort, dieet, grootte, activiteit, beveiliging, enz.

5.4 Class diagram



Class Diagram Uitleg – Dierentuin App

Het bovenstaande class diagram beschrijft de kernstructuur van de **Dierentuin App**. Het geeft inzicht in de belangrijkste klassen binnen het systeem, hun verantwoordelijkheden en onderlinge relaties. De app maakt gebruik van Entity Framework Core voor database-interactie.

1. Dierentuin_AppContext (DbContext)

Beschrijving:

- Deze klasse beheert de verbinding met de database.
- Bevat twee DbSet<>-eigenschappen:
 - DbSet<Animal> Animals
 - DbSet<Stall> Stalls
- Verantwoordelijk voor het opslaan van wijzigingen via SaveChanges().

Relaties:

- 1..* relatie met Animal en Stall: één context beheert meerdere instanties van deze entiteiten.
- Wordt gebruikt door de AnimalsController voor databasebewerkingen.

2. Animal

Beschrijving:

- Vertegenwoordigt een dier in de dierentuin.
- Bevat eigenschappen zoals Name, Species, Category, Size, ActivityPattern, SecurityRequirement, enz.
- Bevat een verwijzing (StallId) naar de stal waarin het dier zich bevindt.
- Heeft een methode CalculateSpaceNeeded() die de benodigde ruimte berekent.

Relaties:

- Heeft een *many-to-one* relatie met Stall: elk dier behoort tot exact één stal.
- Wordt beheerd door de AnimalsController.

3. Stall

Beschrijving:

- Vertegenwoordigt een stal of verblijf voor dieren.
- Eigenschappen omvatten Name, Climate, HabitatType, SecurityLevel, en Size.
- Bevat een collectie van dieren (List<Animal> Animals).

Relaties:

- Heeft een *one-to-many* relatie met Animal: één stal kan meerdere dieren bevatten.
- Bevat de methode UpdateTotalSpace() die de benodigde ruimte van alle dieren in de stal bijwerkt.

4. AnimalsController

Beschrijving:

- Verzorgt de interactie tussen gebruiker en systeem.
- Beheert alle CRUD-operaties voor dieren:
 - Index() met zoek- en filtermogelijkheden
 - Create(), Edit(id), Delete(id)
 - Details(id)
- Bevat ook UpdateStallSize(stallId) om de ruimte van een stal automatisch aan te passen.

Relaties:

- **Gebruikt** de Dierentuin_AppContext om te communiceren met de database.
- **Beheert** Animal-objecten direct.
- **Indirect** gekoppeld aan Stall via de Animal-relatie.

6. Solid Principles

Toepassing van de SOLID-principes in het Class Diagram van de Dierentuin App:

In de Dierentuin App zijn de **SOLID-principes** toegepast om de code **duidelijker, uitbreidbaarder en beter onderhoudbaar** te maken. Hieronder wordt per principe uitgelegd hoe deze terugkomt in het ontwerp van de applicatie.

1. Single Responsibility Principle (SRP)

Principe:

Elke klasse heeft één verantwoordelijkheid en één reden om te veranderen.

Toepassing in de Dierentuin App:

- **Animal:** Beheert alleen de data en logica die horen bij één dier (zoals naam, soort, ruimtebehoefte).
- **Stall:** Verantwoordelijk voor eigenschappen en logica van een dierenverblijf, zoals klimaat, grootte, en verzameling dieren.
- **AnimalsController:** Handelt enkel acties af zoals het aanmaken, aanpassen of verwijderen van dieren.
- **Dierentuin_AppContext:** Richt zich alleen op database-interactie.

Impact:

Elke klasse heeft een duidelijke taak. Dit maakt de code makkelijk aanpasbaar en beperkt bugs bij veranderingen.

2. Open/Closed Principle (OCP)

Principe:

Klassen zijn open voor uitbreiding, maar gesloten voor aanpassing.

Toepassing:

- **Factory Pattern** (zoals bij de dieren templates) laat toe om nieuwe diersoorten toe te voegen zonder bestaande code aan te passen.
- **Stall en Animal** kunnen worden uitgebreid met extra eigenschappen door het maken van subklassen of extra services, zonder de bestaande klassen zelf te hoeven aanpassen.
- In de controllers kunnen nieuwe functionaliteiten worden toegevoegd via extra methodes of services.

Impact:

De app is makkelijk uit te breiden, zonder risico dat bestaande functionaliteit breekt.

3. Liskov Substitution Principle (LSP)

Principe:

Objecten van een subklasse moeten probleemloos vervangbaar zijn voor objecten van hun superklasse.

Toepassing:

- Hoewel er momenteel weinig subklassen zijn, zou je bijvoorbeeld `PredatorAnimal` of `AquaticAnimal` kunnen introduceren als subklassen van `Animal`. Deze kunnen zonder probleem gebruikt worden waar `Animal` verwacht wordt.
- Hetzelfde geldt voor stallen: een gespecialiseerde `TundraStall` zou `Stall` kunnen uitbreiden zonder de rest van het systeem aan te passen.

Impact:

De toepassing is voorbereid op toekomstige uitbreidingen zonder compatibiliteitsproblemen.

4. Interface Segregation Principle (ISP)

Principe:

Klassen zouden niet gedwongen moeten worden om methodes te implementeren die ze niet gebruiken.

Toepassing:

- Hoewel de app momenteel nog geen expliciete interfaces gebruikt voor controllers of services, zou dit eenvoudig kunnen worden toegevoegd.

Bijvoorbeeld:

- IAnimalRepository interface voor databasebewerkingen
 - IStallCleaningStrategy interface voor reinigingslogica
- Dit zorgt ervoor dat klassen alleen gebruiken wat ze nodig hebben.

Impact:

De logica blijft overzichtelijk en testbaar. Interfaces kunnen makkelijk worden gemoockt bij unit tests.

5. Dependency Inversion Principle (DIP)

Principe:

Hogere modules mogen niet afhankelijk zijn van lagere modules, maar van abstracties.

Toepassing:

- De applicatie gebruikt **dependency injection** via de constructor in controllers, bijvoorbeeld:

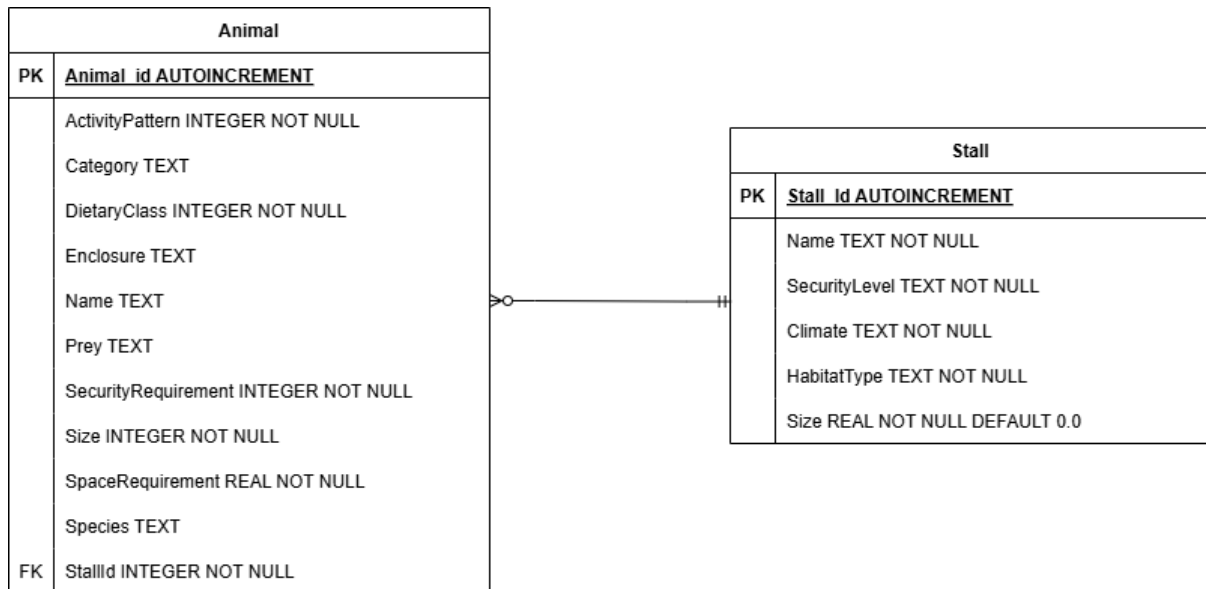
```
public AnimalsController(Dierentuin_AppContext context) { ... }
```

- Dit maakt het mogelijk om eenvoudig een andere context te injecteren, bijvoorbeeld bij testen.
- Abstracties zoals toekomstige IAnimalService of IStallService zouden de afhankelijkheid van concrete implementaties verder kunnen verminderen.

Impact:

De code is makkelijker testbaar en aanpasbaar zonder directe afhankelijkheden tussen klassen.

7. Entity Relationship Diagram (ERD)



Uitleg van het ERD en Externe Applicatiecontext

De **Dierentuin App** is een publieke applicatie die gebruikers op een eenvoudige manier inzicht geeft in de dieren en hun verblijven binnen de dierentuin. De applicatie biedt basisfunctionaliteiten zoals het aanmaken, bewerken en verwijderen van dieren en stallen. Daarnaast biedt het interactieve elementen zoals het voeren van dieren en het volgen van een dag- en nachtcyclus.

De applicatie heeft geen inlogsysteem en is bedoeld voor algemeen gebruik. Het doel is om op een leuke, educatieve manier inzicht te geven in het leven van dieren in hun stallen.

Entiteiten en hun Relaties

1. Stall

Beschrijving:

- Een verblijfseenheid waarin dieren worden ondergebracht.
- Bevat informatie over de grootte, het klimaat en het type habitat.

Attributen:

- Id: Unieke identificatie van de stall (Primair sleutel).
- Name: Naam van de stall.
- SecurityLevel: Niveau van beveiliging.
- Climate: Type klimaat (bijv. Tropisch, Woestijn).
- HabitatType: Beschrijving van de habitat.
- Size: Grootte van de stall in m².

Relatie:

- Heeft een **one-to-many** relatie met Animal (één stall bevat meerdere dieren).

2. Animal

Beschrijving:

- Een individueel dier dat zich in een specifieke stall bevindt.
- Bevat informatie over gedrag, dieet, status en ruimtebehoefte.

Attributen:

- Id: Unieke identificatie van het dier (Primair sleutel).
- Name: Naam van het dier.
- Species: Soort (bijv. Leeuw, Zebra).
- Category: Classificatie (bijv. Zoogdier, Reptiel).
- ActivityPattern: Actief overdag of 's nachts.
- DietaryClass: Type dieet (bijv. Herbivoor, Carnivoor).
- Prey: Eventuele prooien.
- Size: Grootte van het dier.
- SpaceRequirement: Benodigde ruimte in m².
- SecurityRequirement: Beveiligingsniveau nodig voor het dier.
- Enclosure: Type omgeving.
- StallId: Foreign sleutel naar de stall waarin het dier zich bevindt.
- IsHungry: Boolean (wel of niet hongerig).
- IsSleeping: Boolean (slaapstatus, afhankelijk van dag/nacht).

Relatie:

- Heeft een **many-to-one** relatie met Stall.

Relaties in het ERD

Animal > Stall

- **Relatie:** many-to-one
- Meerdere dieren kunnen gekoppeld zijn aan één stall.
- Elk dier hoort bij exact één stall.

Functionaliteiten van de Dierentuin App

CRUD Functionaliteiten

- **Stall:**
 - Aanmaken, bekijken, bewerken en verwijderen van stallen.
 - Op de detailpagina wordt:
 - De **lijst van dieren in de stall** weergegeven.
 - De **totaal gebruikte ruimte** van de dieren berekend t.o.v. de grootte van de stall.
- **Animal:**
 - Aanmaken, bewerken en verwijderen van dieren.
 - Elk dier wordt aan een stall gekoppeld via StallId.

Interacties

- **Voeren:**
 - Als een dier honger heeft (IsHungry = true), kan deze gevoerd worden.
 - Na voeren wordt IsHungry op false gezet.
- **Dag/Nacht-cyclus:**
 - Een schakelbare cyclus bepaalt of het dag of nacht is.
 - Dieren reageren afhankelijk van hun ActivityPattern:
 - Diurnaal (dagactief): slapen 's nachts.
 - Nocturnaal (nachtactief): slapen overdag.
 - Slaapstatus (IsSleeping) wordt automatisch aangepast.

8. Design Patterns

8.1 Creational pattern

Wat is een creational pattern?

Een patroon dat helpt bij het maken van objecten zonder afhankelijk te zijn van specifieke klassen. Dit maakt de code flexibeler en gemakkelijker te onderhouden.

Toepassing 1 (Animal template)

Voor de applicatie is een methode geïmplementeerd om vooraf geconfigureerde dieren te maken, zodat de gebruiker het formulier niet handmatig hoeft in te vullen als het dier al bestaat (factory method).

Uitleg toepassing 1 (Animal template)

1. Product (Animal class)

De Animal class is wat de factories maken. Het bevat eigenschappen zoals Name, Species, Size, and DietaryClass , die de kenmerken van de dieren in de dierentuin beschrijven.

2. Factory interface

Er wordt een interface gedefinieerd met een methode die alle factories moeten gebruiken. Deze interface zorgt ervoor dat elke factory een manier biedt om een dier te maken.

```
// Interface for Animal
7 references
public interface IAnimalFactory
{
    7 references
    Animal CreateAnimal();
}
```

3. Factory implementaties

Voor elk dier is een aparte factory class gemaakt met ieder zijn eigen kenmerken.

```
// Factory implementations
1 reference
public class LionFactory : IAnimalFactory
{
    2 references
    public Animal CreateAnimal()
    {
        return new Animal
        {
            Name = "Lion",
            Species = "Panthera leo",
            Category = "Mammal",
            Size = Animal.AnimalSize.Large,
            DietaryClass = Animal.AnimalDietaryClass.Carnivore,
            ActivityPattern = Animal.AnimalActivityPattern.Cathemeral,
            Prey = "Zebra, Wildebeest, Antelope",
            Enclosure = "Savanna",
            SpaceRequirement = 1000,
            SecurityRequirement = Animal.AnimalSecurityRequirement.High,
        };
    }
}
```

4. Factory (provider)

Deze class werkt zodat de juiste factory gekozen wordt op basis van de naam van het dier.

```
// Factory (provider)
1 reference
public static class AnimalFactory
{
    1 reference
    public static IAnimalFactory GetFactory(string animal)
    {
        return animal switch
        {
            "lion" => new LionFactory(),
            "monkey" => new MonkeyFactory(),
            "elephant" => new ElephantFactory(),
            "penguin" => new PenguinFactory(),
            "tiger" => new TigerFactory(),
            "giraffe" => new GiraffeFactory(),
            _ => throw new ArgumentException($"Unsupported animal type: {animal}"),
        };
    }
}
```

5. Werking UI

De gebruiker klikt op een template button van een dier.

Create Animal

Quick Templates:

Elephant

Monkey

Lion

Penguin

Tiger

Giraffe

Name

Species

Category

Size

Choose size

▼

DietaryClass

Choose dietary class

▼

ActivityPattern

Choose activity pattern

▼

Prey

Enclosure

SpaceRequirement

SecurityRequirement

Choose security requirement

▼

Stall

Sadek

▼

Create Animal

[Back to List](#)

De JavaScript-code slaat het dier op en stuurt een AJAX-aanvraag naar de controller action GetAnimalTemplate.

```
@section Scripts {
    @{
        await Html.RenderPartialAsync("_ValidationScriptsPartial");
    }

    <script>
        $(document).ready(function() {
            $('#template-btn').click(function() {
                // Get animal type
                const animal = $(this).data('animal');

                // Get template data from server
                $.ajax({
                    url: '@Url.Action("GetAnimalTemplate", "Animals")',
                    data: { animal: animal },
                    success: function(data) {
                        $('#Name').val(data.name);
                        $('#Species').val(data.species);
                        $('#Category').val(data.category);
                        $('#Size').val(data.size);
                        $('#DietaryClass').val(data.dietaryClass);
                        $('#ActivityPattern').val(data.activityPattern);
                        $('#Prey').val(data.prey);
                        $('#Enclosure').val(data.enclosure);
                        $('#SpaceRequirement').val(data.spaceRequirement);
                        $('#SecurityRequirement').val(data.securityRequirement);
                    }
                });
            });
        });
    </script>
}
```

```
<!-- Animal templates -->
<h5>Quick Templates:</h5>
<div class="d-flex gap-3 flex-wrap">
    <!-- Elephant -->
    <button type="button" class="btn btn-secondary template-btn" data-animal="elephant">
        Elephant
    </button>

    <!-- Monkey -->
    <button type="button" class="btn btn-brown template-btn" data-animal="monkey"
        style="background-color: #8B4513; color: white;">
        Monkey
    </button>

    <!-- Lion -->
    <button type="button" class="btn template-btn" data-animal="lion"
        style="background-color: #E6BE8A; color: #333;">
        Lion
    </button>

    <!-- Penguin -->
    <button type="button" class="btn btn-dark template-btn" data-animal="penguin"
        style="border: 2px solid white;">
        Penguin
    </button>

    <!-- Tiger -->
    <button type="button" class="btn template-btn" data-animal="tiger"
        style="background-color: #FF7518; color: black; border: 2px dashed #000;">
        Tiger
    </button>

    <!-- Giraffe -->
    <button type="button" class="btn template-btn" data-animal="giraffe"
        style="background-color: #F9DC5C; color: #7E4A1C; border: 2px dotted #7E4A1C;">
        Giraffe
    </button>
</div>
```

De controller vraagt aan de factory voor het gekozen dier en krijgt het object van het dier terug met al zijn kenmerken.

```
[HttpGet]
0 references
public IActionResult GetAnimalTemplate(string animal)
{
    try
    {
        Animal template = AnimalFactory.GetFactory(animal).CreateAnimal();
        return Json(template);
    }
    catch (ArgumentException)
    {
        return NotFound();
    }
}
```

De gegevens van het dier worden teruggestuurd naar de browser, die de invoervelden automatisch invult.

6. Eindresultaat

Create Animal

Quick Templates:

Elephant

Monkey

Lion

Penguin

Tiger

Giraffe

Name	<input type="text" value="Elephant"/>
Species	<input type="text" value="Loxodonta africana"/>
Category	<input type="text" value="Mammal"/>
Size	<input type="text" value="VeryLarge"/>
DietaryClass	<input type="text" value="Herbivore"/>
ActivityPattern	<input type="text" value="Diurnal"/>
Prey	<input type="text" value="Grass, Leaves, Fruit"/>
Enclosure	<input type="text" value="Savanna"/>
SpaceRequirement	<input type="text" value="2000"/>
SecurityRequirement	<input type="text" value="Medium"/>
Stall	<input type="text" value="Sadek"/>

Create Animal

[Back to List](#)

Voordelen van dit patroon:

- **Encapsulatie van logica:** De manier van dieren maken is verborgen in de factory, waardoor de rest van de code eenvoudiger en overzichtelijker wordt.
- **Makkelijk uitbreidbaar:** Nieuwe dieren toevoegen zonder bestaande code te wijzigen.

Toepassing 2 (Stall Template)

Voor de applicatie is een methode geïmplementeerd om vooraf geconfigureerde **stallen** te maken. Dit bespaart tijd bij het toevoegen van stallen aan de diertuin, doordat veelvoorkomende configuraties beschikbaar zijn als template-knoppen. Deze aanpak maakt gebruik van het **Factory Method**.

Uitleg Toepassing 2 (Stall Template)

1. Product (Stall class)

De Stall class is het object dat wordt gegenereerd via de factory. Het bevat eigenschappen zoals Name, Climate, HabitatType, SecurityLevel, en Size.

```
public class Stall
{
    public int Id { get; set; }
    public string Name { get; set; }
    public string Climate { get; set; }
    public string HabitatType { get; set; }
    public string SecurityLevel { get; set; }
    public double Size { get; set; }
}
```

2. Factory class (StallFactory)

De StallFactory centraliseert de creatielogica van vooraf ingestelde stallen. Hier worden presets opgeslagen in een dictionary en via CreateStall() omgezet in nieuwe Stall-objecten.

```
public static class StallFactory
{
    private static readonly Dictionary<string, Stall> Presets = new()
    {
        { "Tropical Rainforest", new Stall { Name = "Tropical Paradise", Climate = "Tropical" } },
        { "Savannah Enclosure", new Stall { Name = "Savannah Plains", Climate = "Arid", HabitatType = "Savannah" } },
        { "Arctic Zone", new Stall { Name = "Frozen Habitat", Climate = "Polar", HabitatType = "Arctic" } }
    };

    public static Stall CreateStall(string presetName)
    {
        if (Presets.TryGetValue(presetName, out var preset))
        {
            return new Stall
            {
                Name = preset.Name,
                Climate = preset.Climate,
                HabitatType = preset.HabitatType,
                SecurityLevel = preset.SecurityLevel,
                Size = preset.Size
            };
        }

        return new Stall();
    }
}
```

3. Werking UI (JavaScript + Buttons)

Gebruikers klikken op een template-knop voor een bepaalde stall:

```
<button type="button" class="btn btn-success stall-template-btn" data-preset="Tropical Rainforest">Tropical Rainforest</button>
```

4. Controller: Factory aanroepen

De controller ontvangt het preset-verzoek en vraagt aan de factory om een nieuw Stall-object te maken:

```
[HttpGet]
public IActionResult GetStallTemplate(string preset)
{
    var stall = StallFactory.CreateStall(preset);

    return Json(new
    {
        name = stall.Name,
        climate = stall.Climate,
        habitatType = stall.HabitatType,
        securityLevel = stall.SecurityLevel,
        size = stall.Size
    });
}
```

5. Eindresultaat

- Gebruiker kiest een knop zoals "Tropical"
- De velden in het formulier worden automatisch ingevuld
- Het Stall-object kan direct opgeslagen worden

Create Stall

Quick Templates:

Tropical Savannah Arctic

Name

Climate

HabitatType

SecurityLevel

Size

0

Create Stall

[Back to List](#)

8.2 Behavioral pattern

Wat is een behavioral pattern?

Een behavioral pattern bepaalt hoe objecten met elkaar communiceren en wie welke taak op zich neemt binnen de samenwerking tussen die objecten.

Toepassing 1 (Animal behavior)

Gezien een activiteitspatroon niet altijd gemakkelijk te begrijpen is wat het is, moest er een duidelijke beschrijving worden geïmplementeerd om dit probleem op te lossen. Daarom is er in de applicatie een strategie (patroon) toegewezen, zodat er eenvoudig een dynamische beschrijvingsfunctie kan worden toegevoegd op basis van het activiteitspatroon van het dier, zonder dat er conflicten kunnen ontstaan met bestaande functionaliteiten.

Details of Lion

Name:	Lion
Species:	Panthera leo
Category:	Mammal
Size:	Large
DietaryClass:	Carnivore
ActivityPattern:	Cathemeral
Prey:	Zebra, Wildebeest, Antelope
Enclosure:	Savanna
SpaceRequirement:	1000
SecurityRequirement:	High
Stall:	Sadek

[Edit](#) | [Back to List](#)

Uitleg toepassing 1 (Animal behavior)

1. Strategie interface en concrete strategieklassen

Er wordt een interface gedefinieerd waaraan alle concrete gedragsstrategieën moeten voldoen. Elke strategie biedt een specifieke implementatie van het gedrag van dieren, gebaseerd op hun activiteitscycli (AnimalActivityPattern).

```
// Interface animal activity pattern behaviors
4 references
public interface IAnimalBehavior
{
    5 references
    string Behavior();
}

// Behaviors based on ActivityPattern
1 reference
public class DiurnalBehavior : IAnimalBehavior
{
    3 references
    public string Behavior()
    {
        return "Active during daylight hours, rests at night.";
    }
}

1 reference
public class NocturnalBehavior : IAnimalBehavior
{
    3 references
    public string Behavior()
    {
        return "Sleeps during the day and becomes active at night.";
    }
}

1 reference
public class CathemeralBehavior : IAnimalBehavior
{
    3 references
    public string Behavior()
    {
        return "Active during both day and night, alternating periods of activity.";
    }
}
```

2. Contextintegratie

De Animal class bevat de gedragstrategie als een eigenschap (wordt niet naar de database weggeschreven).

```
23 references
public enum AnimalActivityPattern
{
    Diurnal,
    Nocturnal,
    Cathemeral
}

[NotMapped]
5 references
public IAnimalBehavior AnimalBehavior { get; set; }

0 references
public string? Behavior()
{
    return AnimalBehavior?.Behavior();
}
```

3. Juiste strategie pakken

In de AnimalsController wordt de juiste strategie opgehaald tijdens het ophalen van de details pagina voor een specifieke animal.

```
// GET: Animals/Details/5
0 references
public async Task<IActionResult> Details(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var animal = await _context.Animal
        .Include(a => a.Stall)
        .FirstOrDefaultAsync(m => m.Id == id);

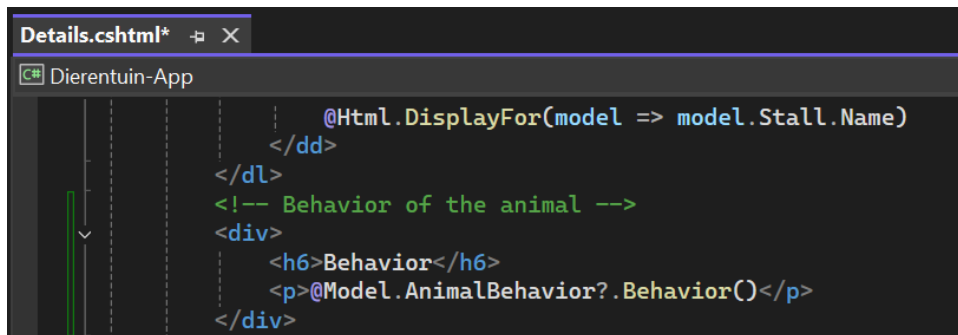
    if (animal == null)
    {
        return NotFound();
    }

    // Set the animal behavior based on the activity pattern
    switch (animal.ActivityPattern)
    {
        case Animal.AnimalActivityPattern.Diurnal:
            animal.AnimalBehavior = new DiurnalBehavior();
            break;
        case Animal.AnimalActivityPattern.Nocturnal:
            animal.AnimalBehavior = new NocturnalBehavior();
            break;
        case Animal.AnimalActivityPattern.Cathemeral:
            animal.AnimalBehavior = new CathemeralBehavior();
            break;
    }

    return View(animal);
}
```

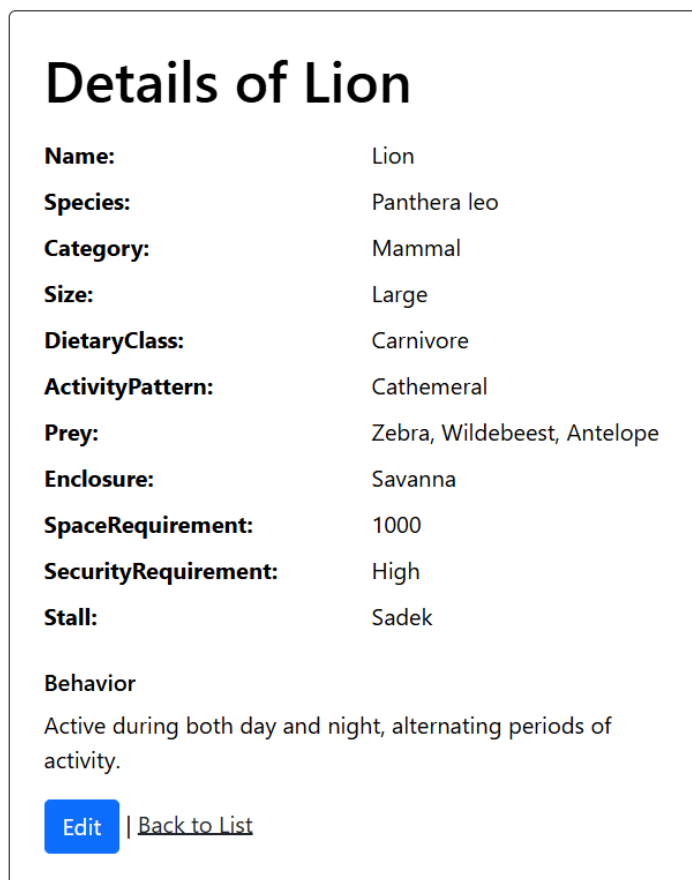

4. Weergave beschrijving

Het gedrag wordt tot slot weergegeven aan de gebruiker.



```
Details.cshtml*  X
Dierentuin-App

@Html.DisplayFor(model => model.Stall.Name)
</dd>
</dl>
<!-- Behavior of the animal -->
<div>
    <h6>Behavior</h6>
    <p>@Model.AnimalBehavior?.Behavior()</p>
</div>
```



Details of Lion

Name:	Lion
Species:	Panthera leo
Category:	Mammal
Size:	Large
DietaryClass:	Carnivore
ActivityPattern:	Cathemeral
Prey:	Zebra, Wildebeest, Antelope
Enclosure:	Savanna
SpaceRequirement:	1000
SecurityRequirement:	High
Stall:	Sadek

Behavior

Active during both day and night, alternating periods of activity.

[Edit](#) | [Back to List](#)

Voordelen van dit patroon:

- **Eenvoudige uitbreiding:** Nieuwe functies kunnen snel worden toegevoegd.
- **Losse koppeling:** Wijzigingen beïnvloeden geen andere delen van de applicatie.
- **Eenvoudig te onderhouden:** behaviors zijn geïsoleerd.

Toepassing 2 (Stall cleaning)

Voor stallen is een strategie geïmplementeerd om het **schoonmaakgedrag** dynamisch toe te passen op basis van het habitatype. Hierdoor hoeft de schoonmaaklogica niet hardcoded te zijn in de Stall class, en kan het gedrag makkelijk uitgebreid worden met nieuwe strategieën voor andere habitats.

Uitleg toepassing 2 (Stall cleaning)

1. Strategie-interface en concrete strategieklassen

Er is een interface `ICleaningStrategy` die de methode `Clean()` bevat. Voor elk type habitat is een concrete implementatie gemaakt, zoals:

- `RainforestCleaning`
- `TundraCleaning`
- `DesertCleaning`

Elke klasse geeft een eigen beschrijving van de schoonmaakmethode.

```
namespace Dierentuin_App.Models.Behavior
{
    4 references
    public interface ICleaningStrategy
    {
        4 references
        string Clean();
    }
}
```

```
namespace Dierentuin_App.Models.Behavior
{
    public class DesertCleaning : ICleaningStrategy
    {
        public string Clean() => "Dust off sand and provide fresh water.";
    }
}
```

```
namespace Dierentuin_App.Models.Behavior
{
    1 reference
    public class RainforestCleaning : ICleaningStrategy
    {
        2 references
        public string Clean() => "Use high-pressure water and remove tropical leaves.";
    }
}
```

2. Contextintegratie

De Stall class bevat een referentie naar de gekozen CleaningStrategy. Deze eigenschap wordt **niet opgeslagen in de database**.

```
! reference  
public string PerformCleaning()  
{  
    return CleaningStrategy?.Clean() ?? "No cleaning strategy assigned.";  
}
```

3. Juiste strategie pakken

In de StallsController wordt bij het tonen van de detailpagina automatisch een passende strategie ingesteld op basis van HabitatType.

```
// Clean stall based on habitat type  
switch (stall.HabitatType?.ToLower())  
{  
    case "rainforest":  
        stall.CleaningStrategy = new RainforestCleaning();  
        break;  
    case "tundra":  
        stall.CleaningStrategy = new TundraCleaning();  
        break;  
    case "desert":  
        stall.CleaningStrategy = new DesertCleaning();  
        break;  
    default:  
        stall.CleaningStrategy = null;  
        break;  
}  
  
// Store cleaning result for the view  
ViewData["CleaningInstruction"] = stall.PerformCleaning();
```

4. Weergave beschrijving

De output van de gekozen strategie wordt weergegeven in de view, zodat gebruikers meteen weten hoe de stal schoongemaakt moet worden.

```
@if (ViewData["CleaningInstruction"] != null)
{
    <div class="alert alert-info">
        <strong>Cleaning Instructions:</strong> @ViewData["CleaningInstruction"]
    </div>
}
```

Stall Details

Name	renas kattenhuis
Climate	Arid
HabitatType	Tundra
SecurityLevel	Medium
Size	0.00 m²

Cleaning Instructions: Sweep snow and sanitize heated platforms.

Animals in the Stall:

Feed all animals | [Edit](#) | [Back to List](#)

Voordelen van dit patroon:

- **Eenvoudige uitbreiding:** Nieuwe schoonmaakstrategieën zijn makkelijk toe te voegen zonder bestaande code te wijzigen.
- **Losse koppeling:** De Stall class hoeft geen specifieke schoonmaaklogica te kennen.
- **Herbruikbaarheid:** Eén strategie kan gebruikt worden door meerdere stallen van hetzelfde type.

8.3 Concurrency Pattern

Wat is een concurrency pattern

Een **concurrency pattern** is een ontwerpprincipie dat helpt bij het **tegelijk uitvoeren van acties** (door meerdere gebruikers of processen) **zonder conflicten**.

In webapplicaties wordt dit vaak gebruikt om **race conditions** te voorkomen – situaties waarin meerdere gebruikers dezelfde gegevens tegelijk proberen te bewerken of acties tegelijkertijd uitvoeren.

Het zorgt ervoor dat de applicatie **stabiel en consistent** blijft, zelfs bij **tegelijk gebruik**.

Toepassing 1 (Feeding Animals)

In deze dierentuinapplicatie is een concurrency pattern geïmplementeerd bij het voeren van dieren:

dieren mogen maar eens per 10 minuten gevoerd worden, ongeacht welke gebruiker de actie uitvoert.

Als iemand probeert te voeren binnen 10 minuten, krijgt die gebruiker een duidelijke foutmelding.

Deze bescherming werkt **globaal**, dus is zichtbaar voor alle gebruikers.

Uitleg toepassing 1:

1. LastFedAt in Stall model

Elke Stall heeft een LastFedAt property.

Dit veld houdt bij wanneer de dieren voor het laatst zijn gevoerd.

```
public DateTime? LastFedAt { get; set; }
```

2. Server-side validatie in de controller

In de FeedAllAnimals-methode wordt gecontroleerd of het voeren al is gebeurd in de afgelopen 10 minuten:

```
if (stall.LastFedAt.HasValue && (now - stall.LastFedAt.Value).TotalMinutes < 10)
{
    return Json(new {
        success = false,
        message = "Animals already fed. Try again in X minutes."
    });
}
```

Indien nog niet gevoerd: de server slaat de huidige tijd op als nieuwe LastFedAt.

3. Front-end updates de tijden automatisch

De data-fed-at attributen van elk dier worden geüpdatet zodat de browser kan bepalen of ze gevoerd zijn of niet.

```
animal.setAttribute('data-fed-at', data.fedAt);
```

4. Feed-knop wordt tijdelijk uitgeschakeld

Na het voeren wordt de "Feed all animals" knop **gedeactiveerd** voor 10 minuten met een duidelijke timer.

```
if (diffMinutes < 10) {  
  disableFeedButton();  
}
```

Werking vanuit de UI

1. Gebruiker klikt op "Feed all animals"
2. De server checkt of de 10 minuten al verstreken zijn
3. Als ja: dieren worden gevoerd en nieuwe tijd wordt opgeslagen
4. Als nee: gebruiker krijgt foutmelding
5. Frontend toont: "Animals are fed – X min left"
6. Na 10 minuten: knop wordt automatisch weer klikbaar

Feed all animals

localhost:8080 says
Animals fed successfully!

OK

Feed all animals (available in 10 mins)

Voordelen van dit patroon:

- **Voorkomt dubbele handelingen:** voorkomt dat dieren onbedoeld meerdere keren gevoerd worden.
- **Gebruikersvriendelijk:** gebruikers krijgen duidelijke feedback over wanneer voeren weer kan.
- **Geschikt voor meerdere gebruikers tegelijk:** of je nu 1 of 100 gebruikers hebt, dit patroon houdt het **systeem stabiel**.
- **Volledig server-side gecontroleerd:** dus veilig en betrouwbaar.

Toepassing 2 (AnimalStatistics caching)

Er is een statistiekenpagina die de dieren per stal netjes bijhoudt. Het probleem was dat bij elke refresh van de pagina een nieuwe databasequery werd uitgevoerd, wat zorgde voor onnodige belasting van de database en vertraagde responstijden, vooral bij meerdere gebruikers. Hiervoor is een combinatie van **Monitor Object** concurrency pattern en memory caching geïmplementeerd. Door deze combinatie worden de statistieken slechts één keer per 5 minuten opgehaald en opgeslagen in het geheugen, terwijl gelijktijdige toegang correct wordt afgehandeld.

Uitleg toepassing 2:

1. Controle gegevens in het geheugen

Eerst wordt er gecontroleerd of de statistieken al in het geheugen zijn opgeslagen (_memoryCache) doormiddel van de cache key value te checken. Is er data in het geheugen ga naar stap 4, anders ga naar stap 2.

2. Thread-veilige caching

Bij een cache-miss wordt een thread-veilige aanpak gehanteerd met een lock zodat slechts één thread tegelijk de database benadert. Die controleert nogmaals of de cache inmiddels is gevuld voordat het daadwerkelijk de gegevens ophaalt.

3. Gegevens ophalen

De stall met de bijbehorende animals worden uit de database opgehaald en groepeert per hok. Per stall wordt geteld hoeveel dieren er zijn van elke 'AnimalDietaryClass' (carnivoor, herbivoor, etc.).

4. Caching

Slaat deze data op in de cache voor 5 minuten.

9 references

```
public class AnimalsController : Controller
{
    private readonly Dierentuin_AppContext _context;
    private readonly IMemoryCache _memoryCache;
    private static readonly object _statisticsLock = new object();

    5 references
    public AnimalsController(Dierentuin_AppContext context, IMemoryCache memoryCache)
    {
        _context = context;
        _memoryCache = memoryCache;
    }
}
```

[HttpGet]

0 references

```
public async Task<IActionResult> AnimalStatistics()
{
    // Cache key
    const string cacheKey = "AnimalStatistics";
    Dictionary<string, object> cachedStats;

    // Get statistics from cache miss (without locking)
    if (!_memoryCache.TryGetValue(cacheKey, out cachedStats))
    {
        // Monitor Object
        lock (_statisticsLock)
        {
            // Double check another thread updated the cache
            if (!_memoryCache.TryGetValue(cacheKey, out cachedStats))
            {
                var animals = _context.Animal.Include(a => a.Stall).ToList();

                // Group by stall name and count dietary classes
                var grouped = animals
                    .GroupBy(a => a.Stall.Name)
                    .Select(g => new
                    {
                        Stall = g.Key,
                        Carnivores = g.Count(a => a.DietaryClass == Animal.AnimalDietaryClass.Carnivore),
                        Herbivores = g.Count(a => a.DietaryClass == Animal.AnimalDietaryClass.Herbivore),
                        Omnivores = g.Count(a => a.DietaryClass == Animal.AnimalDietaryClass.Omnivore),
                        Insectivores = g.Count(a => a.DietaryClass == Animal.AnimalDietaryClass.Insectivore),
                        Piscivores = g.Count(a => a.DietaryClass == Animal.AnimalDietaryClass.Piscivore),
                    }).ToList();

                // Store arrays in a dictionary
                cachedStats = new Dictionary<string, object>
                {
                    ["StallNames"] = grouped.Select(m => m.Stall).ToArray(),
                    ["Carnivores"] = grouped.Select(m => m.Carnivores).ToArray(),
                    ["Herbivores"] = grouped.Select(m => m.Herbivores).ToArray(),
                    ["Omnivores"] = grouped.Select(m => m.Omnivores).ToArray(),
                    ["Insectivores"] = grouped.Select(m => m.Insectivores).ToArray(),
                    ["Piscivores"] = grouped.Select(m => m.Piscivores).ToArray(),
                    ["LastUpdated"] = DateTime.Now,
                };

                // Store in cache with a 5 min expiration
                _memoryCache.Set(cacheKey, cachedStats, TimeSpan.FromMinutes(5));
            }
        }
    }

    // Use the statistics (cache or newly grabbed)
    ViewBag.StallNames = cachedStats["StallNames"];
    ViewBag.Carnivores = cachedStats["Carnivores"];
    ViewBag.Herbivores = cachedStats["Herbivores"];
    ViewBag.Omnivores = cachedStats["Omnivores"];
    ViewBag.Insectivores = cachedStats["Insectivores"];
    ViewBag.Piscivores = cachedStats["Piscivores"];
    ViewBag.LastUpdated = cachedStats["LastUpdated"];

    return View();
}
```

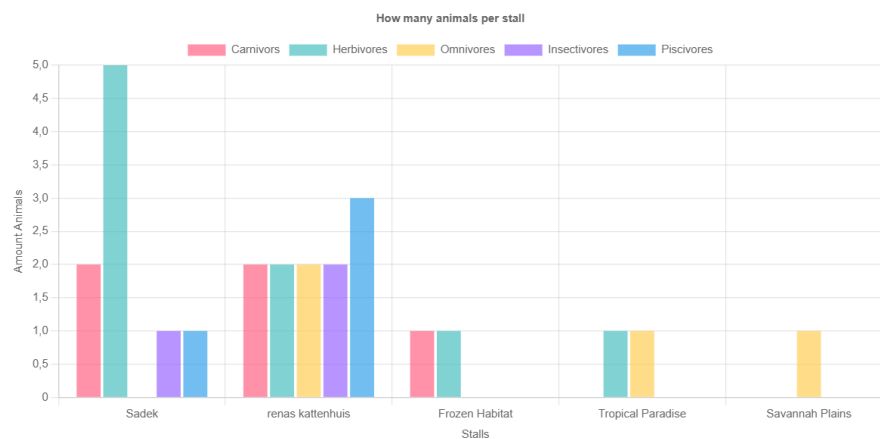

5. Statistieken naar de view

Tot slot worden de statistieken verstuurd naar de view.

```
@section Scripts {
<script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
<script>
const ctx = document.getElementById('dietChart').getContext('2d');
const chart = new Chart(ctx, {
  type: 'bar',
  data: {
    labels: @Html.Raw(Json.Serialize(ViewBag.stallNames)),
    datasets: [
      {
        label: 'Carnivores',
        data: @Html.Raw(Json.Serialize(ViewBag.carnivores)),
        backgroundColor: 'rgba(255, 99, 132, 0.7)'
      },
      {
        label: 'Herbivores',
        data: @Html.Raw(Json.Serialize(ViewBag.herbivores)),
        backgroundColor: 'rgba(75, 192, 192, 0.7)'
      },
      {
        label: 'Omnivores',
        data: @Html.Raw(Json.Serialize(ViewBag.omnivores)),
        backgroundColor: 'rgba(255, 206, 86, 0.7)'
      },
      {
        label: 'Insectivores',
        data: @Html.Raw(Json.Serialize(ViewBag.insectivores)),
        backgroundColor: 'rgba(153, 102, 255, 0.7)'
      },
      {
        label: 'Piscivores',
        data: @Html.Raw(Json.Serialize(ViewBag.piscivores)),
        backgroundColor: 'rgba(54, 162, 235, 0.7)'
      }
    ]
  },
  options: {
    responsive: true,
    plugins: {
      legend: { position: 'top' },
      title: { display: true, text: 'How many animals per stall' }
    },
    scales: {
      y: {
        beginAtZero: true,
        title: {
          display: true,
          text: 'Amount Animals'
        }
      },
      x: {
        title: {
          display: true,
          text: 'Stalls'
        }
      }
    }
  }
});
</script>
```

Dierentuin App [Home](#) [Create Animal](#) [Create Stall](#) [Animals index](#) [Stall index](#) [About Us](#)

Animal Statistics



Voordelen van dit patroon:

- **Betere prestaties:** Vermindert databasebelasting door gegevens tijdelijk in het geheugen op te slaan.
- **Verminderde databasebelasting:** Database wordt slechts elke 5 minuten aangeroepen in plaats van bij elk verzoek wanneer gebruikers de pagina bekijken of verversen.
- **Efficiënt geheugengebruik:** Eén cache voor alle gebruikers, niet per gebruiker.
- **Voorkomen van dubbele berekeningen:** Monitor Object pattern zorgt ervoor dat dat er niet meerdere threads tegelijk dezelfde statistieken berekenen.

8.4 Structural Pattern

Wat is een structural pattern

Een structural pattern beschrijft hoe objecten en klassen gecombineerd kunnen worden tot grotere, beter georganiseerde structuren. Zo'n patroon helpt om systemen te bouwen die niet alleen duidelijk zijn, maar ook flexibel en efficiënt wanneer ze worden uitgebreid of aangepast.

Toepassing 1 (Animal eigenschappen)

De applicatie heeft verschillende Animals. Sommige zijn getraind, sommige hebben medische zorg nodig en sommige hebben beide of geen van beide nodig. In plaats van verschillende classes te maken voor elke combinatie, wordt er een decorator pattern gebruikt om deze kenmerken als lagen rond het basisdier te wikkelen (wrappen).

Uitleg toepassing 1:

1. Base interface

Hierin worden de basisafspraken die alle onderdelen in ons decorator-patroon moeten volgen, wat in dit geval GetCharacteristics is.

```
// Base interface
7 references
public interface IAnimalCharacteristic
{
    8 references
    string GetCharacteristics();
}
```

2. Base implementatie

Vervolgens is er een basiscomponent dat de IAnimalCharacteristic interface implementeert. Het slaat het meegegeven dier op en geeft via GetCharacteristics() de grootte van het dier terug, waarbij de methode virtueel is zodat deze kan worden aangepast in latere decorators.

```
// Concrete base implementation
2 references
public class BaseAnimalCharacteristic : IAnimalCharacteristic
{
    private readonly Animal _animal;

    1 reference
    public BaseAnimalCharacteristic(Animal animal)
    {
        _animal = animal;
    }

    5 references
    public virtual string GetCharacteristics()
    {
        return $"{_animal.Size}";
    }
}
```

3. Base decorator

Een wrapper class die elke kenmerkbeschrijving van het dier kan vasthouden en verzoeken eraan kan doorgeven.

```
// Base decorator class
5 references
public abstract class CharacteristicDecorator : IAnimalCharacteristic
{
    protected readonly IAnimalCharacteristic _characteristic;

    2 references
    public CharacteristicDecorator(IAnimalCharacteristic characteristic)
    {
        _characteristic = characteristic;
    }

    7 references
    public virtual string GetCharacteristics()
    {
        return _characteristic.GetCharacteristics();
    }
}
```

4. TrainedDecorator en MedicalDecorator

Voegt "getraind voor shows"/"heeft speciale medische zorg nodig" toe aan de kenmerken van het dier.

```
// Concrete decorators
2 references
public class TrainedDecorator : CharacteristicDecorator
{
    1 reference
    public TrainedDecorator(IAnimalCharacteristic characteristic) : base(characteristic) { }

    6 references
    public override string GetCharacteristics()
    {
        return $"{_characteristic.GetCharacteristics()}, trained for shows";
    }
}

2 references
public class MedicalDecorator : CharacteristicDecorator
{
    1 reference
    public MedicalDecorator(IAnimalCharacteristic characteristic) : base(characteristic) { }

    6 references
    public override string GetCharacteristics()
    {
        return $"{_characteristic.GetCharacteristics()}, needs special medical care";
    }
}
```

5. Manager (beheerder)

De AnimalCharacteristicDecorator class beheert het proces van de decorator. Het begint met de basiskekenmerken van het dier (de grootte) en voegt daarna de eigenschappen toe, afhankelijk van het dier, met behulp van de juiste decorators.

```
// Manager class
1 reference
public static class AnimalCharacteristicDecorator
{
    1 reference
    public static string GetDecoratedCharacteristics(Animal animal)
    {
        // Start with the base characteristics
        IAnimalCharacteristic characteristic = new BaseAnimalCharacteristic(animal);

        // Apply decorators based on animal properties
        if (animal.SecurityRequirement == Animal.AnimalSecurityRequirement.Critical)
        {
            characteristic = new MedicalDecorator(characteristic);
        }

        if (animal.Size == Animal.AnimalSize.Large)
        {
            characteristic = new TrainedDecorator(characteristic);
        }

        // Return the decorated characteristics
        return characteristic.GetCharacteristics();
    }
}
```

6. View details weergeven

Tot slot moet het nog toegevoegd worden aan de views details pagina van een dier zodat de gebruiker het kan zien.

```
</dd>
<!-- Characteristics section -->
<dt class="col-sm-6">
    Special Characteristics:
</dt>
<dd class="col-sm-6">
    @AnimalCharacteristicDecorator.GetDecoratedCharacteristics(Model)
</dd>
```

Details of Lion

Name:	Lion
Species:	Panthera leo
Category:	Mammal
Size:	Large
DietaryClass:	Carnivore
ActivityPattern:	Cathemeral
Prey:	Zebra, Wildebeest, Antelope
Enclosure:	Savanna
SpaceRequirement:	1000
SecurityRequirement:	Critical
Stall:	Sadek
Special Characteristics:	Large, needs special medical care, trained for shows

Behavior

Active during both day and night, alternating periods of activity.

[Edit](#) | [Back to List](#)



Voordelen van dit patroon:

- **Flexibiliteit:** functionaliteiten (kenmerken) kunnen worden toegevoegd zonder nieuwe classes te creëren.
- **Eenvoudig te begrijpen:** Elke decorator voert één eenvoudige functionaliteit uit, waardoor de code gemakkelijker te begrijpen is.

Toepassing 2 (Stall + Animals)

De applicatie maakt gebruik van het **Composite Pattern** om dieren (Animal) en stallen (Stall) als één geheel te behandelen via een gedeelde interface (IZooComponent). Dit maakt het mogelijk om bijvoorbeeld de benodigde ruimte van een volledige stal inclusief dieren op te vragen, of alle informatie van een stal inclusief dierdetails in één overzicht te tonen.

Uitleg toepassing 2:

1. Interface (IZooComponent)

De interface IZooComponent definieert twee methodes:

```
string GetInfo();  
double GetSpace();
```

Deze methodes worden geïmplementeerd door zowel Animal als Stall.

2. Leaf component (Animal)

De Animal klasse implementeert IZooComponent.

- GetInfo() retourneert de naam, soort en ruimte van het dier.
- GetSpace() geeft de benodigde ruimte terug.
Zo kan elk dier zelfstandig informatie geven over zichzelf.

3. Composite component (Stall)

De Stall klasse bevat een lijst van Animals (componenten) en implementeert ook IZooComponent.

- GetInfo() geeft eerst info over de stal, gevolgd door de info van elk dier in die stal.
- GetSpace() telt alle ruimtevereisten van de dieren op en geeft zo de totale ruimte terug.
Hierdoor kunnen stallen genest worden, of in de toekomst zelfs andere structuren bevatten.

4. Weergave in de View

In de Stall Details View wordt de composite-informatie getoond via:

```
@ViewData["CompositeInfo"]
```

Dit toont automatisch een netjes gegenereerde lijst met alle dieren in de stal en hun eigenschappen.

```
Stall: Sadek (Forest)
Animals:
Animal: Darrick - Species: Elephant - Space: 1.97 m2
Animal: Stacy - Species: Zebra - Space: 3.89 m2
Animal: Maria - Species: Giraffe - Space: 2.9 m2
Animal: Whitney - Species: Lion - Space: 2.82 m2
Animal: Donnie - Species: Lion - Space: 1.73 m2
Animal: Shanelle - Species: Zebra - Space: 2.37 m2
Animal: Cicero - Species: Elephant - Space: 4.23 m2
Animal: Kiera - Species: Tiger - Space: 3.56 m2
Animal: Mandy - Species: Elephant - Space: 1.19 m2
```

Voordelen van dit patroon:

- **Herbruikbaarheid:** Stallen en dieren gebruiken dezelfde interface en logica.
- **Schaalbaarheid:** In de toekomst kunnen stallen bestaan uit sub-stallen of andere structuren zonder extra complexiteit.
- **Eén centrale structuur:** Alle ruimteberekeningen en informatie kunnen vanuit één punt worden aangeroepen.
- **Toekomstbestendig:** Nieuwe types componenten kunnen eenvoudig worden toegevoegd (bijv. nestplaatsen, voederzones).

9. Bronnenlijst

Maheshmaddi. (2023, 12 april). 5.2. Monitor Object - maheshmaddi - medium. *Medium*. <https://medium.com/@maheshmaddi92/5-2-monitor-object-6ed26a1b1fa4>

Refactoring.Guru. (z.d.). *Classification of patterns*. <https://refactoring.guru/design-patterns/classification>

Refactoring.Guru. (2025a, januari 1). *Composite*. <https://refactoring.guru/design-patterns/composite>

Refactoring.Guru. (2025b, januari 1). *Decorator*. <https://refactoring.guru/design-patterns/decorator>

Refactoring.Guru. (2025c, januari 1). *Factory method*. <https://refactoring.guru/design-patterns/factory-method>

Refactoring.Guru. (2025d, januari 1). *Strategy*. <https://refactoring.guru/design-patterns/strategy>

UML Class Diagram Tutorial. (z.d.). <https://www.visual-paradigm.com/guide/uml-unified-modeling-language/uml-class-diagram-tutorial/>

UML Sequence Diagram Tutorial. (z.d.). Lucidchart. <https://www.lucidchart.com/pages/uml-sequence-diagram>

UML Use Case Diagram Tutorial. (z.d.). Lucidchart. <https://www.lucidchart.com/pages/uml-use-case-diagram>

What is Activity Diagram? (z.d.). <https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-activity-diagram/>

What is an Entity Relationship Diagram (ERD)? (z.d.). Lucidchart. <https://www.lucidchart.com/pages/er-diagrams>