Rapportage project vossen en konijnen





Inhoudsopgave

[Inleiding 3](#_Toc410121101)

[1. Probleemstelling 3](#_Toc410121102)

[1.1 Reden van ontwikkeling 3](#_Toc410121103)

[1.2 Geschikt voor deze personen/organistaties 3](#_Toc410121104)

[2. Analyse van huidige situatie 3](#_Toc410121105)

[2.1 Beperkingen 3](#_Toc410121106)

[2.2 Voorgestelde oplossingen 4](#_Toc410121107)

[2.3 Beschikbare gegevens van huidige situatie 4](#_Toc410121108)

[2.4 Varkenscyclus 5](#_Toc410121109)

[3. Uitbreiding 1 5](#_Toc410121110)

[3.1 Veranderingen 5](#_Toc410121111)

[3.2 Klassendiagram oude situatie 6](#_Toc410121112)

[3.3 Sequencediagram 6](#_Toc410121113)

[3.4 Klassendiagram nieuwe sitatie 7](#_Toc410121114)

[4. Het gebruik van MVC in V&K 7](#_Toc410121115)

[4.1 Overeenkomsten en verschillen mvc-projecten 7](#_Toc410121116)

[4.1.1 Overeenkomsten en verschillen tussen MVCDynamicModelThread en -ThreadGeneralized 7](#_Toc410121117)

[4.1.2 Overeenkomsten en verschillen tussen MVCDynamicModelThreadGeneralized en Life 7](#_Toc410121118)

[4.2 Klassendiagram 8](#_Toc410121119)

[4.3 Sequencediagram 8](#_Toc410121120)

[5. Uitbreiding 2 8](#_Toc410121121)

[5.1 Nieuwe toevoegingen 8](#_Toc410121122)

[5.1.1 Gras 8](#_Toc410121123)

# Inleiding

Dit rapport is geschreven om de ontwikkeling van de applicatie in beeld te brengen. Hierin worden een aantal aspecten behandeld, onder andere waarom de applicatie is gebouwd, voor wie de applicatie is gebouwd en welke veranderingen er zijn toegepast.

De applicatie die ontwikkeld gaat worden is een simulatie om dierenpopulaties in bij te houden. De daadwerkelijke reden van ontwikkeling wordt behandeld in paraaf 1.1.

De applicatie zoals is gegeven door de opdrachtgever bevat een aantal tekortkomingen. De oplossingen die hiervoor gemaakt zijn worden ook gemotiveerd in dit rapport. Deze worden eerst algemeen behandeld en vervolgens worden de uitbreidingen per week besproken.

# 1. Probleemstelling

In dit hoofdstuk wordt behandeld voor wie, en waarom deze applicatie is ontwikkeld. Verder wordt er gekeken naar de gegevens er nu beschikbaar zijn.

## 1.1 Reden van ontwikkeling

Vanuit Staatsbosbeheer is er een opdracht gekomen om een applicatie te ontwikkelen die het verloop van een aantal dierenpopulaties kan bijhouden. De gedachte hierachter is dat Staatsbosbeheer op deze manier gemakkelijk en tijdig kan bepalen of er van een bepaalde populatie dieren moeten worden bijgezet, of juist moeten worden weggehaald.

Omdat het hier niet om vaste gegevens gaat wordt het mogelijk gemaakt voor Staatbosbeheer om zelf de gegevens aan te passen. Denk hierbij aan het aantal dieren en welke populaties er in de simulatie moeten worden meegenomen.

## 1.2 Geschikt voor deze personen/organistaties

Deze applicatie wordt ontwikkeld voor Staatsbosbeheer. Deze applicatie beperkt zich niet alleen voor gebruik bij Staatsbosbeheer. Iedereen die interesse heeft in het verloop van dierenpopulaties kan hier gebruik van maken. Door de zwakke afhankelijkheid van de code is het makkelijk om de dieren die standaard geleverd worden te wijzigen of meer dieren toe te voegen.

# 2. Analyse van huidige situatie

In dit hoofdstuk wordt gekeken naar de huidige situatie. Al snel valt te zien dat de huidige situatie een aantal beperkingen heeft. Aan de hand van deze beperkingen is het mogelijk om de applicatie verder uit te breiden. De voorbeelden die genoemd gaan worden, worden niet per definitie geïmplementeerd.

## 2.1 Beperkingen

Bij de huidige situatie zijn er een aantal gebreken. Ten eerste is de functionaliteit van de simulatie zeer beperkt. Er zijn geen instellingen mogelijk waardoor het niet mogelijk is om zelf voorwaarden te bepalen. De simulatie is eentonig en biedt geen inzicht in wat er zich precies afspeelt. Iedere run van de simulatie geeft hetzelfde resultaat (zowel in aantallen als de positie van de dieren). Dit is niet realistisch voor een simulatie wat zich afspeelt in de natuur.

Daarnaast is de code ook niet echt optimaal. Zo is het in de huidige situatie moeilijk om extra’s toe te voegen. De code is nu zo opgebouwd dat alle klassen van elkaar afhankelijk zijn. Dit resulteert in het feit dat een kleine verbetering al heel veel moeite kost om te implementeren.

Ook is de GUI aan de matige kant. Zo valt op dit moment de simulatie niet te stoppen of te resetten. Hierdoor vertoont de applicatie soms gedrag dat niet wenselijk is.

## 2.2 Voorgestelde oplossingen

De in paragraaf 2.1 gestelde beperkingen kunnen met simpele theoretische oplossingen worden aangepakt. Het is mogelijk om te zorgen dat er een aantal eigenschappen instelbaar worden. Een aantal voorbeelden hiervan zijn:

* De maximale leeftijd van dieren veranderen
* Een minimale leeftijd waarop een dier zich kan voortplanten veranderen
* De dierenpopulaties aanpassen

Om meer inzicht te krijgen in de data is het mogelijk gebruik te maken van grafieken en diagrammen. Zo is het makkelijker om de gegevens te gebruiken voor een onderzoek, omdat eventuele trends of afwijkingen dan beter zijn op te sporen. Daarnaast is het ook mogelijk om gebruik te maken van statistieken om meer inzicht te krijgen in de data.

Doordat de code een hoge afhankelijkheid heeft moet de code gerefactord worden. Door de code te refactorren wordt het makkelijker om uitbreidingen toe te gaan voegen. Daarnaast zou de code ook veel beter leesbaar moeten worden.

Voor de GUI is er een hele simpele oplossing. Door knoppen toe te voegen en hier acties aan te verbinden krijgt de eindgebruiker al veel meer invloed op hoe de applicatie werkt.

## 2.3 Beschikbare gegevens van huidige situatie

Omdat het in de huidige situatie nog niet mogelijk is om een geautomatiseerde analyse uit te voeren, wordt dit nu handmatig gedaan. De volgende grafiek geeft een beeld over de verhoudingen na 2000 stappen.

Verticale as zijn de aantallen, en de horizontale as zijn de stappen

De dierenpopulaties zijn over het algemeen stabiel te noemen. Bij de konijnenpopulaties is het duidelijk te zien dat deze steeds op en neer gaan en eigenlijk nooit onder de 2000 uitkomen. Bij de vossen ligt dit iets anders. Doordat de vossen sowieso al met een veel kleinere populatie te maken hebben, zijn de gegevens per 100 stappen niet bepaald representatief. In de bovenstaande grafiek is er namelijk geen duidelijk verband te zien tussen de populatie van vossen en konijnen.

Deze is er echter wel. In paragraaf 2.4 wordt dit uitgelegd aan de hand van een voorbeeld en de reden waarom dit zo is.

## 2.4 Varkenscyclus

Een vraag over deze gegevens zou kunnen zijn: “Is er hier sprake van een varkenscyclus?” Om deze vraag goed te beantwoorden is het eerst van belang om uit te leggen wat een varkenscyclus is.

Een varkenscyclus is een verschijnsel dat voorkomt in de economie. Dit verschijnsel geeft aan dat een overschot van een bepaald product afgewisseld wordt met een tekort, en weer vice versa. Op dit moment speelt er een heel goed praktijkvoorbeeld. De olieprijs wordt steeds lager. Dit is te danken aan het feit dat de olieprijs enkele maanden geleden vrij hoog was. Door de hoge olieprijs wordt er meer olie opgeboord in de hoop een graantje mee te pikken van de hoge prijzen. Na verloop van tijd komt er zoveel olie uit de grond dat er een overschot ontstaat waardoor deze prijzen weer dalen. Vervolgens gaat men minder boren en ontstaat er weer een tekort, waardoor de prijzen juist weer omhoog gaan. (Sijs, 2010)

Het hierboven beschreven verschijnsel is ook van toepassing op de simulatie. Zodra het aantal vossen hoger word, is er een daling te zien in het aantal konijnen. Als het aantal konijnen hoog is, zijn er minder vossen.

# 3. Uitbreiding 1

In dit hoofdstuk worden de veranderingen besproken die in week 2 van het project zijn gemaakt. Er valt in de klassendiagrammen te zien hoe de oude situatie is, en hoe de nieuwe situatie is geworden.

## 3.1 Veranderingen

Tijdens de 2de week van de ontwikkeling zijn er een aantal verbetering toegevoegd. Er is een nieuwe diersoort toegevoegd, namelijk de lynx. De lynx is een diersoort dat jaagt op vossen en konijnen. De eigenschappen van een lynx zijn strenger dan van de vos, dit is gedaan omdat een lynx makkelijker aan voedsel kan komen. In de loop der worden er nog uitbreidingen toegevoegd waarmee het mogelijk wordt om deze eigenschappen zelf in te stellen.

package vossen\_en\_konijnen**;**

**import** java**.**util**.**List**;**

/\*\*

\* De interface die uitgebreid kan worden door elke klasse

\* die deel wil nemen aan de simulatie

\*/

public interface Actor

**{**

/\*\*

\* Voer het gebruikelijke gedrag van de deelnemer uit.

\* @param newActors Een lijst waarin zojuist gemaakte

\* deelnemers worden opgelsagen.

\*/

void act**(**List**<**Actor**>** newActors**);**

/\*\*

\* Is de deelnemer nog steeds actief?

\* @return true als de deelnemer nog actief, anders false.

\*/

boolean isActive**();**

**}**

Verder is er nog een jager toegevoegd. Een jager is vrij simpel van opzet. Hij jaagt op alle dieren en gaat niet dood. Er is ook geen limiet voordat een jager een volgend dier doodschiet.

Verder is er nog een interface aangemaakt genaamd Actor. Deze interface bevat alle dieren, objecten en personen die aanwezig zijn in de simulatie. Door het aanmaken van deze interface is ervoor gezorgd dat alles nu in een lijst staat. Het was ook mogelijk om dit per groep te doen, maar dit zou de code nodeloos ingewikkeld maken. Hierdoor wordt het een stuk moeilijker om andere functionaliteit toe te voegen. De code van de interface Actor staat hierboven.

De code van de nieuwe interface Actor

Ook in de abstracte klasse Animal zijn wijzigen aangebracht. Een aantal methodes die bij ieder dier opnieuw gedeclareerd stonden zijn nu verplaatst naar de klasse Animal. Hierdoor is de code beter leesbaar en een stuk onafhankelijker geworden.

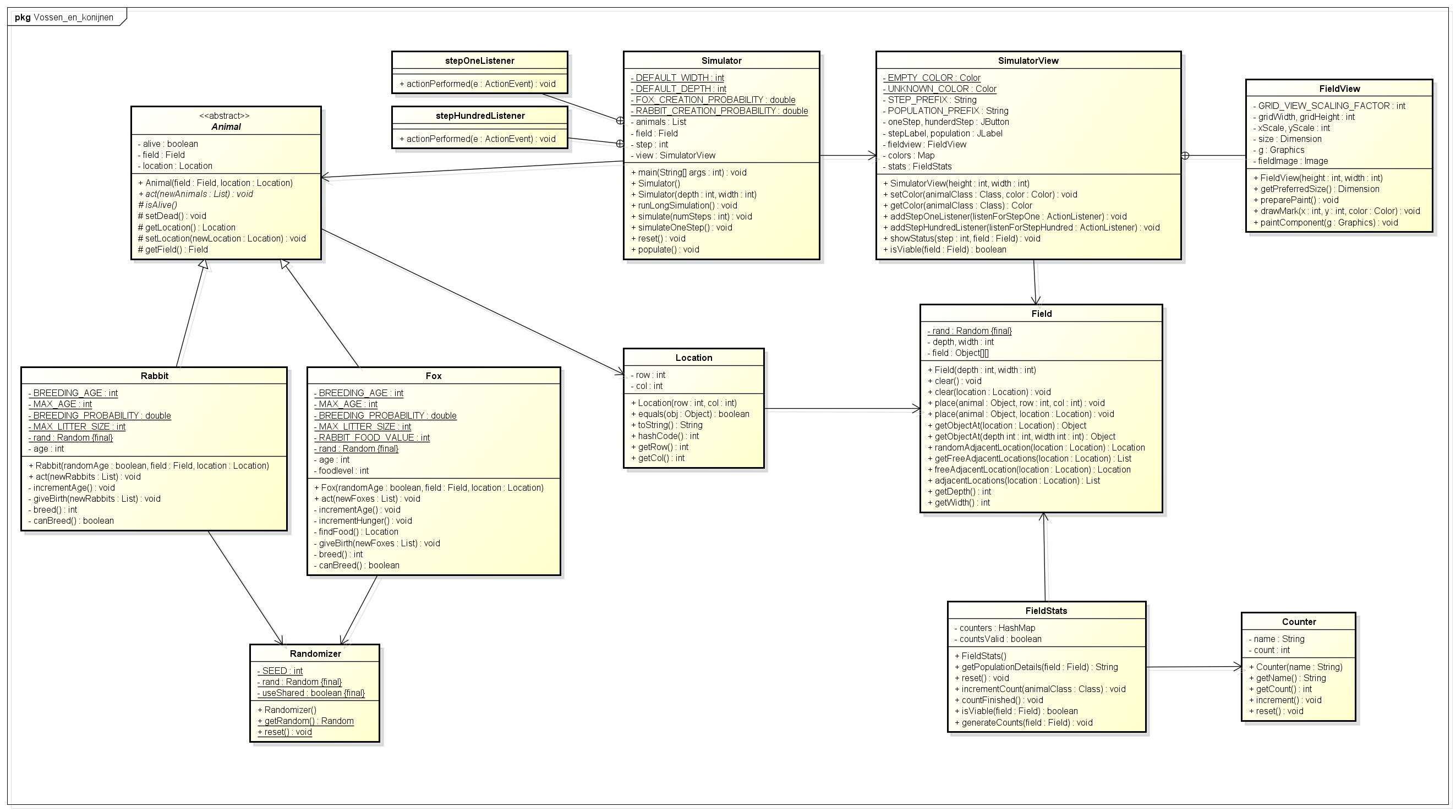
Deze week zijn er ook een aantal views toegevoegd. Deze views omvatten een aantal diagrammen om de gegevens beter mee te kunnen analyseren. De volgende diagrammen zijn toegevoegd:

* Lijndiagram
* Staafdiagram
* Cirkeldiagram

De diagrammen geven de huidige populaties van de dieren weer. Een lijndiagram is een kleine uitzondering, deze houdt ook de populaties bij van de vorige honderd stappen.

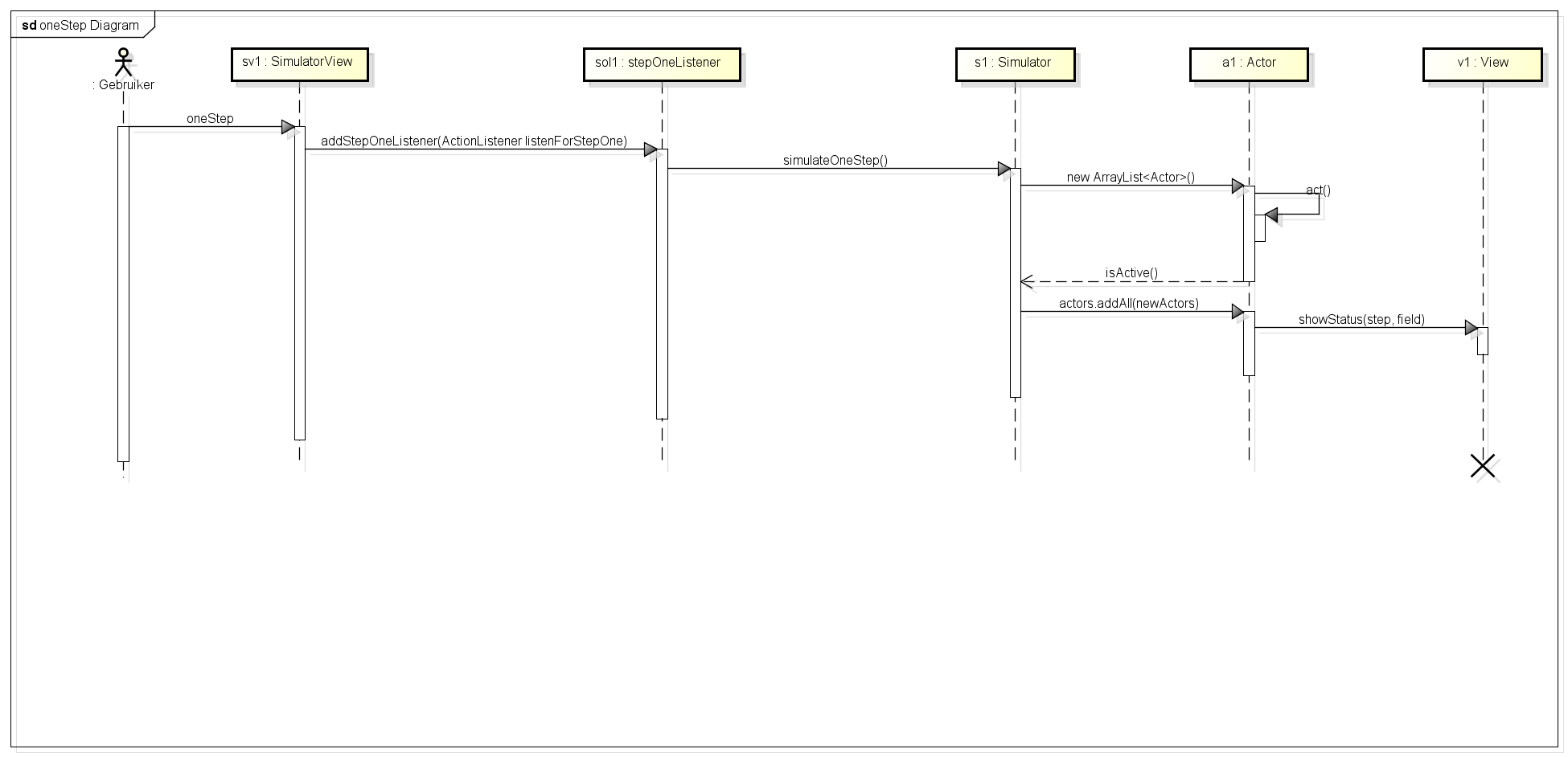
Ten slotte zijn er op diverse plekken kleine wijzigen doorgevoerd. Deze zijn echter qua inhoud niet de moeite waard om afzonderlijk te bespreken.

## 3.2 Klassendiagram oude situatie



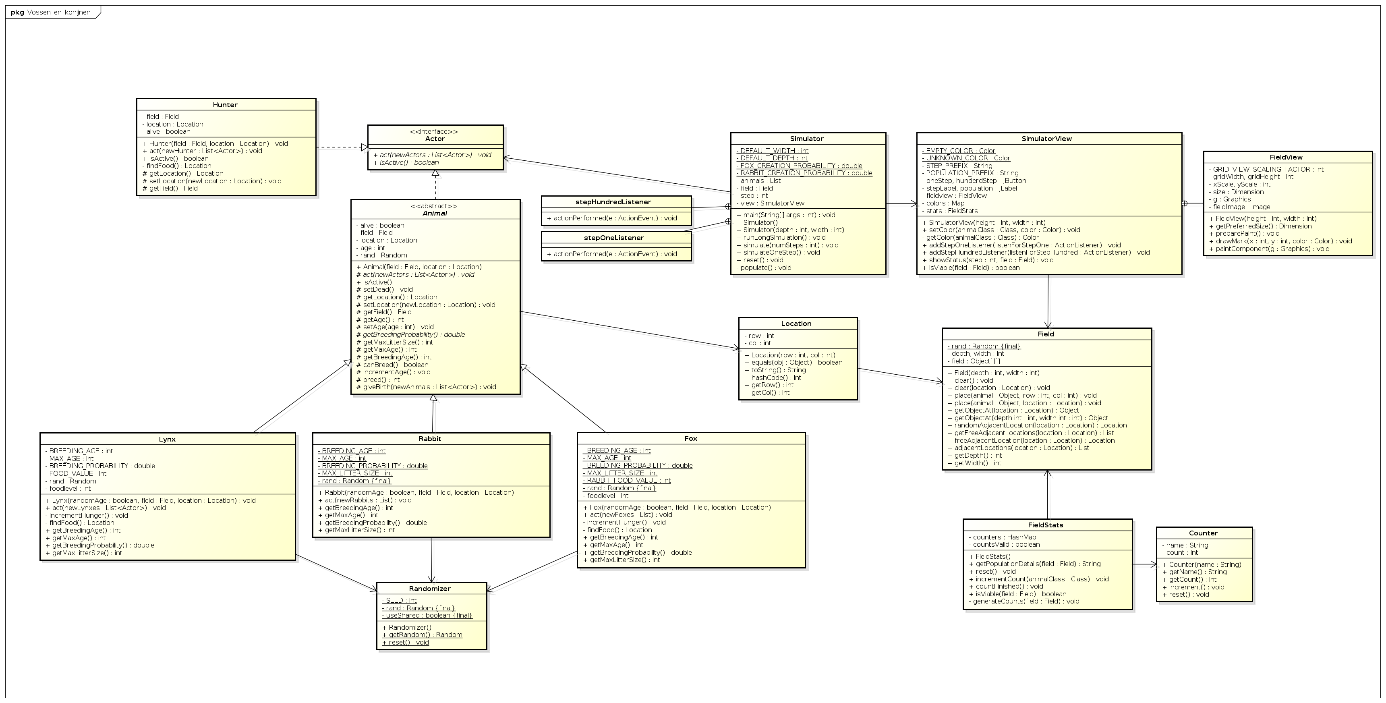
Een klassendiagram van de eerste versie van de applicatie

## 3.3 Sequencediagram



Een sequence diagram van 1 stap in de simulatie

## 3.4 Klassendiagram nieuwe sitatie



Een klassendiagram van de nieuwe situatie, met nieuwe verbeteringen

# 4. Het gebruik van MVC in V&K

## 4.1 Overeenkomsten en verschillen mvc-projecten

### 4.1.1 Overeenkomsten en verschillen tussen MVCDynamicModelThread en -ThreadGeneralized

De projecten MVCDynamicModelThread en -ThreadGeneralized zijn voor het grootste gedeelte aan elkaar gelijk. Beide gebruiken de klassen Controller, CountView, Model, MVCDynamicModelThread (en Generalized), MVCRunner en PieView. De code van deze klassen is dan ook nagenoeg identiek aan elkaar.

Eén verschil is dat ThreadGeneralized abstracte klassen heeft waar zoveel mogelijk van de MVC-aspecten in zitten. Dit zijn dus de klassen AbstractView, -Controller en -Model. Het voordeel hiervan is dat het een stuk gemakkelijker wordt om meer klassen toe te voegen. Er kan bijvoorbeeld nog een View-klasse toegevoegd worden die van AbstractView erft.

Een ander pluspunt is dat de code duidelijker te volgen is en beter is georganiseerd. Het project ModelThread heeft geen aparte abstracte klassen voor MVC. De MVC-aspecten van bijvoorbeeld de ontbrekende AbstractController zitten daar in de klasse Controller. Het kost in zo'n geval meer tijd om wijzigingen aan te brengen in de code.

Een ander verschil is dat de klassen in verschillende packages zijn geordend. De klassen AbstractModel en Model (overerving) zitten in de package nl.hanze.t12.mvc.controller. De folderstructuur is dus nl/hanze/t12/mvc/controller.

### 4.1.2 Overeenkomsten en verschillen tussen MVCDynamicModelThreadGeneralized en Life

Het Project Life volgt dezelfde structuur voor MVC als ThreadGeneralized. Life heeft dus aparte abstracte klassen voor Model, View, en Controller die ook zijn geordend in packages. Een andere overeenkomst is dat ze beide een losstaande main-package hebben, waar de GUI gevormd wordt van de klassen uit de view-package. Ook is er een package voor de runner, waar alleen de main methode zich bevindt.

Een verschil tussen de twee projecten is dat Life in totaal 3 controllers heeft, inclusief een abstracte klasse. ThreadGeneralized heeft er 2; AbstractController en Controller. Life gebruikt het principe van het uitbreiden van de abstracte MVC-klassen dus al meer dan ThreadGeneralized.

## 4.2 Klassendiagram

## 4.3 Sequencediagram

# 5. Uitbreiding 2

In dit hoofdstuk worden de wijzigingen doorgenomen die in de laatste week van de ontwikkeling van het project zijn toegevoegd.

## 5.1 Nieuwe toevoegingen

### 5.1.1 Gras

Een van de nieuwe toevoegingen voor deze week is de klasse Grass. De implementatie van Grass dient als voedsel voor konijnen. Gras ‘plant’ zich voort net zoals dieren zich voortplanten. Dit is gedaan om alle andere populaties in stand te houden. Een veld waar gras in voorkomt kan namelijk geen dieren bevatten. Als gras dus niet zou doodgaan is er geen plek meer voor de dieren, wat de simulatie niet ten goede komt.

### 5.1.2 Ziekte

Voor deze week is er ook een ziekte toegevoegd dat konijnen infecteert. In het begin van de simulatie wordt bepaald of een konijn vatbaar is voor de ziekte. De kans dat een konijn vatbaar is voor de ziekte is ongeveer 90%.

/\*\*

\* Look for rabbits adjacent to the current location.

\* Only the first live rabbit is eaten.

\* @return Where food was found, or null if it wasn't.

\*/

private Location findFood**()**

**{**

Field field **=** getField**();**

List**<**Location**>** adjacent **=** field**.**adjacentLocations**(**getLocation**());**

Iterator**<**Location**>** it **=** adjacent**.**iterator**();**

Location goTo **=** **null;**

**while(**it**.**hasNext**())** **{**

Location where **=** it**.**next**();**

Object animal **=** field**.**getObjectAt**(**where**);**

**if(**animal **instanceof** Grass**)** **{**

Grass grass **=** **(**Grass**)** animal**;**

**if(**grass**.**isActive**()** **&&** goTo **==** **null)** **{**

grass**.**setDead**();**

setFoodLevel**(**GRASS\_FOOD\_VALUE**);**

// Remove the dead rabbit from the field.

goTo **=** where**;**

**}**

**}**

**else** **if(**animal **instanceof** Rabbit**)** **{**

Rabbit r **=** **(**Rabbit**)** animal**;**

**if(**getZiekte**()** **&&** r**.**getZiekteGen**())** **{**

r**.**setZiekte**(true);**

**}**

**}**

**}**

**return** goTo**;**

**}**

Zodra de ziekte wordt geactiveerd, worden 10% van de konijnen die vatbaar zijn voor de ziekte ziek. Deze ziekte verspreid zich vervolgens over de andere konijnen die vatbaar zijn. Als een konijn in de buurt komt van een andere ziek konijn wordt deze ziek. In **figuur XX** is de code te zien waarin een konijn besmet wordt.

De code waar een konijn een ander konijn besmet

Als een konijn ziek is, veranderd de kleur van het konijnen van geel naar oranje. Zo is het makkelijk te zien welke konijnen met de ziekte besmet zijn. Een ziek konijn kan nog vijf stappen lopen voordat het dood gaat.