|  |  |
| --- | --- |
| Vossen & Konijnen  Onderzoeksrapport | A. Elsinga  K. Haitsma  R. Elzen  J. Sikkes  M. Nijboer |

|  |  |
| --- | --- |
| Vossen & Konijnen  Onderzoeksrapport | 01-02-2015  A. Elsinga  K. Haitsma  R. Elzen  J. Sikkes  M. Nijboer  Klas: PIV1C  Opdrachtgever: Hanzehogeschool Groningen |

Voorwoord

Dit rapport is geschreven in het kader van het project: “Vossen & Konijnen”. Het project is een onderdeel van het vak Java binnen het thema "De geprogrammeerde applicatie". De opdracht is het verbeteren en uitbreiden van de applicatie: “Vossen & Konijnen”.

Het rapport is bedoeld voor de opdrachtgever van het project en de Hanzehogeschool Groningen.

De applicatie is onvolledig aangeleverd en werkte niet. Daarom moeten wij als groep deze applicatie verbeteren, werkend maken en uitbreiden met eigen functies. Het doel van deze applicatie is de populatie van vossen en konijnen te monitoren. Uiteindelijk worden er meerdere functies toegevoegd die van invloed zijn op de populatie. Denk hierbij aan beren, jagers of de gekke konijnen ziekte.

Graag willen wij Dhr. Hekman bedanken voor zijn begeleiding tijdens dit project.

FOXRAB

Drachten, 01-02-2015

Samenvatting

Dit rapport bevat het onderzoek van de simulatie: “vossen en konijnen”. Het doel is het uitbreiden en werkend maken van de simulatie. Het probleem is dat de code onvolledig is aangeleverd waardoor de simulatie niet in huidige staat niet voldoet. Om het doel meetbaar te maken en om dit problemen op te lossen zijn de volgende vragen gesteld:

* Wat moet er veranderd worden aan code?
* Wat zijn de eisen van de opdrachtgever?
* Wat zijn de uitbreidingsmogelijkheden?

De aangeleverde code was onvolledig en vooral statisch. Dit is veranderd om het uitbreiden van de simulatie makkelijker te maken. Vervolgens zijn de eisen van de opdrachtgever toegevoegd. Dat zijn de GUI, een nieuwe diersoort, de jagers en het toevoegen van het konijnenvirus. Ook wordt de broedkans van de konijnen verkleind als er veel konijnen zijn.

Om de populatie te monitoren tijdens de simulatie zijn er schermen toegevoegd. Door de verschillende schermen is het monitoren van de populatie zeer overzichtelijk.

Als extra feature is er een configuratie scherm aangemaakt waar instellingen van de simulatie aangepast kunnen worden. Het veranderen van de instellingen heeft invloed op het verloop van de simulatie. Denk hierbij aan de kans dat een konijn voortplant of hoe lang een beer zonder voedsel kan.

Na het veranderen van de code en het toevoegen van de extra features werkt de simulatie uitstekend. De belangrijkste toevoegingen zijn de extra schermen om de populatie te monitoren en het configuratiescherm waar alle instellingen veranderd kunnen worden. Het doel van het onderzoek is bereikt.

Inhoudsopgave

[1. Inleiding 5](#_Toc410734961)

[2. Probleemanalyse 6](#_Toc410734962)

[2.1. Analyse 6](#_Toc410734963)

[2.2. Varkenscyclus 6](#_Toc410734964)

[2.3. Verbeterpunten 6](#_Toc410734965)

[2.4. Wensen opdrachtgever 6](#_Toc410734966)

[2.5. Uitbreidingsmogelijkheden 6](#_Toc410734967)

[3. Test Proces 7](#_Toc410734968)

[4. MVC 10](#_Toc410734969)

[4.1 Model View Controller 10](#_Toc410734970)

[4.1.1 Model 10](#_Toc410734971)

[4.1.2 View 10](#_Toc410734972)

[4.1.3 Controller 10](#_Toc410734973)

[4.1.4 SRC EN BIN 10](#_Toc410734974)

[4.1.5 MVC in Vossen & Konijnen 10](#_Toc410734975)

[5. GUI 11](#_Toc410734976)

[5.1 Uiterlijk 11](#_Toc410734977)

[5.2 Views 11](#_Toc410734978)

[5.2.1 Field view 12](#_Toc410734979)

[5.2.2 Tekst view 13](#_Toc410734980)

[5.2.3 Graph View 14](#_Toc410734981)

[6. Uitbreidingen en extra functies 15](#_Toc410734982)

[6.1 De Beer 15](#_Toc410734983)

[6.1.1 Functie van de beer 15](#_Toc410734984)

[6.2 De Jager 15](#_Toc410734985)

[6.2.1 Functie van de jager 15](#_Toc410734986)

[6.3 Het virus 15](#_Toc410734987)

[6.4 Instellingen 17](#_Toc410734988)

[7. Conclusie 20](#_Toc410734989)

[Literatuurlijst 22](#_Toc410734990)

[Bijlage A – UML Klassendiagram V1.0 23](#_Toc410734991)

[Bijlage B – UML Klassendiagram V2.0 24](#_Toc410734992)

[Bijlage C – Sequence Diagram (Step ++) 25](#_Toc410734993)

# 

# 1. Inleiding

De opdrachtgever Hanzehogeschool Groningen heeft de wens de applicatie van Vossen & Konijnen te verbeteren. De wens van de opdrachtgever is het verbeteren, werkend maken en uitbreiden met extra functies van de applicatie. Onder het verbeteren en werkend maken valt het verbeteren en aanmaken van Java code waardoor de applicatie begint te lopen en geen fouten meer weergeeft. Daarna is het uitbreiden van de applicatie aan de beurt waardoor er beren en jagers toegevoegd worden aan de applicatie. Uiteindelijk wordt er een GUI toegevoegd en wordt er een konijnen ziekte aangemaakt.

Het probleem is dat de Java code onvolledig is en hierdoor werkt de applicatie niet. De probleemstelling luidt: “Waarom werkt de applicatie niet en welke extra features moeten we toevoegen om aan de wens van de opdrachtgever te voldoen?”.   
Naast de probleemstelling zijn er een aantal vragen die gesteld moeten worden om het probleem op te lossen:

* Welke code is fout en welke oplossing is hier voor nodig?
* Welke extra features gaan er toegevoegd worden?

In het eerste hoofdstuk van dit onderzoeksrapport wordt er dieper ingegaan op de probleemstelling door middel van een probleemanalyse. Hoofdstuk twee beschrijft het test proces en de uitkomsten van deze processen. Daarna behandelt hoofdstuk drie de klassendiagrammen & sequence diagrammen. In hoofdstuk vier wordt gekeken naar de extra features die zijn toegevoegd om aan de wensen van de opdrachtgever te voldoen. Daarna wordt er gekeken naar de implementatie en vervolgens gaat het rapport door naar het refractoren. Tot slot eindigt het rapport met een conclusie en een slotadvies.

# 2. Probleemanalyse

In dit hoofdstuk wordt de probleemstelling bekeken door een analyse. En wordt er gekeken naar de verbeterpunten en de uitbreidingsmogelijkheden. Dit wordt gedaan om de werkzaamheden aan deze applicatie duidelijk te maken.

## 2.1. Analyse

De basisapplicatie heeft op dit moment het probleem dat een diersoort uitsterft op een bepaald punt. Het gevolg is een schommelend effect, omdat de verhouding tussen vossen en konijnen verandert. Konijnen zijn de prooi van de vossen. Te weinig vossen betekent een toename van de konijnen oftewel een overschot. Door het overschot aan konijnen hebben de vossen te veel eten en krijg je een toename in het aantal vossen. Hierdoor zijn er meer roofdieren en zal er meer gegeten worden waardoor er weer een afname ontstaat in het aantal konijnen. Ten slotte ontstaat dan een tekort aan voedsel en zullen er roofdieren dood gaan. Hierdoor begint de cyclus weer opnieuw.

## 2.2. Varkenscyclus

Op dit moment lijkt het er op dat de applicatie draait op de bekende varkenscyclus. Deze cyclus wisselt de overschotten en tekorten af van een bepaald product. Dit komt door de reactie op overschotten van de tekorten en wederzijds. Ook kun je het voorbeeld van werkgelegenheid en studiekeuze hiervoor nemen. Hoge werkloosheid onder een bepaalde afstudeerrichting betekent minder eerstejaars voor deze studie. Wales & Sanger, 2013

## 2.3. Verbeterpunten

Er zijn een aantal punten opgevallen tijdens de analyse van de bestaande applicatie. Ten eerste is er geen grafische interface beschikbaar. Ten tweede is er veel te veel code wat kleiner geschreven kan worden. Dit maakt het overzichtelijker om in te werken. Ten slotte zijn variabelen hard-coded. Dit moet dan ook veranderd worden.

## 2.4. Wensen opdrachtgever

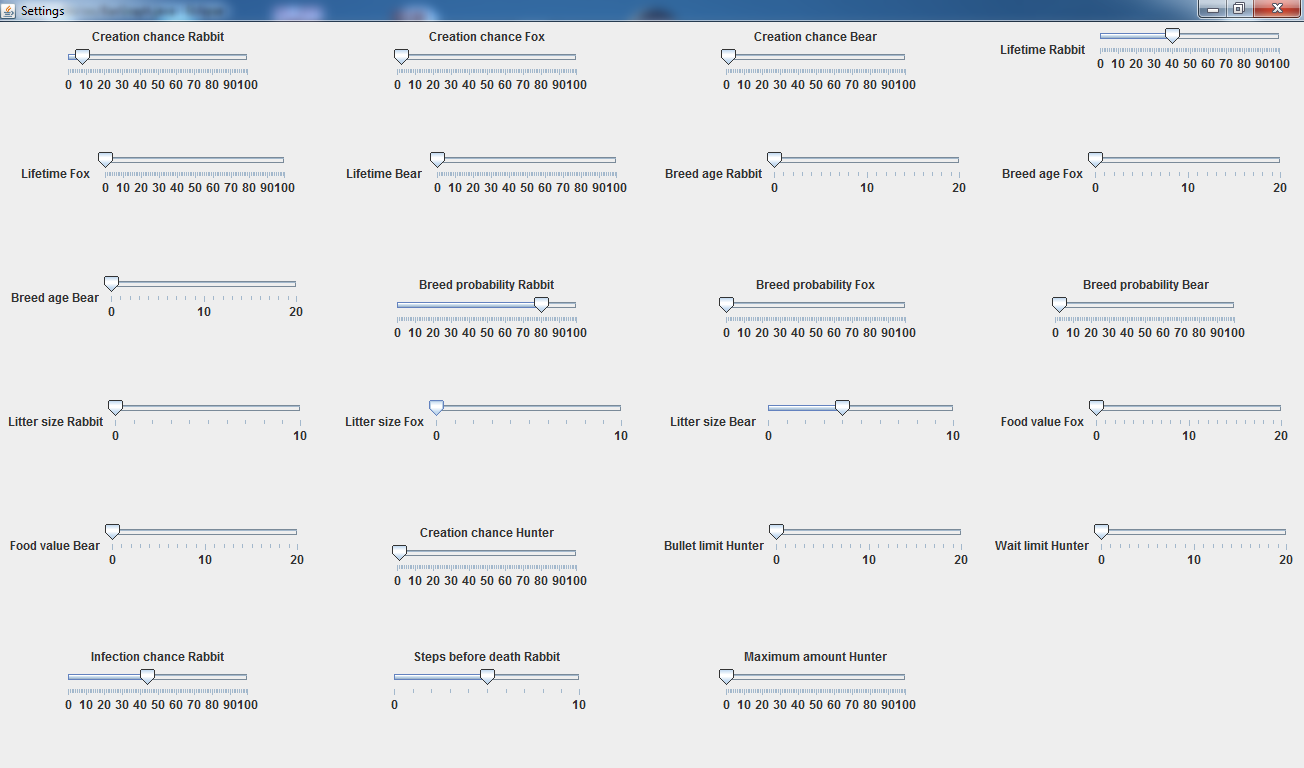
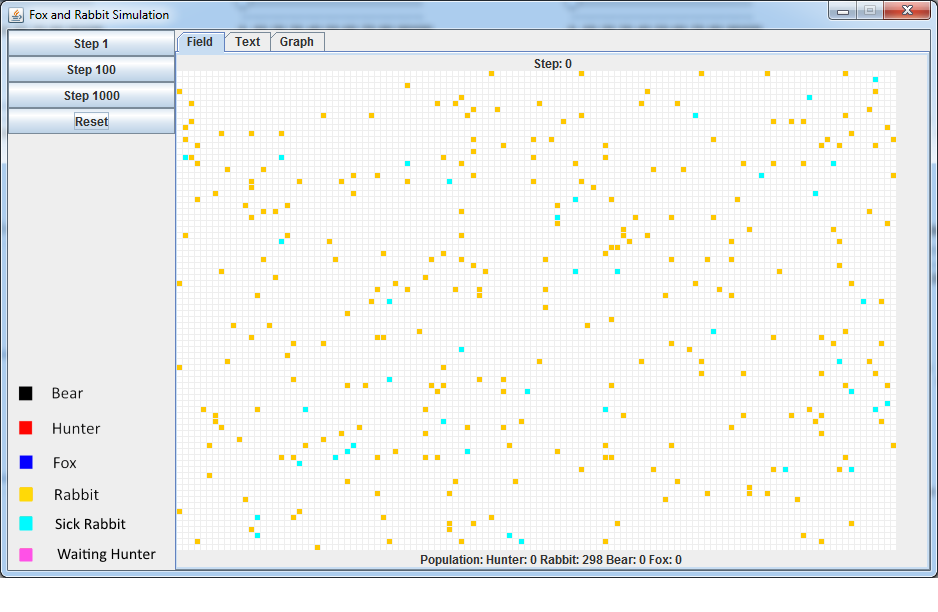
Voor de functionaliteit van de opdrachtgever zijn er twee verbeteringen die moeten worden toegevoegd. Als eerst moeten er beren toegevoegd worden aan de applicatie. Vervolgens worden er jagers toegevoegd die op alle diersoorten kunnen jagen. Daarnaast worden er nog plaatjes en geluiden toegevoegd om het spectaculairder te maken.

## 2.5. Uitbreidingsmogelijkheden

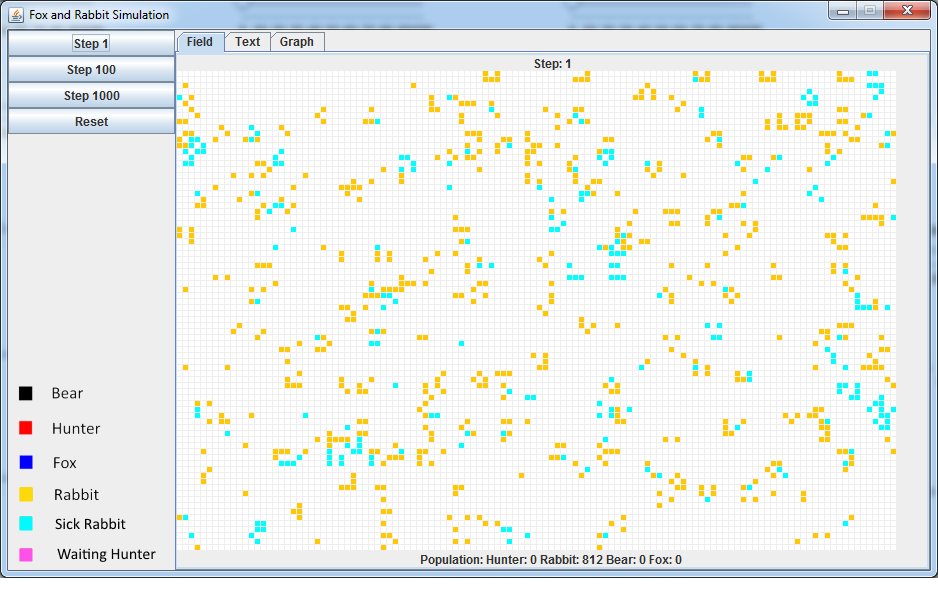
Er zijn veel mogelijkheden om deze applicatie uit te breiden. Bij het bedenken van deze mogelijkheden kwamen de volgende suggesties naar voren. Het toevoegen van een grafische omgeving voor het aanpassen van variabelen zoals levensduur van dieren en het aantal dieren bij het opstarten van de applicatie. Ook is mogelijk om de jagers een maximum aan munitie te geven waardoor deze na een tijdje niet meer kunnen jagen.

# 3. Test Proces

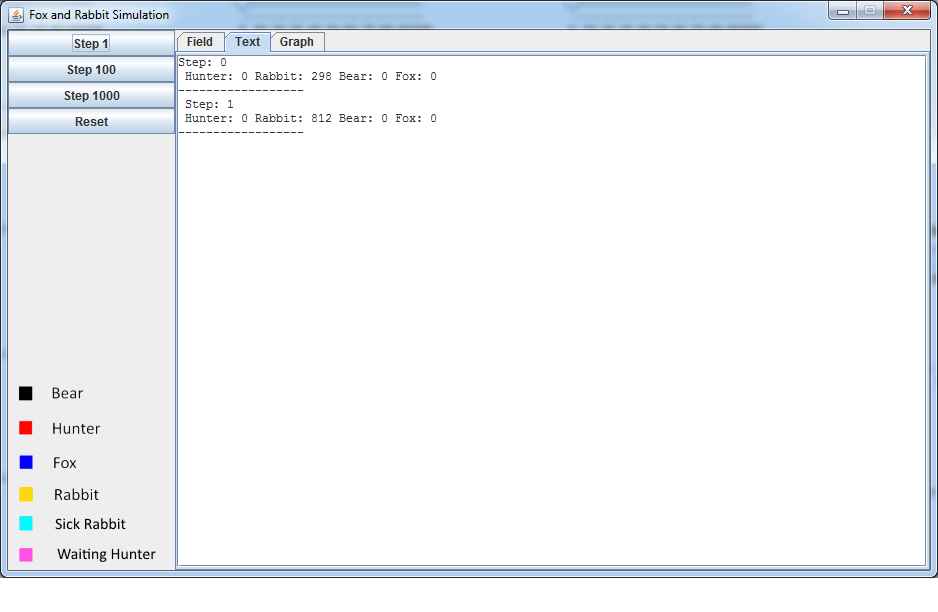
In dit hoofdstuk wordt er een test proces uitgevoerd om te kijken of het proces wat getest wordt werkt zoals het hoort. Er is gekozen om de test uit te voeren door middel van een black-box-test. Dit is overzichtelijk omdat hier gebruik wordt gemaakt van veel plaatjes. Hieronder word een reeks afbeeldingen getoond van de black-box-test die is uitgevoerd na de verbetering van de applicatie. In het begin staan er een aantal konijnen op het veld. Overige dieren zijn niet verwijderd, maar de kans op creatie is op 0 gezet. De ziekte is nog wel actief.

De test is begonnen met deze instellingen.   
  
Hier in het eerste veld wordt het begin van de situatie weergegeven. Er zijn 298 konijnen op het veld.

In het tweede veld, na 1 stap gedaan te hebben zijn er 812 konijnen op het veld.

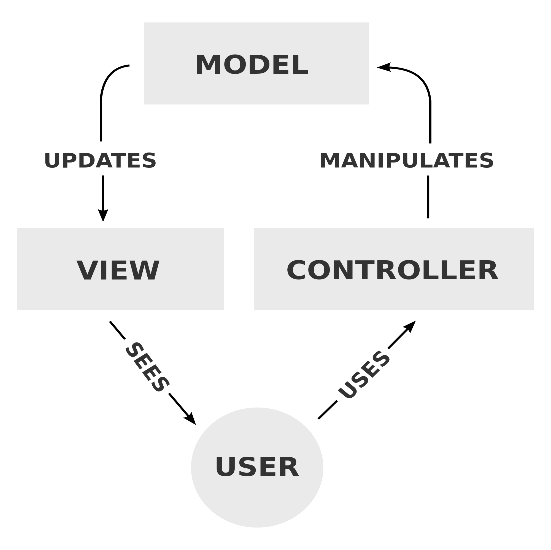


In dit tekstveld kun u dit goed zien. Dit wordt aangemaakt bij elke stap die wordt uitgevoerd.



Het resultaat is dat de konijnen zich hebben voortgeplant, de conclusie daarvan is dat de test is geslaagd.

# 4. MVC

In dit hoofdstuk wordt het gebruik van MVC uitgelegd en hoe dit wordt toegepast in de applicatie.

## 4.1 Model View Controller

MVC staat voor Model-View-controller. Dit houdt in dat het ontwerp van de toepassing, van applicatie wordt opgedeeld in 3 stukken. Dit bestaat uit het datamodel(model), datapresentatie(view) en de applicatielogica(controller).

De scheiding van deze 3 onderdelen zorgt er mede voor dat de code ook beter te lezen is en dat het bruikbaarder en overzichtelijker wordt, zo is hij later ook beter te hergebruiken.

### 4.1.1 Model

Een model is de motor achter de applicatie, waarin hij gerepresenteerd wordt. Een model zorgt ervoor dat data en logica een weg vinden met elkaar, oftewel er wordt een relatie gelegd tussen data en logica. Hij is ook de koppeling tussen de applicatie en de database.  
Per applicatie is er maar één model mogelijk.

### 4.1.2 View

Een view doet geen berekeningen of dergelijke. Deze zorgt er alleen maar voor dat de userinterface weergeven wordt. Hij brengt de gegevens vanuit het model, en zorgt dat deze correct wordt weergegeven. Je kan meerdere views van een enkele applicatie weergeven.

### 4.1.3 Controller

De controller verwerkt en reageert op events, die meestal het gevolg zijn van handelingen van de gebruiker. Als je op een knop drukt, geeft de controller het model een opdracht om te verzetten. Ook kun je meerdere controllers in een applicatie toevoegen.

### 4.1.4 SRC EN BIN

SRC is de afkorting van source - de map waar de broncode in staat.  
BIN is een afkorting van binairy - de map waar uw bestanden worden gecompileerd

### 4.1.5 MVC in Vossen & Konijnen

In het project hebben wij de MVC gebruikt om de klassen te ordenen, zo kun je nu makkelijker een stuk code hergebruiken, maar ook een extra view aan de applicatie toevoegen.

# 5. GUI

In dit hoofdstuk zal er worden gekeken naar de uitbreiding van de GUI. De GUI is het uiterlijk van de applicatie. Om het zo gebruikersvriendelijk te maken is de GUI overzichtelijk gemaakt met tabs en een legenda. De tabs worden uitgelegd in de volgende sub paragrafen.

## 5.1 Uiterlijk

Om de applicatie zo gebruikersvriendelijk mogelijk te maken is er gebruik gemaakt van ‘tabs’ in de applicatie. Door te switchen tussen de tabs krijgt u verschillende overzichten van de applicatie te zien. Op deze manier is het mogelijk de hoeveelheden van diersoorten, jagers en voedsel te monitoren tijdens de voortgang van de simulatie.

## 5.2 Views

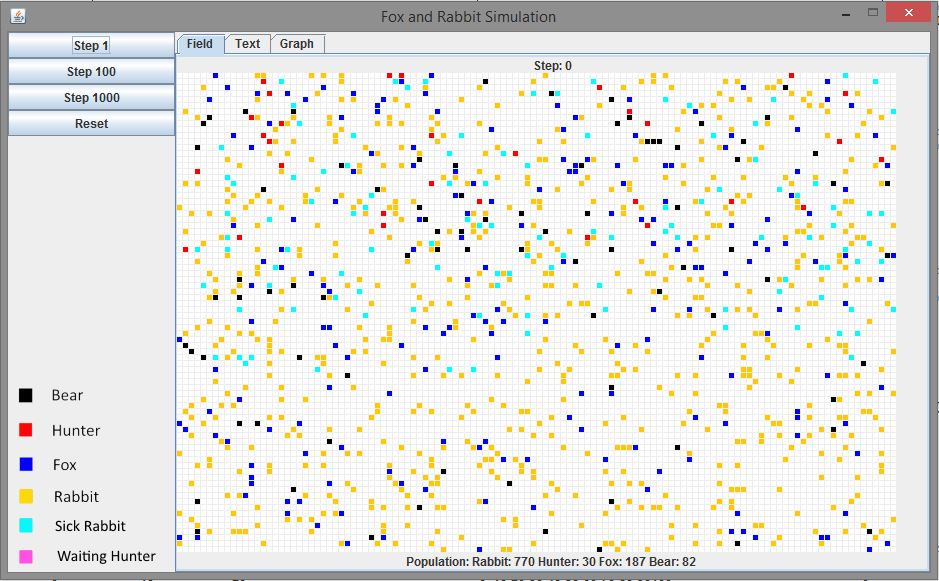
Er wordt in deze applicatie gebruik gemaakt van verschillende soorten views. De verschillende views geven allemaal dezelfde gegevens weer maar op een andere manier.

Ten eerste de field view. Deze view geeft een raster weer. Elk hokje in het raster heeft een kleur. Op die manier is te onderscheiden welk object er in het vakje staat. Ten tweede is er de tekst view. De tekst view geeft alle informatie weer door middel van tekst. Dit wordt gedaan door middel van de naam van het object en het aantal objecten erachter. Ten slotte zijn er de cirkeldiagram, de grafiek en de staafdiagram. Deze zijn allemaal hetzelfde alleen hebben ze een andere vorm. Elk object heeft een eigen kleur tijdens de simulatie. Hieronder worden de views weergegeven met de uitleg hoe deze werken.

De kleuren van de objecten zijn als volgt:

* Konijn is oranje
* Konijn met virus is cyaan
* Vos is blauw
* Beer is zwart
* Jager is rood / Herladen is roze

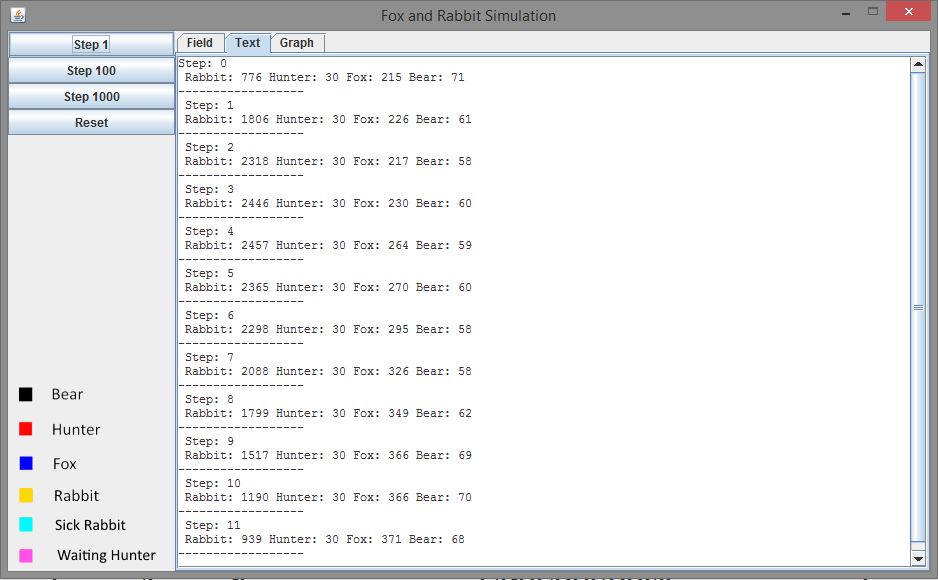
### 5.2.1 Field view

De field view geeft de standaard weergave van de simulatie weer. Hier zie je de vakjes die beschikbaar zijn voor de simulatie. Elk vakje heeft een kleur van een object. Een leeg vakje heeft een witte kleur. De field view is niet gebruikersvriendelijk omdat het niet overzichtelijk is. Daarom zijn er views toegevoegd die dat wel zijn.

#### Afbeelding 1 – Field View Simulatie

### 5.2.2 Tekst view

De tekst view geeft het aantal objecten weer via een lijn tekst. Dit is duidelijk en overzichtelijk voor een snelle weergave van het aantal objecten per stap. Door middel van de tekst view



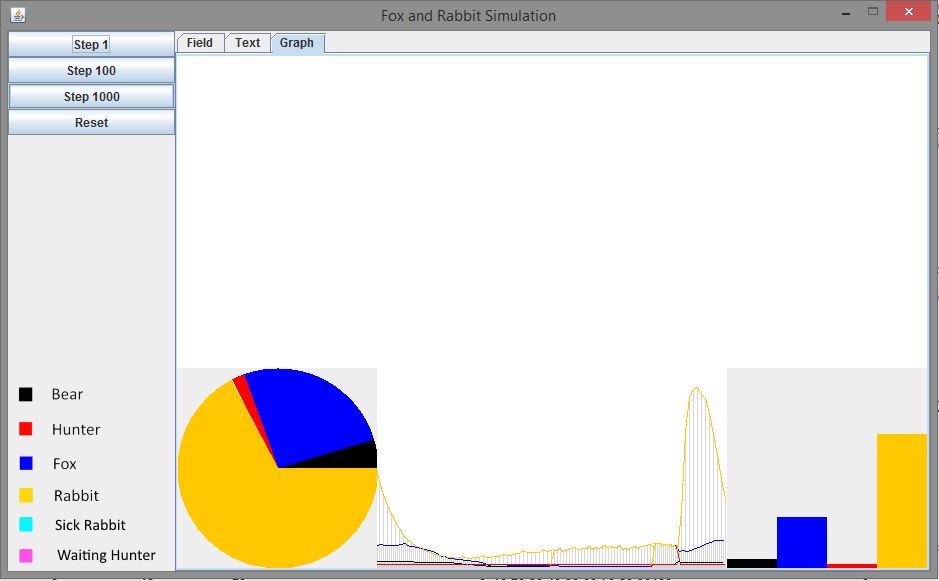
#### Afbeelding 2 – Tekst View Monitor

### 5.2.3 Graph View

De tab graph view geeft de populatie van alle objecten weer in drie verschillende grafieken. Hieronder kun je zien hoe dit in de praktijk er uit zou zien. In het linker menu staat de legenda. Aan de hand daarvan zijn de grafieken gemaakt. Op de cirkeldiagram valt te zien dat bijna drie kwart van de populatie bestaat uit konijnen.

In het midden wordt de normale grafiek weergegeven. Hier worden de laatste stappen gebruikt om een mooie lijngrafiek te genereren. Op die manier is te zien hoe de populatie verandert door de stappen heen.

Rechts in het beeld wordt het staafdiagram weergegeven. Ook hier is snel te zien welke populatie het meest bestaat. De cirkeldiagram en het staafdiagram hebben veel overeenkomsten. Beide kunnen gebruikt worden om globaal de populatie op dat moment te constateren. Om het verloop van de simulatie te monitoren kan het beste de lijngrafiek gebruikt worden.



#### Afbeelding 3 – Graph View Simulatie

# 6. Uitbreidingen en extra functies

In dit hoofdstuk zullen de uitbreiden centraal staan. De volgende uitbreidingen zijn toegevoegd aan de applicatie:

* Het toevoegen van een extra diersoort.
* Het toevoegen van de jager inclusief munitie beperkingen.
* Het konijnenvirus
* Voeding konijnen
* Instellingen Tab

Per sub paragraaf wordt één uitbreiding behandeld.

## 6.1 De Beer

Voor het toevoegen van een extra diersoort is er gekozen voor beren. De beren hebben de kleur zwart gekregen op de legenda.

### 6.1.1 Functie van de beer

Beren vormen een bedreiging voor zowel de vossen als de konijnen. De standaardwaarde voor de maximale leeftijd van de beer is 40 steps. Na 40 steps verdwijnt de beer dan ook van het veld. Ook is het mogelijk dat de beer dood gaan omdat er een tijdje geen voedsel is gevonden. Er bestaat een zeer kleine kans van 0,02 dat de beer zich voortplant. Als dit wel het geval is zullen er maximaal vier beren bijkomen. Al de voorgaande waarden zijn instelbaar.

#### Code 1 – De beer

**public** **class** **Bear** **extends** Animal

{

// Characteristics shared by all Bears (class variables).

// The age to which a Bear can live.

**private** **static** **int** max\_age;

// The maximum number of births.

**private** **static** **final** **int** MAX\_LITTER\_SIZE = **4**;

// The food value of a single rabbit. In effect, this is the

// number of steps a Bear can go before it has to eat again.

**private** **static** **final** **int** FOOD\_VALUE = **11**;

// A shared random number generator to control breeding.

**private** **static** **final** Random rand = Randomizer.getRandom();

// The Bear's food level, which is increased by eating rabbits.

**private** **int** foodLevel;

/\*\*

\* Create a Bear. A Bear can be created as a new born (age zero

\* and not hungry) or with a random age and food level.

\*

\* @param randomAge If true, the Bear will have random age and hunger level.

\* @param field The field currently occupied.

\* @param location The location within the field.

\*/

**public** **Bear**(**boolean** randomAge, Field field, Location location)

{

**super**(field, location);

**if**(randomAge) {

max\_age = SimulatorView.getLifeTimeBear();

setAge(rand.nextInt(max\_age));

foodLevel = SimulatorView.getFoodValueBear();

}

**else** {

setAge(**0**);

foodLevel = SimulatorView.getFoodValueBear();

}

}

## 6.2 De Jager

De tweede toevoeging is de jager. De jager heeft de mogelijkheid om alle diersoorten dood te schieten.

### 6.2.1 Functie van de jager

Jagers kunnen alle diersoorten doorschieten. Als extra functie is er een max munitie toegevoegd voor de jagers. Na zeven kogels geschoten te hebben kunnen de jagers niet meer schieten. De volgende drie steps worden gebruikt om te herladen. Na het herladen kan de jager weer zeven kogels schieten. De jagers zijn tijdens de simulatie onsterfelijk. Tijdens de simulatie wordt uitgegaan van vergunningen waardoor nooit dezelfde jagers actief blijven. Ook de maximale munitie en het aantal stappen dat een jager moet wachten als zijn munitie op is zijn waarden die instelbaar zijn.

##### Code 2 – De jager

**private** Location **findAnimal**()

{

Field field = **this**.field;

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

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

**if**(wait == **0**) {

**if**(bullets < SimulatorView.getBulletLimitHunter()) {

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

Location where = it.next();

Object animal = field.getObjectAt(where);

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

Rabbit rabbit = (Rabbit) animal;

**if**(rabbit.isAlive()) {

rabbit.setDead();

// Remove the dead rabbit from the field.

bullets++;

**return** where;

}

}

**if**(animal **instanceof** Fox) {

Fox fox = (Fox) animal;

**if**(fox.isAlive()) {

fox.setDead();

// Remove the dead rabbit from the field.

bullets++;

**return** where;

}

}

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

Bear bear = (Bear) animal;

**if**(bear.isAlive()) {

bear.setDead();

// Remove the dead rabbit from the field.

bullets++;

**return** where;

}

}

}

} **else** {

wait = SimulatorView.getWaitLimitHunter(); // 2 + 1 = waiting 3 turns

color = Color.pink;

}

} **else** {

wait--;

**if**(wait == **0**) {

color = Color.red;

bullets = **0**;

}

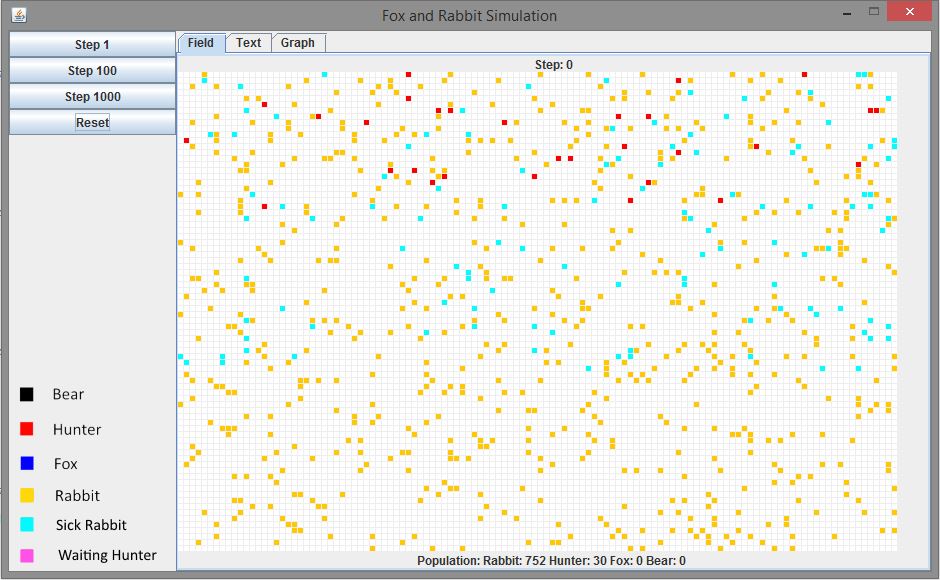
}

**return** **null**;

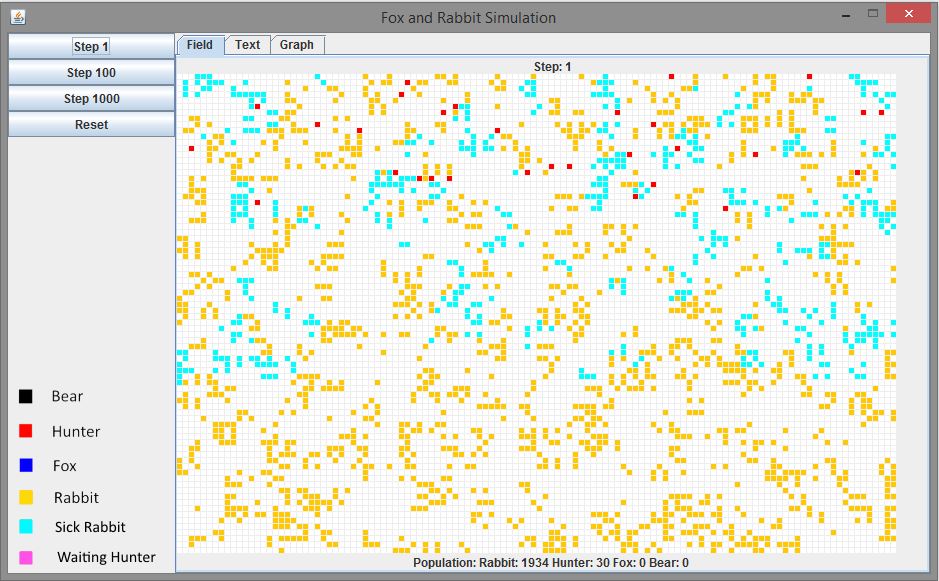
}

## 6.3 Het virus

Konijnen kunnen besmet raken met het virus. Dit virus is besmettelijk en kunnen andere konijnen besmetten met het virus. De kans dat een konijn het virus overbrengt is 90%. Na uiterlijk vijf dagen gaat het konijn dood. Ook deze twee waarden zijn instelbaar. Het virus verdwijnt zodra het laatste konijn met het virus sterft zonder een ander konijn te besmetten. Hieronder is de beginsituatie te zien met konijnen zonder virus en konijnen met virus. In deze test zijn geen andere dieren meegenomen.



#### Afbeelding 4 – Begin situatie zieke konijnen



#### Afbeelding 5 – Situatie zieke konijnen na stap 1

##### Code 3 – Het virus

**import** **java.awt.Color**;

**import** **java.util.Iterator**;

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

**import** **java.util.Random**;

/\*\*

\* A simple model of a rabbit.

\* Rabbits age, move, breed, and die.

\*

\* @author David J. Barnes and Michael KÃ¶lling

\* @version 2011.07.31

\*/

**public** **class** **Rabbit** **extends** Animal

{

// Characteristics shared by all rabbits (class variables).

// The age to which a rabbit can live.

**private** **static** **int** max\_age;

// The likelihood of a rabbit breeding.

**private** **double** breedingProbability;

// A shared random number generator to control breeding.

**private** **static** **final** Random rand = Randomizer.getRandom();

// A boolean to see whether the rabbit is sick or not

**private** **boolean** ziek;

// The chance the rabbit is one of the first to be infected

**private** **final** **int** FIRST\_INFECTED\_CHANCE = **20**;

// Maximum of first infected rabbits

**private** **final** **int** FIRST\_INFECTED = **500**;

// Integer for how many steps the rabbit has been sick

**private** **int** timeSick = **0**;

// Count neighbors

**private** **int** counter;

// Minimum free space required around the rabbit

**private** **final** **int** MIN\_FREE\_SPACE = **2**;

// Factor to reduce breeding probability

**private** **final** **double** LESS\_BREEDING\_PROBABILITY = **0.5**;

// Individual characteristics (instance fields).

/\*\*

\* Create a new rabbit. A rabbit may be created with age

\* zero (a new born) or with a random age.

\*

\* @param randomAge If true, the rabbit will have a random age.

\* @param field The field currently occupied.

\* @param location The location within the field.

\*/

**public** **Rabbit**(**boolean** randomAge, Field field, Location location, **int** numberInfected)

{

**super**(field, location);

color = Color.orange;

OFFICIAL\_COLOR = Color.orange;

setBreedingProbability(((**double**) SimulatorView.getBreedProbabilityRabbit()) / **100**);

max\_age = SimulatorView.getLifeTimeRabbit();

**if**(randomAge) {

setAge(rand.nextInt(max\_age));

} **else** {

setAge(**0**);

}

Random randomGenerator = **new** Random();

**int** random = randomGenerator.nextInt(**100**);

**if**( random <= FIRST\_INFECTED\_CHANCE && numberInfected < FIRST\_INFECTED)

{

setZiekteGen(**true**);

}

}

/\*\*

\* This is what the rabbit does most of the time - it runs

\* around. Sometimes it will breed or die of old age.

\* @param newRabbits A list to return newly born rabbits.

\*/

**@Override**

**public** **void** **act**(List<Actor> newRabbits)

{

**if** (getZiekteGen()){

**if** (timeSick < SimulatorView.getStepsBeforeDeath()) {

timeSick++;

} **else** {

setDead();

}

}

incrementAge();

**if**(isAlive()) {

checkFood();

**if** (counter >= MIN\_FREE\_SPACE) {

setBreedingProbability(LESS\_BREEDING\_PROBABILITY \* **this**.breedingProbability);

counter = **0**;

}

**if** (neighborInfected()) {

**if** (ziekteGen()) {

setZiekteGen(**true**);

}

}

giveBirth(newRabbits);

// Move towards a source of food if found.

Location newLocation = getField().freeAdjacentLocation(getLocation());

**if**(newLocation != **null**) {

setLocation(newLocation);

}

**else** {

// Overcrowding.

setDead();

}

}

}

/\*\*

\* Check whether or not this rabbit is to give birth at this step.

\* New births will be made into free adjacent locations.

\* @param newRabbits A list to return newly born rabbits.

\*/

**@Override**

**public** **void** **giveBirth**(List<Actor> newRabbits)

{

// New rabbits are born into adjacent locations.

// Get a list of adjacent free locations.

Field field = getField();

List<Location> free = field.getFreeAdjacentLocations(getLocation());

**int** births = breed();

**for**(**int** b = **0**; b < births && free.size() > **0**; b++) {

Location loc = free.remove(**0**);

Rabbit young = **new** Rabbit(**false**, field, loc, Simulator.getInfected());

**if** (getZiekteGen()) {

young.setZiekteGen(**true**);

}

newRabbits.add(young);

}

}

/\*\*

\* Check whether the rabbit has enough space around it

\*/

**public** **void** **checkFood**()

{

Field field = getField();

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

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

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

Location where = it.next();

Object animal = field.getObjectAt(where);

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

Rabbit rabbit = (Rabbit) animal;

**if**(rabbit.isAlive()) {

counter++;

}

}

}

}

/\*\*

\* Check whether the rabbit will get infected or not

\* @return boolean true if yes

\* @return boolean false if not

\*/

**public** **boolean** **ziekteGen**()

{

**int** random = (rand.nextInt(**100**) + **1**) / **100**;

**if**( random <= ((**double**) SimulatorView.getInfectionChance()) / **100**)

{

**return** **true**;

}

**return** **false**;

}

/\*\*

\* Check whether the neighbor of the rabbit is infected

\* @return true if yes

\* @return false if not

\*/

**public** **boolean** **neighborInfected**()

{

Field field = getField();

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

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

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

Location where = it.next();

Object animal = field.getObjectAt(where);

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

Rabbit rabbit = (Rabbit) animal;

**if**(rabbit.getZiekteGen()) {

**return** **true**;

}

}

}

**return** **false**;

}

/\*\*

\* Set the rabbit's breeding probability.

\* @param the rabbit's breeding probability.

\*/

**public** **void** **setBreedingProbability**(**double** breedingProbability)

{

**this**.breedingProbability = breedingProbability;

}

/\*\*

\* Return if the rabbit is sick

\* @return true if the rabbit is sick

\* @return false if the rabbit is not sick

\*/

**public** **boolean** **getZiekteGen**()

{

**return** ziek;

}

/\*\*

\* Make the rabbit sick

\* @param true if the rabbit gets sick

\* @param false if the rabbit doesn't get sick

\*/

**public** **void** **setZiekteGen**(**boolean** ziek)

{

**this**.ziek = ziek;

color = Color.cyan;

}

## 6.4 Instellingen

Er is een extra scherm met instellingen. In dit scherm is het mogelijk om de standaardwaarden van variabelen aan te passen die van invloed zijn op de simulatie. Hierdoor is het simpel om de instellingen van de simulatie aan te passen naar eigen wens. Om een situatie controle uit te voeren hoeft u enkel de desbetreffende slider te verplaatsen naar de waarde die u wilt gebruiken.



#### Afbeelding 6 – Simulatie Configuratie

Na het verplaatsen van de slider worden de waardes direct gebruikt. Tenzij het om create animal gaat, dan moet de simulatie eerst gereset worden alvorens de waarden gebruikt kunnen worden.

##### Code 4 – Instellingen

**import** **java.awt.\***;

**import** **javax.swing.\***;

**import** **java.util.LinkedHashMap**;

**import** **java.util.Map**;

/\*\*

\* A graphical view of the simulation grid.

\* The view displays a colored rectangle for each location

\* representing its contents. It uses a default background color.

\* Colors for each type of species can be defined using the

\* setColor method.

\*

\* @author David J. Barnes and Michael KÃ¶lling

\* @version 2011.07.31

\*/

**public** **class** **SimulatorView** **extends** JFrame

{

// Initiate the sliders

**public** **static** Configuration creationChanceRabbit;

/\*\*

\* Create a view of the given width and height.

\* @param height The simulation's height.

\* @param width The simulation's width.

\*/

**public** **SimulatorView**(**int** height, **int** width, FieldStats fieldStats, MonitorView monitorView, TextView textView)

{

// Make the sliders

creationChanceRabbit = **new** Configuration("Creation chance Rabbit",**0**,**100**,**8**);

// Add sliders to the settings frame

Settings settings = **new** Settings();

settings.add(creationChanceRabbit);

}

/\*\*

\* Getters for the values of the slider

\* @return the value

\*/

**public** **static** **int** **getLifeTimeRabbit**() {

**return** lifeTimeRabbit.getValue();

}

/\*\*

\* Create a frame for the sliders

\* @author Ronald elzen

\*

\*/

**public** **class** **Settings** **extends** JFrame {

**public** **Settings**() {

setTitle("Settings" );

setExtendedState(JFrame.MAXIMIZED\_BOTH);

setLayout(**new** GridLayout(**6**, **2**, **15**, **0**));

setVisible(**true**);

}

}

/\*\*

\* Create the sliders and initialize them

\* @author Ronald Elzen

\*

\*/

**public** **class** **Configuration** **extends** JPanel{

**private** JSlider a;

**private** JLabel b;

**public** **Configuration**(String labelText, **int** boundsMin,**int** boundsMax,**int** d) {

a = **new** JSlider(JSlider.HORIZONTAL, boundsMin, boundsMax, d);

b = **new** JLabel(labelText);

a.setMajorTickSpacing(**10**);

a.setMinorTickSpacing(**1**);

a.setPaintTicks(**true**);

a.setPaintLabels(**true**);

add(b);

add(a);

}

/\*\*

\* Get the value of the slider

\* @return the value

\*/

**public** **int** **getValue**() {

**return** a.getValue();

}

/\*\*

\* Set the value of the sliders

\* @param value

\*/

**public** **void** **setValue**(**int** value)

{

a.setValue(value);

}

}

# 7. Conclusie

Na het analyseren van de problemen en het opzetten van een werkplan is de simulatie verbeterd en uitgebreid naar wens van de opdrachtgever. Tijdens het analyseren viel op dat veel code statisch was aangeleverd. Dit is veranderd door een configuratie scherm aan te maken waarin alle variabelen veranderd kunnen worden aan de hand van sliders. Vervolgens zijn de beren, vossen en jagers toegevoegd. Als laatste object uitbreiding is er voor gezorgd dat er tijdens de simulatie een ziekte ontstaat. Deze ziekte ontstaat op het moment dat de simulatie begint. Als de ziekte het niet overleeft in de simulatie, komt het ook niet meer terug.

De doelstelling was het verbeteren van de huidige code en het toevoegen van extra features om deze simulatie zo goed mogelijk te kunnen laten verlopen. Dit is gelukt door middel van het configuratiescherm toe te voegen. Alle objecten werken zoals ze horen te werken en daarom is de doelstelling volbracht.

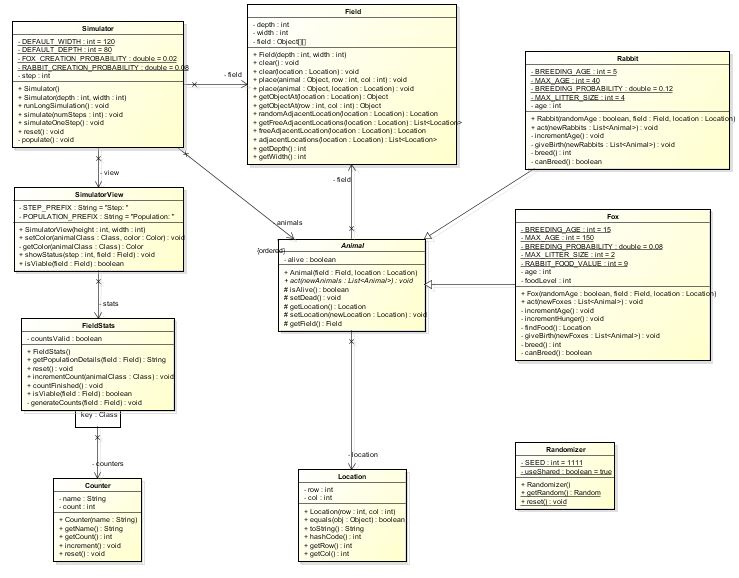
# Literatuurlijst

Wales & Sanger, (2013) Varkenscyclus. Geraadpleegd op 27 Januari, 2015, op

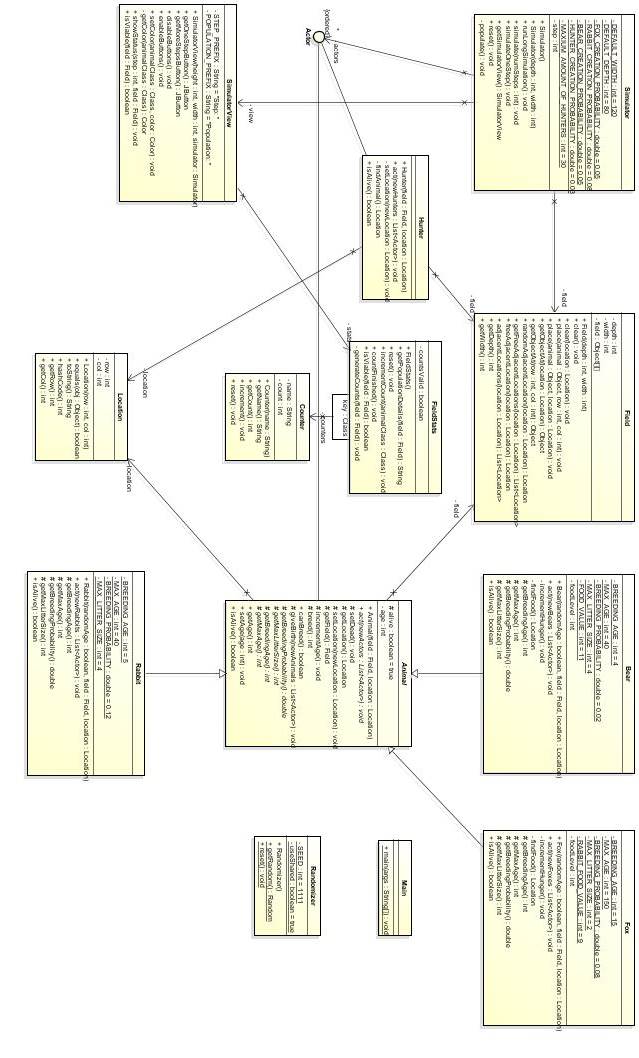
<http://nl.wikipedia.org/wiki/Varkenscyclus>

# Begrippenlijst

# Bijlage A – UML Klassendiagram V1.0



# Bijlage B – UML Klassendiagram V2.0



# Bijlage C – Sequence Diagram (Step ++)

