|  |  |
| --- | --- |
|  | **Vossen en Konijnen Groep 5** |
|  | Datum: 09/02/2012  Arjan Frans Brendan Kanters  Alwin Rombout Jelte Verbree  Onder opdracht van Jolanda van der Lee en Jan-Wiepke Knobbe |

|  |
| --- |
| **[Project Vossen en Konijnen]** |
|  |





Inhoudsopgave

[Hoofdstuk 1 – Inleiding 3](#_Toc316550643)

[Hoofdstuk 2 – Probleemanalyse 4](#_Toc316550644)

[2.1 Probleemstelling 4](#_Toc316550645)

[2.2 Analyse van de huidige situatie 4](#_Toc316550646)

[2.2.1 Varkenscyclus 5](#_Toc316550647)

[2.2.2 Beperkingen en Uitbreidbaarheid 5](#_Toc316550648)

[Hoofdstuk 3 – Uitbreiding 1 6](#_Toc316550649)

[3.1 De abstracte superklasse Animal 6](#_Toc316550650)

[3.2 De interface Actor 7](#_Toc316550651)

[3.3 De Wolf 8](#_Toc316550652)

[3.4 De Jager 9](#_Toc316550653)

[Hoofdstuk 4 – Het Gebruik van MVC in V&K 11](#_Toc316550654)

[Hoofdstuk 5 – Uitbreiding 2 13](#_Toc316550655)

[5.1 De GUI 13](#_Toc316550656)

[5.1.1 De ‘look’ 13](#_Toc316550657)

[5.1.2 De ‘views’ 14](#_Toc316550658)

[5.1.3 De Instellingen 17](#_Toc316550659)

[5.1.4 Het Gras 18](#_Toc316550660)

[5.1.5 De Weg 19](#_Toc316550661)

[5.1.6 Het Virus 19](#_Toc316550662)

[Hoofdstuk 6 – Testen 21](#_Toc316550663)

[Hoofdstuk 7 – Conclusie 23](#_Toc316550664)

[Bronnenlijst 24](#_Toc316550665)

[Bijlage A – UML Klassendiagram V1 25](#_Toc316550666)

[Bijlage B – UML Klassendiagram V2 26](#_Toc316550667)

[Bijlage C – Reflecties 28](#_Toc316550668)

[Alwin 28](#_Toc316550669)

[Arjan 28](#_Toc316550670)

[Jelte 28](#_Toc316550671)

[Brendan 28](#_Toc316550672)

# Hoofdstuk 1 – Inleiding

Het project van Vossen en Konijnen simuleert het leven van deze dieren in het wild. Groep 5, bestaande uit 4 studenten, heeft de opdracht gekregen om het programma Vossen en Konijnen te verbeteren en uit te breiden. Deze opdracht is gegeven door de Hanzehogeschool Groningen onder leiding van Jolanda van der Lee en Jan-Wiepke Knobbe.

In dit rapport wordt beschreven hoe het programma Vossen en Konijnen werkt. Er zal gekeken worden naar de tekortkomingen van de simulatie en wat er verder nog voor nuttige dingen aan toegevoegd kunnen worden. Er zal duidelijk gemaakt worden met welke verbeteringen het programma beter aan de eisen zal voldoen om een nuttig hulpmiddel te zijn voor gebruik in de praktijk.

Om te beginnen zal er een probleemanalyse worden gegeven waarin de probleemstelling wordt uitgelegd en waar er gesproken wordt over de beperkingen en mogelijke uitbreidingen. Daarna zal er gesproken worden over de verbeteringen en uitbreidingen aan het programma, toegelicht met behulp van klassendiagrammen en stukken programmacode. Tot slot zal er een conclusie gegeven worden waarin er gesproken zal worden over aanbevelingen en een reflectie van groep 5.

# Hoofdstuk 2 – Probleemanalyse

In dit hoofdstuk zal een probleemanalyse worden beschreven. Eerst zal er wat meer vertelt worden over de probleemstelling. Daarna wordt de basisversie van de applicatie geanalyseerd en zullen de tekortkomingen van deze eerste versie besproken worden

2.1 Probleemstelling

In de natuur is er een sterk verband tussen de populatie van het roofdier en zijn prooi. Als de balans tussen het roofdier en zijn prooi niet goed is dan kan dit grote gevolgen hebben. Als er bijvoorbeeld te veel van een soort roofdier zijn dan zal deze snel een gebied overnemen en zijn prooi uitroeien. De roofdieren zullen na het uitroeien van hun prooi uiteindelijk ook uitsterven om dat er gebrek aan voedsel is. Aan de andere kant als het roofdier in een sterke minderheid is en er is geen prooi in de buurt, dan zal het roofdier uitsterven en de prooi kan dan ongecontroleerd voortplanten.

In het programma Vossen en Konijnen zal er gesimuleerd worden hoe dit er uit zou kunnen zien in de natuur en wat de gevolgen zijn. Dit kan een handig hulpmiddel zijn voor bijvoorbeeld natuurbeheerders. Het wordt pas echt bruikbaar als de gebruiker een invoer kan doen met gegevens uit een realistische situatie. Er moeten echter nog wat aanpassingen aan het programma gemaakt worden voordat dit mogelijk is. De probleemstelling is dus dat er gekeken zal worden naar welke verbeteringen en uitbreidingen er nodig zullen zijn om de simulatie bruikbaar te maken in de praktijk.

## 2.2 Analyse van de huidige situatie

Bij het analyseren van het programma foxes-and-rabbits-v2 kan er gekeken worden naar twee dingen. Hoe het er aan de buitenkant uitziet (de GUI) en hoe het programma werkt achter de schermen (de code).

Als er een nieuwe simulatie gestart wordt dan maakt het programma een raster van 50 bij 50 pixels, deze wordt gevuld met een willekeurig aantal ‘vossen’ en ‘konijnen’ door het inkleuren van deze pixels. Onder het raster is er te zien hoeveel vossen en konijnen er op dit moment zijn en erboven staat in welke stap de simulatie zich bevindt. Er is in de GUI nog geen mogelijkheid om één of enkele stappen verder te gaan in de simulatie.

De code zorgt voor de werking van de simulatie, bijvoorbeeld waar en wanneer nieuwe diersoorten opspringen. Bij het langslopen van de simulatie valt op dat er een nauw verband bestaat tussen de twee diersoorten. Als het aantal vossen stijgt daalt het aantal konijnen waardoor het aantal vossen weer daalt door te weinig voedsel en zo de konijnen populatie weer de kans heeft om te groeien. Wat opvalt is dat als er maar genoeg stappen gemaakt worden dat er maar één diersoort zal overleven. Er is dus geen ‘balans’ die oneindig doorgaat.

### 2.2.1 Varkenscyclus

Wat gezien wordt in de simulatie van Vossen en Konijnen komt overeen met het economische principe van de ‘varkenscyclus’. Het verschijnsel van de ‘varkenscyclus’ heeft te maken met het elkaar afwisselen van overschotten van een bepaald product waardoor de prijzen hiervan dalen, wat gevolgd word door een massale reactie hierop waardoor het producten overschot daalt en de prijs weer omhoog gaat. Dit is in de simulatie te zien, als het aantal vossen stijgt daalt het aantal konijnen omdat er niet genoeg konijnen zijn voor de vossen populatie. Hierdoor daalt de vossen populatie omdat er niet meer genoeg voedsel is voor de vossen, dit geeft de konijnen weer de kans om te groeien. Dit is een cyclus die steeds weer herhaald word.

### 2.2.2 Beperkingen en Uitbreidbaarheid

In de analyse van het programma zijn al enkele beperkingen aan bod gekomen. De gebruiker heeft geen mogelijkheid om input te geven aan de simulatie. Het is bijvoorbeeld nog niet mogelijk om de verhouding tussen de verschillende dieren in te stellen. Dit zal aangepast worden in een latere versie.

In de simulatie zijn op dit moment maar twee diersoorten. In een latere versie van het programma zullen er meer diersoorten en ook een ‘niet-dier’, een jager bijvoorbeeld, worden toegevoegd.

De structuur van het programma is op dit moment niet handig om uitbereidingen op te doen. Daarom zal er gebruik gemaakt worden van een zogenaamd ‘Model-View-Controller’ structuur. Deze structuur maakt het veel makkelijker om uitbereidingen te implementeren en wijzigingen te maken. In hoofdstuk 4 is er meer informatie te vinden over deze structuur en hoe deze word toegepast.

#### 2.2.2.1 De veranderingen

Hier is een opsomming van de aanpassingen die gemaakt zullen worden aan het programma om te zorgen dat de gebruiker meer invloed kan hebben op de simulatie.

* De GUI zal verbeterd worden door enkele knoppen en velden toe te voegen waarin de gebruiker parameters voor de simulatie kan instellen.
* Er zal een extra diersoort worden toegevoegd en een (‘niet-dier’) jager.
* De structuur van het programma zal worden veranderd naar een MVC om het makkelijker te maken om uitbreidingen toe te voegen zoals extra views.

# Hoofdstuk 3 – Uitbreiding 1

In dit hoofdstuk zullen de eerste uitbreidingen aan het programma worden besproken. De volgende uitbreidingen en verbeteringen zullen besproken worden:

* Implementatie van de abstracte superklasse Animal.
* Implementatie van de interface Actor.
* Het toevoegen van een diersoort.
* Het toevoegen van een ‘niet-dier’, de jager.



UML Klassendiagram basisversie, zie Bijlage A voor een uitgebreid diagram.

## 3.1 De abstracte superklasse Animal

De klassen Fox en Rabbit hebben verschillende gedeelde eigenschappen zoals leeftijd en de mogelijkheid om te broeden. Ook zijn er verschillende methodes die hetzelfde zijn of veel op elkaar lijken.

De eigenschappen age, alive, field en location zijn verplaatst naar de klasse Animal.

|  |
| --- |
| **private** **int** alive; **private** Location location;  **private** Field field;  **protected** **int** age;  **protected** **int** breed() |

De getters en setters die bij deze eigenschappen horen en de methodes incrementAge(), giveBirth(..) en breed() zijn ook verplaatst.

|  |
| --- |
| **protected** **void** giveBirth(List<Animal> newAnimals)  **public** **void** incrementAge()  **public** **boolean** canBreed()  **protected** **int** breed() |

De methode giveBirth(..) zit in zowel de klasse Fox als Rabbit, maar het verschil is dat er in de klasse Fox een lijst met nieuwe Foxes wordt geretourneerd en in de klasse Rabbit een lijst met nieuwe Rabbits. De methode is verplaatst naar Animal, maar er wordt nu eerst gekeken welk diersoort de Animal is en vervolgens wordt een lijst met de nieuwe dieren van de gevonden soort geretourneerd.

|  |
| --- |
| **protected** **void** giveBirth(List<Animal> newAnimals)  **public** **void** incrementAge()  **protected** **int** breed() |

Omdat de body van deze methodes voor de klassen Fox en Rabbit hetzelfde zijn hoeven ze niet abstract te worden gemaakt. De methode act(…) is bijvoorbeeld wel abstract, dit moet omdat in zowel de klasse Fox als Rabbit de methode een andere body heeft.

## 3.2 De interface Actor

De interface Actor word geïmplementeerd door de klasse Animal en zal ook door de klasse Hunter worden geïmplementeerd. Een Hunter is namelijk geen animal, deze interface is hier de oplossing voor.

In de interface wordt alleen de signatuur van de methodes aangegeven. Deze interface is geïmplementeerd zodat deze methodes later gebruikt kunnen worden als er een jager word toegevoegd aan het programma.

|  |
| --- |
| **public** **interface** Actor {  **public** **void** act(List<Actor> actor);  **public** **boolean** isAlive();  **public** **void** setDead();  **public** Location getLocation();  **public** Field getField();  **public** **void** setLocation(Location location);  } |

## 3.3 De Wolf

Als nieuw diersoort is er een wolf aan de simulatie toegevoegd. De wolf eet alleen vossen en net als de twee andere diersoorten kan hij ook jongen krijgen. De manier waarop hij eten zoekt en zich voortplant is vergelijkbaar met dat van de vossen en konijnen, wel zijn de variabelen aangepast. Waaronder de leeftijd waarop een wolf zich kan voortplanten en hoeveel jongen de wolf kan krijgen.

|  |
| --- |
| **public** **class** Wolf **extends** Animal  {  // Characteristics shared by all wolves (static fields).    // The age at which a wolf can start to breed.  **private** **static** **int** *BREEDING\_AGE* = 15;  // The age to which a wolf can live.  **private** **static** **final** **int** *MAX\_AGE* = 200;  // The likelihood of a wolf breeding.  **private** **static** **double** *BREEDING\_PROBABILITY* = 0.15;  // The maximum number of births.  **private** **static** **final** **int** *MAX\_LITTER\_SIZE* = 5;  // The food value of a single fox. In effect, this is the  // number of steps a wolf can go before it has to eat again.  **private** **static** **final** **int** *FOX\_FOOD\_VALUE* = 10;    // The wolf's food level, which is increased by eating foxes.  **private** **int** foodLevel;  *Methodes weg gelaten*  } |

Na het toevoegen van de wolf is de balans van de simulatie veranderd. Het “uitsterven” van diersoorten wordt voorkomen door te zorgen dat er een goede balans is in de variabelen die gebruikt worden door de klassen. Met name bij de dieren; vanaf welke leeftijd en hoeveel jongen ze kunnen krijgen, hoe oud ze kunnen worden en hoeveel ze eten.

## 3.4 De Jager

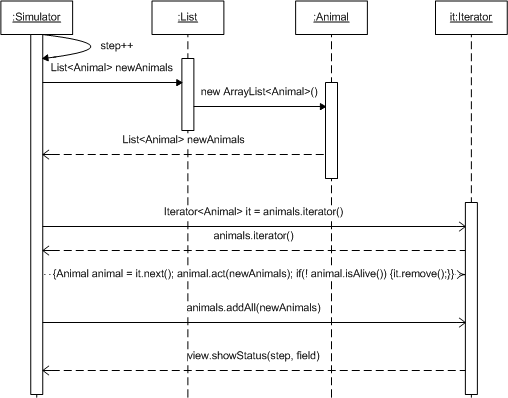
De klasse Hunter stelt de jager voor in deze simulatie. De klasse implementeert de interface Actor. Een jager kan alleen op de dieren schieten, om te voorkomen dat ze op elkaar kunnen schieten is in de code de volgende check toegevoegd die eerst kijkt of het object wel een dier is.

|  |
| --- |
| **private** Location shoot(Location location)  {  Field field = getField();  List<Location> adjacent = field.adjacentLocations(getLocation());  Iterator<Location> it = adjacent.iterator();  **while**(it.hasNext()) {  Location where = it.next();  Object actor = field.getObjectAt(where);  **if**(actor **instanceof** Animal) {  Animal animal = (Animal) actor;  **if**(animal.isAlive()) {  animal.setDead();  **return** where;  }  }  }  **return** **null**;  } |

Als het gevonden Object een Animal is dan zal deze ‘geschoten’ worden. Als het geen Animal is, dus bijvoorbeeld een andere jager, dan zal er niks gebeuren.



UML Klassendiagram na uitbreidingen, zie Bijlage B voor een uitgebreide uitvoering.

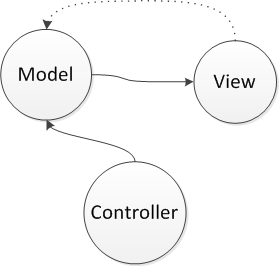


Sequentie diagram van 1 stap simulatie.

# Hoofdstuk 4 – Het Gebruik van MVC in V&K

Het vossen en konijnen project had in het begin geen handige structuur die het uitbreiden makkelijk maakte. Bij het toevoegen van een nieuwe klasse moest er op een andere plek ook een wijziging gemaakt worden. Daarom is het project omgebouwd naar een Model-View-Controller structuur, afgekort MVC. Een MVC komt voor in verschillende variaties.

Het MVC structuur dat in dit project gebruikt word ziet er zo uit:



* Het model is de ‘motor’ achter de simulatie, deze zorgt voor het gedrag van de objecten op het veld en houdt bij wat er na elke stap gebeurd. Er kan maar één model zijn.
* Een view is een weergaven van het model, dit kan op verschillende manieren. In het vossen en konijnen project is dit een raster waarop de objecten van het veld zichtbaar zijn in verschillende kleuren. De view haalt gegevens uit het model om een weergave te maken. Een view veranderd zelf niks aan het model. Er kunnen meerdere views toegevoegd worden.
* De controller maakt het mogelijk om iets aan het model te kunnen veranderen. In vossen en konijnen is er bijvoorbeeld een knop die de simulatie naar de volgende stap brengt. Deze knop is dan de controller die verteld dat het model een stap verder moet zetten. Er kunnen meerdere controllers toegevoegd worden.

In ons project is het nu mogelijk om nieuwe views toe te voegen zonder dat het model aangepast hoeft te worden. Er moet bijvoorbeeld misschien wel een getter gemaakt worden in het model die gegevens retourneert die de view dan kan gebruiken, maar er hoeft in het model geen directe verwijzingen naar één view gebruikt te worden. Alle views worden tegelijk geüpdatet met één regel in het model.

Er wordt gebruik gemaakt van een package structuur om het model, de views en de controllers gescheiden te houden. In de package van het model zit een abstracte klasse AbstractModel. In view zit een klasse AbstractView en in de controller package zit AbstractController. De klasse simulator is het model.

In AbstractModel zit een functie addView(AbstractView view) en notifyViews(). De functie addView(…) wordt in de klasse AbstractView aangeroepen. De view wordt dan toegevoegd aan het model. De methode notifyViews() wordt aangeroepen in het model om alle views in één keer te laten updaten. In de body van notifyViews() wordt de functie updateView() voor elke view aangeroepen. AbstractController heeft geen functies, maar heeft wel net als AbstractView de simulator als parameter in de constructor

Er is een klasse Window die alles samenbrengt. Er wordt een nieuwe simulator aangemaakt en vervolgens worden de views gedefinieerd als AbstractView en de controllers als AbstractController en wordt de simulator (het model) meegegeven.

Er hoeft nu alleen maar in de klasse Window een nieuwe view of controller gedefinieerd te worden. Een nieuwe controller of view toevoegen is nu dus veel makkelijker met een MVC structuur.

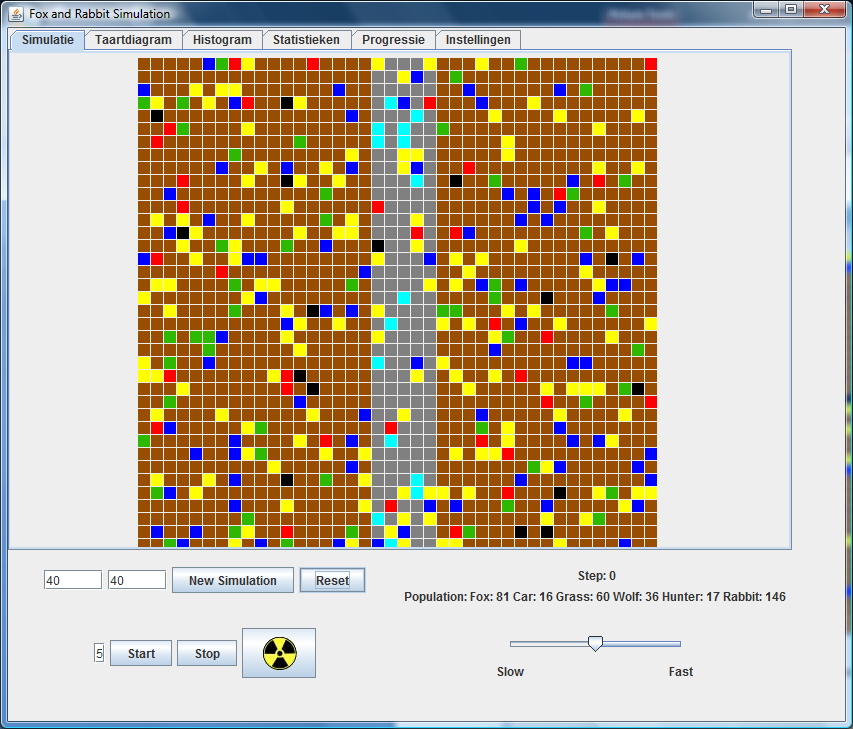
# Hoofdstuk 5 – Uitbreiding 2

In dit hoofdstuk zal er gesproken worden over de tweede laag van uitbreidingen die zijn toegepast op het simulatie programma Vossen en Konijnen. Er is al gesproken over het toevoegen van meerder diersoorten en een jager. Ook is er gesproken over het implementeren van het MVC structuur. In dit hoofdstuk zal er geconcentreerd worden op de uitbreiding van de GUI en de functionaliteit van de simulatie.

## 5.1 De GUI

Er waren enkele uitbreidingen nodig in de GUI om alles duidelijker en overzichtelijker te maken, ook was het nodig om de instellingen aan te kunnen passen in de GUI zelf. Hoe dit gedaan is zullen we aan de hand van wat afbeeldingen en tekst laten zien.

### 5.1.1 De ‘look’

Zoals te zien is in de afbeelding is er gebruik gemaakt van ‘tabs’ in het programma. Hier is voor gekozen omdat dit de applicatie overzichtelijk en makkelijk te gebruiken maakt. De knoppen balk onderaan het scherm is te zien onder alle tabs. Hier kan de selectie gemaakt worden over de grote van het raster, ook is hier te bepalen hoeveel stappen de simulatie moet lopen en op welke snelheid dit gebeurt.

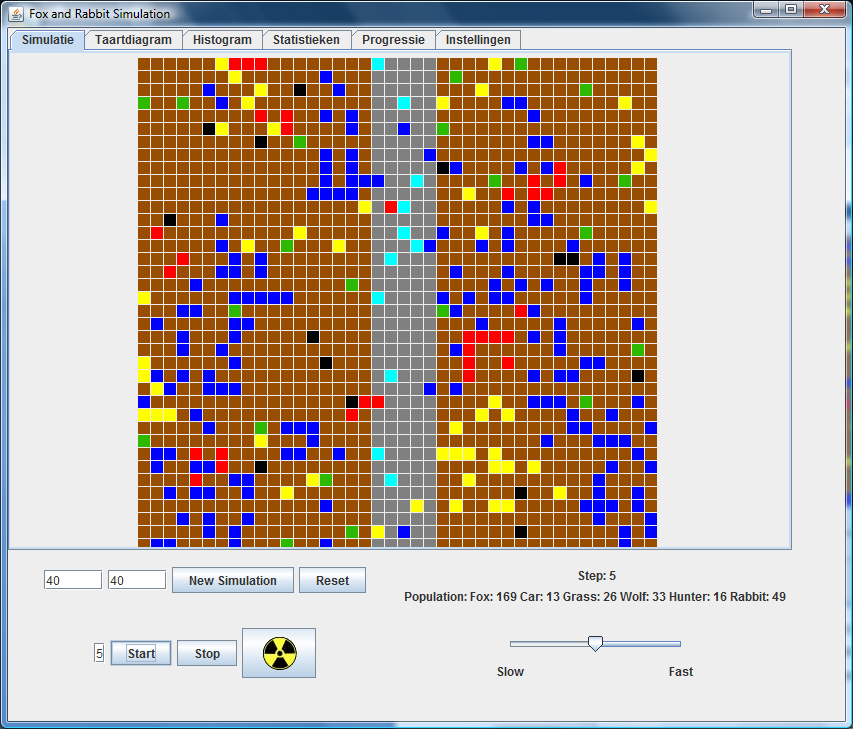
De nieuwe 'look'

### 5.1.2 De ‘views’

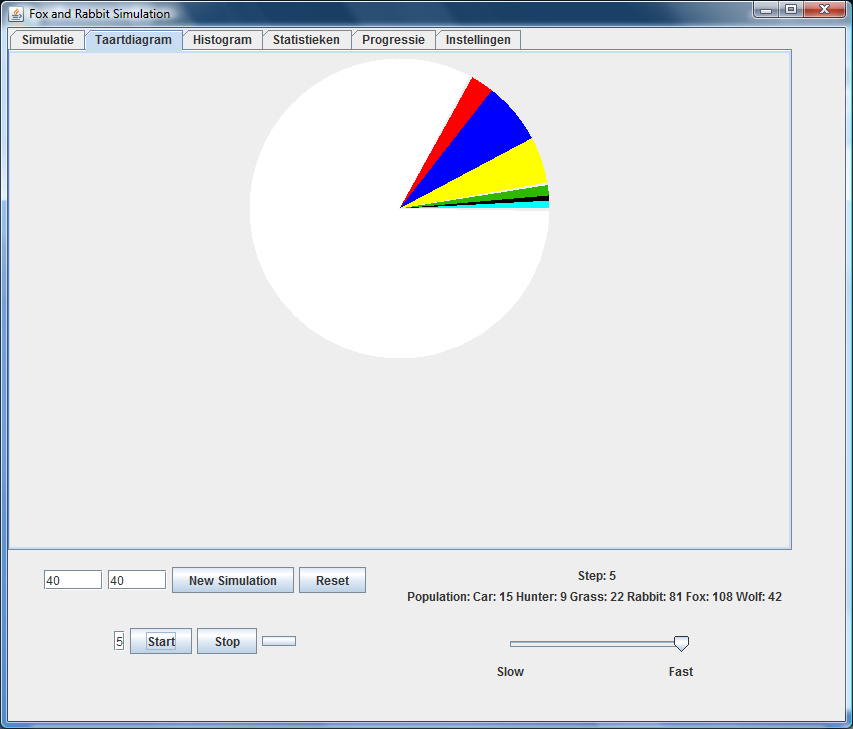
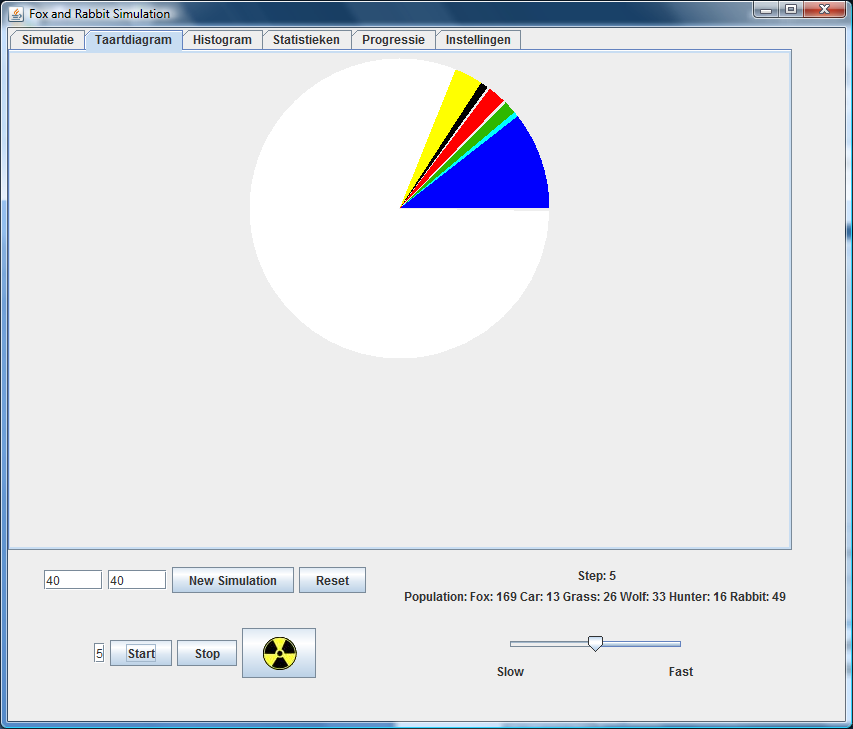
Er wordt in de simulatie gebruik gemaakt van 5 views. Hieronder worden enkele views laten zien na 5 stappen van de simulatie doorlopen te hebben. Elk object in de simulatie heeft een eigen kleur:

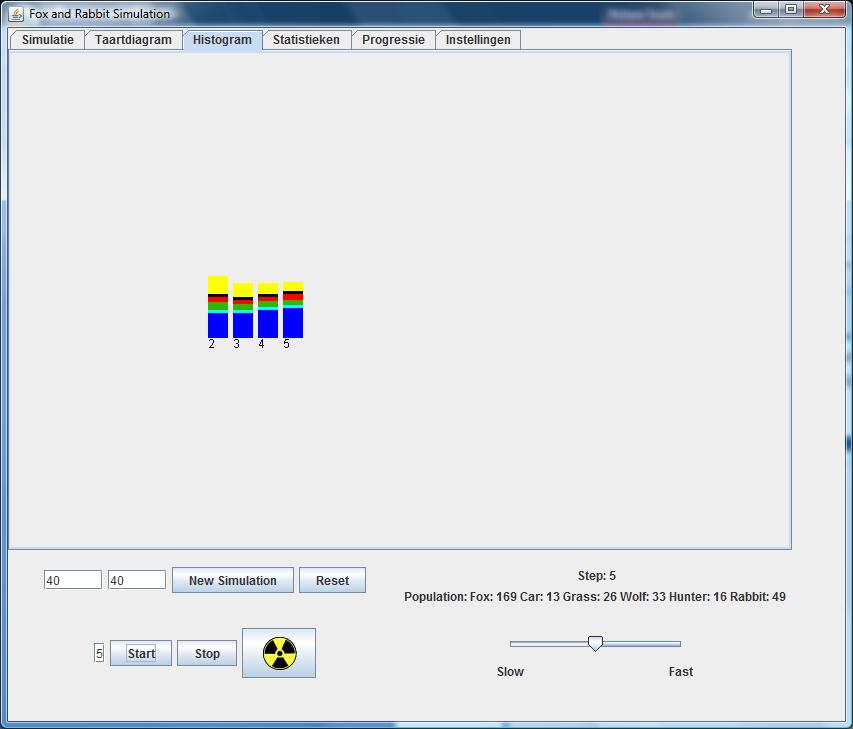
* Het konijn is geel
* De vos is donker blauw
* De wolf is rood
* Het gras is groen
* De auto is licht blauw
* De jager is zwart
* De weg is grijs
* Lege vakjes zijn bruin

#### 5.1.2.1 De standaard simulatie view

  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
Hier is het standaard raster te zien met alle objecten die hierin zijn getekend. Onderaan het scherm kan je zien in welke stap van de simulatie het programma is en ook de hoeveelheid van de verschillende objecten, dit onderste gedeelte is te zien bij elke view.

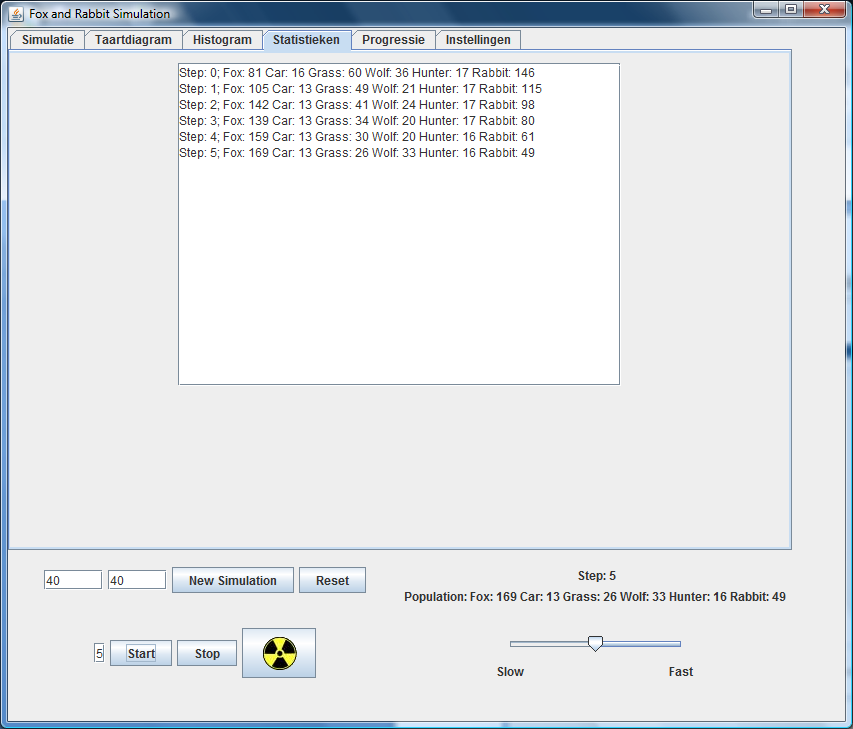
#### 5.1.2.2 De Taartdiagram



  
  
  
  
In het taartdiagram krijgt de gebruiker een duidelijk beeld van het percentage van elk object in de simulatie. In dit geval staat wit voor het gebied van de simulatie dat leeg is.  
  
5.1.2.3 De Histogram

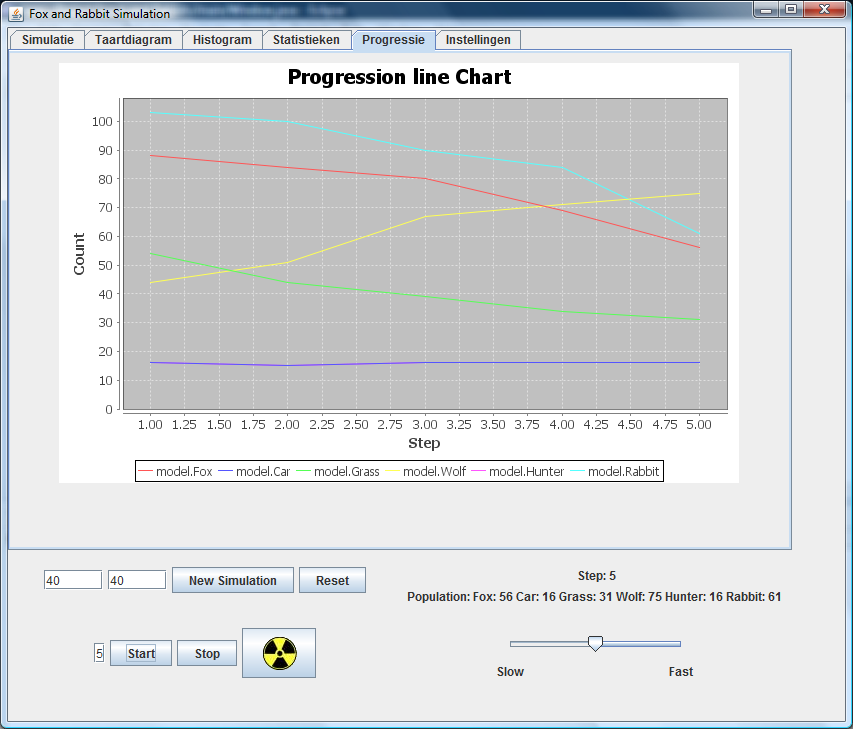
De histogram laat de verhouding tussen de hoeveelheden objecten goed zien aan de hand van staven.

#### 5.1.2.4 De Statistieken



In de statistieken view is aan de hand van tekst te zien hoeveel er van elk object aanwezig is in de simulatie.

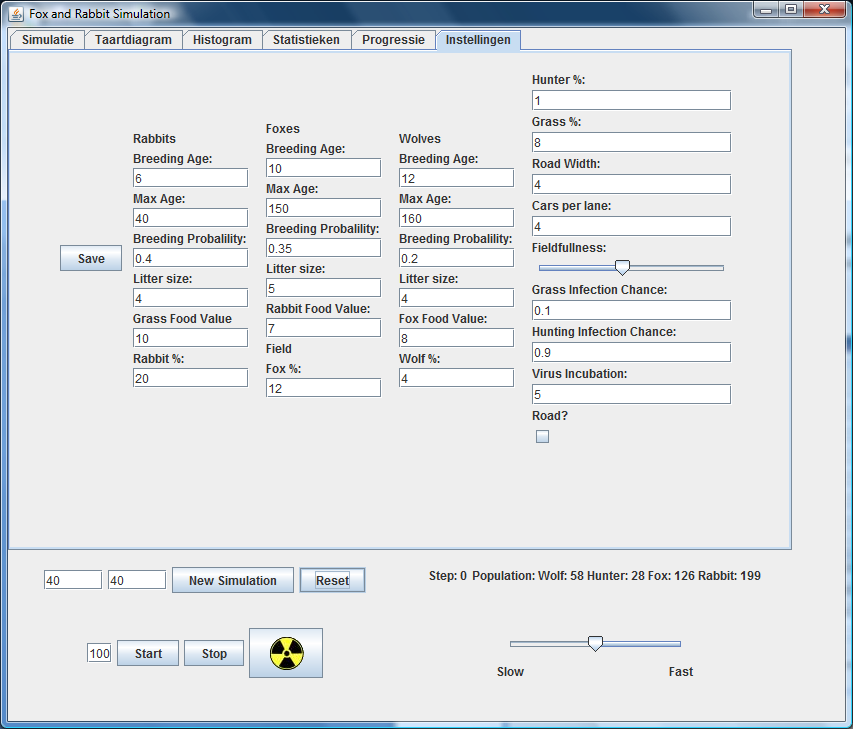
#### 5.1.2.5 De Progressie



In de progressie view is aan de hand van een lijnen grafiek te zien hoeveel er van elk object aanwezig is in de simulatie.

### 5.1.3 De Instellingen

Voor de gebruiksvriendelijkheid is er de mogelijkheid ingebouwd om alle instellingen aan te kunnen passen in de GUI zelf. De standaardwaardes van de instellingen worden uit de code gehaald.



Hier kan de gebruiker voor elk diersoort de volgende instellingen aanpassen:

* De leeftijd waarop een dier zich kan voortplanten.
* De maximale leeftijd dat een dier kan halen.
* De kans waarop een dier zich kan voortplanten.
* De maximale nakomelingen in elk nest van een dier.
* De waarde van de voeding die een dier gebruikt.
* De kans dat een dier word aangemaakt.

Ook zijn er enkele algemene dingen aan te passen waaronder:

* De kans dat een jager en gras word gemaakt.
* De breedte van de weg.
* Het aantal auto’s dat op elke laan van de weg kan zijn.
* De kans op infectie van het virus.
* Het aantal stappen dat een ziek dier besmettelijk is.
* Of er wel of niet een weg is.

### 5.1.4 Het Gras

In de simulatie is er gras toegevoegd als voeding voor de konijnen.

|  |
| --- |
| **private** **void** grow(List<Actor> newGrass)  {  Field field = getField();  List<Location> free = field.getGrassLocations(getLocation());  Grass grass = **null**;  **int** patches = **new** Random().nextInt(*MAX\_LITTER\_SIZE*);  **for**(**int** b = 0; b < patches && free.size() > 0; b++) {  Location loc = free.remove(0);  **if**(field.getRoadLocation(loc.getRow(), loc.getCol()) == **false**) {  grass = **new** Grass(field, loc);  newGrass.add(grass);  }  }  } |

Bij het maken van nieuwe konijnen zal er gezocht worden naar deze voeding en indien gevonden zal het nieuwe konijn in een locatie dichtbij deze voeding worden geplaatst.

|  |
| --- |
| **private** Location findGrass(Location location)  {  Field field = getField();  List<Location> adjacent = field.adjacentLocations(getLocation());  Iterator<Location> it = adjacent.iterator();  **while**(it.hasNext()) {  Location where = it.next();  Object actor = field.getObjectAt(where);  **if**(actor **instanceof** Grass) {  Grass grass = (Grass) actor;  **if**(grass.isAlive()) {  grass.setDead();  currentLitterSize++;  // Remove the dead grass from the field.  **return** where;  }  }  **else** {  **if**(currentLitterSize > 0) currentLitterSize--;  **return** where;  }  }  **return** **null**;  } |

Ook zal er met deze informatie worden bepaald over het aantal nakomelingen van de konijnen aangepast moet worden.

### 5.1.5 De Weg

Als extra aanvulling is er gekozen voor een weg met auto’s. Deze weg loopt door het midden van het gebied en erop rijden een aantal auto’s dat in te stellen is door de gebruiker. Deze auto’s werken hetzelfde als auto’s in het echte leven en bewegen dus ook.

|  |
| --- |
| **public** **void** drive(Location location)  {  **int** row = location.getRow();  **int** col = location.getCol();  **if**(row+direction < 0 || row+direction >= field.getDepth())  {  **if**(direction == +1) row = 0;  **if**(direction == -1) row = field.getDepth()-1;  }  **else** {  row += direction;  } |

Als er een dier op het pad van een auto komt dan zal deze worden ‘doodgereden’. Als er een jager op het pad van een auto komt dan zal er niks met de jager gebeuren.

|  |
| --- |
| **if**(field.getObjectAt(row,col) **instanceof** Animal)  {  ((Animal) field.getObjectAt(row,col)).setDead();  }  Location newLocation = **new** Location(row,col);  **if**(field.getObjectAt(row,col) **instanceof** Hunter)  {  //do not move;  newLocation = location;  }  setLocation(newLocation);  } |

### 5.1.6 Het Virus

Als extra mogelijkheid is er een besmettelijke ziekte aan de simulatie toegevoegd. Konijnen hebben standaard een kans van 90% (instelbaar) om deze ziekte te krijgen. Het virus word geïntroduceerd door besmettelijk gras. Elk stukje gras heeft een kans om het virus over te geven aan een konijn.

|  |
| --- |
| **if**(actor **instanceof** Grass) {  Grass grass = (Grass) actor;  **if**(grass.isAlive()) {  **if**(*rand*.nextDouble() <= *GRASS\_INFECTION\_CHANCE*){  setZiekteGen(**true**);  } |

Als een konijn besmet is met dit virus dan is deze voor 5 (instelbaar) stappen besmettelijk en kan deze het virus over geven aan een vos dat dit konijn op eet.

|  |
| --- |
| **if**(animal **instanceof** Rabbit) {  Rabbit rabbit = (Rabbit) animal;  **if**(rabbit.isAlive()) {  **if**(rabbit.getZiekteGen()){  setZiekteGen(**true**);  }  rabbit.setDead();  foodLevel = *RABBIT\_FOOD\_VALUE*;  // Remove the dead rabbit from the field.  **return** where;  }  } |

De vos die een besmet konijn heeft opgegeten heeft een kans om dan ook het virus te krijgen en is ook voor een instelbaar aantal stappen besmettelijk voor een ander dier dat deze vos opeet.

|  |
| --- |
| protected void infect()  {  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 Animal) {  Animal carrier = (Animal) animal;  if(carrier.isAlive() && carrier.getZiekteGen()) {  if(*rand*.nextDouble() <= *INFECTION\_CHANCE*){  setZiekteGen(true);  }  }  }  }  } |

Als een besmet dier niet opgegeten of geschoten word voor het aantal stappen over is dan zal deze doodgaan.

# Hoofdstuk 6 – Testen

Er is gebruik gemaakt van J-Unit om de code te testen, dit wordt ook wel een ‘whiteboxtest’ genoemd. Omdat de interne werking van het product bij de groep bekend is was bekend wat de verwachtte resultaten waren. Hieronder volgt 1 van deze J-Unit testen. Deze J-Unit test kijkt of een Rabbit kan broeden. Er wordt één Rabbit op het veld gezet. Deze Rabbit zal op een leeftijd van vijf (dit is een eigenschap in de klasse Rabbit) moeten kunnen broeden. De kans op broeden is voor deze test op 100% gezet. Er worden vijf stappen gesimuleerd. Na het nemen van deze stappen moeten er minstens twee Rabbits zijn. Als dit zo is zal de test slagen.

|  |
| --- |
| @Test  **public** **void** testRabbitBreeding()  {  System.*out*.println("Test breeding of rabbits");  //3x3 field  **int** arraySize = 9;  //Make a location for the rabbit, this location should allow the rabbit to breed 4 little rabbits  //because of the free space around it  location1 = **new** Location(1,1);  //create the rabbit  rabbit1 = **new** Rabbit(**false**,field,location1);  //to make sure they breed, set the breeding probability to 100%;  rabbit1.setBreedingProbability(1);  //Place a fox and a rabbit next to each other on the field  newAnimals = **new** ArrayList<Actor>();  newAnimals.add(rabbit1);  simulator = **new** Simulator(newAnimals, 3, 3);  Animal[] animal = **new** Animal[arraySize];  //no step has been taken, the rabbit should be there;  animal = getActors(simulator, arraySize);  *assertTrue*(animal[0] **instanceof** Rabbit);  //the other locations should be null:  **for**(**int** i=1;i<9;i++) {  *assertNull*(animal[i]);  }  //the rabbits should not be able to breed, because he is not 5 years old yet but 0  *assertEquals*(0, animal[0].getAge());  *assertFalse*(animal[0].canBreed());  //simulate 5 steps so that the rabbit will be able to breed  simulator.simulateNoStop(5);  animal = getActors(simulator, arraySize);  **boolean** breedingRabbit = **false**;  **int** rabbitCount = 0;  **for**(**int** i=0;i<arraySize;i++){  System.*out*.println("Step " + i);  **if**(animal[i] **instanceof** Rabbit)  {  **if**(animal[i].canBreed()) breedingRabbit = **true**;  System.*out*.println("Location " + i + ": " + animal[i].getAge() + "; breed?" + animal[i].canBreed()  + "; alive?" + animal[i].isAlive());  rabbitCount++;  }  **else** {  System.*out*.println("Location " + i + ": No animal");  }  }  //There should be more than one rabbits now  //Most of the time it will be five but it could be less because there might be overpopulation  //which results in death of some rabbits  *assertTrue*(rabbitCount > 1);  } |

Ook is er gebruik gemaakt van een ‘blackboxtest’. Nadat de interne werking van het programma goede resultaten had gegeven hebben we de applicatie vanuit het gebruikersoogpunt onderzocht. Dit is gedaan door extensief de simulatie te testen met verschillende optie en instellingen om te controleren dat alles werkte zoals het moest en dat er geen fouten werden terug gegeven.

# Hoofdstuk 7 – Conclusie

Het programma foxes-and-rabbits-v2 dat als uitgangspunt werd gebruikt had nog enkele tekortkomingen. Om het programma realistischer en functioneler te maken voor de eindgebruiker, bijvoorbeeld een natuurbeheerder, moesten er enkele veranderingen en uitbreidingen gemaakt worden.

Door het uitbreiden van de GUI is het programma een stuk gebruiksvriendelijker geworden. Dit maakt de simulatie een stuk overzichtelijker en geeft de gebruiker ook de mogelijkheid om zelf aanpassingen te maken aan de instellingen.

De simulatie bestaat nu uit 3 diersoorten, een jager, auto’s op een weg en de mogelijkheid om ziek te worden. Dit brengt de resultaten van de simulatie dichterbij de realiteit. Ook zijn er meerdere views aan de simulatie toegevoegd die op verschillende manieren aan de gebruiker laat zien hoe de situatie op dat moment is.

De aanpassingen die gemaakt zijn aan het programma maken het al een stuk realistischer en gebruiksvriendelijker.

# Bronnenlijst

MVC - <http://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller>  
Varkenscyclus - <http://nl.wikipedia.org/wiki/Varkenscyclus>

# Bijlage A – UML Klassendiagram V1



# Bijlage B – UML Klassendiagram V2





# Bijlage C – Reflecties

## Alwin

Het werken aan de leertaak foxes-and-rabbits was voor mij een echte uitdaging. Ondanks de modulaire functionaliteit van Java viel het mij zwaarder om in een groep te werken dan tijdens het PHP project @tafel van periode één. In mijn ogen komt dit doordat ik bij dit project moeilijker het gehele project in beheer kon houden. Bij het @tafel project lukte mij het om alle code van groepsgenoten na te lopen en te controleren, bij foxes-and-rabbits lukt dat simpelweg niet door tijdsgebrek. Dit komt gedeeltelijk doordat Java aanzienlijk moeilijker is dan PHP maar ook door een moeilijke periode in mijn privé leven.

Als perfectionist is het erg belangrijk voor mij om alle code te beheren, ook al leveren mijn groepsgenoten goed werk. Het gevolg hiervan was dat ik moeite had gemotiveerd te raken voor het project.

Ondanks dat het me niet lukte om alles van het project “in beheer” te houden, snap ik wel alle code van het project. Dit was een interessant leermoment omdat ik in voorgaande groepsopdrachten het me nooit is gelukt om aan een project bij te dragen zonder alle details te snappen. Deze keer lukte dat echter wel door de modulaire functionaliteit van Object Oriented Design.

Al met al hoop ik spoedig weer een Java project te krijgen omdat het me met mijn nieuwe ervaringen een leuke uitdaging lijkt.

## Arjan

Het project verliep naar mijn mening beter dan het project uit het vorige thema. De programmeurtaal Java zelf vond ik al een veel betere structuur hebben. Hierdoor was het samen werken ook makkelijker, ieder kan afzonderlijk aan een onderdeel werken zonder dat het invloed heeft op de andere. Ik vond het ook een project met meer uitdaging, aan het programma was het mogelijk om veel ideeën toe te passen. Onze zelfbedachte uitbreiding, de weg door het midden, was ook zo’n idee dat ons gelukt is om toe te passen. Door de tijdsdruk in de laatste paar dagen zijn meerdere ideeën ook niet gelukt, wat ik zelf jammer vindt.

Onze focus voor de laatste dagen was om de applicatie goed werkend te krijgen. De applicatie werkt, maar het zou beter kunnen. Ik vind zelf dat ik netter had moeten programmeren. In het begin van het project had ik het idee dat de moeilijkheidsgraad niet al te hoog was, maar dat viel tegen. Een aantal aanpassing uit de laatste week vond ik erg lastig om goed werkend te krijgen. Ik merkte ook dat naarmate we dichter bij de eindstreep kwamen dat de code steeds slordiger werd.

In vergelijking tot het eerste thema ging dit project veel beter. De werklast was beter verdeeld en de structuur was beter. Ik heb geleerd dat een belangrijke structuur voor zowel het programma als teamwork belangrijk is.

## Jelte

Het vossen en konijnen project was erg leerzaam. Ik had van te voren niet gedacht dat er zoveel tijd in ging zitten. Vooral het MVC model sprak mij aan. Omdat ik denk dat dit een goede applicatie structuur is om aan te leren en waarmee je in veel andere projecten ook iets mee kan.

In het begin vond ik het wat lastig om de MVC structuur onder de knie te krijgen. Wat voor mij geholpen heeft om dit onder de knie te krijgen is dat we als groep vanaf het begin af aan er na gestreefd hebben om de applicatie direct in een MVC structuur te bouwen, omdat er anders op een later tijdstip grote aanpassingen aan de applicatie gedaan moest worden.

Het samenwerken vond ik goed gaan. We hadden duidelijke afspraken over wie wat deed. Ik vond de projectafspraken hierbij een goede manier om te werken. Omdat je dan snel kunt overleggen wanneer iets niet duidelijk is en gezamenlijk problemen aan problemen kunt werken.

De begeleiding van tijdens dit project was prima. Ik heb eigenlijk geen problemen gehad die ik moest bespreken met de projectbegeleider tijdens het ontwikkelen van dit project.

Het project vond ik een leerzaam project en nuttig omdat dit denk dat dit een realistisch beeld geeft van de taken en verantwoordelijkheden van een ontwikkelaar.

## Brendan

Als 2e groeps-project voor thema 1.2 moesten we met een groep van 4 personen aan een simulatie applicatie werken. De bedoeling was om de simulatie uit te breiden en te verbeteren.

Dit leek mij wel een interessant project om te doen. Het was leuk om de balans tussen dieren in het wild te kunnen simuleren op de computer. Ik had gedacht dat het niet zo heel veel werk zou zijn maar het kostte toch nog best wat tijd.

Het samenwerken met de groep verliep goed. We zijn een paar keer samen op school bijeen gekomen om aan het project te werken en dingen te bespreken. Iedereen werkte goed mee en zorgde niet voor problemen.