|  |
| --- |
| Bart Bakker  Thijs Kuilman  Kevin Lankhuizen  Sergen Nurel  Rik Jan Schuringa  *Adviesbureau Caffeïne - Voor al uw advies voor IT toepassingen!*  Datum geschreven: 27 januari 2015  Datum van Inleveren: 28 januari 2015 |

|  |
| --- |
|  |
| Vossen & Konijnen |
| Einderapport werkzaamheden |



Samenvatting

Vossen en Konijnen is een simulatie die geschreven is in de programmeertaal Java. Het is ontworpen om een virtueel ecosysteem na te bootsen. De applicatie werd werkend maar incompleet aangeleverd door de opdrachtgever. Vervolgens is de applicatie doorontwikkeld en zijn er tal van nieuwe onderdelen toegevoegd aan de applicatie.

De al bestaande structuur is compleet veranderd en van de grond af opnieuw ingericht. Het oude model ging nog niet uit van een MVC. MVC staat voor Model-View-Controller. Dit is een bepaalde manier van het inrichten van programmacode die het mogelijk maakt de verschillende onderdelen van het programma modulair te schrijven. Daar zijn namelijk een aantal voordelen aan verbonden, waaronder de vergemakkelijking van het onderhouden van de applicatie.

De grafische interface heeft een update gekregen, waardoor de gebruiker nu eenvoudiger de applicatie kan bedienen en instellen. Er zijn namelijk een aantal grafieken, knoppen en andere elementen toegevoegd. Bovendien zijn een aantal van de knoppen voorzien van eenvoudige afbeeldingen zodat de gebruiker de functies eenvoudig kan herkennen.

Ook een instellingenpaneel ontbreekt niet aan het nieuwe programma. De gebruiker kan nu eenvoudig de verschillende variabelen aanpassen die de simulatie beïnvloeden. Zo kan de gebruiker bijvoorbeeld de vossen minder jongen laten werpen waardoor konijnen en dodo's een langer leven is beschoren.

Om de applicatie compleet te maken zijn ook geluiden en een heus weersysteem toegevoegd aan de applicatie. Zo wordt de gebruiker een compleet pakket aangeboden waarmee goed en productief onderzoek mee gedaan kan worden.

Er zijn echter wel wat verbeterpuntjes aan de applicatie. Zo wordt, naarmate de simulatie draait, de simulatiesnelheid lager. Dit moet nog een keer uitgezocht worden, maar voor nu is dit voldoende.

# 1. Inleiding

Het aangeleverde project had veel gebreken. Zo waren er geen invloeden van buitenaf en had de eindgebruiker geen inzicht in de werking van het ecosysteem. Tevens was het niet mogelijk om iets te doen in de simulatie aangezien alle knoppen ontbraken.

Het programma “Vossen en Konijnen” is ontwikkeld om inzicht te geven op de fluctuatie van de dierenpopulaties onder bepaalde situaties. De resultaten van de simulatie kunnen worden geanalyseerd om bijvoorbeeld het uitsterven van dieren tegen te gaan.

Uit onze analyse blijkt dat er een verband is tussen de vossen en de konijnen. Wanneer er veel konijnen zijn, begint het aantal vossen te groeien. Dit resulteert erin dat er meer konijnen worden opgegeten. Hierdoor krimpt het aantal konijnen weer, iets dat weer resulteert in een mindering in het aantal vossen. Er is een trend aanwezig dat vergelijkbaar is met de varkenscyclus. De varkenscyclus wordt hieronder uitgelegd.

Wanneer er veel vraag is naar varkens is de prijs erg hoog en zullen veel fokkers de dieren gaan fokken. Na verloop van tijd worden er echter minder varkens gegeten. Dit resulteert in een overvloed aan varkens, waardoor de prijs daalt en fokkers minder varkens zullen fokken.

Het programma dat ons werd aangereikt mist een aantal belangrijke onderdelen. Hieronder staat opgesomd wat er momenteel mist in het programma:

* Eindgebruikers moeten de mogelijkheid krijgen wijzigingen aan te brengen aan de simulatie
* Het systeem moet achter de schermen worden aangepast zodat het mogelijk wordt om dieren toe te voegen en de attributen van de dieren te wijzigen.
* In de code moet gebruik worden gemaakt van overerving. In het huidige systeem heeft een wijziging in een attribuut teveel invloed op het programma. Wanneer er gebruik wordt gemaakt van overerving, wordt het mogelijk wijzigingen door te voeren die weinig invloed hebben op het totale programma.
* De grafische interface moet worden uitgebreid. Er moeten knoppen komen om het programma te starten, te stoppen en te resetten.

Een mogelijkheid om het programma te verbeteren is het maken van een superklasse genaamd ‘Dier’. Vervolgens dient van elk dier een subklasse aangemaakt te worden. Doordat de super klasse alle basis onderdelen van een dier bevat, is het erg eenvoudig later nieuwe dieren toe te voegen.

# 2. Uitbreidingen

In het begin van dit project was de applicatie zeer beperkt. De opdrachtgever wenst dat de applicatie realistisch en volledig flexibel wordt. In dit rapport wordt uitgelegd hoe en waarom verschillende onderdelen aan de applicatie is toegevoegd.

* Hoofdstuk 3 - hierin wordt een korte uitleg gegeven over de implementatie niveau van de functies
* Hoofdstuk 4 - dit hoofdstuk zal zich voornamelijk richten op de herstructurering van het programma
* Hoofdstuk 5 - in dit hoofdstuk staat alles dat te maken heeft met wat de gebruiker ziet. De nieuwe grafische interface wordt ook toegelicht.
* Hoofdstuk 6 - hierin staat hoe systeem van de statistieken in elkaar steekt. Er wordt gesproken over de manier waarop data wordt verzameld en de manier waarop grafieken worden getekend.
* Hoofdstuk 7 - dit hoofdstuk zal gaan over de actoren van de simulatie. Deze actoren kunnen op het veld geplaatst worden.
* Hoofdstuk 8 - in dit hoofdstuk wordt uitgelegd hoe het programma is getest en op welke factoren hierbij is gelet.
* Hoofdstuk 9 - hierin wordt gesproken over bepaalde onderdelen die niet haalbaar bleken te zijn. De problemen worden in dit hoofdstuk toegelicht.
* Hoofdstuk 10 - hierin staat de conclusie van het project
* Hoofdstuk 11 - in dit hoofdstuk worden de uitbreidingen beschreven die het project niet hebben gehaald.

Helemaal achteraan kunnen bijlagen gevonden worden.

# 3. Beschrijving implementatie niveau

Bij de implementatie van de nieuwe functies en bij het verbeteren van bestaande code van het programma Vossen & Konijnen is er rekening mee gehouden dat de verschillende stukken code van de verschillende klassen zo min mogelijk afhankelijk van elkaar moeten zijn. Er is gebruikt gemaakt van de inkapseling strategie. Dit houdt in dat een klasse alleen code bevat dat strikt noodzakelijk is voor de methodes van deze klasse. Op deze manier heeft elke klasse een specifieke taak. Ook wordt er voorkomen dat als er iets bij moet komen of weggehaald moet worden er onnodige wijzigingen plaatsvinden.

# 4. Het gebruik van MVC in de simulatie

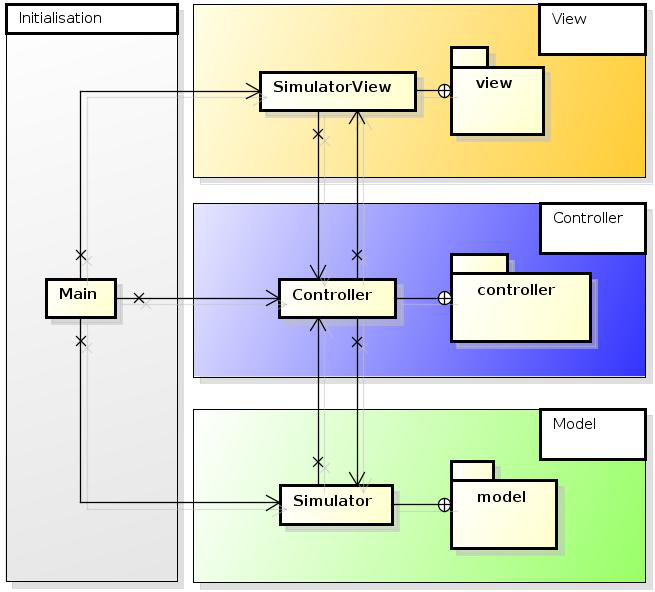
MVC staat voor model view controller. Dit is een standaard die wordt gebruikt bij het schrijven van software. Het doel van MVC is om de code overzichtelijk en makkelijk aanpasbaar te maken. Wanneer er gebruik wordt gemaakt van het MVC model worden stukken code opgedeeld in drie verschillende packages. Zoals de naam al zegt heten deze packages: Model, View en Controller. In de package model worden alle klassen neergezet die berekeningen uitvoeren. In controller komen klassen die de verbinding tussen View en Model verzorgen. Ten slotte komt in View het grafische gedeelte dat de gebruiker te zien krijgt.

Het project "Vossen en Konijnen" moest worden omgezet naar het MVC model. Doordat het project nog niet was voorzien van deze structuur was het erg onoverzichtelijk en werkte de software minder logisch. Alle klassen zaten namelijk in een folder en waren zeer verweven. Nu is alles opgedeeld in de drie packages en is de code zo onafhankelijk mogelijk. Hierdoor is het eenvoudig om dingen toe te voegen of te wijzigen.

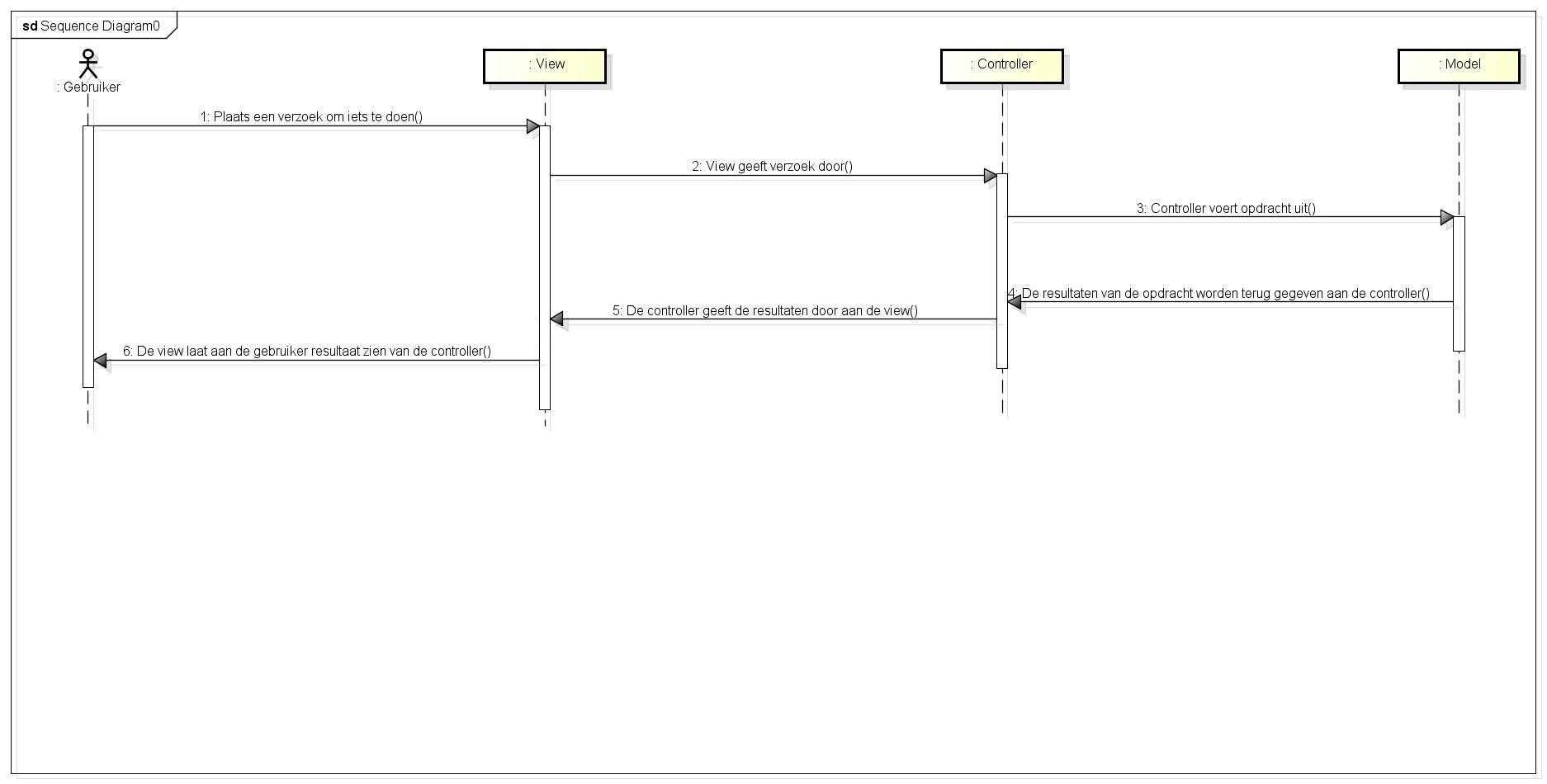
In de onderstaande afbeelding is te zien hoe de structuur er uiteindelijk uitziet.



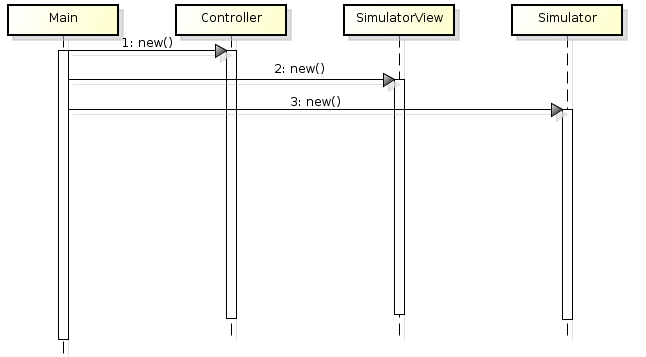
Op onderstaande afbeelding is de structuur van de packages schematisch weergegeven.



En in de vorm van een sequentie-diagram is te zien wat er gebeurd wanneer een gebruiker het programma gebruikt.



Het onderstaande sequentie diagram geeft het initialiseren van klassen weer.



In bijlage1 is de uitgebreide uitwerking van alle klassen terug te vinden in de klassendiagram.

# 5. De grafische interface

De grafische interface bevat alle grafische elementen van de applicatie buttons, grafieken en het rasterveld. Om de grafische interface te kunnen tonen dient de applicatie eerst te worden gestart. Dit wordt gedaan door de onderstaande code uit te voeren.

**public static void** main(String[] args) {

**new** Simulator().simulateOneStep();

}

Met deze functie wordt de simulatie aangeroepen, waarna de grafische interface wordt getoond en de gebruiker aan de slag kan gaan met de applicatie.

## 5.1 De grafische interface

De grafische interface bestaat uit een frame met daarin veel componenten. Hierbij kan gedact worden aan een bedieningspanelen, knoppen, labels en tekstvelden. De interface maakt het mogelijk dat de eindgebruiker de applicatie kan bedienen, aanpassen en de voortgang kan zien. Ook is het voor de gebruiker mogelijk om de instellingen van de applicatie aan te passen. Zo zou een gebruiker bijvoorbeeld bij een konijn de voortplanting sneller kunnen laten verlopen.

## 5.2 Het frame

Het frame is het venster waarin de grafische interface van de applicatie wordt weergegeven. Binnen het frame bevindt zich de grafische interface waarin alle componenten van de applicatie worden weergegeven.

## 5.3 De componenten

De componenten zorgen voor interactie tussen de software en de eindgebruiker. De belangrijkste componenten van de applicatie zijn:

* **De één stapknop** - deze knop zorgt ervoor dat de applicatie één stap zet.
* **De honderd stapknop** - deze knop zorgt ervoor dat de applicatie honderd stappen zet.
* **De duizend stapknop** - deze knop zorgt ervoor dat de applicatie duizend stap zet.
* **De start en stopknop** - wanneer de applicatie onafgebroken door moet gaan kan de start knop worden gebruikt. De stopknop zorgt ervoor dat de applicatie dieren stopt met het zetten van de volgende stap.
* **De resetknop** - om de stappen en de statistieken te resetten.
* **De instellingenknop** - om de instellingen van de applicatie aan te passen.



*Een aantal componenten die zijn toegevoegd aan de grafische interface.*

Elke knop roept de 'Actionhandler' aan. De actionhandler kijkt welke actie uitgevoerd moet worden en roept de juiste methode aan. Nadat de juiste methode is aangeroepen volgt er een bewerking plaats binnen de applicatie. Na de bewerking krijgt de gebruiker het resultaat van de bewerking te zien.

**if** (doThis.equals(**"plusEen**")) **simulator**.start(**1**);

Wanneer de applicatie gestart wordt, moet het eerst een aantal stappen doorlopen. Als de applicatie bijvoorbeeld oneindig door zou moeten gaan, dan moet de parameter veranderd worden naar -1.

Dit is gedaan bij de start knop. Nadat op de start knop of één van de andere stap knoppen is geklikt, wordt er een thread gestart. De thread stopt wanneer het aantal opgegeven stappen is bereikt of wanneer er op de stopknop wordt geklikt.

# 6. Grafieken en statistieken van dieren

In de applicatie Vossen & Konijnen wordt het straks mogelijk om data te monitoren in vorm van verschillende soorten grafieken. Er zijn vier soorten data die worden opgeslagen:

* **Aantal geboortes per dier** – Elke stap kunnen dieren zich voortplanten. Er wordt opgeslagen hoeveel geboortes er per dier gedurende een beurt plaatsvinden.
* **Aantal doden per dier** – Elke beurt sterven er dieren wanneer ze bijvoorbeeld een te hoge leeftijd hebben bereikt. Het aantal doden per dier wordt per beurt opgeslagen.
* **Aantal stappen per dier** – Wanneer een dier succesvol in het veld kan bewegen, wordt deze data opgenomen in het aantal stappen per dier per beurt.
* **Aantal levende dier** – Elk levend wezen wordt geteld. Zo kan worden gekeken wanneer de populatie daalt en wanneer de populatie stijgt.

Het gehele systeem moet het toevoegen van nieuwe soorten statistieken relatief eenvoudig maken.

Wanneer de data’s worden opgeslagen, moeten deze in een grafiek worden geplaatst. Voor elk type data wordt er gebruik gemaakt van verschillende soorten grafieken. Zo kan er gekozen worden tussen een staafdiagram, een lijngrafiek of een punt grafiek.

## 6.1 Implementatie van het systeem

Allereerst is er gekeken hoe de statistieken het beste opgeslagen kunnen worden. Er is gekozen om een statistieken model te maken die alle data kan opslaan en verwerken. Deze klasse heeft de volgende functies:

**Statistieken.java bevat de volgende belangrijke methodes:**

* AddData methode – Deze methode kan worden gebruikt om alle data per beurt op te slaan
* AddDataToHistory methode – Deze methode zorgt ervoor dat de data van een beurt worden toegevoegd aan een arraylist met de geschiedenis. Standaard worden de data van de laatste honderd beurten opgeslagen. Dit kan echter gemakkelijk aangepast worden met de statische variabele HISTORY\_TURNS.
* updateData() methode - Deze methode wordt aan het einde van elke stap aangeroepen. Dit stukje code is verantwoordelijk voor het aanroepen van de addDataToHistory() methode voor elk soort statistiek.
* convertToGraphData() methode – Voor de grafieken wordt er gebruik gemaakt van een externe bibliotheek. Een bibliotheek is een soort starterspakket voor een bepaald systeem. Voor het aanmaken van een grafiek is een double array een vereiste. Dit is een type variabel met een lijstje aan komma getallen. Deze methode zet de arraylist met de geschiedenis om naar een double array type. De laatst genoemde variabele wordt vervolgens teruggegeven. Zie de methode hieronder

**public double**[] convertToGraphData(List<Double> list){

*// Limitedlist contains the last HISTORY\_TURNS (100 standard) values of the history list.*

List<Double> limitedList = **new** ArrayList<>();

*// If the history list is empty, then fill it with data to prevent errors.*

**if**(list == **null** || list.isEmpty()){

**return new double**[]{**0**};

}

*// If the history list contains more than 100 values, then add the last 100 values to the limitedlist*

**if**(list.size() > **HISTORY\_TURNS**) {

**for** (**int i** = (**int**)**HISTORY\_TURNS**; i > **0**; i--) {

limitedList.add(list.get(list.size() - i));

}

} **else** {

limitedList = list;

}

*// Convert limitedlist into a double array. This is needed to create an graph*

**double**[] returnDouble = **new double**[limitedList.size()];

**for** (int i = **0**; i < returnDouble.**length**; i++){

returnDouble[i] = limitedList.get(i);

}

return returnDouble;

}

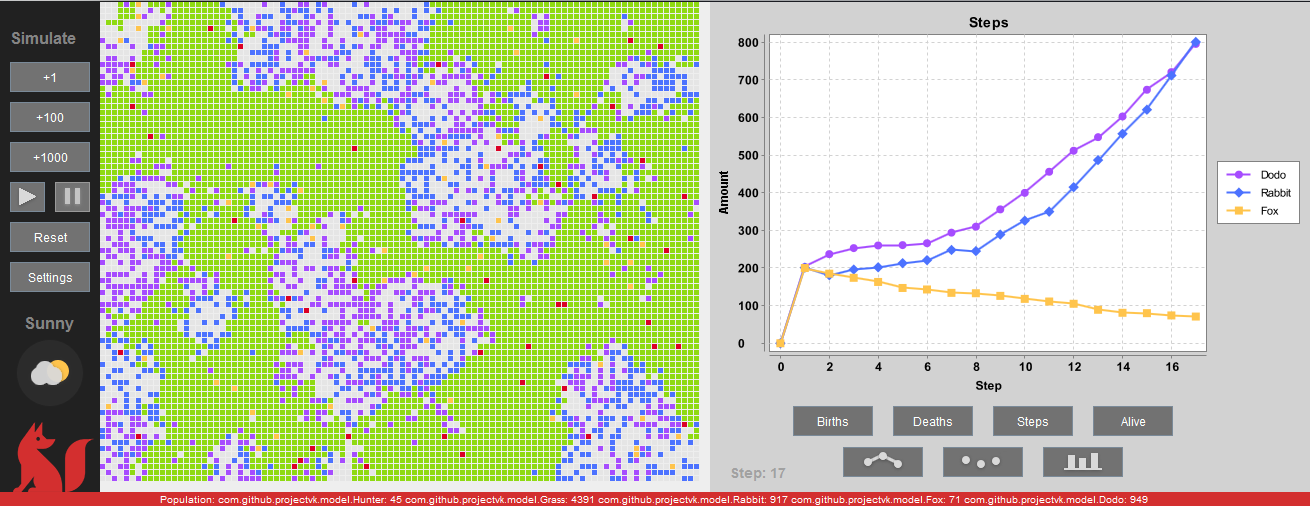
Met dit statistieken model is het erg makkelijk om elders data toe te voegen aan de statistieken. Hieronder is een voorbeeld waarbij een geboorte wordt geregistreerd. Allereerst wordt gekeken of er sprake is van een dier. Als dit correct is, dan wordt er bij dit dier één geboorte opgeteld.

**if** (getEntityClass() != Grass.class){

addData(Statistics.births, getEntityClass(), 1);

}

Met al deze data moet er uiteraard nog een grafiek getekend kunnen worden. Hierbij wordt gebruik gemaakt van XCharts. Dit is een lichtgewichte bibliotheek die het makkelijk maakt om verschillende soorten grafieken te tekenen. We hebben in totaal drie soorten grafieken: een lijngrafiek, een punt grafiek en een staafdiagram.



Zoals hierboven te zien is, is er voor gezorgd dat de gebruiker een goed overzicht heeft op zowel de simulatie als de grafieken. Met een druk op de knop kan de gebruiker wisselen tussen verschillende soorten grafieken. Daarnaast worden alle grafieken realtime weergegeven en aangepast.

## 6.2 Besmettelijke ziekte

In de applicatie Vossen & Konijnen is het mogelijk dat verschillende dieren besmet raken met een ziekte. Deze ziekte ontstaat bij de dieren en kan worden overgedragen aan andere dieren. De dieren die deze ziekte krijgen kunnen niet genezen worden en zullen dus dood gaan binnen vijf stappen.

De ziekte wordt bij de dieren bepaald door een willekeurige variabel. Als het dier de ziekte krijgt, dan heeft deze nog maximaal 5 stappen te leven. Andere dieren te dicht bij een ander ziek dier in de buurt komen worden ook besmet. Ook wanneer een vos een ziek dier op eet, zal het ook besmet worden. Bij de geboorte van een dier is er een kans aanwezig dat het dier besmet is met deze ziekte. Een jager kan niet besmet worden, omdat de jager geen dier is.

Hieronder ziet u een stukje code waarin wordt bepaald of het dier ziek wordt:

**public** **void** setSick() {

**if**(getRandom().nextDouble() <= getSicknessCatchProbability())**isSick** = **true**;

**if**(isSick() && getAge() > getMaxAge()-**5**){

setAge(getMaxAge()-**5**);

}

}

Verder is het mogelijk om de instellingen van de ziekte van verschillende dieren aan te passen. Zo kan bijvoorbeeld een konijn veel sneller ziek worden dan andere dieren. Hiermee kunnen natuurlijk interessante situaties ontstaan.

Het enige verschil tussen het ontwerp en de uiteindelijke implementatie is het gemis van een cirkeldiagram. Dit type diagram wordt namelijk niet ondersteund door de gebruikte bibliotheek. Dit is echter geen probleem, aangezien het erg gemakkelijk was om het te vervangen door een staafdiagram.

## 6.3 Voedselvoorraad

Er is een voedselvoorraad ingebouwd. Wanneer er veel konijnen zijn, is er weinig gras met als gevolg minder nakomelingen. Als er echter weinig konijnen zijn, dan is er veel gras en met als gevolg meer nakomelingen.

Er is een nieuwe klasse aangemaakt genaamd ‘Grass’. Deze klasse is, net als de verschillende dieren, onderdeel van de klasse NaturalEntity. Het gras heeft in principe veel eigenschappen die de dieren ook bevatten, zo kan er op ‘gejaagd’ worden door de konijnen. Ook kan gras zich ‘voortplanten’. Enkele verschillen tussen gras en een dier is dat het gras niet kan lopen en geen voedsel nodig heeft.

# 7. De aanwezige actoren

Om de applicatie soepel te laten lopen is de klasse actor toegevoegd, elk object dat een rol speelt in de applicatie is een actor. Elke actor heeft een locatie, een veld en een actie. Op dit moment heeft de applicatie vijf actoren, namelijk: gras, jager, konijn, vos en de dodo. Onder de actoren zijn ook actoren die in de natuur voorkomen. Zo kunnen dieren zich voortplanten, lopen en sterven. De jager kan op dieren jagen. Mocht een actor niet aan de voorwaarden voldoen dan implementeert de applicatie deze klasse niet.

**Gras**

Gras begint te groeien op het dichtsbijzijnde lege vak, dat betekent dat er geen dier op mag staan. Gras verspreidt zich alleen als het oud genoeg is om zaden te kunnen maken en het gaat pas dood als het opgegeten wordt. Het gras dient als voedsel voor de konijnen en dodo's. Echter kan het gras ook worden vertrapt door jagers en Vossen.

**Het konijn**

Een konijn is een prooidier dat gras eet. Konijnen worden voornamelijk opgegeten door vossen en worden doodgeschoten door jagers. Wanneer een konijn oud genoeg is en voldoende voedsel om zich heen heeft, plant het zich sneller voort dan wanneer er een voedselschaarste is.

**De vos**

In de applicatie speelt de vos een belangrijke rol. Hij jaagt op konijnen en dodo's. Ondanks dat de vos een langere levensduur heeft hoeft hij niet het langst te leven. Als de vos namelijk te dicht bij een jager komt wordt hij dood geschoten. Binnen de applicatie is de vos dus een prooi voor de jager.

**De dodo**

Binnen de applicatie heeft de dodo de minst gunstige positie, het dier eet alleen gras en er wordt op gejaagd door vossen en jagers. Omdat dodo's relatief kort te leven hebben en ze zich niet snel voortplanten sterven ze redelijk snel uit.

**De jager**

De jager heeft de unieke eigenschap dat hij niet dood kan gaan en zich ook niet kan voortplanten, daarom is de jager geen NaturalEntity. De klasse jager implementeert wel een methode van NaturalEntity namelijk het zoeken van een prooi. De jager doet dit op dezelfde manier als de vossen, konijnen en dodo's.

**Methoden**

Een entiteit heeft een aantal methoden om te kunnen jagen, voortplanten, lopen of sterven. Deze methoden staan allemaal in de klasse NaturalEntity en hoeven dus niet te worden overschreven door de klasse van het dier. Alle dieren zijn gebaseerd op NaturalEntity en kunnen dus dezelfde dingen doen. Zo wordt voorkomen dat vaak dezelfde code opnieuw geschreven moet worden.

**Acteren**

Wanneer de simulatie draait word de methode act regelmatig aangeroepen. Deze methode zorgt ervoor dat de NaturalEntity ouder wordt, hongeriger wordt, gaat broeden, voedsel zoekt, gaat jagen, gaat lopen of sterft. Wanneer de waarde binnen een entiteit verandert wordt dit door gegeven aan een grafiek.

**public** **void** act(List<Actor> actors) {

incrementAge();

incrementHunger();

**if**(isAlive()) {

giveBirth(actors);

*// Move towards a source of food if found.*

Location newLocation = findPrey();

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

*// No food found - try to move to a free location.*

newLocation = getField().freeAdjacentLocation(getLocation(), canOverrideGras());

}

*// See if it was possible to move.*

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

setLocation(newLocation);

if (getEntityClass() != Grass.**class**) Statistics.addData(Statistics.**steps**, getEntityClass(), **1**);

}

**else** {

*// Overcrowding.*

setDead();

}

}

}

**Sterven**

Wanneer een NaturalEntity voor een langere tijd geen voedsel heeft gehad of een bepaalde leeftijd heeft bereikt wordt de methode 'setDead' aangeroepen. Dan verdwijnt het dier van het scherm en van de simulatie. Ook de sterftecijfers worden bijgehouden in de statistieken en weergeven op de grafiek.

**protected void** setDead() {

alive = **false**;

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

field.clear(**location**);

location = **null**;

field = **null**;

}

**if** (getClass() != Grass.**class**) Statistics.addData(Statistics.**deaths**, getEntityClass(),**1**);

}

**Broeden**

Wanneer een entiteit volwassen is geworden en er voldoende voedsel aanwezig is, zal er een variabele aangemaakt worden waarin het aantal nakomelingen geregistreerd worden. Dit hangt af van andere variabele en de hoeveelheid open velden in de directe omgeving van de entiteit. Omdat broeden gaat via de klasse NaturalEntity wordt er gebruik gemaakt van een speciale manier om een ‘constructor’ aan te roepen van de klasse van de entiteit om een nieuwe instantie aan te kunnen maken.

actors.add((Actor)getEntityClass().getDeclaredConstructor(Boolean.**class**, Field.**class**, Location.**class**).newInstance(**false**, field, loc));

Deze methode zoekt een constructor in de klasse entiteit met drie variabelen: een boolean, locatie en veld. Door middel van deze constructor wordt er een nieuwe entiteit aangemaakt. Door het gebruik van deze manier van het creëren van nieuwe instanties is er minder code nodig.

## 7.1 Weersysteem (zelf bedachte opdracht)

Het weersysteem is een extra element aan de simulatie. Dit systeem introduceert de drie onderstaande weertypen:

* Zonnig: Wanneer het zonnig is, fluiten de vogels en spreidt het gras sneller.
* Regen: regen zorgt ervoor dat er meer gras groeit. Ook is het regen te horen voor

de gebruiker.

* Sneeuwstorm: Een sneeuwstorm zorgt ervoor dat er minder gras groeit.

De gebruiker wordt goed op de hoogte gebracht van het weer. Naast een dynamisch icoontje, zorgt het geluid ervoor dat een weersverandering duidelijk naar voren komt. Hieronder zijn de drie soorten icoontjes te zien.



### 7.1.1 De werking van het weersysteem

Het weersysteem is volledig willekeurig. Na een willekeurig aantal stappen wordt er een dobbelsteen gegooid om het weertype te bepalen. Wanneer er een weertype is bepaald, wordt de audio klasse aangeroepen om het geluid af te spelen. Ook wordt de controlpanel klasse aangeroepen om het weer-icoontje aan te passen.

Binnen de gras klasse wordt er continu gekeken welk weertype er op dat moment is. Afhankelijk van het type weer, wordt de bijbehorende gras variabele vermenigvuldigd met een getal.

### 7.1.2 Stukjes code

Hieronder wordt bepaald hoe groot de kans is op een bepaald type weer. In dit geval is het dus 50 procent kans op zon, 25 kans op sneeuw en 25 procent kans op regen.

**public void** randomWeather(){

*// Chances of a particular type of weather*

**double** randomDouble = Math.random();

if(randomDouble < **0.5**){

changeWeather(**"Sunny"**);

} **else if** (randomDouble < **0.75**){

changeWeather(**"Snow"**);

} **else** {

changeWeather(**"Rain"**);

}

}

Hieronder word een bijpassend geluidje afgespeeld. Aan deze methode dient de locatie van een geluidsbestand meegegeven te worden. Wanneer er problemen zijn met het afspelen van het geluidsbestand, wordt dit opgevangen in een zogenaamde exceptie. Dit laatste zorgt ervoor dat er een melding verschijnt met de bijbehorende foutcode.

**public void** playSound(String path) {

**try**{

**if**(!(getPlayingAudio.equals(**"default"**)) && !(**getPlayingAudio**.equals(path))){

clip.close();

}

**if**(!(**getPlayingAudio**.equals(path))) {

**getPlayingAudio** = path;

File soundFile = **new** File(path);

AudioInputStream sound = AudioSystem.getAudioInputStream(soundFile);

**clip** = AudioSystem.getClip();

**clip**.open(sound);

**clip**.start();

}

} **catch**(Exception e) {

e.printStackTrace();

}

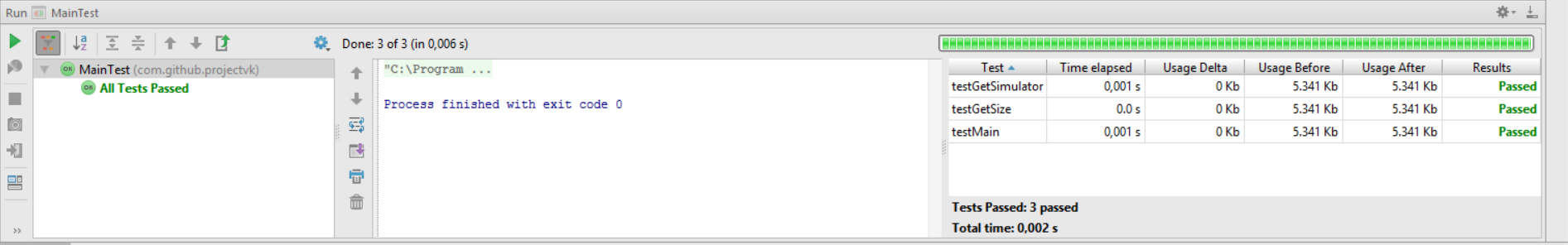
}

# 8. Testproces

## 8.1 Junit

JUnit is een framework en bibliotheek waarmee je java code kunt testen. Programmeurs kunnen hiermee veel tijd besparen wanneer er gezocht wordt naar een probleem.

Om JUnit kunnen gebruiken moet de bibliotheek eerst gedownload worden van de officiële site. Vervolgens kan het geïmporteerd in het project.



*Junit test package.*

## 8.2 Debuggen

Het vinden van bugs in een systeem, oftewel het vinden van een insect in een systeem, klinkt toch wel een beetje raar. Het woord is ontstaan in 1947 toen Grace Murray Hopper een probleem in een van de eerste computersystemen vond. Later bleek dat er vlieg/mot in relais nummer 70 op paneel F zat. In het logboek schreef zij "er zat een vlieg in het systeem". Nadien vond men de term debuggen uit. Debuggen betekent dus het opsporen van problemen binnen een programma of systeem. Tijdens het debuggen wordt de code stap voor stap doorlopen en kan er bepaald worden vanaf welke regel het programma niet meer loopt zoals het zou moeten lopen. In bijna alle programma's zit een debugger ingebouwd.

## 8.3 Testproces

### 8.3.1 Blackbox-testing

Bij blackbox-testing worden de functionaliteiten van de applicatie getest zonder dat er naar de werking van de code wordt gekeken. De functionaliteit van de applicatie is hierbij alleen belangrijk. Bij blackbox-testing worden de functies van verschillende klassen getest. Dit kan geautomatiseerd worden door bijvoorbeeld een ander stuk software. Er moet bij blackbox-testing ook getest worden als gebruiker. Hiervoor dient een tester alle mogelijke functionaliteiten van het systeem uit te voeren. Voordat Er wordt bij blackbox-testing voor de test genoteerd wat de verwachte resultaat zal zijn. De blackbox-test is in principe pas afgelopen als de verwachte resultaten gelijk zijn aan de resultaten van de tests.

### 8.3.2 Whitebox-testing

Bij whitebox-testing worden ook de functionaliteiten van de applicatie getest maar gebeurt dit op een lager niveau. Er wordt gekeken naar de code en wordt precies gekeken wat de code uitvoert. Bij whitebox-testing mag de code ook tijdelijk worden aangepast. Zo kan er bijvoorbeeld worden gekeken wat de resultaten zijn van een stuk code door bijvoorbeeld na het draaien van stuk code tekst te printen naar het scherm. Ook wordt er bij whitebox-testing verbeteringen toegevoegd aan de code van de applicatie. Net zoals bij blackbox-testing kan bij whitebox-testing het testen worden geautomatiseerd(bijvoorbeeld Junit).

### 8.3.3 Regressietesten

Bij regressietesten worden de niet-gewijzigde code getest op de functionaliteiten. Ook worden regressietesten uitgevoerd als bijvoorbeeld hardware of benodigde software veranderd. Een regressietesten is pas succesval als de functionaliteiten nog hetzelfde zijn.

### 8.3.4 Testproces Vossen & Konijnen

Voor het testproces van Vossen & Konijnen is gebruik gemaakt van de Java-package Junit. Er zijn verschillende test klassen aangemaakt zodat alle code grondig wordt getest(whitebox-testing). Eventuele fouten die tijdens het testen voorkwamen zijn verbeterd. Aan het einde van de whitebox-testing wordt erover gegaan naar de blackbox-testing. Nu worden alleen nog de functionaliteiten getest zonder de code aan te passen.

# 9. Refactoring

Er is regelmatig een stuk code gerefactord. Refactoren is het herstructureren van de broncode van een computerprogramma met als doel de leesbaarheid en onderhoudbaarheid te verbeteren of het stuk code te vereenvoudigen.

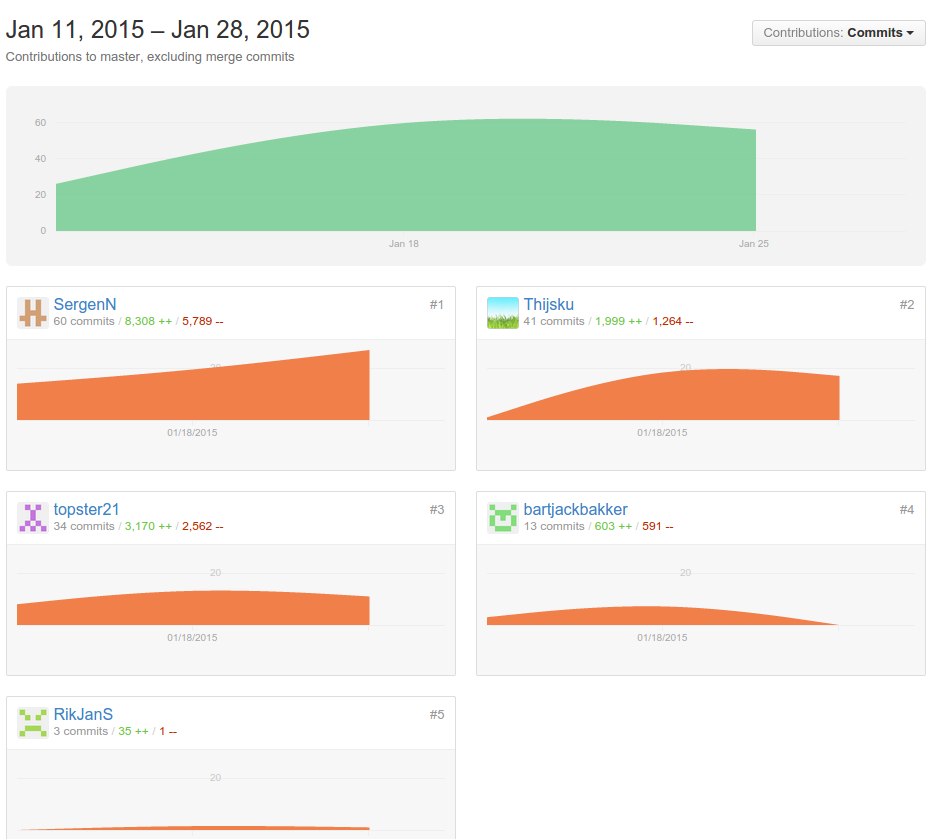
De ‘Animal’ klasse is één van de klassen die onder handen is genomen. Allereerst is de naam van de klasse veranderd naar ‘NaturalEntity’ om naast dieren ook bijvoorbeeld gras te accepteren. Ook zijn zoveel mogelijk methoden, zoals bijvoorbeeld act en breed verplaatst van een specifiek dier naar de hoofdklasse NaturalEntity.

Deze refactoring heeft tot resultaat dat de code makkelijker leesbaar is. Ook is overbodige code hiermee voorkomen.

# 10. Versiebeheer

Voor het schrijven van de programmacode is gebruik gemaakt van het versiebeheersysteem genaamd git. Git houdt wijzigingen in de code bij en zorgt er voor dat er een mogelijkheid is om terug te gaan naar een eerdere versie.

Om al deze wijzigingen te verzamelen en bij te houden is gebruik gemaakt van de website Github.com. Deze website geniet ruime bekendheid en is erg gebruiksvriendelijk. Zo is het erg gemakkelijk om overzichten op te vragen over bijdragen van verschillende teamleden aan de code.



# 11. Resterende problemen en mogelijke uitbreidingen

Helaas is de applicatie niet volledig en zitten er nog een aantal foutjes in. Zo ontbreekt een aantal grafieken en zijn er prestatieproblemen wanneer de simulatie lang draait.

In een later stadium kan dit prima verholpen worden, maar op dit moment is daar geen tijd voor.

Een lijst van toevoegingen en oplossingen die in een later stadium toegevoegd kunnen worden:

* Het oplossen van het prestatieprobleem
* Het toevoegen van de nieuwe grafieken
* Het toevoegen van een nucleaire explosie inclusief radioactief verval
* Het toevoegen van grasstatistieken
* Het toevoegen van meerdere dieren
* De applicatie beter schaalbaar maken
* Ziektes weergeven

# 12. Java documentatie

Het Java documentatie kan worden gegenereerd vanuit het ‘Intergrated Development Environment’ (IDE). De Java documentatie is handig als andere ontwikkelaars verder willen werken met het project, door het Java documentatie weten zij dan precies waar de methoden voor zijn. Tijdens het genereren worden er HTML bestanden gemaakt van alle documentaties die boven de methoden staan. De HTML bestanden kunnen daarna worden gehost op Github of een ander webhost. De documentatie van het vossen en konijnen project is momenteel gehost op Github en kan gevonden worden op de volgende site: <http://sergenn.github.io/ProjectVK/>

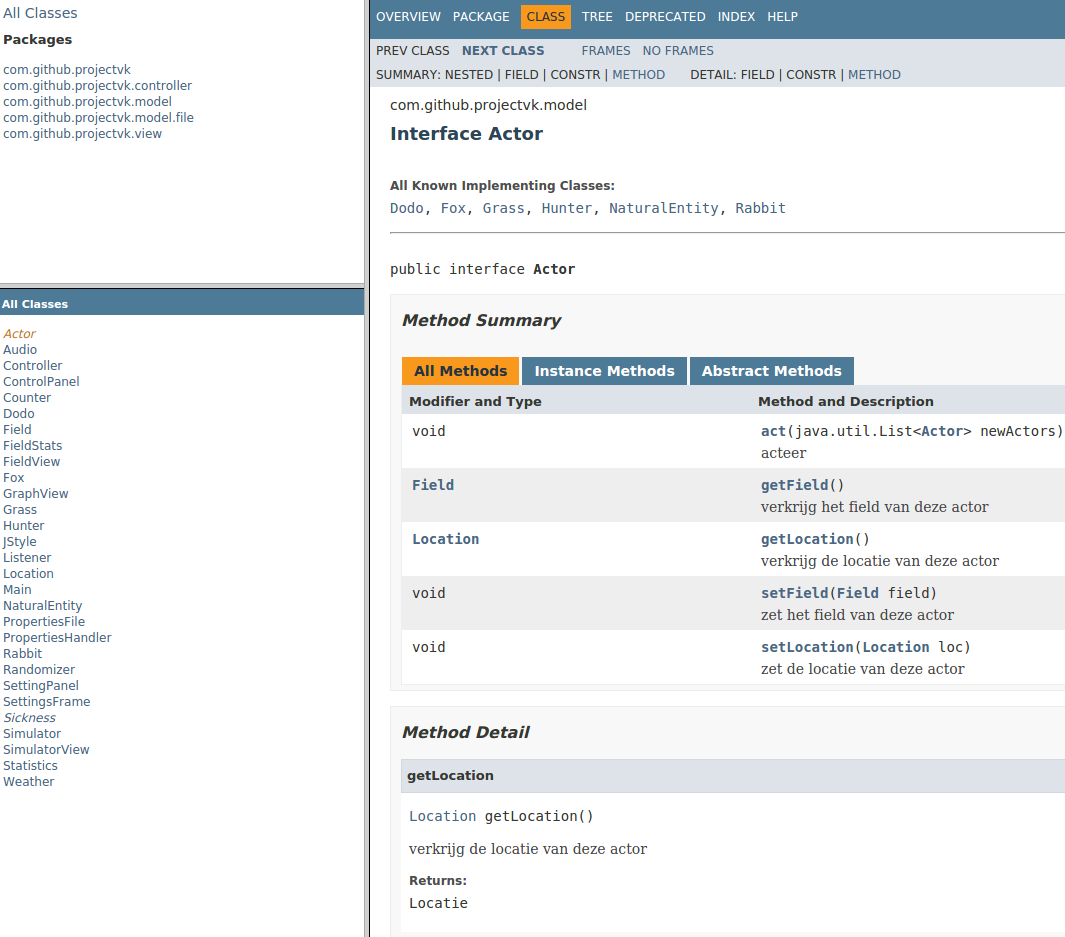
*/\*\**

*\* verkrijg de maximum leeftijd van een dier*

*\* @return int maximum leeftijd*

*\*/*

Een voorbeeld van een Java documentatie boven methodes in de bron code.



*Voorbeeld van de java documentatie dat momenteel gehost is op Github*

# 13. Conclusie

De applicatie was bij aanlevering zeer beperkt. Hierdoor was de applicatie tevens erg onrealistisch. Als de applicatie gebruikt werd, waren maar drie half werkende knoppen zichtbaar. De applicatie liep vast wanneer op de derde knop gedrukt werd. Bij het vooruitspringen met 100 stappen tekende de applicatie de tussenliggende stappen niet.

Het programma heeft een complete metamorfose gekregen. Zowel het visuele gebied als onderliggende infrastructuur werden verbeterd. De applicatie heeft een gloednieuwe grafische interface, met uitgebreide grafieken die de verzamelde data eenvoudig leesbaar maken. Er is nu tevens een scherm waarop de gebruiker instellingen van de verschillende onderdelen van de simulatie kan aanpassen.

Uniek aan deze applicatie is het weersysteem. Dit systeem heeft invloed op de groei van gras en dus op de gehele voedselketen. Op deze manier wordt de applicatie zo realistisch mogelijk.

Verder is de applicatie volledig in Model View Controller (MVC) structuur herschreven. Dit houdt in dat de verschillende onderdelen van de applicatie grotendeels los van elkaar opereren en via een controller met elkaar verbonden zijn.

Dan zijn er ook een aantal nieuwe actoren bijgekomen. Deze actoren zijn de Hunter of jager en Dodo. De jager jaagt op dodo's, konijnen en vossen. De dodo eet eigenlijk alleen maar gras.

Literatuurlijst

StackOverFlow

<http://stackoverflow.com/>

Xeiam Java XChart Library

<http://xeiam.com/xchart/>

Oracle Java Documentatie

<http://docs.oracle.com/javase/7/docs/api/>

<http://docs.oracle.com/javase/8/docs/api/>

Derek Banas - Java MVC Tutorial

<https://www.youtube.com/watch?v=dTVVa2gfht8>

JUnit

<http://junit.org/>

JetBrains IntelliJ Idea Documentatie

<https://www.jetbrains.com/idea/help/intellij-idea.html>

Whitebox & blackbox testing

<http://en.wikipedia.org/wiki/Black_box>

<http://en.wikipedia.org/wiki/White_box_(software_engineering)>

Jos Warmer en Anneke Kleppe - Praktisch UML (vijfde editie)

Programmeren in Java met Bluej (vijfde editie)

Bijlagen

Bijlage 1: Klassendiagrammen

Bijlage 1.1:

Bijlage 1.2:

Bijlage 1.3:

Bijlage 1.4:

Bijlage 1.5:

Bijlage 2: Individuele reflecties

Bijlage 2.1: Reflectie Bart Bakker

Bijlage 2.2: Reflectie Thijs Kuilman

Bijlage 2.3: Reflectie Kevin Lankhuizen

Bijlage 2.4: Reflectie Sergen Nurel

Bijlage 2.5: Reflectie Rik Jan Schuringa