WheninDoubt, use hashmaps

Project Diary

André Birk
Bror Hansen
Ayushmaan Bordoloi

21 December, 2023

Contents

1 Week 1

1.1 Resume

In the first week, we began our project journey. As the first thing, we created a group contract going over our work ethics and methods used in our group. We also finished the 'kom-i-gang' guide to get a good overview of the library use and it's methods. We went in depth with reading the description of this weeks theme to get a good idea of the requirements of this weeks task. We decided it would be a good idea to start with making a class diagram using UML. We began first working on the world, and then the first two classes, Grass and Rabbits as well as the file reader. Regarding the burrow, we have made some specifications for it regarding design but have not implemented it yet.

1.2 Design

We first started by implementing the grass class, which will inherit from our "IFauna" interface which currently has no special features other than extending Actor. This is for future proof of our design structure if more classes are regarded as "Fauna" or plants are to be implemented. This was relatively simple to implement as it has no real interaction with other classes yet. It needs to grow out and expand over time, and it needs to wither over time and it also needs to be able to be eaten by rabbits. We have then made our Rabbit class, which to just spawn in and eat grass was also relatively simple. But as we started designing the requirement that age was to change the energy level of the rabbit and then also deciding what energy actually dictates, that was a bit more difficult. Wer've decided that energy can decided three actions. How many tiles it can move, if it can reproduce or not as well as if it can dig borruws or not. This in conjunction with hungerlevel makes it a pretty good system depending on the different thresholds we've decided to put on these actions.

We decided that for the age of the rabbit, we would make it so rabbits have three different age stages: baby and adult, determined by the tick rate. Age transitions involve temporarily removing the current rabbit, spawning a new older rabbit in its place, and then deleting the old rabbit. The different ages determine max energy level.

The Burrows section details the creation of burrows, either through input files or if a rabbit has over 50 percent energy and hunger. A rabbit creating a burrow will always be connected to it, ensuring they enter it every night. If created by an input file, the first 6 rabbits within a specified tile radius will be connected. Nighttime behavior involves rabbits with connections entering burrows and those without connections being assigned to the nearest burrow. The maximum number of rabbits in a burrow is currently set at 8. However, the mechanism for rabbits searching and returning to burrows at night remains undecided.

We later on also have an idea to to abstract methods like eat(), checkHungerLevel(), and age(), with the intention to concretize these abstractions in the next theme of development.

We have not yet finished making the burrows, but we plan to make some sort of search method for the rabbits to search for the nearest burrow. This will probably also be used in the following weeks.

1.3 Testing

We have not yet made any progress on testing, as we have not yet made all the classes yet or finished the world. We will be doing this next week once we have finalised making the burrow and rabbit interactions finished.

2 Week 2

2.1 Resume

In this second week, we have finished making all the first week's requirements. So we have finished making the rabbit, borruw interaction system and also started doing jUnit testing for the different methods to document that they work as the requirements state. Of course, some of the requirements are a bit abstract and up to interpretation, and in that case we will be justifying out interpretation and how the test results are correct. We have also started on week 2 requirements with predators, flock animals and territories. We've tried to catch up from last week, but I unfortunately believe that we will still not be able to catch up completely with week 2 as week 2 is understandable more difficult than week 1. We've started creating the bear as it's at the top of the food chain, as well as the berries, and then we will be working on the wolf's interactions later on as they have a lot more complicated requirements than the bear. As stated in the recommended startup for this week, we will most definitely be making use of some of the abstract classes as well as interfaces that we made for last week like IFauna, IStructure and Animal.

2.2 Design

Regarding the borruws, we made it as it's supposed to. Rabbits can make burrows by random chance when they are above a certain threshold in terms of hunger level and energy level. When there's room available in a borruw, a rabbit can enter it at night. But it doesn't "search" towards it at night, so we might chance that later. Right now the rabbit is just removed from the map within a certain radius of the closes burrow. And then when it's morning that burrow will spawn the rabbits around the burrows.

For week two, we've started making the bear. And also the wolf. We will continue writing as we get more design made.

2.3 Testing

We've started doing jUnit testing on IntelliJ. It took a bit of time to figure out how to do it, but we've made a few tests and still need to finish a few more for documentation for week 1. As we continue to finish methods and class for week 2, we'll continue making more tests to ensure that everything works as it should

3 Week 3

3.1 Resume

In this week, we found out that we had already implemented a lot of the features for fungi and carcass' when we made week 1. So all the animals already created carcasses after they died as well as the carcass naturally disappearing after a while. So we actually finished fungi and carcass' relatively quickly. So we delegated the work where one finished week 3, and the other two worked on the wolfs, bears and berry-bushes. At this point we also updated our file-reader so we could add any entity to the map when we started using a custom txt file.

We've uploaded 2 UML-diagrams under the section *"Bilag"*, which includes diagrams of our current Classes and Interfaces [1]

3.2 Design

Carcass' vs Fungi, *and their interactive life cycle.*

In the prior weeks, when an Entity dies, the DisplayInformation got updated to a carcass, and then got deleted after some time. Now following the introduction of the fungal kingdom, the carcass is now a class in itself, alongside it's fungal counterparts, which extends carcass.

Speaking strictly in terms of the carcass, it's functionality and what event triggers the occurrence of creating a new carcass, is when any living entity dies of age, hunger or getting eaten by an entity of a higher status in the food chain.
The functionality of the carcass in itself is not really impressive, as it'll sit still and get removed up until a certain threshold. - This behavior still remains the same.

With this, there's some redundancy of code in each of the Animal-classes. So to counteract this, we've placed this logic in the abstract-class Animal. - more on this in a moment.

Now introducing the fungi, the carcass itself has the functionality of being infected with fungal spores and will change the behaviour of the carcass, as it can now spread the spore to neighbouring carcass'. This will also speed up the removal process of the individual carcass, as the fungal spore "eats" the carcass. Towards the end of a carcass' lifecycle, a new fungi will then sprout in wake of the devoured corpse.

So a Fungi can spread its spores to neighbouring carcasses, and the fungi kingdom will then flourish, at a rapid pace. - Though the only downside of a fungi, being that this can only happen if there is a carcass nearby, and if no other carcass' dies nearby then the entire fungi kingdom falls down.

So to counteract this, we've made sure the fungal class will stay on the map for a longer duration of time, so the seed of the fungi kingdom won't be completely destroyed. In other words, they're playing the long game as they're an extension of the animal-kingdom and will clean the map up, rather than following the death and life cycle of a typical animal.

DeathException and relating to the Abstract animal, *it's functionality and its motivation.*

The thought of an animal dying is rather grim, but also natural, which we've previously established. The reasoning behind the DeathException, being that since animals can die, and as of now in 3 different ways, and implying that the act of dying is a rare event, would mean that in terms of an animals life- and death-cycle *this should only occur once.* Another reasoning behind this, is to localise this logic in one method.

[1] Please do note that all the classes implements the interface IEntity, and IEntity extends Actor

Furthermore, including the cause of death in the exception, we're also able to print the specific cause of death in the console, for us to see the event log of the simulation.

The use of our DeathException also extends our abstract Animal as it allows us to catch the death of an animal in their respective act-method, and more specifically, in the event of an animal goes to sleep,

> " *'cause sleep is the cousin of death*" [2]

Some other features of the abstract animal is the eating, and the breeding logic is also implemented in the Act-method, so it also runs automatically, whilst the animal-specific behaviours can be coded in the individual classes.

The requirement **K3-1b** did force us to change some of the internal logic and behaviour of the energylevel field of the abstract Animal, but alas, nothing too major.

Now when it came to making new classes for bear, wolf and berry-bushes. At least for berry-bushes and bear, it was relatively easy. Off course each class extends and implements each of their respective interfaces and super classes. But for:

Berry-bushes: We made an int, berries which counts the number of berries the bush has at all times. Each step, a new berry is added. When it has above 5 berries, it becomes a berry bush. When a bear eats the berries, we assume it eats all the berries, hence the berry-bush becomes a normal bush again and then continues counting up (growing more berries). And bushes can only be put into the world if the file says so. Since it takes a rather long time in the real world to grow a lot of berries, the probability of it becoming a berry-bush are rather low.

Bear: We made it be able to eat everything as said in the description, this includes berries, grass, wolfs and rabbits. But Once it's placed in the world, it has a radius of 5 around the tile it was placed in where it can move around - it's territory. It can't go outside of that zone, and during night it sleeps and becomes inactive.

Rabbits: The only change we made to this is that they now run away from predators. They still randomly move around, but if a predator in a radius of x comes within, they begin to move in the opposite direction of the predators. Although I have not accounted for if there are multiple predators surrounding the rabbit. It most likely gets eaten there as it most likely also would in the real world.

Wolves: Wolves is probably the most challenging feature/animal to implement in the whole project due to it's pack requirement.

3.3 Testing

Our testing continues, although our testing has come to a halt as of now as we're doing maintenance on our fileReader and implementing methods; to simplify unit testing.

[2] Nas - "N.Y. State Of Mind", 1994; vers: 5, line: 7

4 Week 4

4.1 Resume

So for this week, we've done a whole lot of refactoring and cleaned up the code from our previous weeks by observing which parts can be improved with the use of ***WorldService*** which is our wrapper of ***World***. As part of the refactor the animals behaviours have changed. Wolves have yet to be fully implemented, specifically regarding the pack behaviour. We've begun on our rapport.

4.2 Design

WorldService *wrapping World and simplifying with Generics*

We've recognised that the ***world*** that we've been provided isn't perfect, in terms of its functionality.

So we've made a ***WorldService*** class that is a wrapper class. It contains several helper methods and uses generics so we can specify the type of ***IEntity*** we want for example getting a list of surrounding predators or finding the nearest animal of a specific type. They can be specified with type such as ***Bear*** or more generically like getting a list of animals by passing ***IFauna***. A few other helper methods include event listeners for day/night events which abstracts time away, moving an entity towards or away from another and simulating the world. This reduces work needed to be done in Animal classes and simplifies code. As of week 4 not all of the helper methods are completely implemented but due to the use of an interface ***IWorldService*** these methods can be defined and used by other classes while designing and the interface hides the inner implementation.

Observer pattern *day and night events.*
We created "events" for when there is a new day or night by implementing the Observer pattern. This makes defining entity behaviours much simpler as instead of having to do complicated calculations with tick counts, now simply "listening" for when it is day or night can be used to invoke behaviour such as sleep() or wake().

New Animal behaviours *reforming health/energy, breeding, dying, fungi*
As part of the refactor a new abstract Animal class was created and all "Small" variants killed of. Additionally as every animal shares the same behaviour of dying the behaviour of carcass was implemented in Animal and subsequently carcass classes killed of. The new Animal class implemented mechanisms for growing, energy, health, dying, breeding, sleeping and being infected. These methods could be overriden to define custom behaviours like Rabbit overrides sleep() for custom Burrow sleeping. For breeding reflection was used to dynamically add a new class of same type. Properties can also be overriden to define custom health, aging settings etc.

Now animal behaviours have also changed for example all animals have up to 100 energy which can be depleted and replenished. Any energy surplus is gained as extra health points for healing. Below 0 energy causes starvation with depletion of energy. 0 health causes death turning into a carcass. The carcass lasts for a defined amount of time before disappearing and if it was infected a Fungi takes it's place. The night and day event is listened to for waking/sleeping. Animals breed if they both are the opposite gender, adults and have enough energy.

Using enums *providing more flexibility than primitive types*
Primitive types that used to define age and health have been replaced to use enums such as AgeStatus which can be Child or Adult. This replaced using complicated techniques to calculate age with imagekey. Other enums also include EntityStatus giving info if animal is alive, dead, infected, carcass all in one replacing several booleans. Now it makes interacting with other animals easier for example a Bear can check if another bear is alive and a child and then not eat it. Or to find if an animal is

alive then kill it, or if its a carcass then directly eat it and if its infected carcass then gain less energy just with a few if statements.

More exceptions *since the code is exceptional*
NoEntityException if no entities nearby when called from WorldService or other purposes like no burrows. A LocationBlockedException allows entities to not be out of bounds.

***Our entity of choice**, its behaviour and also how it will impact the ecosystem*
So we've previously discussed how we wanted to implement a dragon for the simulation, that will come in and just destroy the whole world, ofcourse only a probability that it will instatiate. But that wouldn't fit the theme of the world. So we've opted for a lighter design of a dragon; it's real life counterpart, the salamander.

***The salamanders life- and death cycle** and what it eats*
Now the nature of a salamander is quite tame, as it will go around and only eat worms in the grass. This means that we've also changed the grass classes functionality, as it'll also have a *secret* counter which will increase with time, and also infect neighbouring grass tiles if they also had some to begin with. Thinking of backwards compatibility, this wont effect a thing, of the grass class, but just add another layer of complexity

The salamanders are a pretty docile creatures, as they'll primarily only be walking freely around and just eat the worms in the grass. This means that the primary competition the salamander would be the rabbit. But since the salamander is docile, it'll be shy and run away and find somewhere new it can eat.

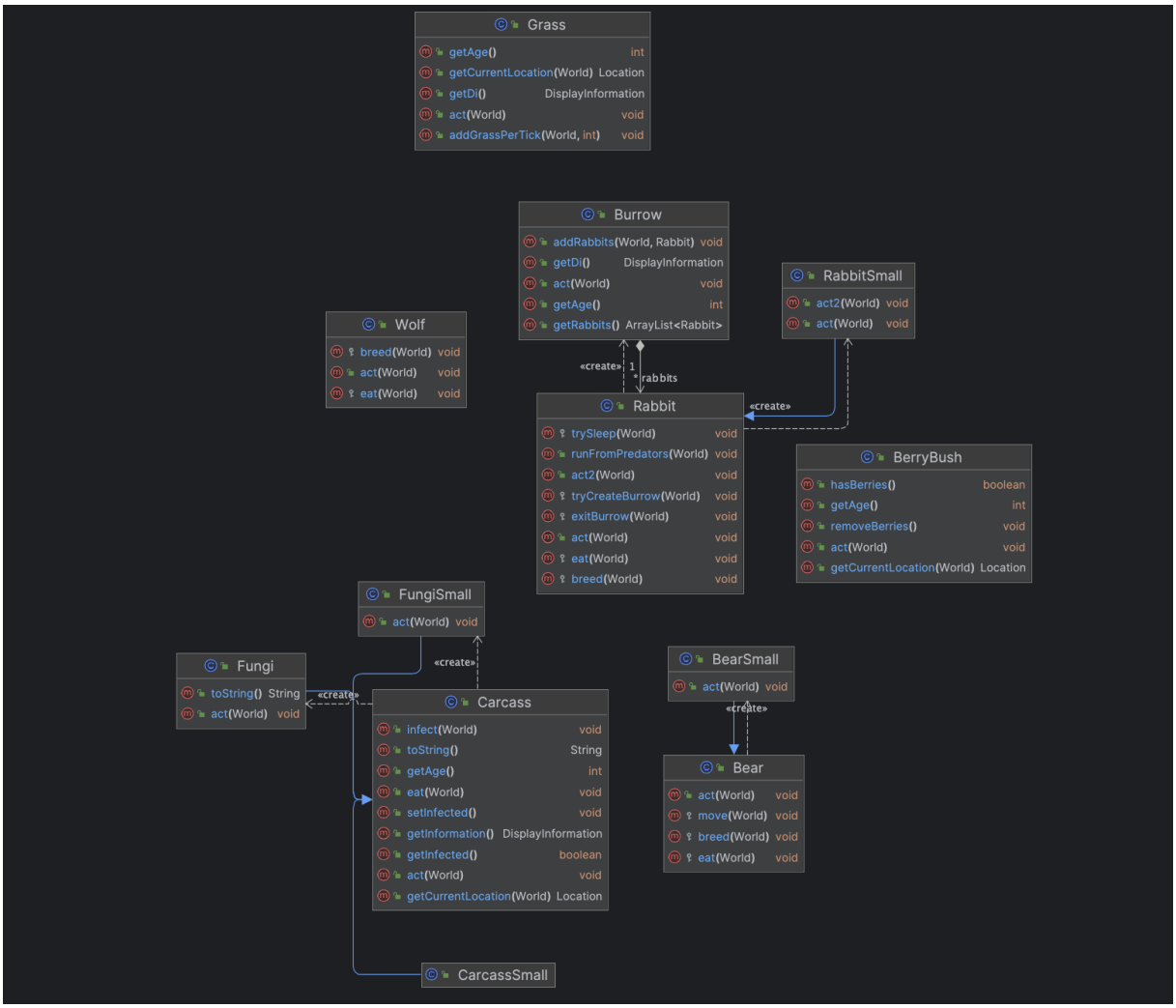
So to add an extra feature, or rather fun visual effect of the salamander, and also taking our personal spin on the creature *salamander*, we've made sure it'll breath fire when eating the worms for a nice and delicious hot toasted meal.

Now when a salamander encounters a predator this behaviour changes. Since the salamander in real life is poisonous, we've made sure that when a predator eats the salamander, it'll also rapidly decrease the life expectancy of the set predator.

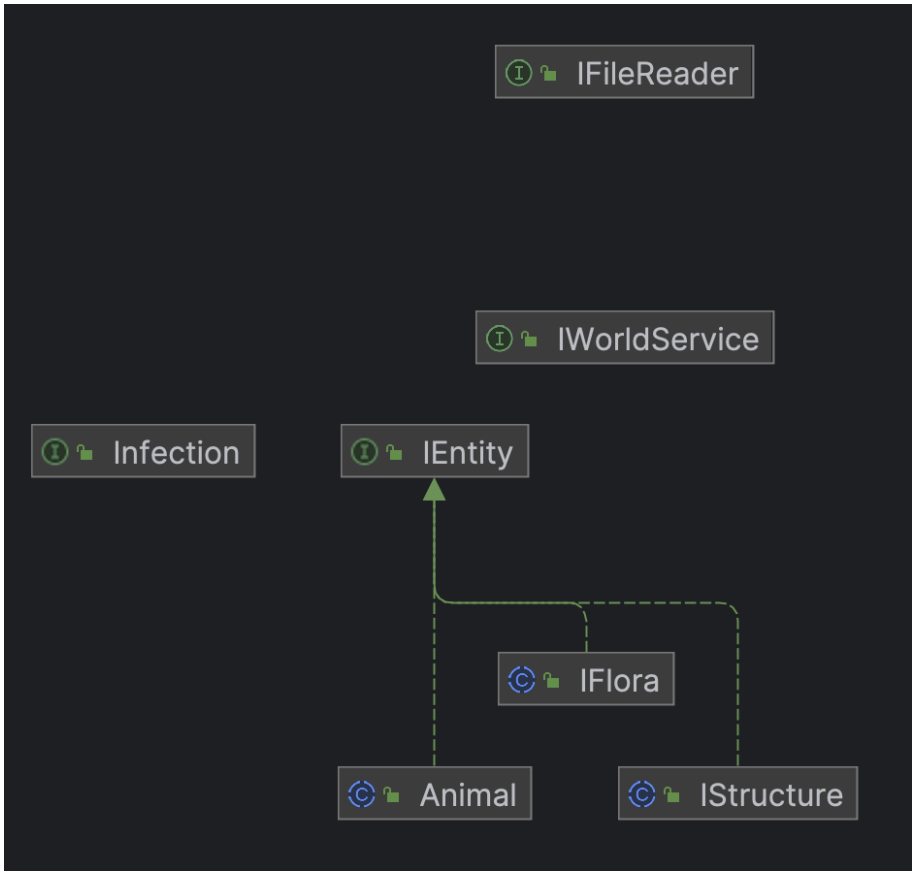
4.3 Testing

Regarding our tests. Since we've reworked our system, our previous tests has to be modified, which includes some last minute coffee hours, but we'll make do.

5 Bilag



Class Diagram for week 3



Interfaces Diagram for week 3