Week 1.

This week we have implemented the following features:

- **TestReader** - reads test files for information
- **Distributer** - distributes testfile paths
- **Borrow** - Rabbit holes
- **Grass** - Sweat green grass for the simulation
- **Rabbit** - A cute rabbit with various functions
- **BabyRabbit** - A cute baby rabbit (inherets from Rabbit)

TestReader

The TestReader reads the test files for information out the types and amount of entities in the simulation. The TestReader is being used along side the **Distributer**, to distribute the test files to the simulation. There are numores functions in the TestReader, so it is easy to get the information out of the test files. The TestReaders main function is **getMap** function, that returns a map of the test file with the corrusponting entity type and thier amount. TestReader also has a corrusponting test file, that is used to test the TestReader.
TestReader has the following functions:

getFilePath(); getFileContent(); getFileContentString(); getMap(); getRandomIntervalNumber(); getWorldSize(); isNumeric();

Distributer

The Distributer distributes the test files path for easy use in the simulation. The Distributer is a simple enum that stores all the test file paths and has some functions to get the test file paths.
Distributer has the following functions:

getUrl();

Borrow

The Borrow is a simple class that is used to create rabbit holes in the simulation. The does don't have a constructor and is only used to create and dispay rabit holes. This could be interface
Borrow has the following functions:

none

Grass

Grass is a class with diffrent functions that is used for the behavior of the grass object. One of the functions is the **spread** function. The function is used to spread the grass object to the surrounding tiles with a 10% chance of the grass spreading.
Grass has the following functions:

act($World$ world); spread($World$ world)

Rabbit

The Rabbit class constructs cute rabbits to the simulation with normal rabbit behaviors. Rabbits can move and eat grass, but unfortunately die of starvation as well. Most of the development time was spend on the Rabbit class. all requirements has not been fully developed this week, but the Rabbit class is somewhat functional. There has also been made a baby rabbit class, that is used to create baby rabbits for the reproduction function.
Rabbit has the following functions:

act(*World* world); digBorrow(*World* world); move(*World* world); die(*World* world); eat(*World* world); reproduce(*World* world); getRandomSurroundingTile(*World* world);

Baby Rabbit

The Baby Rabbit class is a class that is used to create baby rabbits for the reproduction function. The baby rabbit class is a simple class that is used to create baby rabbits. This class is not fully functional after this week, but will be worked on.
Baby Rabbit has the following functions:

@override act(*World* world);

Week 2.

Resumé

We used this week to make our program a lot more object oriented, utilizing the animal superclass a lot more often. We experimented with making a predator subclass or interface.

Design

Animal class

We've made a lot of changes to the rabbit class, and added other new classes. With 3 animals, we decided to make an Animal superclass.
We figured it would make sense to make a moveAway and moveTowards function, as a lot of the code was already written for the rabbit class. This utilizes our already created move function. We implemented a vision varaiable, that is used to determine how far away the animal can see, and a hunger variable, that is used to determine how hungry the animal is.

Rabbit class

We reworked the rabbit class to be more object oriented. We made a lot of the functions of rabbit into functions of the Animal superclass. We made a kind of behavior tree for rabbits, where they first will move towards their home if it is night, then move away from potential predators, then eat if possible, otherwise move towards grass, dig a burrow if it does not already have a home, a 20% chance to reproduce if possible, and finally a 50% chance move randomly. We also removed the baby rabbit class, as it was not needed.

Predator interface

We made a predator interface, that is implemented by the wolf and bear class. This is to make it easier to check if a class is a predator. We experimented with making an abstract predator class, but decided against it, as we had some issues with figuring out exactly how to make it work. The interface could be compared to the NonBlocking interface from the libary. The only function in predator is the abstract attack function, that the predators can use to attack other animals. This is primarily meant to be used by the bears if another animal is in their territory, or wolves if they are in a pack strong enough to attack a bear.

Bear class

The bear class is supposed to have a territory based on where it was spawned. The bear has different priorities based on some varaiables. If it is starving (has a hunger level below 3) it will search for food, not nessesarily in its territory. Food sources can be berrybushes or rabbits. If it is not starving, it will try to protect it territory. If any class that inherets from Animal is in its territory, it will attack it. if it is instead hungry, (has a hunger level below 8) it will search for food in its territory. We have not yet implemented the behavior of only finding food in the territory if it is not starving. otherwise it will move towards the center of the territory, or a random tile in the territory if it is already in the center.

Bush class

The bush (or berry as it is called in the input files) are a food source for the bear. We decided to not make them spread like grass, as they don't disappear when eaten. After the bear has eaten from the bush, it will take 5 steps during the day before the berries regrow.

Wolf class

The wolf, like the bear, implements the predator interface. It is supposed to hunt rabbits, and eat them. It functions a bit like the bear, except it does not have a territory. Instead it has a pack, that is a list of wolves.

Wolfpack class

The wolfpack is a class that is supposed to make the wolves hunt in packs. The leader of a wolfpack is the first wolf in the list containing the wolves. This class is responsible for adding the wolves to the lair.

Lair class

We remade the burrow into a lair class, that can also be used by the wolf.

FileReader class

We had to make changes to our FileReader class, as we had to read the input files in a different way. Since the way wolf packs and bears are defined in the input files, we had to slightly alter the way it reads them.

Thoughts for coming weeks

We still have some things missing from the requirements. When we refactored the animal class, we didn't find a way to incorporate energy. Although we are missing some things, we feel that we are on track to finish the project in time. We have had a lot of issues with the itumulator libary not working, sometimes because we were using an outdated version, and sometimes because they hadn't released a fixed version yet. Most of those problems are fine now, but we still wasted a lot of time on it. We are going to be more aware next week.

Week 3.

Resumé

For this week we had to rework the way animals eat, as they now had to eat carcasses, instead of simply eating another animal whole. We also used the start of the week of getting every member on the group up to speed on the different parts of the program. We talked about potential ways to refactor the code to be easier to understand and work with. Our filereader had to be redesigned again, as it did not work with how the input files process fungi.

Design

We had to reconsider how we wanted eating to work for animals. Earlier they consumed all of the prey in one "bite", but we figued it would make more sense for carcasses to have x amount of meat, and each predator consuming x amount of meat per "bite". Carcasses have a different amount of meat depending on what animal it was. We decided to make a default value for meat, in case a carcass is spawned via the input files.
We decided to make a fungus class as well. The fungus is supposed to spread to nearby carcasses, and consume them. We have made fungi be kind of "symbiotic" with a carcass. The carcass class has a function to add a new fungus to it. Each carcass can only have one fungus. The fungus has a eat function similar to predators, although with a "bite size" of only 1. It increases its size by 1 as well, each time it eats, which it has a certain chance to do every step. When it is big enough it has a chance to spread to other nearby carcasses. We thought about making the size increase the reach, but as of writing, fungi just has a static reach. When it spreads to other carcasses it uses the carcasses addFungus function to add a new fungus to that carcass. We thought about making the carcass have the same fungus that "infected" it, as it's own fungus, but we decided it made more sense to have a new one.
We also had to rework how we could fulfill the k1-2d requirement from week 1. It now makes the rabbit have a certain chance of doing nothing on a step, which is then multiplied by their age. All animals ages increment on the 19th step of a day/night cycle, or 1 step before the next day.
We also decided to make predator a class instead of an interface after all. It was a lot easier to understand how the program works with predator as a class instead of an interface. We figured out how to fix the issues with having predator as an abstract class, and decided to make it a class instead of an interface.

Testing

We experimented a bit with systematic testing, but mostly just tested cases as we went and discovered bugs. We had a lot of issues with the wolf packs from the previous week.

Week 4.

Resumé

This week we focused a lot on unit testing and making sure the program works as intended. For this weeks requirements we decided to make a snake. We thought it would be interesting to have a non-mammal animal, that lays eggs. Overall we have just fixing bugs, and getting ready to hand in the project and the report.

Design

We decided that the snake class would be a subclass of the predator class, as it should mimic snakes in real life. It does not do as much damage as the wolf or bear, but it applies poison damage to the target. The target will take 1 damage every step for 5 steps. We also made the bite-size of the snake 50, as we wanted to mimic the way real life snakes eat their food in one big bite. Snakes also lay an egg when they reproduce. The egg will unlike other animal's babies not be born as a baby snake, but instead as a snake egg. When an egg is laid, it will hatch after 15 steps. The egg will hatch into a baby snake, that will grow up to be a normal snake, much like other animals. Like other animals, the snake will not reproduce if it is not in its reproductive age, which is 3 days, as all other animals. Although we have not made an explicit food chain, we think of the snake as not that far up the food chain, while still being a wildcard. There is a chance that it will attack any animal it enocunters, but a wolfpack or a bear i generally going to be stronger.
We again reworked our filereader, as we had to make sure it supported the format of the input files. It should now hopefully also be ready to support theoretical new requirements, although it is of course not needed for this specific project.

Testing

Our primary focus for a lot of the week was on unit testing. We decided to try to make tests to prove as many of the requirements as possible. We discovered a few bugs while doing this, and fixed them. We made seperate test files for the different classes, and tried to make a test for each requirement we figured would make sense and was possible to test. We labeled them after the names of the requirements, and explained what they were testing in the comments.