WheninDoubt, use hashmaps

Project Report - Mini Worlds

André Birk
Bror Hansen
Ayushmaan Bordoloi
Total characters: 13 296

21 December, 2023

Contents

1 Introduction

Purpose

This mini-world project we are creating consists of creating a world simulation full of life. Each week we get introduced to themes introducing new elements and requirements, such as animal/plant life, where we are required to expand, refine, or rework previous solutions we have already made. We will be analyzing the requirements and creating designs that simulate real-world life and its many different ecosystems and interactions. To do this, we will use the IDE IntelliJ, to make use of everything we learned during the course in Java such as classes, methods, interfaces, abstract classes, loops etc. We will utilize Java documentation for each class and method to provide references and insight into our code. Furthermore, we will be making sure to test the program to ensure its accuracy and then make a report. This report will delve into our design and implementation decisions, offering a detailed explanation. Additionally, we'll assess the program's functionality and test coverage.

Project Description

For the mini-world, envision building a small world with animals, plants, and other actors. Each weekly theme delves into new aspects of interaction among these actors. We will use a project library called ITU-Emulator to visualize the life cycle of actors over weeks. Themes will focus on life, food chains, and decay. Each theme comes with specific requirements and accompanying input files that will shape and further develop our simulation.

To conclude the introduction, we have made a virtual world, which depending on the input files creates different plants and animals that live, grow, breed, die, eat, and interact with each other. Each plant and animal has specific relationships with each other and abide by logic and rules that we designed, which is meant to imitate the relationships they have in the real world.

2 Design

2.1 Creating the *world*

We created a ***fileReader*** which will take input files, with .txt in the end, and translate it to how many and what entities to load into the simulated world at compile-time. The file reader is modifiable so that any class can be created, added, and instantiated into the world.

The ***WorldService*** is a wrapper class for the given ITU-Emulator and includes helper methods for all the entities in our world to utilize. It also includes a multitude of useful methods for interacting with the ITU-Emulator. It abstracts processes of creating and simulating the world into just two methods. There are also helper methods which has abstracted away more complex code of working with World. Furthermore, they significantly simplify creating implementation of entity classes to interact with ***world*** in regards to the application of generics. The ***IWorldService*** interface allows defining the helper methods without implementing ***WorldService*** immediately and abstracting away details.

Explaining the use of:

2.1.1 Enums:

Also formally known as *Enumerations* offers an easy way to work with sets of related constants. An enum, is a symbolic name for a set of predefined values. This means that the set of enums can't be changed.[1] the ***entities*** of the world will have certain characteristics that should be predetermined before instantiating. (jf. project diary, week 4)

2.1.2 Generics

Generics is a feature in programming that allows to write code that works with a variety of data types while maintaining type safety. [2] In our case, We use type parameters to check for types without sacrificing manually written compile-time type checking. It also reduces code redundancy by enabling re-usability without the need for duplicate code.

2.2 Key concepts of the *entities'* design choices

The requirements for the given weeks have been open for interpretation. This section aims to explain and justify the more significant design choices and key concepts for different entities that have been created.

The ***rabbit*** is the prey of the simulation. They will jump around and eat ***grass***. They'll also conveniently use either provided or created ***burrow***s to rest in during the night time. It's a safe space for it to escape from roaming ***predators***, which are hunting.

The ***grass plant*** is a food source for multiple classes and will spread to the whole map. Another closely related ***plant*** is the ***berrybush***. Unlike the ***grass***, the bush itself isn't a food source, and cannot be eaten. But it will grow berries that can get consumed by ***bears***.

There are two predators and in terms of the food chain, the ***bear*** stands atop, with the ***wolves*** right beneath. While ***wolves*** move around in packs and hunt for ***rabbits***, and even occasionally fight other wolf packs, ***bears*** stays in their designated territory and protect it at all cost.

The ***fungi*** will thrive on the byproduct, the carcass, of the ***animal*** kingdom and will grow a ***fungi***. They do have the downside of not being able to grow if other predators eat the carcass.

[1] Coding with John. "Java Enums Explained in 6 Minutes". YouTube. 11 Oct. 2021. https://www.youtube.com/watch?v=wq9SJb8VeyM&t=352s

[2] Hodkinson, Jack. "Java Generics Explained." YouTube, YouTube, 20 Feb. 2021, www.youtube.com/watch?v=CN27X68YO4I&ab_channel=JackHodkinson.

2.3 The day- and night-cycle of our *world* and its inhabitants

The day and night cycle of our world restricts the ***entities*** flow of motion. This will add a dynamic element to the simulation that will influence the interactions of the individual creature of the virtual ***world***.

Another potential route of design is prioritizing the ***animal***'s state, like searching for food, running from predators, eating, etc. This could've been a realistic model of an ***animal***, as they will be assigned different tasks throughout the day and night.

So this program ***entities*** will have a constant flow of commands it can execute, rather than focusing on one "state".

2.4 What is the abstract *animal*

In terms of the modularity for the program, the design of an abstract ***Animal***, which hides the logic of the common behaviours as when it needs *to eat, to sleep and to move, and get killed.*

On one side, this means that all the extending classes from ***animal*** will inherit these behaviours, and the only specific behaviours that can be customized are the aforementioned three previous behavioural-logic.

On the other side, the requirements explicitly specify that there has to be a form of food-chain system implemented in the program, which makes a lot of sense. However, the abstract class in itself doesn't support this functionality. Therefore the predators implements the interface ***IPredator***, and likewise the prey implements ***IPrey*** (cf. Figure 6).

A huge part of an ***animal*** are it's life-stages. There's a multitude of ways to solve this issue, where one possible solution is creating sub-classes for the ***animal***. But this also means code redundancy for each ***animal***, by extending their "adult" self. So to solve this design issue, the ***animal*** has two constructors; One for it being an adult and another for being a baby or dead.

Having the stages of life in multiple classes may follow the single responsibility principle and can also lead to different life-stage behaviours. A downside of this could be issues with "adult"- and "baby"-behaviours.

Thinking of the application of an ***animal*** in the simulation, it should simply be born, grow up, and die. This introduces some flexibility in terms of modifying the "adult" methods so the babies can't use them. A plausible downside to this, would of course be a more complex ***animal*** where each method or behaviour needs to be considered.

2.5 The cycle of death & life of our *entities*

Since all living things must die at some point, there's an abstraction of life and death. This is represented by enums and the health system.

Alive

The health/hunger/energy system of an animal has been inspired by the videogame "Minecraft"[3]

Figure 1: Minecraft health system for a player.

The abstraction of life and death is linked to a single concept of health, as it's easier for the animal to be able to lose its health, as well as recover after a near-death experience.[4]

The health system is also closely intertwined with the common behaviour of "breeding" for an ***animal*** since considering the age of animals.

Dead

There are primarily 3 different ways, which include but are not limited to, death by starvation, age, or death by a predator. There's also the fungi kingdom that will thrive upon the carcasses left behind by the ***animal***s. This fungal infection will after a limited time end in the fungi kingdoms takeover. So the ***fungi*** kingdom, can be seen as a kind of "clean-up" of the deceased ***entities***, but in the end its about adding another layer of complexity during the runtime.

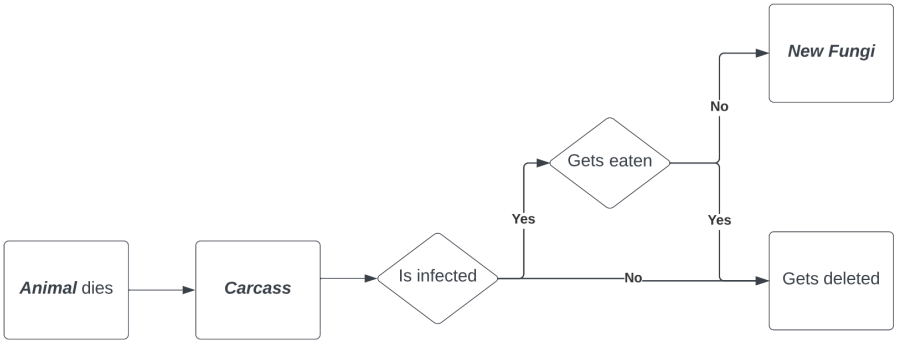

Figure 2: Flow chart of a dead ***animal***

Figure: 2 demonstrates the flow of a deceased animal, and also on how the ***fungi*** is an extension on the ***animal***-kingdom.

Plants (jf. Figure 5) doesn't have a "life- and death-cycle", as they serve the purpose of being fuel for the ***animals***. They'll increase in numbers, so not to starve the consumers. The same concept was also kept in mind when creating the bush and its food source, which are berries.

Figure 3: All **Salamander**s are created with the help of AI-generation. We've used the openAI DALLE-gpt4. The prompt for each design can be found in the appendix.

2.6 *Our entity of choice*

For our optional ***animal*** of choice, we've chosen a salamander.

We've previously discussed implementing a dragon, but for the sake of keeping it to the same theme of "simulating the nature of different ecosystems", the mythical dragon doesn't fit in. Chinese dragons therefore inspired this idea of the salamander.

When deciding on the ***animal***, we wanted one that could interact with the environment and other ***animal***s, while also having a unique trait. We found it fitting that their diet consists of worms[5], that they're poisonous to eat [6] and some species live in grasslands and forests [7] which are features of the existing environment. These traits make it a suitable candidate for the ecological environment of the ***world***.

The salamander extends the abstract class ***animal***, and will inherit all the common behaviours. Regarding having unique traits, the we came up with *poison*. This trait was added to the abstract ***animal***, so it's an added layer of complexity. The logic follows that any ***predator*** that eats the ***salamander*** will become very sick and die much faster. For the worms it eats, we modified ***grass*** to have a chance of generating worms. These worms spread to nearby ***grass*** filled tiles. Another unique trait is that it randomly breathes fire which can set bushes and grass on fire. The fire spreads to surrounding bushes and will also hurt nearby ***animal***s.

[3]Minecraft Fandom. "Health" Minecraft Wiki, https://minecraft.fandom.com/wiki/Health.
[4]The specific logic of this can be found in the JavaDoc
[5]The Editors of Encyclopædia Britannica. "Salamander." Encyclopædia Britannica, Encyclopædia Britannica, inc., 24 Nov. 2023, www.britannica.com/animal/salamander.
[6]Mike. "Are Salamanders Poisonous (and Dangerous for People)?" Amphibian Life, 18 Feb. 2023, www.amphibianlife.com/are-salamanders-poisonous-and-dangerous-for-people/.
[7]The Editors of Encyclopædia Britannica. "Salamander." Encyclopædia Britannica, Encyclopædia Britannica, inc., 24 Nov. 2023, www.britannica.com/animal/salamander.

3 Testing

Test design

The test design is based on the principles of unit testing, where each component of the system is tested in isolation. This approach allows us to verify the functionality of individual classes and methods, ensuring that they behave as expected.

In each test, we create different world sizes to speed up the testing depending on what the particular test needs. There has been used the SimulateSilent-method to run tests without the GUI to speed up the testing.

Choice of tests

The tests for the ***animal***, ***plant***, and mechanic classes, has tests that cover a wide range of scenarios. These tests are designed to verify the behaviour of the classes under both typical and edge conditions. For example, in the case of the ***WolfPack*** class, we have tests that verify the behaviour of the pack when it is hunting, when it is resting, and when it is interacting with other packs.

Besides the ***animal*** and ***plant*** classes, we've created tests for the ***Animal*** abstract class as well as the ***WorldService*** and ***FileReader*** classes as these are crucial for the core functionality of the mini world.

We chose the ***Animal*** abstract class as it contains core methods that all our animal classes share such as ageing, moving, breeding, infecting carcass, animals being killed, and consumed. If these methods don't work correctly, then any class extending ***Animal*** won't function properly. A UML diagram of all our test classes and their methods can be seen in Figure 8 of the appendix.

4 Conclusion

The program supports a dynamic and lively world simulation, which has evolved several times as we attempted to create seamless implementations of entities that each have their unique relationships with each other. The primary objective was to construct a virtual ecosystem, which we have achieved by making design choices consisting of several life forms, each adhering to specific rules and logic. To make sure it all worked out we have also thoroughly tested the methods to make sure the logic is sound.

Our design phases have been key moments in our world simulation's functionality. Specifically, the ***FileReader*** allows us to seamlessly integrate new entities specified in the input files. The ***WorldService*** also was a key design choice that acted as a wrapper class, facilitating all our methods and interactions with the ITU-Emulator library. Our day-and-night cycle was a big design aspect that influenced all of our virtual inhabitant's behaviours. The cycle emulates real-world conditions which adds realism to the entity's cross-interactions. Furthermore, our abstract ***Animal*** class was part of our foundation for inheriting common animal behaviours while allowing specializations through extending classes and interfaces. Lastly, just like our abstract Animal class, our ***Enums*** played a huge role in abstracting death behaviours and making behaviours such as starvation, ageing, and predation much easier to work with.

There are also different life cycles of the ***entities***. One of them is in terms of the ***animal*** kingdom, and the other the ***fungi*** kingdom as a consequence.

In conclusion, our mini-world project displays a complex and real-life ever-changing ecosystem. The world, with the help of well-thought-out design choices and implementations, is a dynamic simulation that we made sure works with the help of testing.

5 Appendix

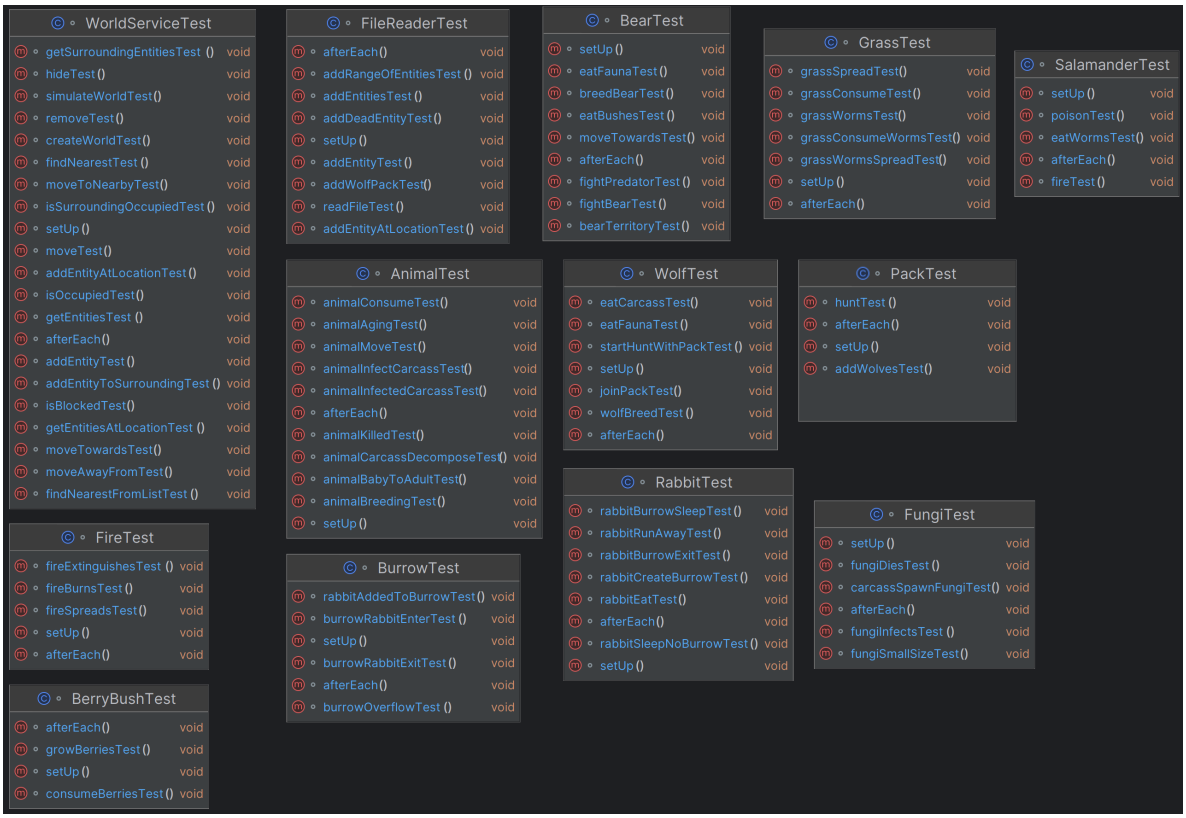

Figure 4: UML diagram of all our test classes and individual unit tests

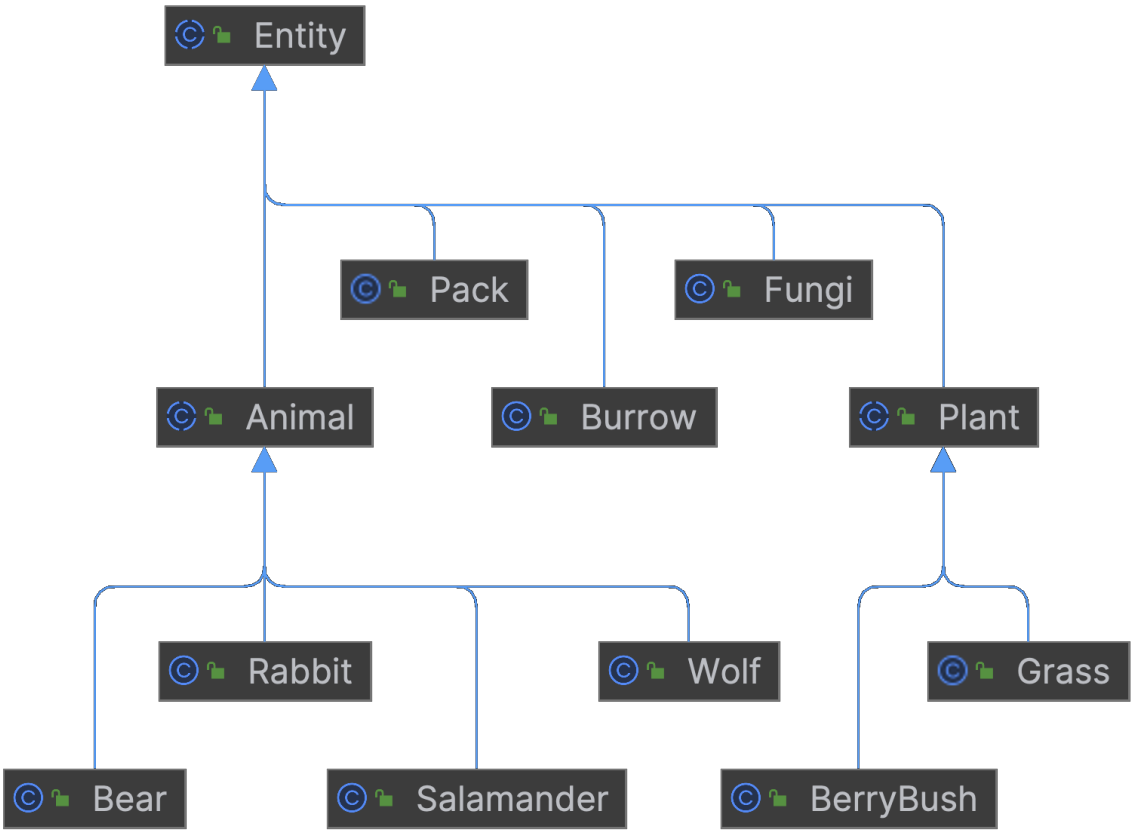

Figure 5: UML-Diagram of the classes

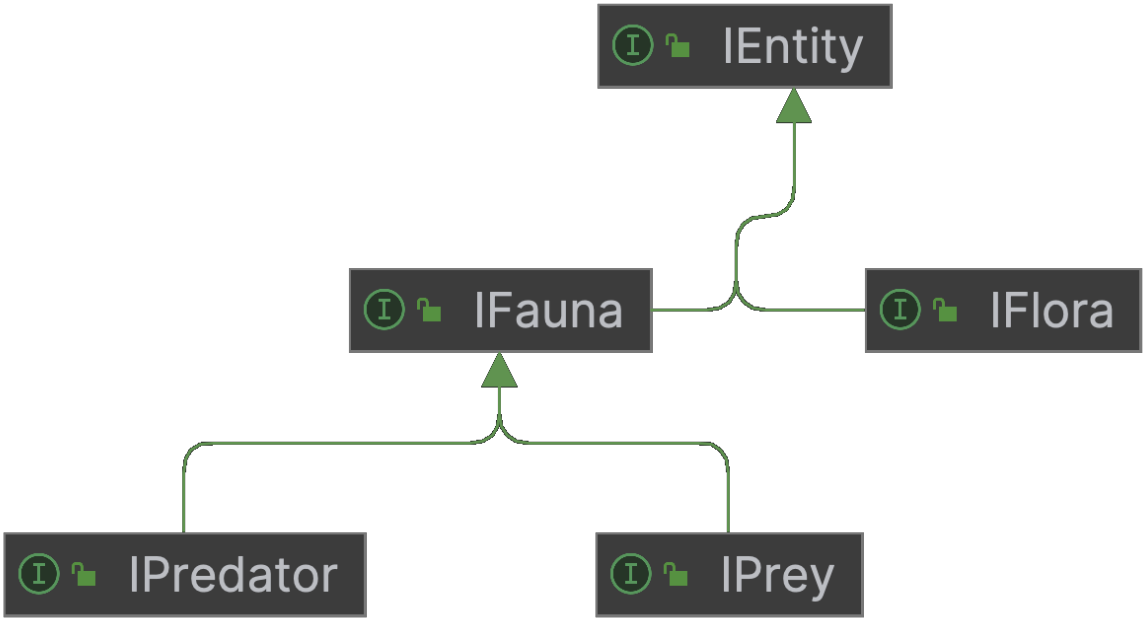

Figure 6: UML-Diagram of the interfaces

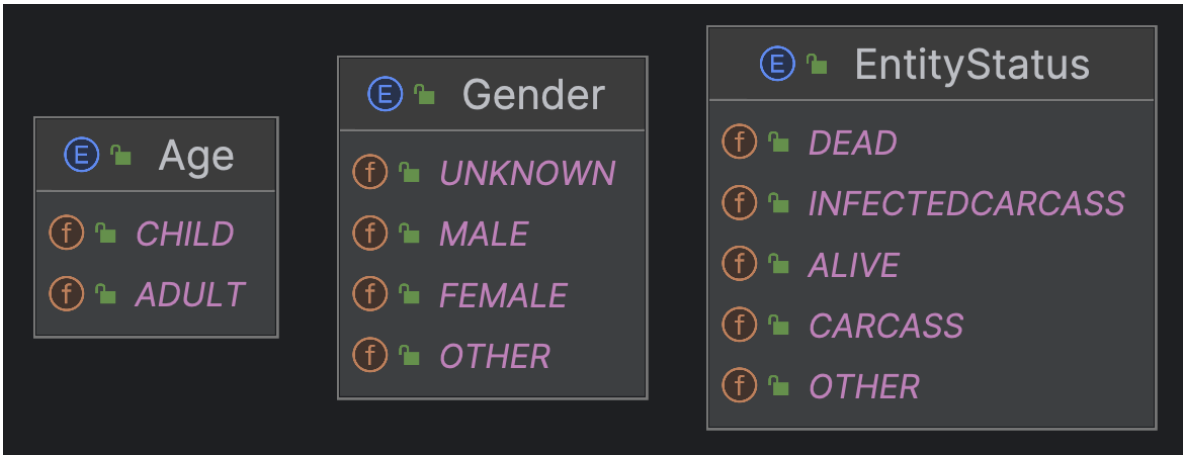

Figure 7: UML-Diagram of the Enums

```
@Test
void animalKilledTest(){
    world.Create( size: 3);
    Rabbit a1 = new Rabbit(world);
    world.Add(a1, new Location( x: 2,  y: 2));
    world.Add(new Fire(world), new Location( x: 2,  y: 1));
    world.Add(new Fire(world), new Location( x: 1,  y: 1));
    world.Add(new Fire(world), new Location( x: 2,  y: 0));
    world.SimulateSilent( Amount: 80);
    assertSame(a1.getStatus(), EntityStatus.CARCASS);
}
```

Figure 8: Example code of animalKilledTest

Figure 9: Image has been created by using Chat-GPT4 using the following prompt: *Subject: A salamander character. Style: 8-bit pixel art, similar to early video game graphics. Pose: The salamander standing upright with a prominent, curved tail. Color Palette: Predominantly red for the salamander with white and black for details and highlights. Special Features: Large eyes to give the character a friendly appearance. Additional Elements: A shadow under the salamander to ground it within the scene.*

Figure 10: Image has been created by using Chat-GPT4 using the following prompt: *Subject: A baby salamander character. Style: 8-bit pixel art, characteristic of classic video game graphics. Pose: The salamander standing upright, with a slight curve to its body and tail prominently displayed. Color Palette: Body: Different shades of red for the main body. Belly: A lighter color, possibly white or pale yellow. Eyes: Large with white, blue, and black to give a glossy, friendly appearance. Accents: Orange or similar warm tones for detailing and patterns on the body. Shadow: Include a pixelated shadow under the salamander to create a sense of depth and ground the character. Background: A dark or black background to ensure the character stands out.*

Figure 11: Image has been created by using Chat-GPT4 using the following prompt: *Subject: A salamander character. Style: 8-bit, reminiscent of classic video game graphics. Action: Breathing fire. Pose: The salamander should be standing with an upright posture. Color Palette for the Salamander: Shades of red for the body, with lighter tones for the belly and darker tones for the outline and details. Fire Details: A dynamic gradient of yellows, oranges, and reds to create the flame, designed in pixel art style to appear as if bursting from the salamander's mouth. Ambience: The image should capture the energy and intensity of the salamander's fiery breath.*

Figure 12: Image has been created by using Chat-GPT4 using the following prompt: *Subject: A salamander character. Style: 8-bit pixel art, typical of vintage video games. Action: The salamander breathing fire. Pose: The salamander should be standing with its mouth open, facing to the side, with a flame emerging from its mouth. Color Palette: Salamander: Reds and oranges with white for the belly and details, and darker tones for the outlines. Fire: A gradient from yellow at the base to reds and oranges at the tips with some white for highlights to represent the intensity of the flame. Background: Black or another dark color to help the bright colors of the salamander and the fire stand out. Effects: Add some glow around the fire to enhance the effect of light on the surrounding area.*

Figure 13: Image has been created by using Chat-GPT4 using the following prompt: *Subject: A sleeping baby salamander character. Style: 8-bit pixel art, resembling retro video game sprites. Pose: The salamander should be curled up in a circular shape, reminiscent of a sleeping position. Color Palette: Predominantly red for the body with variations in shading to suggest curvature, and white along with a few blue pixels for the eyes to indicate they are closed. Additional Elements: Pixelated 'Zzz' symbols above the salamander to denote sleep. Background: A transparent or solid color background that does not distract from the character.*

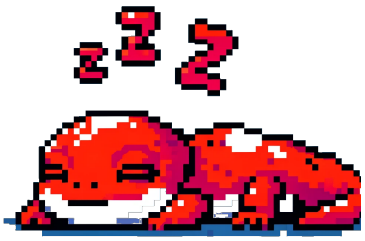

Figure 14: Image has been created by using Chat-GPT4 using the following prompt: *Subject: A sleeping salamander character. Style: 8-bit pixel art, reminiscent of classic video games. Pose: The salamander should be in a lying down, curled position, similar to how a dog sleeps. Color Palette: Main body: Various shades of red for depth and contouring. Belly and face: Lighter shades like white or pale yellow. Eyes: Closed, with blue accents to suggest peaceful sleep. Additional Elements: Pixelated 'Zzz' symbols in red above the character to represent sleep. Background: Black or very dark to contrast sharply with the character and emphasize the pixel art style. Overall Mood: Tranquil and restful, capturing the essence of sleep.*