Assignment 3

Group 71

Name	Student Nr.	Email
Kloena Burazeri	2723423	k.burazeri@student.vu.nl
Andrei Ioan Daraban	2727462	a.daraban@student.vu.nl
Lara Ichli	2730901	l.ichli@student.vu.nl
Håvard Skjærstein	2780502	h.skjaerstein@student.vu.nl

Summary of changes from Assignment 2

Authors: Håvard, Kloena

Provide a bullet list summarizing all the changes you performed in Assignment 2 for addressing our feedback.

In summary, Assignment 2 was a success, and there is not much to address regarding feedback. Our team worked diligently to complete the assignment, and we were able to meet all the requirements outlined in the project brief. The code was well-structured, and we made efforts to maintain consistency with the original design. We received positive feedback from the TA, but the sequence and state machine diagram had some flaws:

State machine diagram

The feedback we received on the state diagram indicated that the descriptions were generally good, but there were some minor errors and the level of detail was basic. Hence, we have taken measures to enhance the diagram by addressing the identified errors and augmenting its comprehensiveness. Main concerns regarded the second State Machine diagram, presenting a few errors that needed to get addressed.

Additionally to the feedback provided,the diagrams were also changed in order to accommodate and reflect the behavior of the system.

Sequence diagram

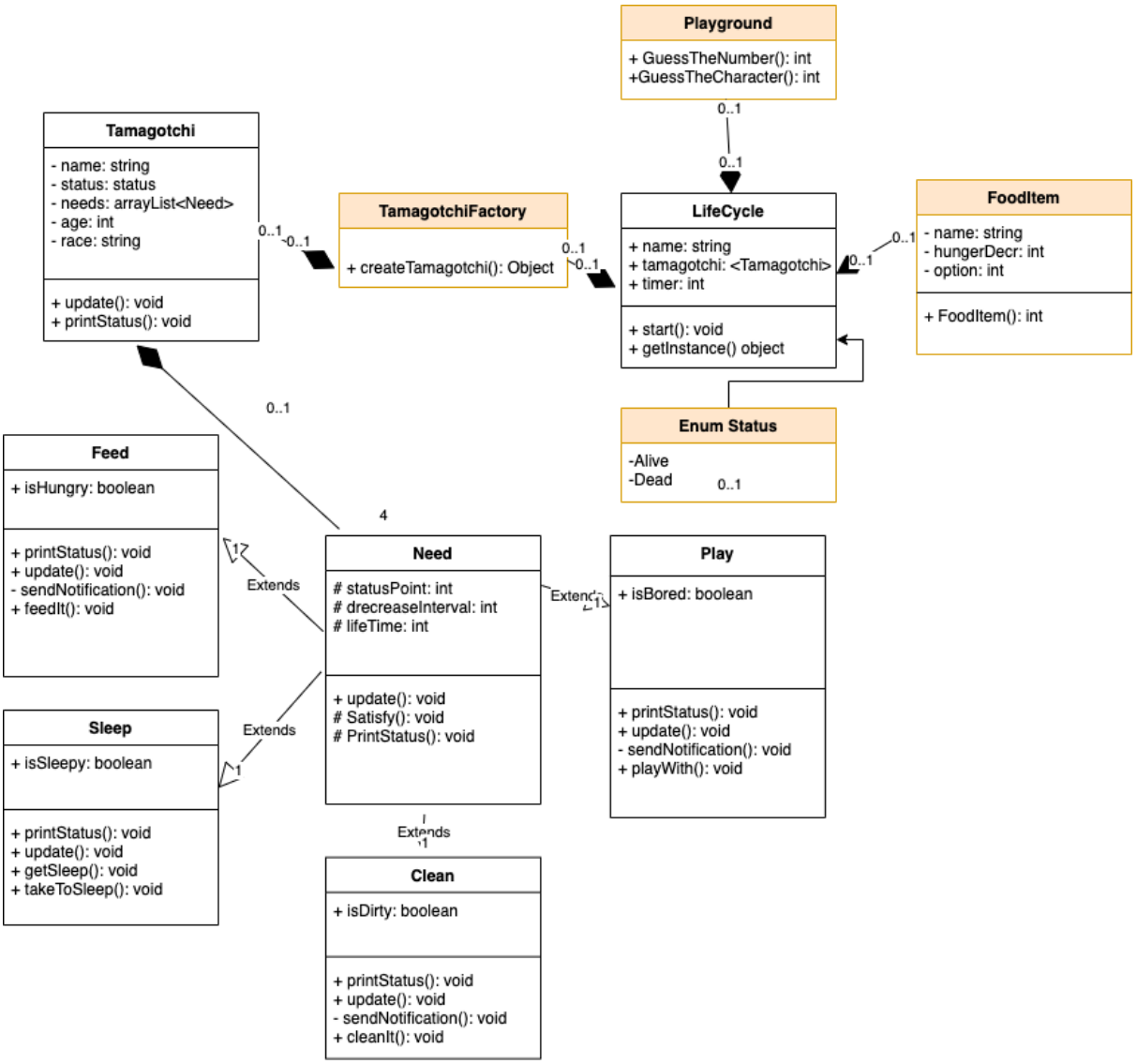
As for the sequence diagram, the feedback suggested that it had room for improvement. Specifically, there were errors present and the description lacked clear reasoning. In response, we made these changes to enhance its quality.

We made several improvements to our program, including changing the synchronous messages to asynchronous ones. To better represent the actor's role in the program, we changed their representation to a stickman. We also provided more thorough reasoning for the choices we made in the program.

To demonstrate the full functionality of our program, we added to the already-existing diagrams. We made sure that all the diagrams accurately matched the implementation of the program, ensuring that there were no discrepancies between the two.

Revised class diagram

Author: Håvard



The changes in this UML are highlighted in yellow, in respect to the previous versions.

The **Tamagotchi Factory** is a specialized class that generates instances of the **Tamagotchi** class. Its implementation allows us to centralize the creation process of Tamagotchi objects and abstract it from the rest of the code. This results in a more modular and maintainable codebase over time.

By using the **Tamagotchi Factory**, we can create new **Tamagotchi** instances easily without worrying about the intricate details of its creation process. This class also offers greater flexibility as it enables us to modify how Tamagotchi objects are created without having to modify the code that uses them.

In order to work around the limitation of Java not supporting multiple inheritance, some changes were made to the **MiniGame class**. It was rewritten to utilize composition instead, and now exists as an instance within the **LifeCycle** class rather than being a subclass of it.

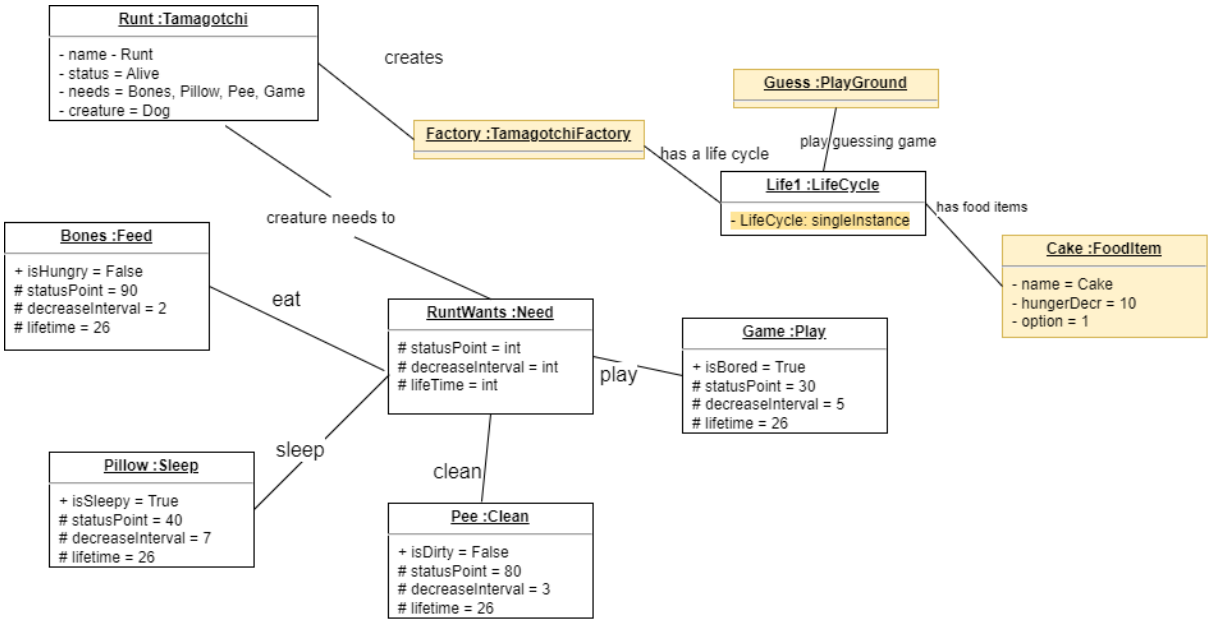
To further improve the code's modularity and maintainability, the **FoodItem class** was introduced. It allows the player to choose from a menu of food items, and abstracts away the implementation details of different food items from the **LifeCycle** class. This makes it easier to maintain the code and make changes in the future.

We made some minor changes to the names of the classes and attributes in the code. Rather than updating the code directly, we opted to update the diagram instead. This was simply a matter of convenience for us.

Overall, the program's design underwent some changes, but the majority of the diagram remained faithful to the original plan. This group made a concerted effort to stay on track with the initial design, resulting in a cohesive and effective system that meets all the original methods, attributes, and class relationships.

Revised object diagram

Author: Kloena



An object diagram is a visual representation of objects of the classes of a system and their relationships during a specific point in time. It allows us to examine how our objects interact with each other and can be used to analyze and understand the behavior of a system at a specific point of time during execution. Therefore we applied all of the changes we made to this object diagram so it matches our new class diagram.

Firstly, the Factory object which creates the Runt object was created as an instance of the *TamagotchiFactory* class. Now this object has one LifeCycle. Furthermore, Life1, has only one attribute now which is an instance of itself, since the class LifeCycle is now a Singleton. In Life1 you can now play 1 out of 2 guessing games represented by Guess. Lastly, Life1 has several food items for the user to choose from like Cake. This object has a *name* and *hungerDecr* which decrements the hunger level and an *option* relating to the user's choice.

Overall, the object diagram did not undergo that many changes except for the choices made in order to apply design patterns like Singleton and Factory Method.

Application of design patterns

Authors: Kloena, Andrei

	DP1
Design pattern	Singleton
Problem	We want to ensure that there is only one instance of the **LifeCycle** object at any given time because if there were multiple instances, it could lead to inconsistent states, conflicting updates, and unexpected behavior. Because the **LifeCycle** object manages the different stages of the **Tamagotchi's** life cycle, it needs to be ensured that only one instance of the **LifeCycle** object can exist at a time.
Solution	A singleton is a creational design pattern that lets you ensure that a class has only one instance, while still providing a global access point to this instance. By using a private constructor and a static instance variable, we can ensure that only one instance of a class is ever created. The Singleton class also provides a global method to access the instance, which can be used to reach the instance throughout the program. We can ensure that there is only ever one instance of **LifeCycle** at any given time and that all access to the **LifeCycle** object is through a global method.
Intended use	At runtime, the Singleton pattern ensures that only one instance of the **LifeCycle** object exists. When the program first starts, the Singleton instance is created, and subsequent attempts to create a new **LifeCycle** object will always return the same instance. Any code that needs to access the **LifeCycle** object can do so by calling the *getInstance*() method of the Singleton class, which returns the single instance of **LifeCycle**. The *getInstance*() method should be the only way of getting the Singleton object.
Constraints	The Singleton Pattern imposes the constraint that the Singleton class must have a private constructor to prevent external instantiation and that the Singleton instance must be accessed through a public static method. Additionally, care must be taken to ensure thread safety in a multithreaded environment.
Additional remarks	The Singleton Pattern is a commonly used design pattern in software development, especially in situations where there should be only one instance of a particular class. In a Tamagotchi game, it can be used to ensure that certain game elements are consistent throughout the game, and that memory usage is optimized by preventing unnecessary object creation. However, overuse of the Singleton Pattern can lead to inflexibility and poor code organization.

	DP2
Design	Factory Method

pattern	
Problem	The problem with creating **Tamagotchi** game objects is that the specific type of **Tamagotchi** that needs to be created is not known until runtime. It would be impractical to have a constructor for each possible type of **Tamagotchi**.
Solution	Factory Method is a creational design pattern that provides an interface for creating objects in a superclass but allows subclasses to alter the type of objects that will be created. The Factory Method Pattern solves the problem way, at runtime, the specific type of Tamagotchi object can be created without modifying the existing code.
Intended use	At run-time, the client code will use the Factory Method to create **Tamagotchi** objects. The client code will not be aware of the specific type of **Tamagotchi** object being made, as that is determined by the concrete factory subclass that is being used.
Constraints	The creation of **Tamagotchi** objects must be done through a factory interface, which requires creating additional classes. However, this constraint leads to better code organization and more flexibility in the creation of **Tamagotchi** objects.
Additional remarks	The Factory Method Pattern is a common design pattern used in many applications, including game development. It is a useful pattern to use when creating objects with multiple possible types, but the specific object type is not known until runtime.

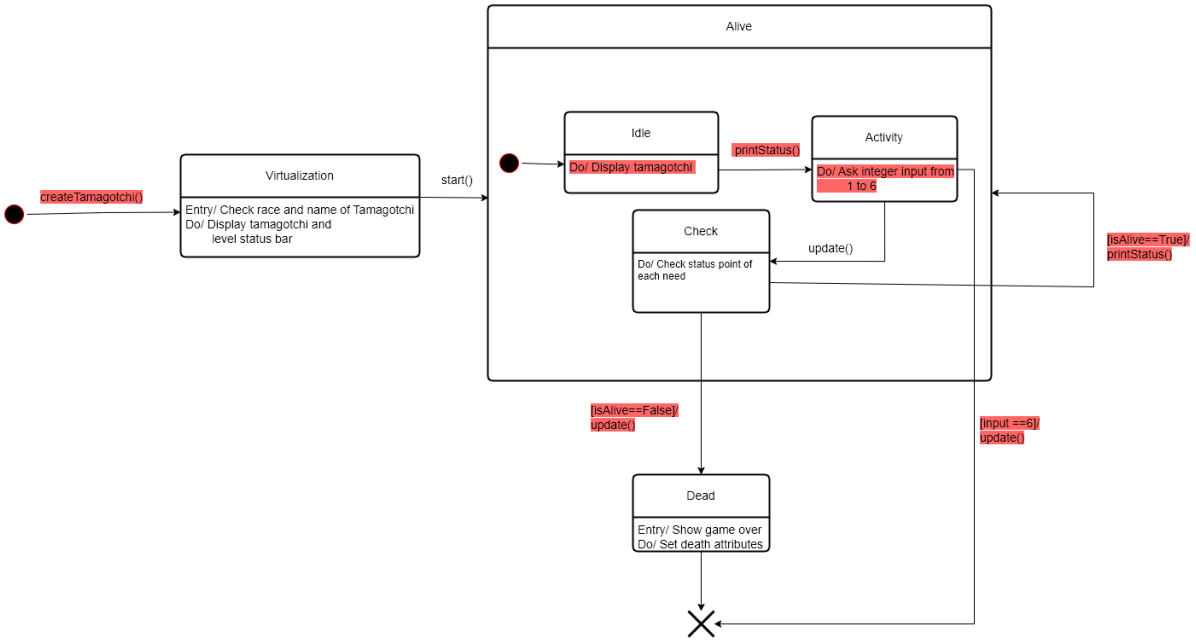
	DP3
Design pattern	Observer
Problem	In a **Tamagotchi** game, there are multiple game elements that need to be notified when a **Tamagotchi's** state changes (*hunger, happiness, sleepiness, and cleanliness*). However, coupling these elements with the **Tamagotchi** class by hardcoding the notification logic would go against the Single Responsibility Principle and make the code difficult to maintain and expand.
Solution	The Observer Pattern solves this problem by defining a dependency between objects so that when one object changes its state, all its dependents (observers) are notified and updated automatically. In the context of a Tamagotchi game, the game elements that need to be notified are observers.
Intended use	At runtime, the **Tamagotchi** object will notify its registered observers (the **Needs**) whenever its state changes. The observers will receive the notification and update their state accordingly.
Constraints	The Observer Pattern imposes the constraint that the subject (**Tamagotchi**) and observers (**Needs**) must implement the same interface or extend the same abstract class. This constraint ensures that the subject

	and observers are decoupled and can be modified independently.
Additional remarks	The Observer Pattern is a commonly used design pattern in software development, especially in user interfaces and event-driven systems. It is a useful pattern to use when there are multiple objects that need to be notified of changes to a single object, without tightly coupling them together. In a **Tamagotchi** game, it can be used to ensure those game elements are updated dynamically and appropriately when the **Tamagotchi's** state changes.

Revised state machine diagrams

Author: Andrei

Title: Revised General LifeCycle (Game)

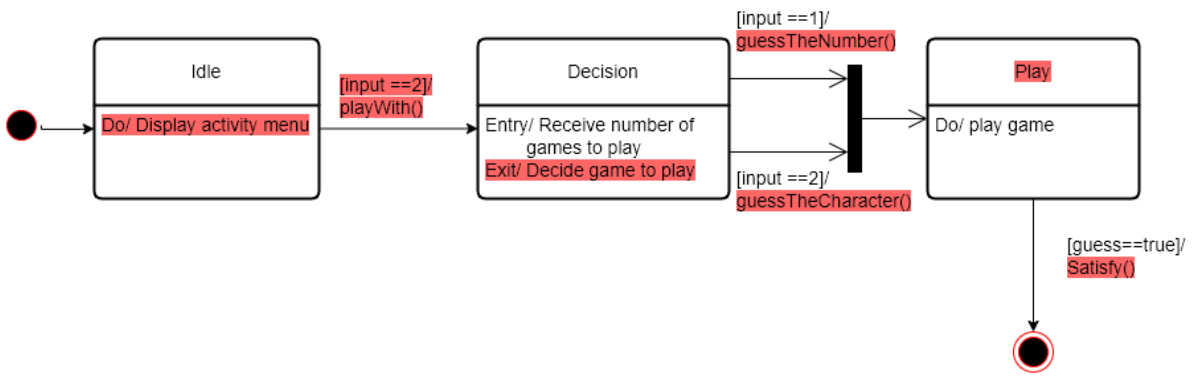


The initial feedback received on the state machine diagram highlighted concerns with the activities in the transitions. It is customary for transitions in a state machine diagram to depict the movement of a system or entity from one state to another in response to an event or a condition. This aspect was addressed by refining the transitions to better align with the functions of the class diagram.

Moreover, various activities within the states were revised to reflect the behavior of the program when it is executed. Following the execution of the *PrintStatus()* function, the user is presented with not four, but rather six options, with two newly added options. The addition of these options, "skip" and "quit," is critical to the Tamagotchi program, as it facilitates smoother gameplay by enabling users to progress through the game more efficiently, while also requiring them to take better care of their virtual creature. Option five, "skip," allows for the activation of the *Update()* function, which decreases the status values, thus increasing the level of user engagement. Option six, "quit," terminates the object, enabling users to exit the game when they desire.

Overall, these changes were instrumental in improving the state machine diagram and providing a more comprehensive representation of the program's behavior.

Title: Revised Game State Machine



The second state machine diagram has undergone significant improvements as a result of feedback received and changes to the program's internal dynamics.

First, there were significant changes made to the diagram's actions and transitions. To adhere to the idea of naming states as nouns, the titles of the states were changed. Reinterpreting the diagram and altering the functions that cause these transitions improved. Additionally, a few minor changes were made to the activities taking place within the states.

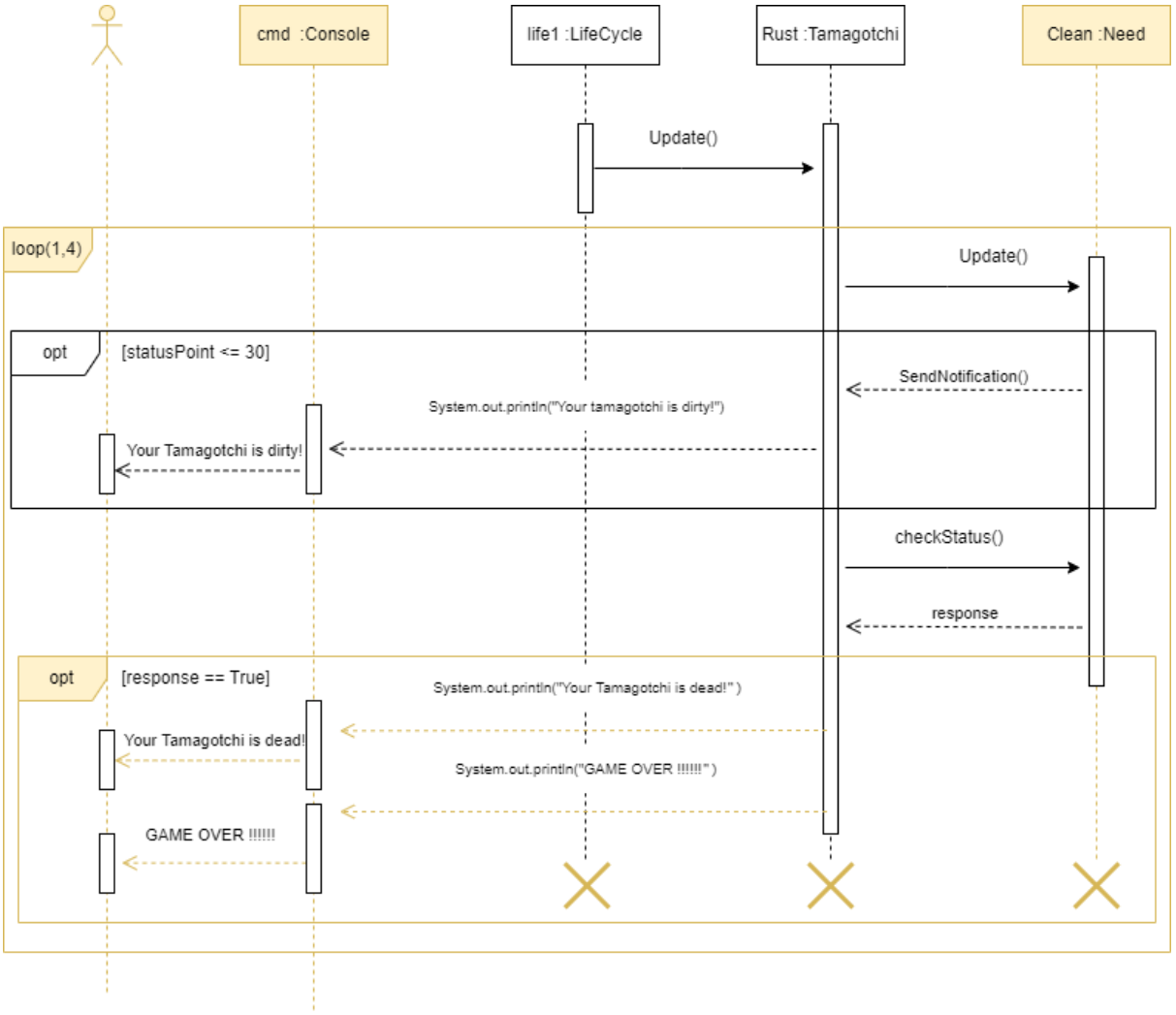
Furthermore, the synchronization node was introduced in this diagram. This node has at least two incoming edges, triggered by the guard and activities *GuessTheNumber()* and *GuessTheCharacter()*) and one outgoing edge. It is used to merge multiple concurrent flows. Since the user will ultimately end up in the **Play** state regardless of input, this node plays a crucial role in maintaining the flow of control.

Lastly, a significant change was made to the diagram by removing the **Winning** state from the previous version. This was replaced with a condition and a transition event *Satisfy()* that rewards the player with an increase in the attribute of happiness. With this current implementation, a **Winning** state was deemed unnecessary and redundant.

In conclusion, these changes have significantly improved the second state machine diagram, resulting in a more precise and comprehensive representation of the program's behavior. The introduction of the synchronization node and the removal of the **Winning** state are significant modifications that contribute to a more efficient and effective program design.

Revised sequence diagrams

Author(s): Kloena



Title: The user gets alerted when the Tamagotchi is dirty or when Tamagotchi dies

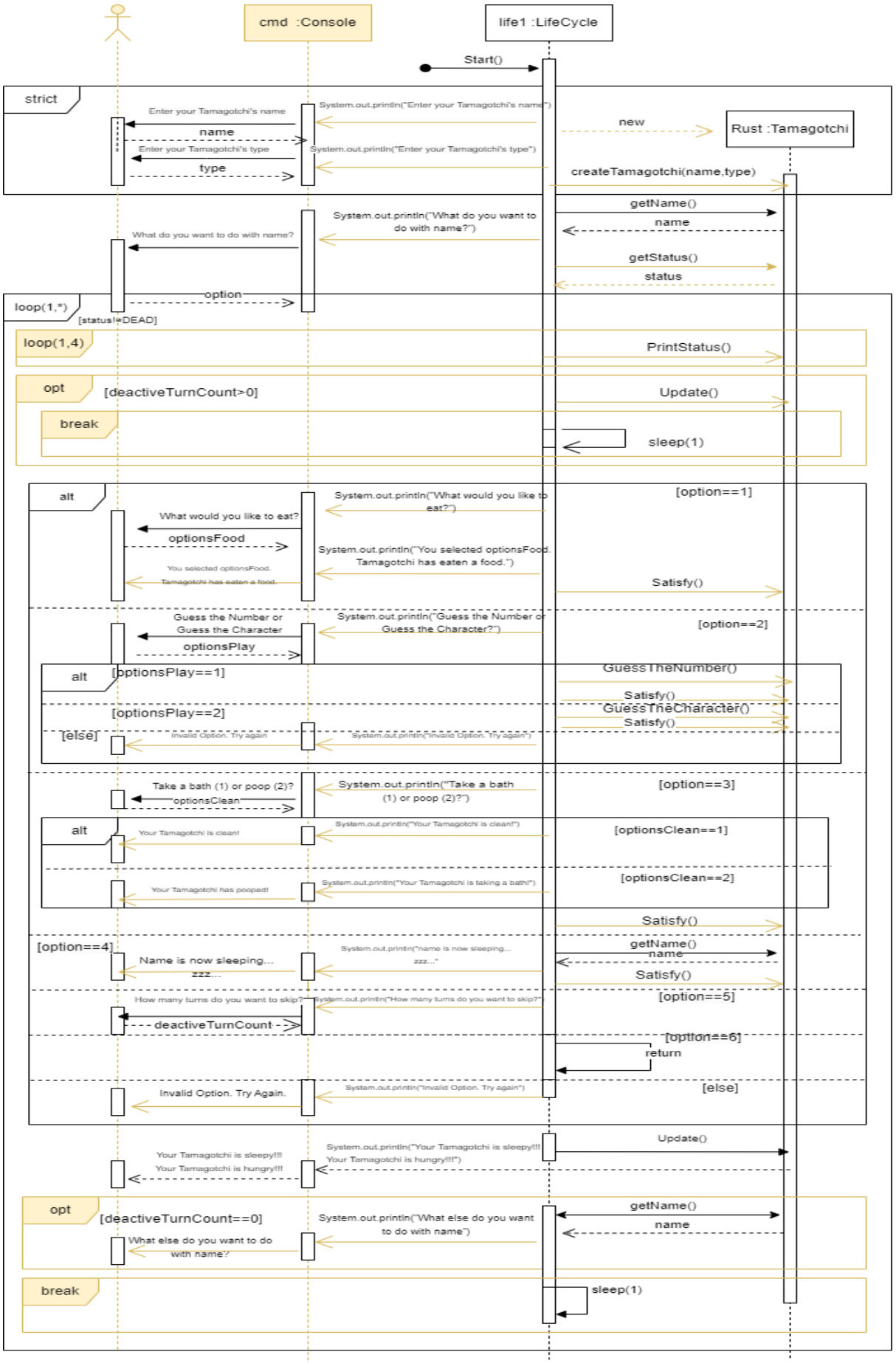
This sequence diagram represents the user getting an alert when the creature's cleanliness level is low or when the creature dies. The classes are the Actor which would be the **User**, the **Console** which is the Command Line, the **LifeCycle**, the **Tamagotchi**, and the **Need** class. After every game loop which is in the **LifeCycle** class, the **Needs** of the **Tamagotchi** get updated by the function *Update()*. In this function, we enter a loop that goes through all the needs one by one, therefore we loop 4 times or as long as the array of needs is. We do this to update all of the needs without repeating code

For each **Need** we call the function *Update*(). In this function, the *statusPoint* variable of all of the Needs gets decreased by 10. This happens because as time passes all attributes of the Tamagotchi should naturally decrease as the game is played. Therefore the *statusPoint* of the **Clean Need** gets decreased by 10. Then, we check if the *statusPoint* is below or equal to 30 with an if statement.

If it is, the **Need** class sends back the *sendNotification()* function.
This sends the message *System.out.println*(“Your Tamagotchi is dirty!”) to the **Console**, which sends the message “Your Tamagotchi is dirty!” to the **User**. This process happens with all the **Needs** of the **Tamagotchi** and aims to alert the user when the **Tamagotchi** is in critical condition so they can nurse it back to health.

After each **Need** is updated, the **Tamagotchi** checks the status by the function *checkStatus*(). This function in the **Need** class returns true only if the *statusPoint* is below or equal to 0. If it returns true, this means that the status of the **Tamagotchi** is *Dead*, meaning the **LifeCycle** of the **Tamagotchi** ends and it dies. When an instance of the **Tamagotchi** dies, so do all of the **Needs** that inherit from it. Game over is printed to the console.

The main improvements (highlighted in yellow) were making a better layout of the objects used in the sequence diagram. An actor representing the user was added to better display the interaction the system has with the user. The **Console** class helps convey the messages sent to create a CLI better. Furthermore, now the function *Update*() is represented more accurately, as the loop was added showing how it goes through each vital part of the **Tamagotchi** one by one. Lastly, now this sequence diagram also shows the termination of the objects **Tamagotchi, Need,** and **LifeCycle** in the case where the **Tamagotchi’s** *status* is *Dead*.


cmd :Console
life1 :LifeCycle
Start()
strict
Enter your Tamagotchi's name
System.out.println("Enter your Tamagotchi's name")
name
new
Rust :Tamagotchi
Enter your Tamagotchi's type
System.out.println("Enter your Tamagotchi's type")
type
createTamagotchi(name,type)
getName()
System.out.println("What do you want to do with name?")
What do you want to do with name?
name
getStatus()
status
option
loop(1,*)
[status!=DEAD]
loop(1,4)
PrintStatus()
opt
[deactiveTurnCount>0]
Update()
break
sleep(1)
alt
[option==1]
System.out.println("What would you like to eat?")
What would you like to eat?
optionsFood
System.out.println("You selected optionsFood. Tamagotchi has eaten a food.")
You selected optionsFood. Tamagotchi has eaten a food.
Satisfy()
[option==2]
Guess the Number or Guess the Character
System.out.println("Guess the Number or Guess the Character?")
optionsPlay
alt
[optionsPlay==1]
GuessTheNumber()
Satisfy()
[optionsPlay==2]
GuessTheCharacter()
Satisfy()
[else]
Invalid Option. Try again
System.out.println("Invalid Option. Try again")
[option==3]
Take a bath (1) or poop (2)?
optionsClean
System.out.println("Take a bath (1) or poop (2)?")
alt
Your Tamagotchi is clean!
System.out.println("Your Tamagotchi is clean!")
[optionsClean==1]
Your Tamagotchi has pooped!
System.out.println("Your Tamagotchi is taking a bath!")
[optionsClean==2]
Satisfy()
[option==4]
Name is now sleeping... zzz...
System.out.println("name is now sleeping... zzz...")
getName()
name
Satisfy()
[option==5]
How many turns do you want to skip?
System.out.println("How many turns do you want to skip?")
deactiveTurnCount
[option==6]
return
Invalid Option. Try Again.
System.out.println("Invalid Option. Try again")
[else]
Update()
Your Tamagotchi is sleepy!!! Your Tamagotchi is hungry!!!
System.out.println("Your Tamagotchi is sleepy!!! Your Tamagotchi is hungry!!!")
opt
[deactiveTurnCount==0]
getName()
name
System.out.println("What else do you want to do with name")
What else do you want to do with name?
break
sleep(1)

Title: The Main Game Loop

This sequence diagram represents the main game loop of our implementation of Tamagotchi. The classes are the Actor which would be the **User**, the **Console** which is the Command Line, the **LifeCycle**, and the **Tamagotchi.**

Firstly, **LifeCycle** gets a call from the *Start*() function. The message “Enter your Tamagotchi’s name” is shown on the screen and the user enters a name and “Enter your Tamagotchi’s type” where the user enters a type. This is done to create an instance of the **Tamagotchi** with the *createTamagotchi*() function.

This sequence of actions happens strictly in this order because the name and race are required to initialize the **Tamagotchi** class. Furthermore, each time we want to use the name of the **Tamagotchi** we have to send a *getName*() message that returns the *name* attribute of the class. We do the same with getStatus() so that we can constantly check in the loop that is about to be executed if the **Tamagotchi** is *Alive* or *Dead*.

As long as the **Tamagotchi** is not *Dead* the loop continues because the game depends on the livelihood of the **Tamagotchi**. Before entering the alt frame we call the *PrintStatus*() function while looping 4 times. The reason for this is that we loop through all the **Needs** of the **Tamagotchi** and print their *status individually*.

This is done in order for the player to be aware of the current *status* of the creature and to know which action they should perform next. In addition, we check through an opt frame if the variable *deactiveTurnCount* is bigger than 0, and if so we call the *Update*() functions that the attributes can decrease when the user chooses to not do any activity with the **Tamagotchi**. Then, we wait for one second, which is displayed by a break frame to represent a try-catch statement.

Secondly, the switch statement is represented by an alt frame where with each option the user chooses to send a *Satisfy*() message to each of the **Tamagotchi’s Needs**. However, if the user selects *option* 6 they quit the game and return out of the loop.

Thirdly, the *Update*() function gets called which checks the **Tamagotchi’s** attributes. In return, the **Tamagotchi** sends back what **Needs** have to be taken care of to the user. In case the *deactiveTurnCount* is 0 we ask the user what other actions they'd want to do. This is done in order for us to keep proper track of how many times the user skips turns, representing how they are neglecting the **Tamagotchi**.

Lastly, we perform a last try-catch statement represented by a break frame in order to wait for 1 more second before looping again and showing the messages to the user.

The man improvements (as highlighted in yellow) were again making a better layout of the objects used in the sequence diagram. An actor representing the user was added to better display the interaction the system has with the user. Secondly, the creation of the **Tamagotchi** instance from the **LifeCycle** class is demonstrated by the new notation. Thirdly, all throughout the diagram wrongly used synchronous messages were replaced with

asynchronous messages, where needed. Lastly, additional frames like the break and opt were added to better represent our implementation and the details of the functionality of the game and all of the messages are functions used in the class diagram.

Implementation

Author: Håvard

During the preliminary stages of the project, the main goal was to identify the essential aspects and attributes of the software. These aspects encompassed creature customization, day/night cycle, Graphical User Interface (GUI), and more. Through collaborative meetings, the team arrived at a consensus on the crucial features that the system needed to possess.

With the system's features and plans defined, the development of Unified Modeling Language (UML) diagrams and source code progressed simultaneously. The development process involved frequent adjustments to both the code and diagrams to accommodate modifications made in either of the two.

To translate the UML model into implementation code, we first reviewed the UML models created in the previous assignment to understand the system's requirements, classes, and relationships. Then, we implemented the classes that were identified in the UML models, ensuring that each class had the appropriate attributes, methods, and relationships as specified in the UML models.

The key solutions and implementation of the system

The most important aspect of the Tamagotchi system is the implementation of activities. In order to keep the game engaging and interesting, the **Tamagotchi** must have a variety of activities that the player can engage and have fun with. The activities we implemented were sleep, play, eat and clean.

The **activities classes** are made as instances in the **Tamagotchi** class. By implementing it this way we ensured that the data and functionality of the inner class are encapsulated within the outer class. That composition allowed us to implement more complex, modular, and reusable objects inside the **Tamagotchi** class. It also abstracts away complexity in the outer program.

The activity classes share a similar layout. Each class has a boolean value, which indicates whether a notification of the need should be sent out to the terminal. Additionally, each class has an *'update'* method that increases the *Status* of the need.

To keep our tamagotchi entertained, we implemented a **playground class** that can be initiated whenever the player wants to play with their *tamagotchi*. The **playground** class features two guessing games that boost the **tamagotchi's** happiness levels. One of the games generates a random character, while the other generates a random number. In both

games, the player is prompted to input their guesses. Once the games are complete, the **tamagotchi's** happiness levels increase as a result.

The **life cycle** of the **Tamagotchi** is managed within a while loop. As the loop runs, a timer keeps track of the passage of time, generating the **Tamagotchi's** various needs at the appropriate intervals. Once the loop completes, the **Tamagotchi's life cycle** comes to an end.

Another important aspect of the Tamagotchi system is the implementation of **needs.** In order for the **Tamagotchi** to thrive, it must have certain needs met, such as food, rest, and playtime. We implemented that in a **Need** class which contains the bla bla bla

To alert the player to changes in the **Tamagotchi's** needs or status, the Tamagotchi system can implement various forms of notifications. For example, the system may display a message or icon when the **Tamagotchi** is *hungry* or *tired*, or it may make a sound to get the player's attention. These were implemented by having print statements print out a notification to the player if the needs of the **Tamagotchi** class got too low.

In order to provide information to players within the **command line interface,** we implemented print statements that output text directly to the console. By doing so, players can easily read and respond to relevant information. To capture player input, we utilized the scanner utility package from Java, which allows for registering keystrokes.

To fulfill the **Tamagotchi's** needs and increase its attributes, each instance of the N**eed** class is equipped with its own S*atisfy* method. This method is called whenever the **Tamagotchi's** corresponding need gets attention. It adds 10 points to whatever need is satisfied. By using individual instances of the **Need** class, the **Tamagotchi's** needs can be addressed and fulfilled in a targeted and efficient manner.

As the game progresses through each iteration of the loop, the value of each need attribute is reduced by 10 points and updated. This ensures that players are continuously challenged to maintain their character's well-being and avoid negative consequences. By implementing this mechanic, players must be strategic in managing their resources and prioritizing actions accordingly.

The **Tamagotchi's life comes to an end** when one or more of its needs reach zero. Each game loop checks the status of its needs, and if any of them are at zero, the system exits the loop and shuts down.

Design Patterns implementation:

Singleton pattern:

The Singleton Pattern is a design pattern that restricts the instantiation of a class to a single instance, ensuring that there is only one instance of the class in the entire system. It is useful

in situations where there is a need for a single instance of a class that can be accessed and shared across multiple parts of the codebase.

We implemented that with a single instance of a class created by defining a private static constructor to ensure that the lifecycle class can only be instantiated once, and that the same instance is used throughout the system.

Factory pattern

The Factory pattern provides an interface for creating objects, but allows subclasses to alter the type of objects that will be created. In this case, we can create a factory that will create instances of Tamagotchi based on the type of creature.

The Factory Pattern was implemented as a separate class, called tamagotchiFactor, that serves as an interface for creating Tamagotchi instances. Inside that class we also implemented a switch case where the user can select a cat, dog, human or bird avatar that gets printed out to the terminal. The factory method is responsible for creating and returning Tamagotchi objects without directly initializing them within the system.

Observer pattern

The Observer Pattern is a behavioral pattern that defines a one-to-many dependency between objects, such that when one object changes its state, all its dependents are notified and updated automatically. This pattern is useful in situations where there is a need to keep multiple objects in sync with each other.

In the implementation, the Tamagotchi object serves as the subject, or the object that is being observed. The needs objects, on the other hand, act as the observers that are notified whenever changes are made to the Tamagotchi object. By establishing this observer pattern, the need objects can stay up-to-date on the status of the Tamagotchi and respond accordingly to ensure its well-being.

The location of the main Java class needed for executing your system in your source code;

The location of the main Java class needed for execution is:
"/SoftwareDesign/src/main/java/softwaredesign/Main.java"

The location of the Jar file for directly executing your system;

Gradle manages the process of making a jar file. It can be found at:
"/SoftwareDesign/build/libs/software-design-vu-2020-1.0-SNAPSHOT.jar"

The 30-seconds video showing the execution of your system

The video demonstration below provides a comprehensive showcase of all the features described in the initial project pitch from the first assignment. The system is fully functional and operational, allowing for an immersive and engaging gaming experience. Through this

demo, viewers can gain a clear and concise understanding of the system's capabilities, from its intuitive interface to its advanced gameplay mechanics. If the pace of the demonstration seems too fast, feel free to pause the video and read at your own pace.

https://www.youtube.com/watch?v=aNwHCIIyTBM&ab_channel=H%C3%A5vardS.

Time logs

Member	Activity	Week number	Hours
Håvard Skjærstein	Implemented the system	3	8
Håvard Skjærstein	Wrote the implementation section	4	8
Håvard Skjærstein	Made the demo video	4	2
håvard Skjærstein	Revisited the class diagram	4	3
håvard Skjærstein	Summary feedback	4	1
Kloe	Revisited the object diagram	4	1
Kloe	Summary feedback	4	1
Kloe	Revisit of the sequence diagram	4	3
Kloe	Implemented the system	4	3
Kloe	Design patterns	4	3
Kloe	Design patterns section in docs	4	1
Andrei	Revisted the state machine	4	4
Andrei	Design patterns	4	4
Andrei	Implemented the system	4	4
		TOTAL	46