Assignment 2

Team number: 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 1

1. In the overview, we have elaborated more on *our* version and vision of the Tamagotchi game. (page 3.)

We have indicated clearly that we will be using CLI to interact with the creature and explained some basic features of the game more clearly than the first time, such as the life cycle.

2. We have clarified the descriptions for our functional features. (pages 4,5.)
3. Some descriptions are deleted.

F1: The possible characters of the creature are listed as human, cat, bird, and bunny.

F2: We have set quantitative features for the life cycle.

F7: We have added a more detailed numerical explanation for the level bars indicating the current hunger, sleep, cleanness, happiness, and energy levels of the Tamagotchi.

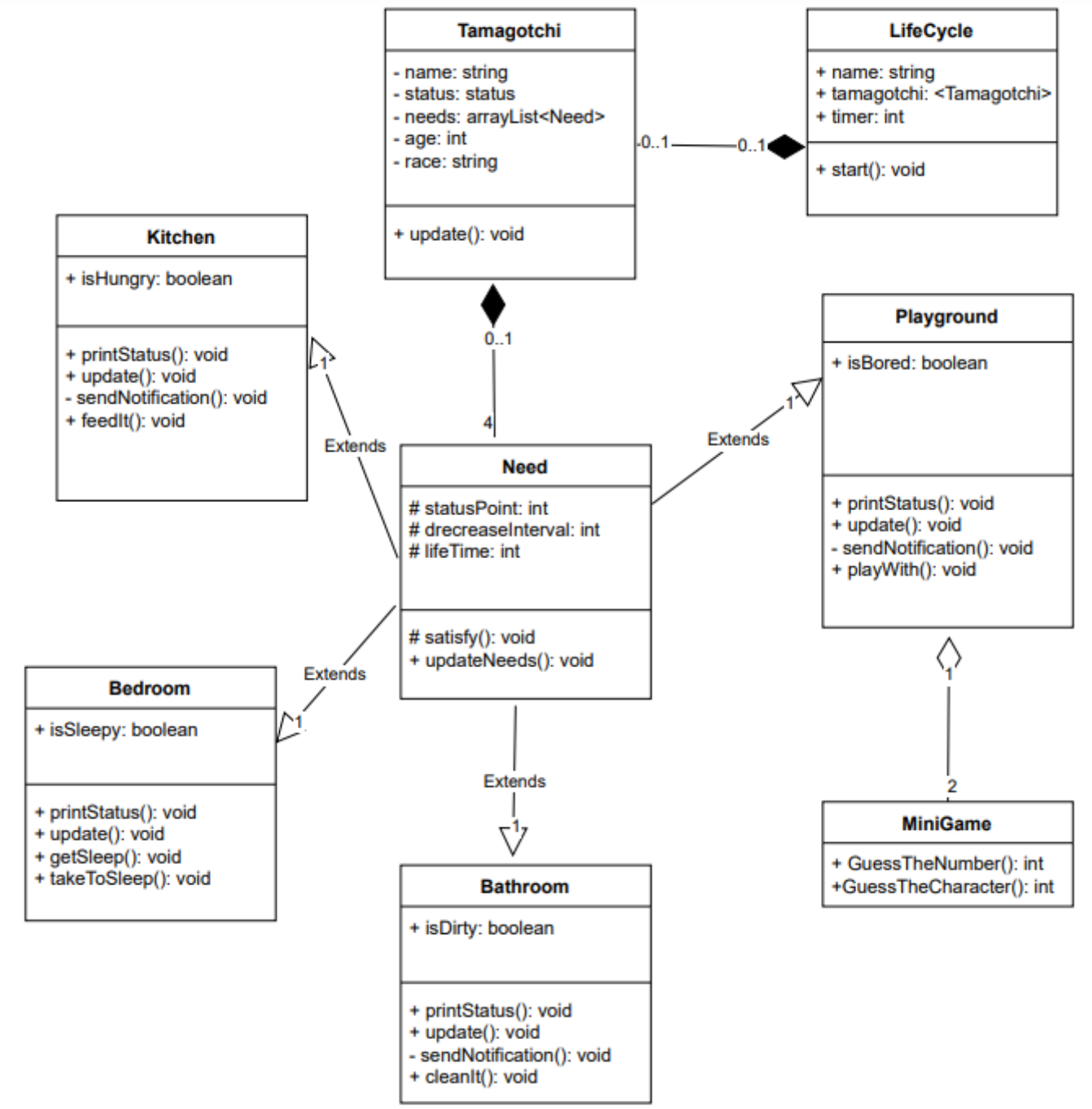
F3/F4/ F6: How the Tamagotchi sends the notification to the user is clarified as we have indicated that it will print a message on the screen.

4. Mistakes in quality attributes and ambiguous descriptions are fixed.

Q3: The quality attribute is changed to maintainability and the description of the requirement is re-written.

Q4: The performance requirement is explained more in detail including the numerical value in which we expect our system to react which is 3 seconds.

Class diagram



The **Tamagotchi class** is a fundamental class that represents the virtual pet that the user takes care of. It is designed to have certain attributes that reflect the pet's identity such as name, status (alive or dead), age, and race. It also provides various useful methods and operations that allow the user to access and modify the pet's attributes during each game loop. These methods update and set values, such as the name and race of the creature. The Tamagotchi has certain needs such as sleep, cleaning, playing, and eating. They are implemented respectively in the classes of Bedroom, Bathroom, Kitchen, and Playground. They are extended from the superclass of Need as they all share some common functions such as satisfy(), which means increasing the status point by 10 by satisfying a need. Status point represents the satisfaction of a need out of a hundred points, represented by 10 hashtags as a status bar on the screen. These attributes are found inside an arraylist, in the Tamagotchi class and are essential to keep the pet alive, healthy, and the game going. We use composition when handling the relationship between the Tamagotchi and the Need class because 1 Tamagotchi is composed out of 4 Needs. The Tamagotchi could contain many needs, while each need belongs to only one Tamagotchi.

The Bedroom class brings together variables and functions regarding the sleep need of the creature by extending the Need class. The Tamagotchi can sleep to increase its sleepiness status point by 10 each time with the takeToSleep() function. We found it most efficient to store this value as an isSleepy boolean that determines if the Tamagotchi is sleepy or not, which becomes true if statusPoint is below 30. The Bedroom class provides a method that allows the user to put the Tamagotchi to sleep and to set and get values regarding the sleep needs of the creature.

The Playground class represents a virtual playground where the Tamagotchi can have fun and increase its happiness level. This class contains a boolean attribute to indicate if the Tamagotchi is bored (statusPoint<30) or not. The methods consist of ways to communicate the boredom status and also ways to initiate play.

The MiniGame class has two mini-games methods "Guess the Number" and "Guess the Character", in which the user guesses the number or character that the Tamagotchi is thinking of. If the player guesses the right answer, the Tamagotchi gains happiness. If not, its status point decreases by 10. We use aggregation to describe the relationship between the MiniGame and the Playground class because the Playground is composed of 2 mini-games. However, the mini-games can exist outside of the playground.

The Bathroom class represents the hygiene virtual space where the Tamagotchi can take a bath or use the toilet in order to increase its cleanliness attribute. The class

contains methods that initiate hygienic activities. The first one allows the Tamagotchi to take a bath, the second one allows the Tamagotchi to use the toilet in order to increase its cleanliness. By using these methods, the user makes sure the Tamagotchi is still healthy. It's a subclass of the Need class.

The Kitchen class is responsible for the Tamagotchi's food requirements, which can be selected from a menu to increase its hunger attribute by 10 each time. In addition to storing a boolean value that indicates whether the Tamagotchi is hungry or not, this class also includes methods to feed the creature, feedIt(). As a subclass of the need class, it helps to fulfill the Tamagotchi's hunger needs.

The Need class manages the Tamagotchi's needs by controlling its attributes, and it also initializes activity classes such as the Bedroom. This class includes an integer value that specifies the time interval for the creature's needs to decrease and its overall lifespan. If any of the needs of the creature reach a critical level, the creature will communicate with the user through a printed message indicating its health will start to deteriorate. It may eventually die if not taken care of properly. All of the needs of the Tamagotchi have status points and a specific interval period in which the points decrease. The attributes of the need class are common for specific needs. By declaring some functions such as update() in the needed superclass, we save ourselves from repetition of code in various needed sub-classes.

The LifeCycle class is responsible for initializing the game instance by starting the timer which determines when the Tamagotchi needs to be fed, cleaned, played with, or taken to sleep. It controls the flow and execution of the program. We use composition when handling the relationship between the Tamagotchi and the LifeCycle class because 1 Lifecycle is composed of 1 Tamagotchi. The LifeCycle could contain many Tamagotchis, while each Tamagotchi belongs to only one LifeCycle.

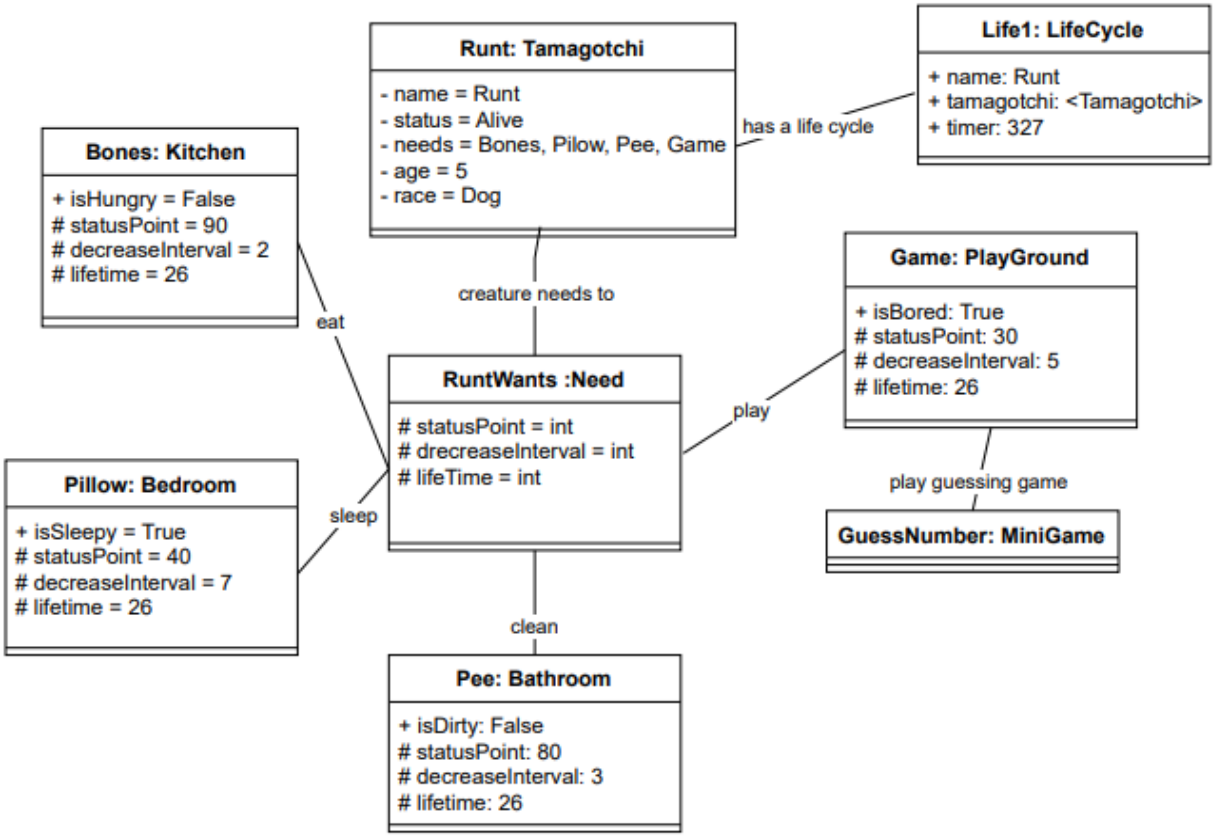
Meanwhile, the main class, which is not included in the class diagram intentionally, is specifically designed to handle the creation and management of Tamagotchi instances within the game. This class serves as the central coordinator for all gameplay interactions, receiving input from the player and communicating with other classes as needed.

We made our design decisions considering the "separation of concern" approach regarding the classes. The systems classes are designed to have clear, well-defined responsibilities. This separation of concern makes the codebase easier to understand,

debug and maintain. Overall these design decisions made for the system prioritize modularity, maintainability, and ease of use for both the developer and the end user.

Object diagram

Title: Tamagotchi object when sleepy and bored



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.

In the case of a **Tamagotchi** game, an object diagram can be used to represent the state of a particular instance of the Tamagotchi class, such as "**Runt**". This diagram provides a snapshot of the game at a certain time of execution, showing the current values of the attributes associated with the Runt instance.
We can see that the creature's name is **‘’Runt’’**, its status is **‘’Alive’’**, it is a dog and its needs are listed as an arraylist.

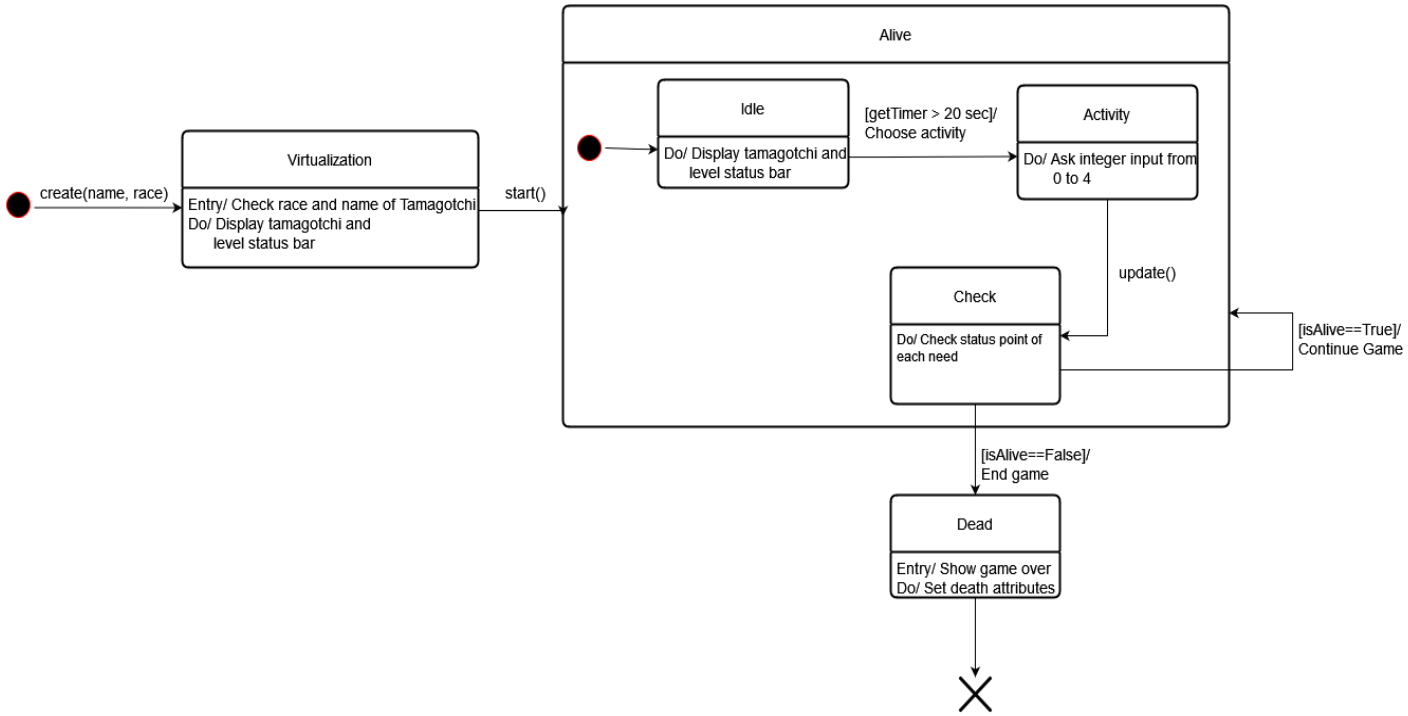
Each object in the diagram represents a different aspect of the game environment that interacts with the Runt instance. For example, the **Bones** object is responsible for increasing Runts' **hunger levels** when it is used. Similarly, the **Pillow** object increases Runts' **sleepiness statusPoint** by 10, while the **Game** object increases Runts' **happiness** attribute.

By examining the relationships between these objects and Runts' instance, we can gain insight into how the game functions.

At this point of time, the Tamagotchi object Runt is sleepy and bored as illustrated with the booleans. The status points can be seen in the object diagram. The common attributes of those different objects are extended from the **Need** class. These are **statusPoint**, **decreaseInterval** and **lifeTime**. The tamagotchi needs to be fed in every 2 minutes, played with every 5 minutes, taken to sleep in every 7 minutes and cleaned every 3 minutes.

State machine diagrams

Title: General Life Cycle (Game)



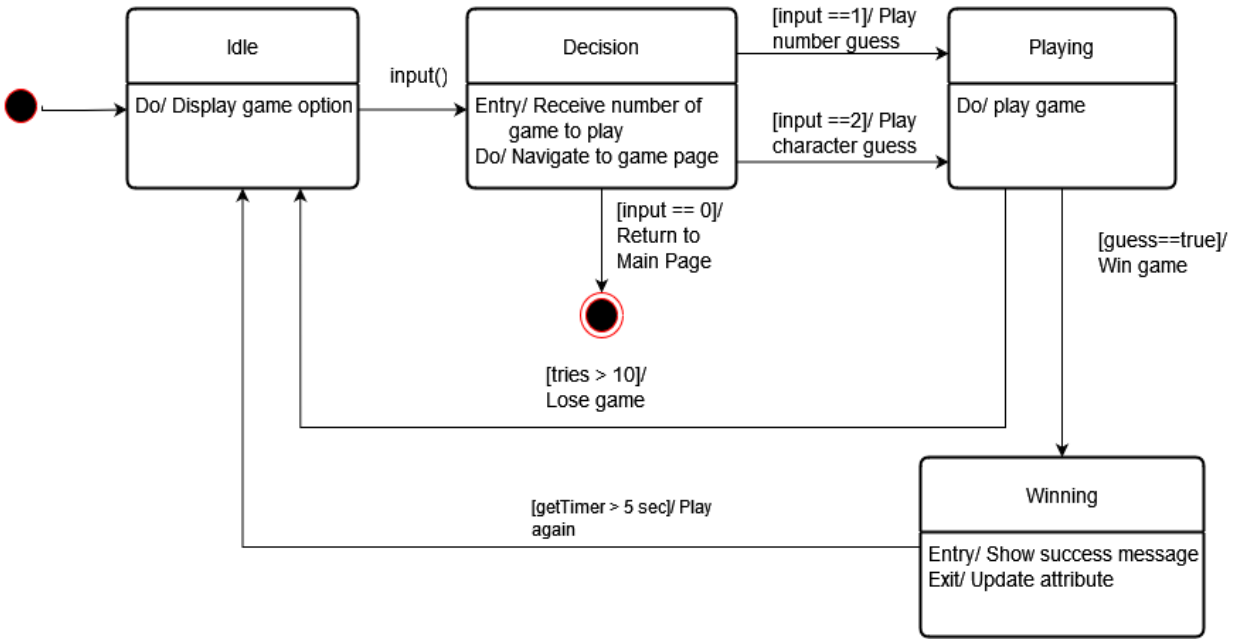
Description:

The diagram above illustrates the general life cycle state of the **Tamagotchi**. It begins with the **create(race, name)** function, which takes the race and name of the creature that the user inputs from the terminal. After this call event, the Tamagotchi enters the **'Virtualization'** state, where it checks for the race and name of the creature and its status levels, such as **hunger**, **energy**, **cleanliness**, and **happiness**, are displayed to the user. The **start()** function starts the timer for the Tamagotchi, where one year is equivalent to one real hour for the creature. The virtualization state is different from the Alive state and comes before as the timer only starts when the race and name is recieved, so the life cycle of tamagotchi starts afterwards.

Upon starting the game, a more complex state representation is required. The **Alive** state represents a **Nested state,** where all the substrates work exclusively inside and only in the **Alive** state. The **Alive** state starts with the initial state of **Idle** state, as this is a new “environment” where the Tamagotchi activates in. Every 20 seconds, the Tamagotchi asks for user input between 0 and 4 (0: **skip**, 1: **feed**, 2: **play**, 3: **clean**, 4: **sleep**). After the user chooses a command, the **update()** function triggers the checking state. This state checks whether any of the status points for the **hunger**, **energy**, **cleanness**, and **happiness** level bars have reached 0, indicating that the Tamagotchi should enter the **Dead** state.

The **isAlive** boolean is used to control and guide the state transitions. If the Tamagotchi is alive after each activity, it continues the **Game**,continues to be alive, and the same cycle repeats where it asks for user input every 20 seconds. However, if it is dead, the end game is reached and it enters the **Dead** state, where the game over message is displayed and the attributes are set to the default ones opposite of alive. Since the Tamagotchi ceases to exist, a termination node is added, terminating the object. Using a final state dot would be incorrectly implemented, because the object “Tamagotchi” is terminated, gets removed and needs to be created once more if a new game is started.

Title: Minigame State Machine



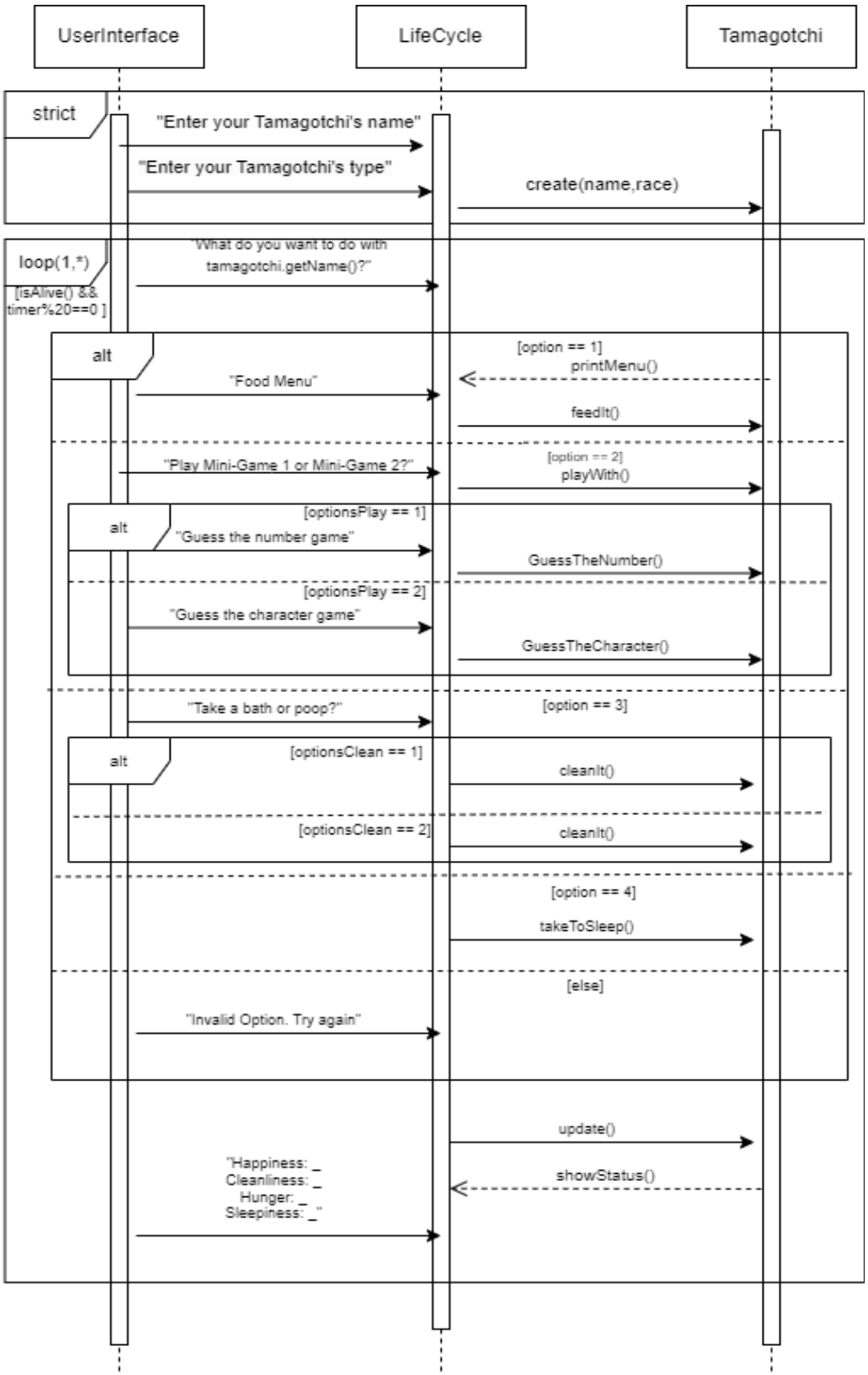
Description:

The State Machine diagram above illustrates the various states of the **Minigame**. The diagram begins with Tamagotchi's initial state, **Idle**, where two game options, "**guessNumber**" and "**guessCharacter**," are displayed on the terminal. Following the input(), the decision state is reached, which determines which game page to open.

Regardless of the game selected, the creature enters the **Playing** state, which involves the actual gameplay by guessing either the number or the character. It is essential that the user correctly guesses the number within 10 trials. If the user exceeds the 10 trials, they will lose the game, exit the **Playing** state, and be prompted with a "**GAME OVER**" message. Additionally, the creature's happiness attribute will decrease as a penalty for not making the correct guess. Conversely, if the guess is correct, the creature enters the **Winning** state, where the user receives a positive message and an increase in the creature's happiness attribute by 10 points. To exit the **Minigame** state, the user must input 0 in the **Decision** state.

Sequence diagrams

Title: The main game loop



Description:

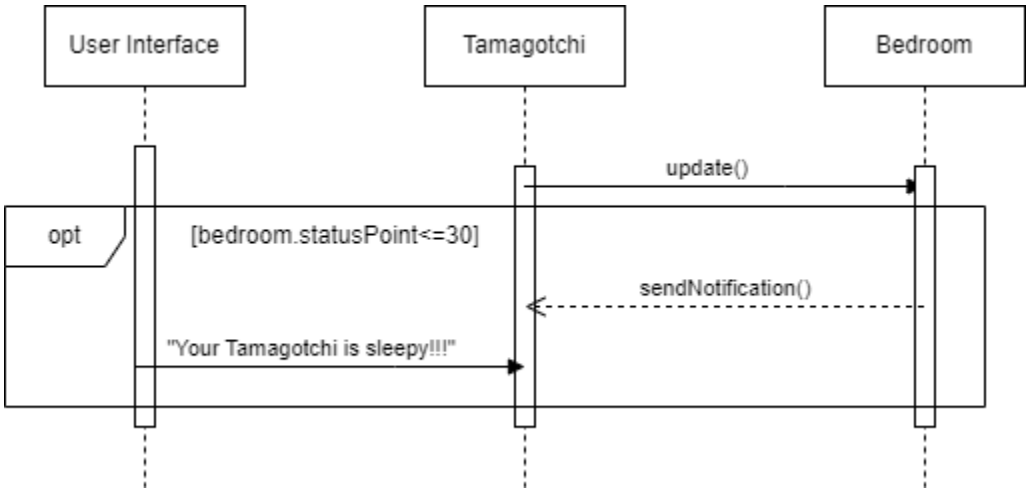
This sequence diagram represents the main game loop of our implementation of Tamagotchi. The three classes are the **User Interface**, the **LifeCycle**, and the **Tamagotchi**.

Firstly, 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. After this, we initialize the Tamagotchi with the **create()** function. This sequence of actions happens strictly in this order because the name and race are required to initialize the **Tamagotchi** class. While constantly checking if Tamagotchi is alive, every 20 seconds the message “What do you want to do with **_tamagotchi.name_**” is shown to the user.

Based on the user’s answer we execute one of the cases. If they select 1 in order to feed the Tamagotchi, the Tamagotchi sends back a message with the **printMenu()** method. The user interface prints the food menu on the screen and the user has the option to choose again. After choosing the method **feedIt()** is called.
If they select 2 in order to play with the Tamagotchi, the message “Mini-game 1 or Mini-game 2?” pops up on the screen after the **playWith()** function is called. If they select 1, the user interface prints the “Guess the number” game on the screen and the user can start playing it. In order to do that, the method **GuessTheNumber()** is called. If they choose mini-game 2 by selecting 2, the user interface prints the “Guess the character” game on the screen and the user can start playing it. In order to do that, the method **GuessTheCharacter()** is called.

If they select 3 in order to clean the Tamagotchi, the message “Take a bath or poop?” pops up on the screen. If they choose to take a bath by selecting 1, the game sends a message to the Tamagotchi by the method **cleanIt()**. If they choose to poop by selecting 2, the game sends a message to the Tamagotchi by the method **cleanIt()**.
If they select 4 in order to put the Tamagotchi to sleep, the game sends a message with the method **takeToSleep()**. If the user doesn’t choose any of the above the User Interface sends a message to the screen saying “Invalid Option. Try again.” After this, the method **update()** is called which checks the Tamagotchi’s attributes. The Tamagotchi responds with the function **showStatus()** which displays all the levels of the features on the screen. 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.

Title: The user getting alerted when the Tamagotchi is sleepy



Description:

This sequence diagram represents the user getting an alert when the creature's sleep level is low. The three classes are the **User Interface**, the **LifeCycle**, and the **Bedroom**. After every game loop, the **Needs** of the Tamagotchi get updated. In this case when the Bedroom gets updated we check if the **statusPoint** is below or equal to 30. In this function **update()**, the **statusPoint** of all of the Needs get decreased by 10. This happens because as time passes all attributes of the Tamagotchi should naturally decrease. Therefore the **statusPoint** of the **Bedroom** gets decreased by 10.

In this sequence diagram, the sleep level is <= 30 and the Bedroom class sends back the **sendNotification()** function. This returns the message "Your Tamagotchi is sleepy!!!" on the screen. This process happens with all the Needs of the Tamagotchi and it aims to alert the user when the Tamagotchi is in critical condition so they can nurse it back to health.

Time logs

Team Member	Activity	Week	Hours
Havard/ Lara	Draw the class and object diagram.	1	1
Kloe/ Andrei	Draw the state machine and sequence diagram.	1	1
Havard/ Lara	Check and give feedback on state machine and sequence diagram.	2	1
Kloe/ Andrei	Check and give feedback on object and class diagram.	2	1
All	Draw the final diagrams on computer.	2	1
All	Evaluate and integrate the feedback given for A1.	1	1