Assignment 2

Team Number: Group 63
Team Name: Team Pineapple

Team members:

Name	**Student Nr.**	**Email**
Turan Yigit	2732933	t.yigit@student.vu.nl
Shadman Sahil Chowdhury	2732169	s.s3.chowdhury@student.vu .nl
Kanushree Jaiswal	2737122	k.jaiswal@student.vu.nl
Wissal Mestour	2742758	w.mestour@student.vu.nl

Summary of changes from Assignment 1

Author(s): Kanushree and Wissal

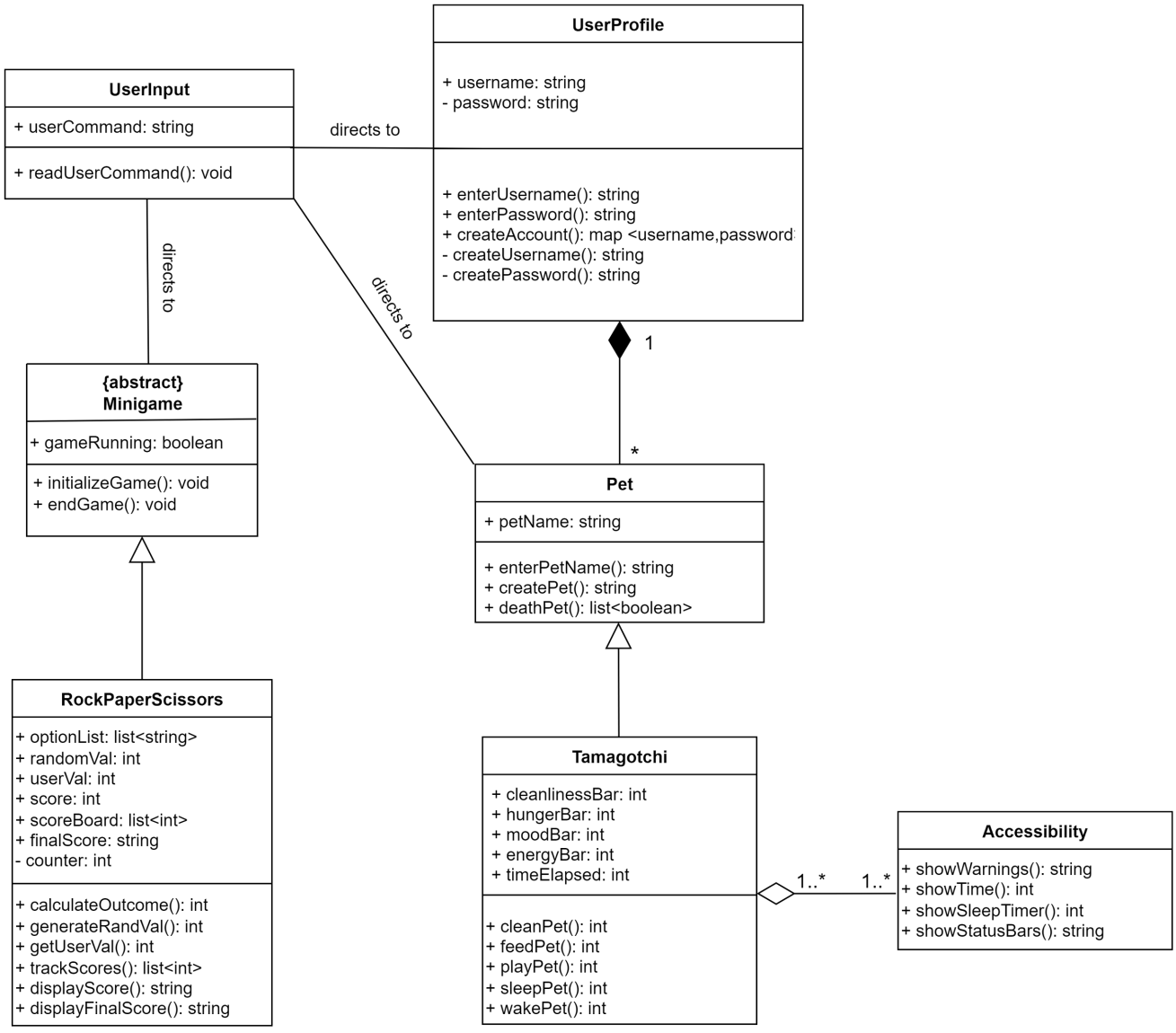
- Made the bonus idea's more precise in the overview section
- *Added a lot of missing functional features, like how the pet, play, feed etc. work.
- Added the functional feature of the death of the pet.
- Changed the minigame from alien shooter to rock paper scissors.
- Removed notifications, only warnings should be enough.
- Added exact timings instead of broad terms like "instantly"
- Added a new quality feature that states that adding a completely new minigame should be done pretty easily
- Removed the inventory.
- We are no longer going with the speech recognition feature.
- Removed the functional feature where the user can move across X and Y-axis in the minigame, since the minigame is no longer an alien shooter but a rock-paper-scissors.
- Removed the "health" bar. Instead the death of the pet will be decided by the combination of all status bars.

All the class changes according to the feedback can be seen in assignment 3

Class diagram

Author(s): Kanushree and Wissal

UML Class Diagram for Tamagotchi Project



The above is a class diagram representing various classes involved in the Tamagotchi Project Implementation. Below are descriptions of each class and its importance in the project.

UserInput

This class essentially takes on the work of what a main class would have involved in it. It reads the user commands typed by the user and directs the user to other parts of the project. The other parts in this case are **UserProfile**, **Pet** and **Minigame**. The **UserInput** consists of one attribute which is the *userCommand* of type string. This is the attribute that stores the command of the user from its method readUserCommand() which returns a void type. This class has three associations that extends to classes **UserProfile**, **Pet** and **Minigame**. It is a binary association and basically directs the user from **UserInput** to either of the classes based on the *userCommand.*

Although this could have been done in the main class as well, in our opinion, it would be better to form a separate class for command handling. This makes it so that you can add more commands without having to change the main code. The command handling process becomes independent and easier to maintain than if we were to keep it in the main class.

UserProfile

This class involves the user authorization and the making of the user account. A user account is necessary to access the game. Therefore, this class ensures that a user can access the game if they have an account or create an account if they do not have one. The attributes it has are *username* and *password*. These are necessary for both creating the account and logging into the account. The important methods that the **UserProfile** has are enterUsername() and enterPassword(). These are used to log in to the account and see if an account exists. The other methods that are createAccount(), createUsername() and createPassword() are used to create the account of the user. Here, createAccount() returns a map of the username and password. Moreover, this map is extended to the **Pet** class where each user its associations with its pets. The **UserProfile** has associations with **UserInput** which is discussed above and **Pet** which will be discussed in the **Pet** class.

The reason this class is separated from the others is so that the user is separated from all the other classes and has its own function. Moreover, in the early implementations of our idea we wanted to merge this class with the **Pet** class. However, we decided against it to ensure that the user is separate from the entire game structure.

Pet

The **Pet** class essentially is used for storing the data of the pet name and creating or deleting a pet based on the game implementation. Its main attribute is *petName* which is used to access the pet with a given name or create one. The main methods in this class are *enterPetName()* that returns a string. As the name suggests, it is used for naming the pet or finding a pet. *createPet()* is used to create a pet with a given name*. deathPet()* deletes the pet with a given name based on the Tamagotchi and its survival. The main associations of pet lead to the **Tamagotchi** class and the **UserProfile** class. The former will be mentioned in the next description. The association of the **Pet** class with the **UserProfile** class is a composition relationship. The **Pet** class cannot exist without an instance of the **UserProfile**. As the relationship shows, one user can have multiple pets.

In the original implementation, we planned to have one class for the **Pet** and **UserProfile**. However, we wanted to have the option of creating multiple pets per user, and hence decided to have **Pet** as a separate class and connect it to the **UserProfile** class.

Tamagotchi

The **Tamagotchi** class is the basis of the Tamagotchi game. All the necessary features for the survival of the **Pet** are implemented using this class. It is essentially the heart of the game. It has multiple important attributes such as the *cleanlinessBar, hungerBar, moodBar, energyBar* and *timeElapsed.* The first four attributes return values for the status bar of the **Tamagotchi**. These status bars are checked for a condition level to keep the pet alive. There is a change in these bars with reference to the user inputs through the methods. The attribute *timeElapsed* keeps track of how long the pet survives for. The methods cleanPet(), feedPet(), playPet(), sleepPet() and wakePet() are called when the user calls an action on the pet. These actions change the bars of the first four attributes. These are essential in maintaining the pet alive. The threshold value for these bars is 15%. If they are all below 15% the pet dies. The Tamagotchi class has two associations. One to **Accessibility** and the other to **Pet**. The former will be written about in the next description. The association of **Tamagotchi** to **Pet** is inheritance. It inherits all attributes and methods of **Pet** and is essentially a subclass of **Pet**. **Pet** can exist without **Tamagotchi** because it only contains the data of the pet.

There was a discussion to merge the **Pet** and **Tamagotchi** classes into one because each instance of the pet would have an instance of the **Tamagotchi**. However, it was later concluded that the **Tamagotchi** class only dealt with the features of the survival of the pet. However, the **Pet** class involved data related to the pet's existence.

Accessibility

The **Accessibility** class deals with the extra functions related to the **Tamagotchi**. It is to make it more accessible by the user. It mostly relates to the display of various attributes of the **Tamagotchi** class. It does not have any attributes but has four methods – showWarnings(), showTime(), showSleepTimer() and showStatusBars(). Each of these methods does what the name suggests. showWarnings() shows warnings when the pet's health is detoriating. showTime() shows how long the pet is alive for. showStatusBars() shows all the bars from the **Tamagotchi** class. showSleepTimer() is to show how long the pet is asleep. The **Accessibility** class has a shared aggregation relationship with **Tamagotchi**. There needs to be at least one **Tamagotchi** class instance for an instance of the **Accessibility** class.

Although this class can pretty much be a part of **Tamagotchi**, it makes more sense to introduce it as another class with a relationship to **Tamagotchi**. It keeps the features of the **Tamagotchi** separate.

{abstract} Minigame

This class is an abstraction of the minigame. In the instructions for the project, the minigame implementation is supposed to be as detached from the main game as possible. For this reason, an abstract class is used. Any minigame can be implemented using this class. However, in our implementation of the project**, RockPaperScissors** is the minigame that we have used. The **Minigame** class has one attribute that is *gameRunning.* It returns a boolean value of true or false to show if the game is running or not. It also has methods initializeGame() and endgame() both of which are void methods. As the name suggests, it starts and ends an instance of the game.

RockPaperScissors

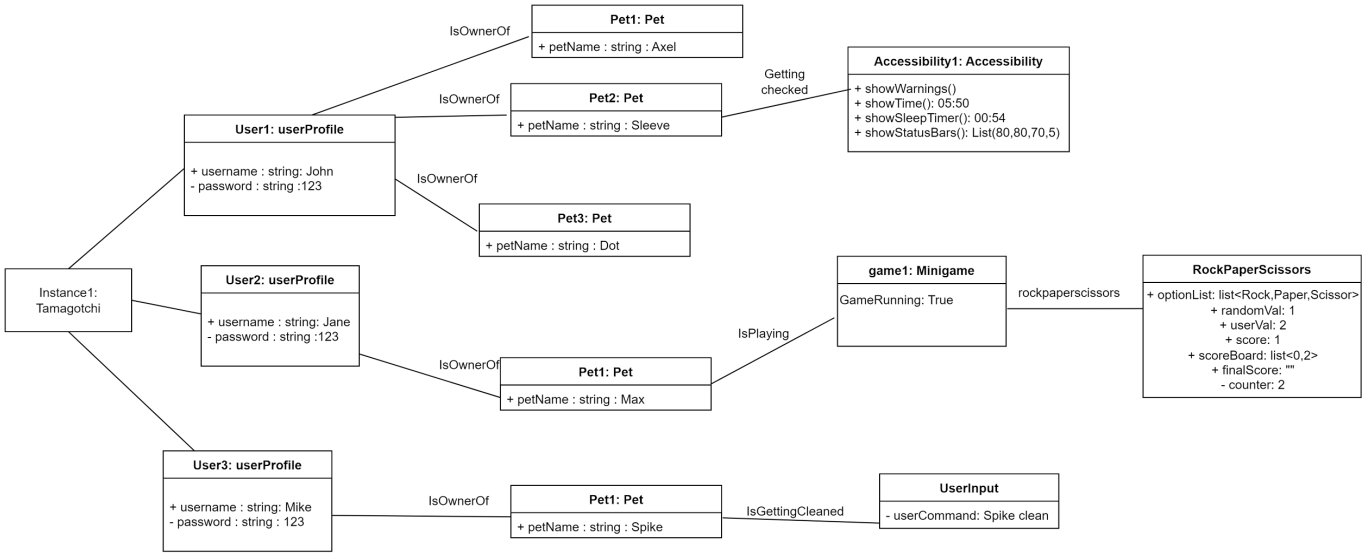
This class is an implementation of the minigame that we are implementing. It is called **RockPaperScissors** and uses the traditional rules of the hand game. However, in this case the user plays with the computer. It has multiple attributes where the *optionList* gives the list of options to choose between Rock, Paper and Scissor. *randomVal* is the random int value between 0 to 3 to pick Rock, Paper, Scissor value from the list. This is the computer generation. *userVal* is the value that the user picks. The score keeps track of the score for each loop and the scoreboard is the cumulative score of all rounds. The *finalScore* shows who wins – the user or the computer. The *counter* keeps track of the number of rounds.

It has methods to such as calculateOutcome() that calculates the final outcome of the game. Next, generateRandVal()generates a random integer to pick. getUserVal() gets the value the user picks. trackScores() tracks the scores for each round and stores it in list form. displayScore() displays score of each round and displayFinalScore() displays who wins – the user or the computer.

The relationship between the **{abstract} Minigame** and **RockPaperScissors** is inheritance. **RockPaperScissors**inherits the values of the **Minigame** class and is a direct implementation of this class.

Object diagram

Author(s): Kanushree and Wissal



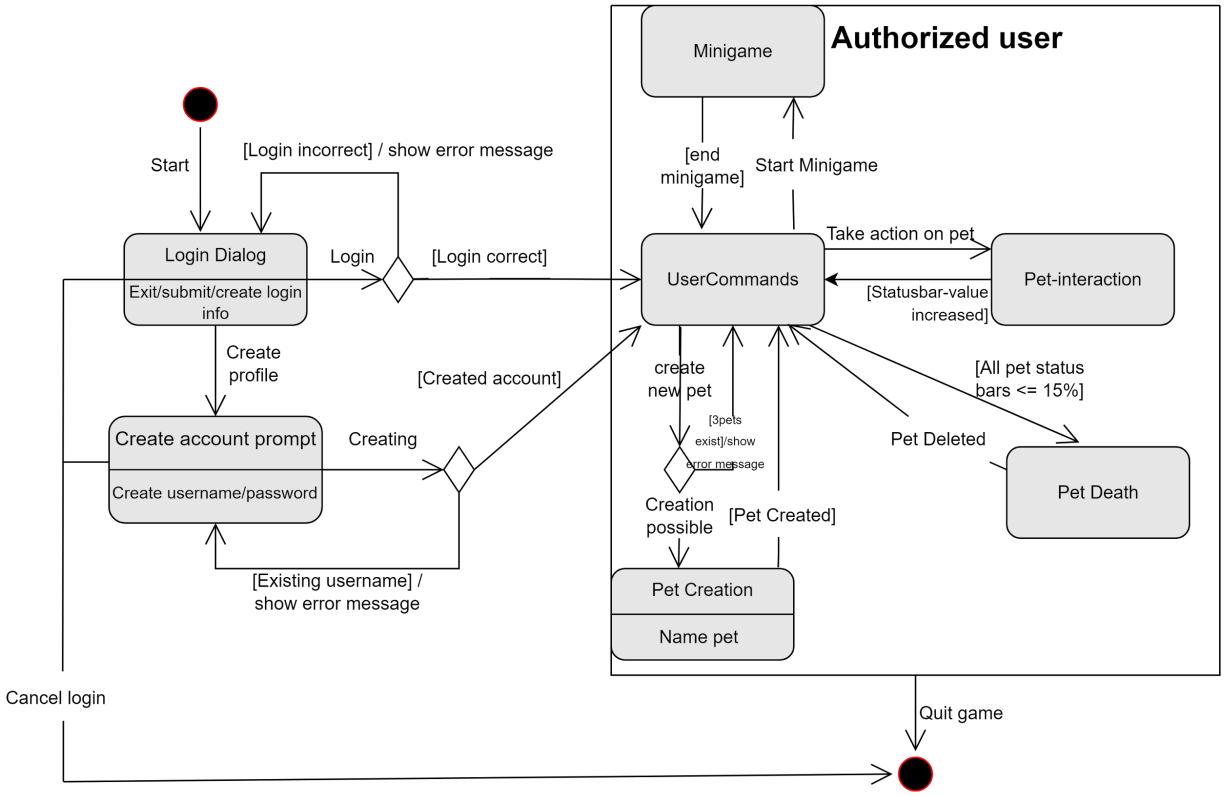
This is a snapshot of the status of our system with 3 users. One of the users has 3 **pets** while the other two have one **pet** each. Our main class of **Tamagotchi** is the game itself. This class holds the 3 **users** which in turn hold their respective **pets**. Each user is only able to interact with one **pet** at a time. The first user is checking his **pets'** status bars. The second user is playing a **minigame** and the third user is cleaning his **pet**. The checking is done through the

Accessibility class, the minigame is played through the **minigame** class and the standard interactions with the **pets** are done through the **UserInput** class.

State machine diagrams

Author(s): Turan and Shadman

Class Name: UserProfile

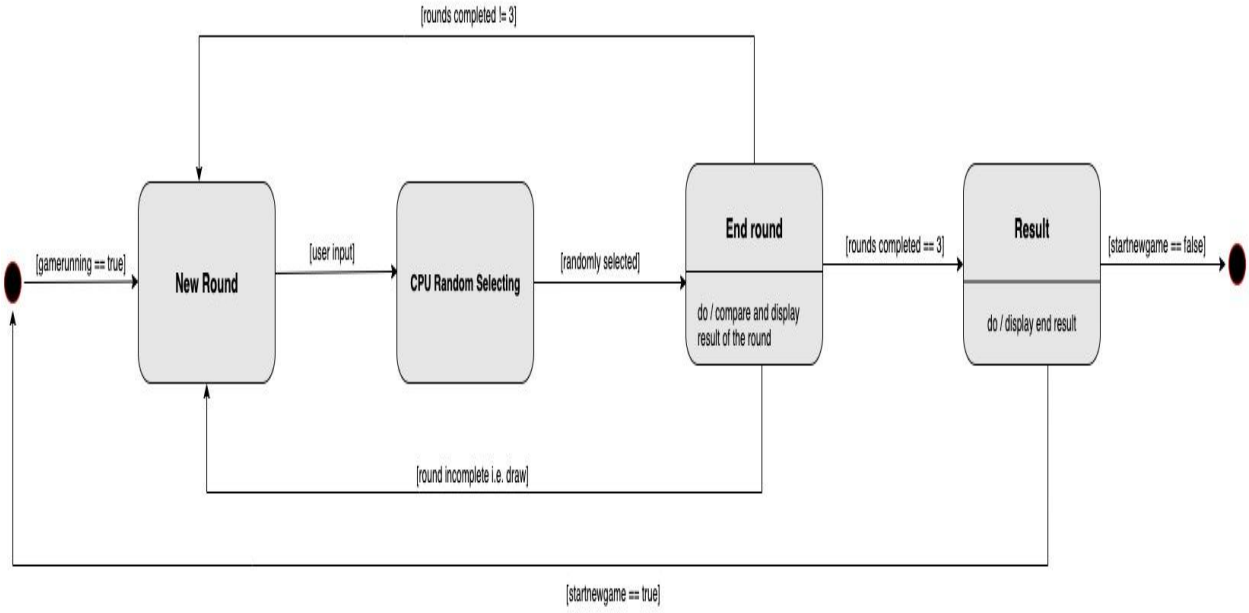


When the **user** launches the application, we will prompt them to sign in or register. We create **user** accounts so that other **users** will not be able to see their **pets** or interact with them. In addition, error handling is performed, which prompts the user to re-enter their credentials or register again if the credentials do not match or if the username has already been taken. An **UserProfile** instance is created as soon as the game gets started up. This instance checks for matching username and passwords. This instance is also able to create a new account. After the **user** has successfully *registered* or *logged in*, the **user** is taken to the game's main screen, where they are able to enter commands. For instance, the **user** is only allowed to *create* a new **pet** if there are fewer than three **pets** that already exist. **Users** are prompted to provide a pet name before an instance of the **Pet** class can be initialized for each of the **pets** that are generated by the system. The **pet** status bars are monitored by the **Tamagotchi** class instance, and as time passes, the bars decide whether the **pet** lives or not. The bars deplete over time. This depletion is being done in order to simulate situations that may occur in real life, such as when a person's **pet** may become hungrier as time passes. In addition, the user can

communicate with their **pet**(s) by issuing commands such as "feed." Because of this, the application will react by displaying an improved status bar. For example, eating will make the hunger bar that is displayed on the screen look better. Warnings are delivered to the user via the **Accessibility** class whenever one or more status bars of a **pet** fall below a predetermined threshold value. Once they all reach below 15%, the **pet** is considered to have passed away and is removed from the system. From the main screen, users also have the option to play a **minigame** that is referred to as Rock, Paper, and Scissors.

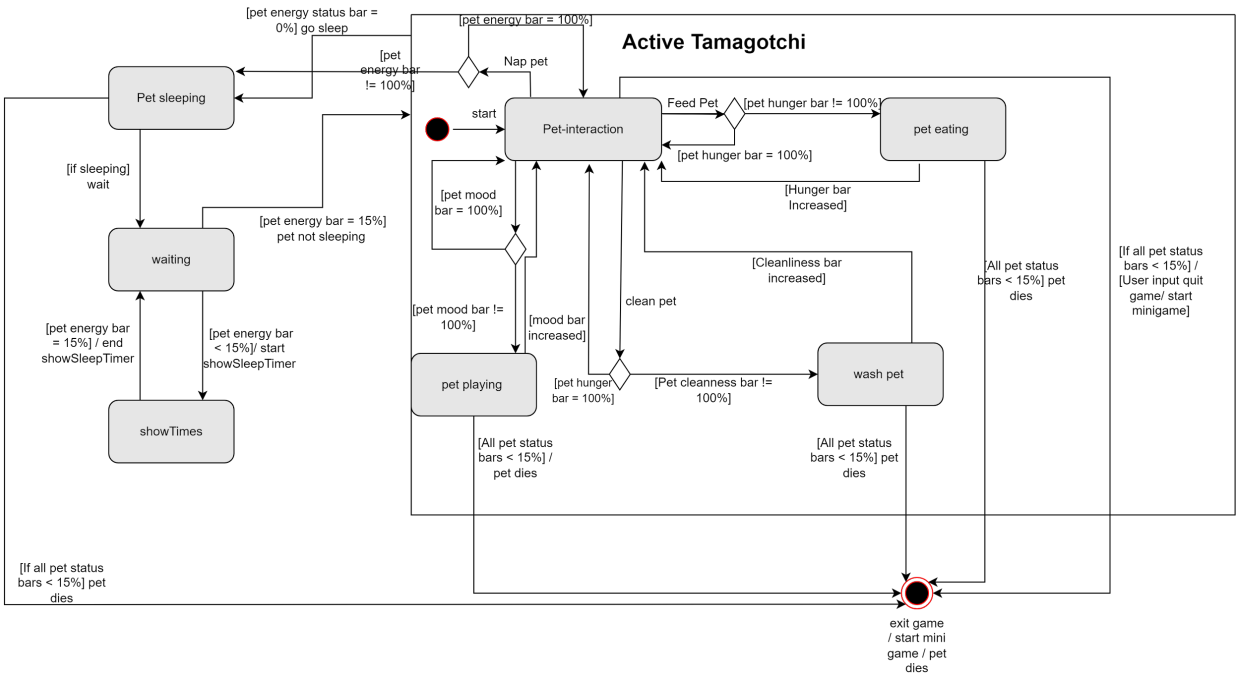
The reason this state machine diagram was designed as it is to make it as clear and straightforward as possible. The **user** firstly has to login just as any other application in reality which uses that function. The **user** will do the rest with commands. The **user** can decide to quit the game as soon as he enters the login dialog. The main abilities of the **user** are starting the **minigame**, *creating a pet*, *interacting with a pet* and quitting the game. The only thing that changes the state which the user does not control is the *death of the pet*.

Class Name: Minigame



We make a new instance of the "**minigame**" class every time a **user** requests to play the **minigame** that we provide. As soon as the program begins, the **user** will be prompted to provide an input, such as "rock, paper, or scissors." After receiving input from the **user**, it is now the turn of the CPU, which plays by choosing an option at random from the three available choices. After the **user's** input and the computer's selection are compared against each other, the round is over, and the result is shown on the screen. In the event that the **user** input and the random selection produce the same result, the round will be restarted. After the three rounds have been played without a tie, the game is over, and the winner's name is shown on the screen. The **user** can then choose whether or not they want to begin a new game, in which case another instance of the **minigame** class is created, and the procedures described above are carried out once more. The **user** may also choose to exit the game and go back to the app's primary menu at any time.

Class Name: Pet interaction



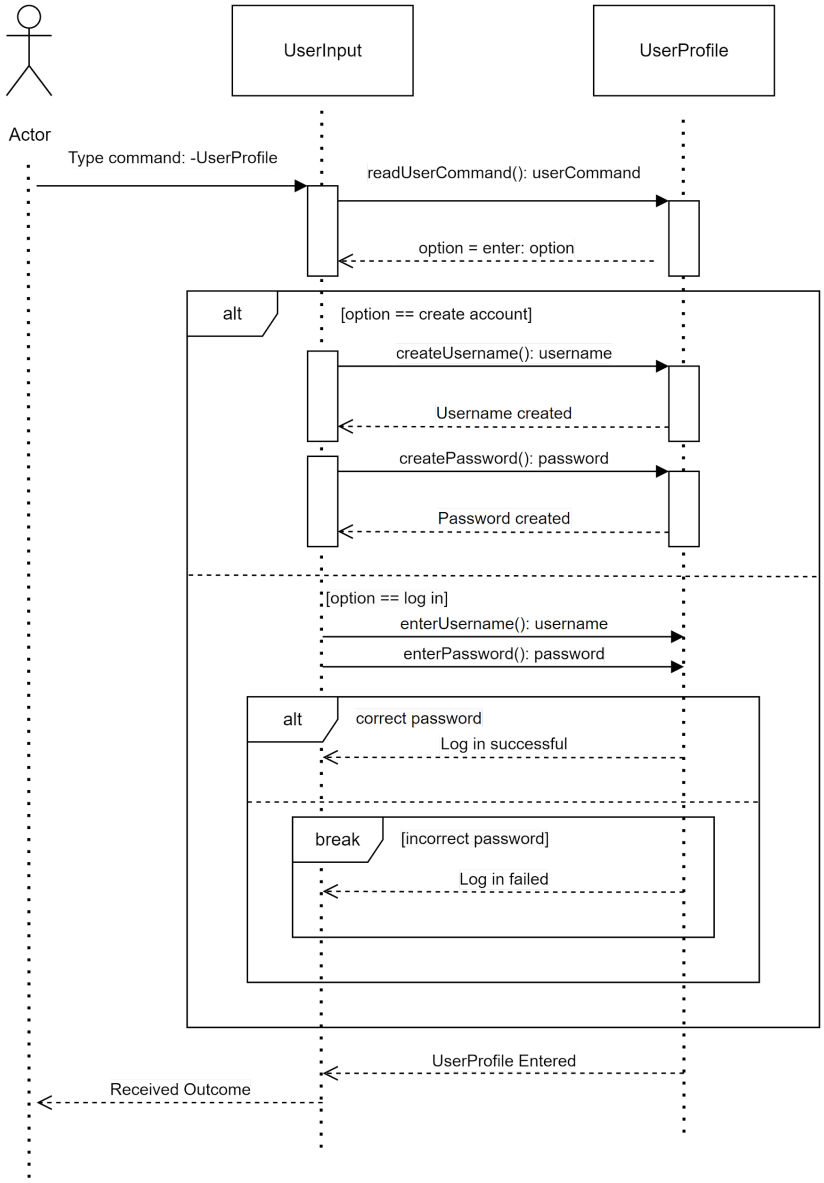
This diagram focuses on the interaction between the **user** and the **pet**. As soon as the **user** enters the pet-interaction state, they will be able to communicate with their **pet**(s) by issuing different commands and also depending on the percentage that the status bars display. The **pet** status bars are monitored by the **Tamagotchi** class instance, which is shown in detail in the first state machine diagram (**UserProfile**). The user can choose to perform different actions on the **pet**, such as feed, play and clean pet. When entering one of these states, the **user** can choose whether to perform the action, in case the status bar is not 100%, or go back to the initial state if the status bar displays 100%. Performing one of these actions means that the corresponding status bar is increased (hunger bar for feed, mood bar for play…). However, for the pet sleeping state, which the **user** can enter if the **pet** needs to take a nap (energy bar < 100%) or if the pet needs to sleep (energy bar = 0%), the **user's** actions are limited in this state, as the **tamagotchi** is no longer active. The **user** cannot perform any other action while the **pet** is asleep, and needs to wait for the sleep timer to end. In the case where the status bars drop

below 15%, the **user** has reached the final state, the **pet** is considered dead, and the user can either exit the game, or start the **minigame**.

Sequence diagrams

Author(s): Turan and Shadman

User Interaction while creating a User Profile



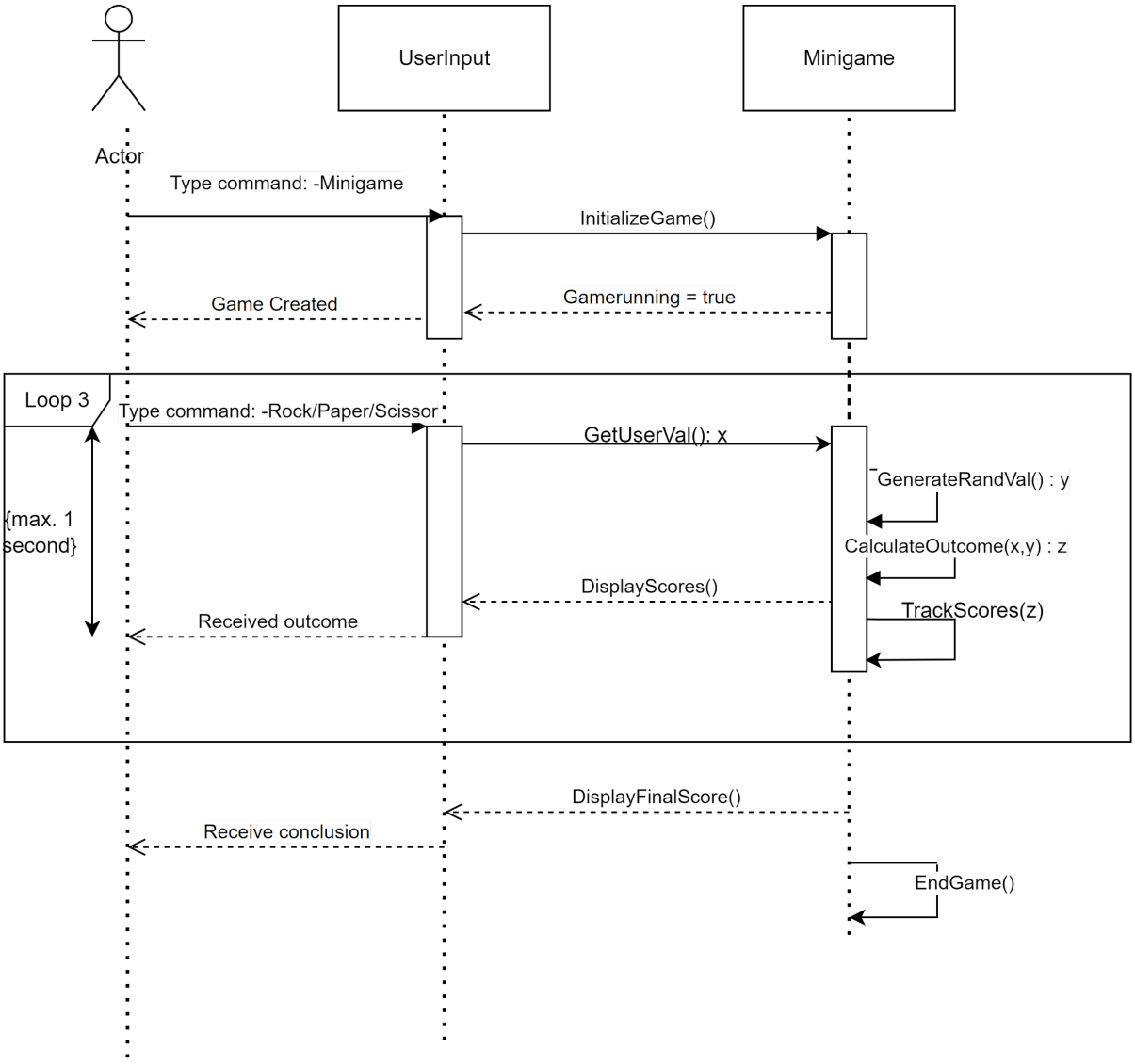
In the above sequence diagram, the interaction between the user and the program is shown when the user is creating a **userprofile** for the **Tamagotchi** game. This diagram shows the interaction between the user (which is the user playing the **Tamagotchi** game), the **UserInput** class and the **UserProfile** class. As both the **UserInput** and **UserProfile** class have been described earlier, we will not go into much detail about it. However, in brief, the **UserInput** class deals with the commands of the user (the actor). The user inputs the command and the **UserInput** reads the command and directs it to the necessary class. In this case, it is the **UserProfile** class. The **UserProfile** deals with the profile of the user such as creating the username and password for the user.

In this sequence, the actor types in a command called **UserProfile** and the **UserInput** class handles the command by reading it and directing it to the **UserProfile** class. The **UserProfile** class then sends back a message to the **UserInput** asking it for an option. The two options that the **user** can choose is to log in to their account or to create a new account. From the sequence diagram, we can see this alternative action from the user. If the user decides to log in to their

account, the command from the user is read by the **UserInput** and the **UserProfile** runs the *createUsername()* function and stores it to the username attribute. After this, the **UserProfile** sends a Username created message to the actor. Similarly, the **UserProfile** creates a password for the same user using the *createPassword()* function and stores it to the password attribute. The other alternative is when the user decides to log in to their existing account. Here the username and password are typed in by the user through **UserInput**. Another alternative case is described here after the user enters their details using *enterUsername()* and *enterPassword()*. If the password is correct, the receiving message from the **UserProfile** is a successful log in. If the password is incorrect, the receiving message from the **UserProfile** is a failed login. In this case, there is a break statement that breaks out of the alternative case and just responds with the failed login message. On the other hand, if the user has created an account or logged in successfully, the **UserProfile** sends a message saying that the **UserProfile** is entered. The actor then receives this final outcome through the **UserInput**.

The authorization of the user is done through a separate database that is excluded from the sequence diagram for the sake of simplification. However, checking if the username or password is correct or incorrect is done through a file that is accessed during run time. In this file, the previous user data is stored, and the new user data is added while creating the account.

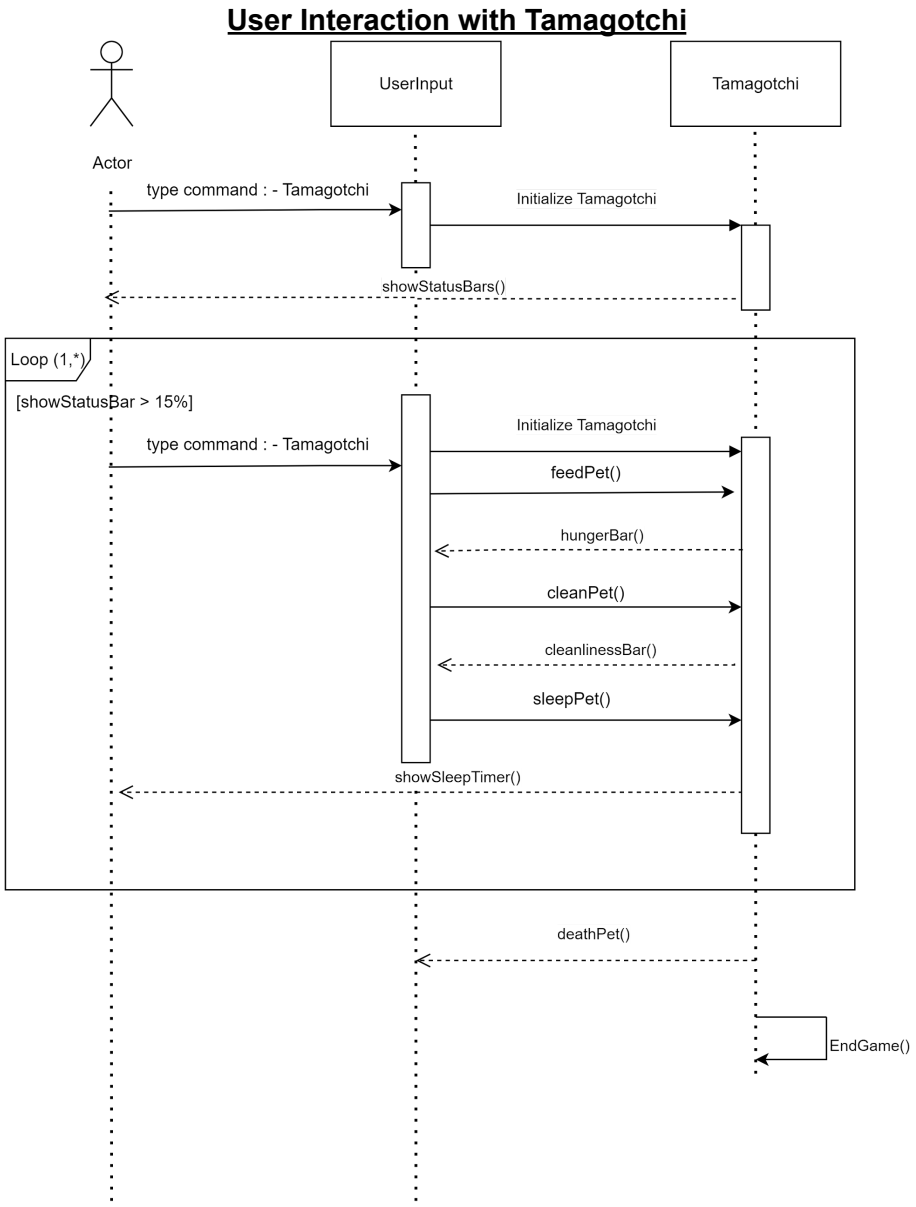
User Interaction with the minigame



This sequence diagram is fairly simple since the **minigame** is a simple rock-paper-scissors game. The user will interact with the **minigame** through the **UserInput** class. The **UserInput** will forward commands from the users and redirect outcomes from the **minigame** to the user. When the user puts the command “-Minigame”, the **UserInput** will make sure the **minigame** gets initialized through *InitializeGame()*. The user will then receive a message of the successful creation of the game. The game will then enter a loop which will repeat itself 3 times. In the loop the user picks his preferred action, the **UserInput** class forwards this to the **minigame**. The **minigame** class will pick a random value with *GenerateRandVal()* and compare the value of the user and the random value with *CalculateOutcome()*. With this the **minigame** sends the outcome. The **minigame** class will also keep a record of these outcomes with *TrackScores()*. This all gets repeated 3 times. In the end the **minigame** class will send the final scores with *DisplayFinalScore()* and will then close his initialization with *EndGame()*.

We decided to make the **minigame** class do everything on its own with just the command input of the user. This will make sure the **minigame** is as independent as possible from the other

classes. This will cause the **minigames** to be easily extendable which in turn makes sure our maintainability attribute gets met.



This sequence diagram models the interaction between the **UserInput** and the **Tamagotchi** class. The user will interact with the **Tamagotchi** through the **UserInput** class. In this sequence, the actor types in a command called **Tamagotchi** and the **UserInput** class handles the command by reading it and directing it to the **Tamagotchi** class. The **Tamagotchi** class then sends back a message to the user to *display the status bars*.

The game will then enter a loop that will execute at least once, if the pet is not dead, which means, all the status bars are above 15%. The user will then put in the command “**Tamagotch**i”. The **UserInput** will make sure **Tamagotchi** is initialized. After that, the **userInput** can send different commands to **Tamagotchi**. In our diagram, the **userInput** first forwards the command *feedPet*, and gets a message back showing the hunger bar. Same is done with the *cleanPet* command. **Tamagotchi** sends a message back, giving the status of the corresponding bar. For the *sleepPet* command, in addition to sending a message to the **userInput** for the energyBar, **Tamagotchi** sends another message to the user showing the sleep timer.

When the guard evaluates to false, which in our case means the status bars are equal or less than 15%, the loop is exited and **Tamagotchi** sends a message to the **userInput** that the **pet** is dead. The game is then over.

Time logs

Member	Activity	Week number	Hours
Kanushree Jaiswal	Summary of changes	3	2
	Class Diagram	3	4
	Object Diagram	4	4
Wissal Mestour	Summary of changes	3	2
	Class Diagram	3	4
	Object Diagram	4	4
Turan Yigit	State Machine Diagram	4	4
	Sequence Diagram	5	6
Shadman Sahil Chowdhury	State Machine Diagram	4	4
	Sequence Diagram	5	6