Assignment 3

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 2:

Author(s): Kanushree

Changes made in the class diagram to improve the quality of diagrams:

- Changed the navigability direction and added multiplicities from the UserInput class to the UserProfile, Pet, and Minigame classes.
- Changed the Pet class to an abstract class from which the Tamagotchi class inherits properties
- Changed the multiplicity from any number to 3 pets in the Pet class.
- The relationship between Tamagotchi and Accessibility has been updated to a composition relationship and the multiplicity is changed.

Changes made in the object diagram to improve the quality, completeness, and description of diagrams:

- The UserInput class represents the system as a whole as it connects all necessary parts of the system
- The Pet class is changed to an abstract class so only instances of the Tamagotchi class are used in the diagram
- All abstracts classes are replaced with their inherited classes as there was a syntax error in the previous diagram
- Object names are given to all instances in the diagram
- Functions are not shown in the class diagram - the syntax error is corrected.
- The diagram is consistent with a system using the command line interface.
- A more descriptive description is added for the diagram in this assignment.

Changes made in the state machine diagrams to improve the quality, completeness, and description of diagrams:

- The design decisions of the models are explained briefly in this assignment to accompany the lack of explanation in the previous assignment.

- The guard conditions in the diagrams are changed and/or accompanied by events to ensure that the states are left (in all diagrams).
- Necessary start and exit nodes are added to the diagrams to ensure that the composite states are entered and exited accurately.
- The Minigame (an abstract class) diagram is changed into a RockPaperScissors diagram.
- The RockPaperScissors diagram indicates a transition from every state to the final state to ensure the game can be exited at any point.

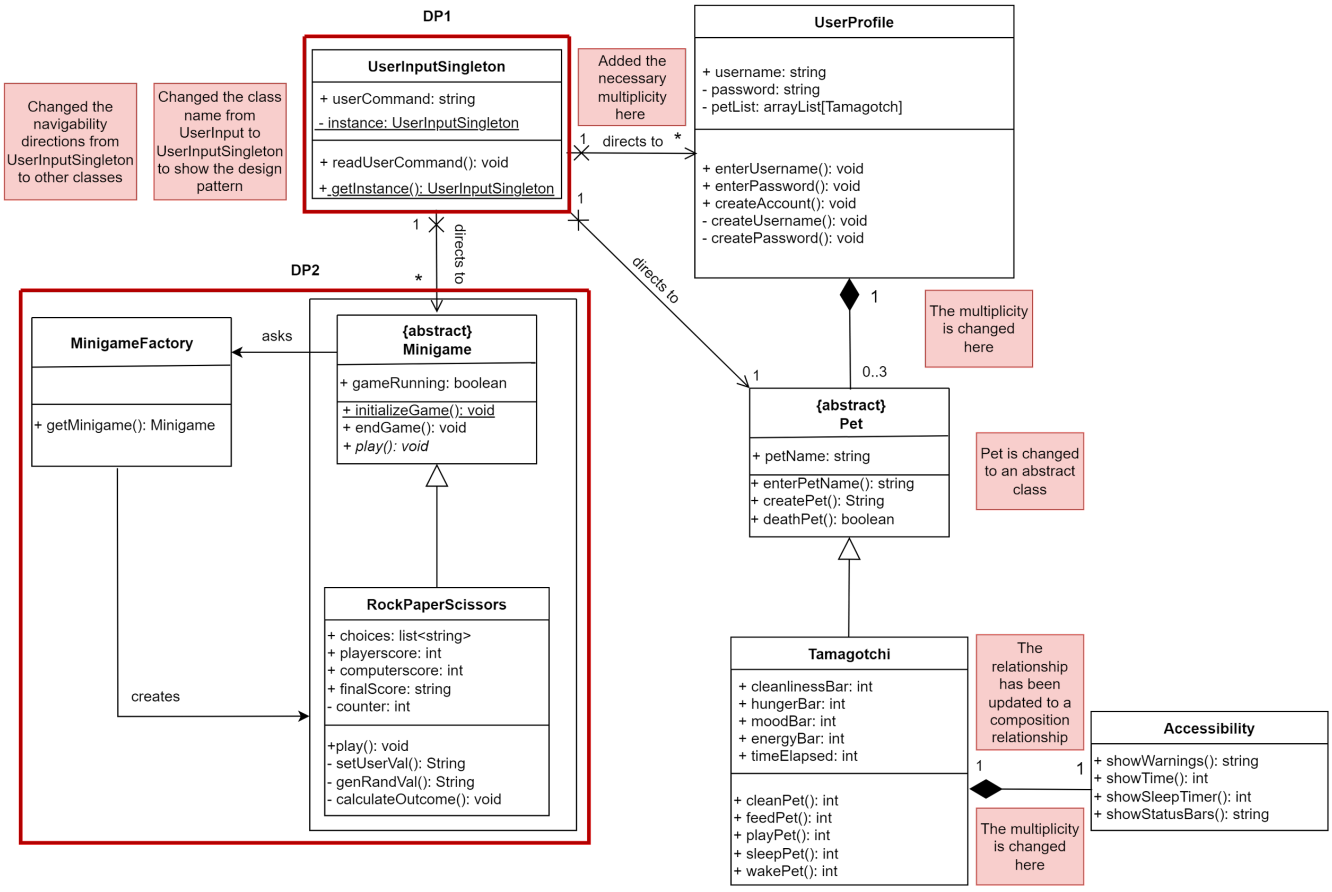
Changes made in the sequence diagrams to improve the quality, completeness, and description of diagrams:

- Added a revised description where the design decisions are discussed in more detail.
- Changed major syntax errors across all sequence diagrams to ensure that it is represented correctly - lifeline representation, synchronous messages, instances, class methods etc.
- Changed the diagrams to ensure only instances of classes are in the diagram.
- Changed the diagrams to ensure it is consistent with functions provided in assignment 1.
- Changed the syntax error where the abstract class is removed and replaced by the inherited class (in the minigame diagram).
- Used getters and setters correctly (in the minigame diagram).
- Objects are created to show the correct sequence in the diagram (in the Tamagotchi diagram).
- Class instances that are not depicted in the previous diagrams are updated in this.

Revised class diagram:

Author(s): Kanushree

UML Class Diagram for Tamagotchi Project



The above is an updated class diagram representing various classes involved in the Tamagotchi Project Implementation. Given below are the main improvements and the use of design patterns in the project.

Main Improvements:

UserInputSingleton

As the name of the class suggests, the name of the class has been changed to **UserInputSingleton**. This is to show the design pattern that has been used in this class. New attributes and functions are added to implement the singleton design pattern which will be explained in depth below. The new attribute that is added is the *instance* attribute which is a class variable of type UserInputSingleton. This variable stores the instance of a given class. The method getInstance() checks if an instance already exists. If not, it creates a new instance for the given class. This is a class method.

The relationships from the **UserInputSingleton** class to **UserProfile**, **Minigame** and **Pet** have also been updated. These are now unidirectional binary association relationships extending from **UserInputSingleton** to **UserProfile**, **Minigame** and **Pet** respectively. The multiplicity from **UserInputSingleton** is always 1 as it is a singleton class. **UserProfile** can have multiple instances which one **UserInputSingleton** class deals with. One **UserInputSingleton** can interact with at most 3 pets. One **UserInputSingleton** has a relationship with multiple minigames. This is a factory design pattern implementation and is explained further below.

UserProfile

A new attribute is added to the **UserProfile** class called *petList* which is an array list consisting of the type **Tamagotchi**. This data structure stores the user pets in the array. Moreover, all the methods of the class return data of type void (no data is returned).

The relationship between **UserProfile** and **Pet** remains the same. However, the multiplicity is changed. Each **UserProfile** class can have at most 3 pets. This is to ensure that the relationship is consistent with previous features. Only 3 pets are allowed per user.

{abstract} Pet

The **Pet** class is now changed into an abstract class from which the **Tamagotchi** class inherits all its properties. This is because an instance of the **Pet** is not created without an instance of the **Tamagotchi**. Both classes share similar information in the class diagrams. Moreover, the **Tamagotchi** class handles information related to the **UserInput** class. Since, **Pet** is an abstract class now, the pet name is entered through the pet class and all the **UserInput** commands are handled in **Tamagotchi**.

Tamagotchi

There are no changes in this class except its relationship with the **Accessibility** class but this will be discussed below.

Accessibility

The class in itself has no major changes but its relationship to **Tamagotchi** is now a composition relationship. Essentially, an instance of the **Accessibility** class can not exist without an instance of the **Tamagotchi**. This also explains the multiplicity. For one instance of the **Tamagotchi** class only one instance of the **Accessibility** class exists.

{abstract} Minigame

There are multiple changes made here to adjust the workings of the **Minigame** into the factory design pattern. The design pattern will be discussed in depth in the next section. However, a new class method initializeGame() has been added to the **Minigame**. As the name suggests, the respective minigame is created using this method. Another method play() is added to the **Minigame** class. This method is where the minigame is played.

Here we can also see that the **Minigame** class asks the **MinigameFactory** to pick a minigame that needs to run. This essentially adheres to the Factory design pattern that is discussed later.

MinigameFactory

This class is to apply the Factory design pattern to the minigame aspect of the project. Although the implementation of the factory design pattern is explained in detail in the next section, in short, this class picks one minigame to be implemented from a range of minigames. Since our system only contains one minigame the **MinigameFactory** picks this game. However, if there were multiple minigames to choose from, the **MinigameFactory** would pick one of these minigames. It is to make the system more dynamic.

This class contains one method called getMinigame(). As the name suggests, it gets the required minigame to be played.

As you can see from the navigation in the class diagram, the **MinigameFactory** "creates" the necessary minigame. In our case, it is **RockPaperScissors**.

RockPaperScissors

Not a lot is changed in this class. It is now connected to the factory design pattern. The new attributes are *choices* of type list<string>, *playerscore* of type int, *computerscore* of type int, *finalScore* of type string and *counter* of type int. The methods are play() of return type void, setUserVal() of return type string, genRandVal() of return type string and calculateOutcome() of return type void. All methods and attributes are self-explanatory and similar to the previous version.

DP1

This class shows the Singleton design pattern that is explained further below.

DP2

These classes show the Factory design pattern that is explained further below.

Application of design patterns:

Author(s): Turan and Kanushree

	DP1
Design pattern	Singleton
Problem	The UserInput class is where the UserProfile class is called. Having multiple UserInput classes would make it so that some users would not be able to log in again because of multiple instances.
Solution	Restricting the UserInput class to only one instance ensures that all the users can access their account once they logout.
Intended use	The UserInput class stores its own instance and only gives this out once to the main class.
Constraints	None
Additional Remarks	None

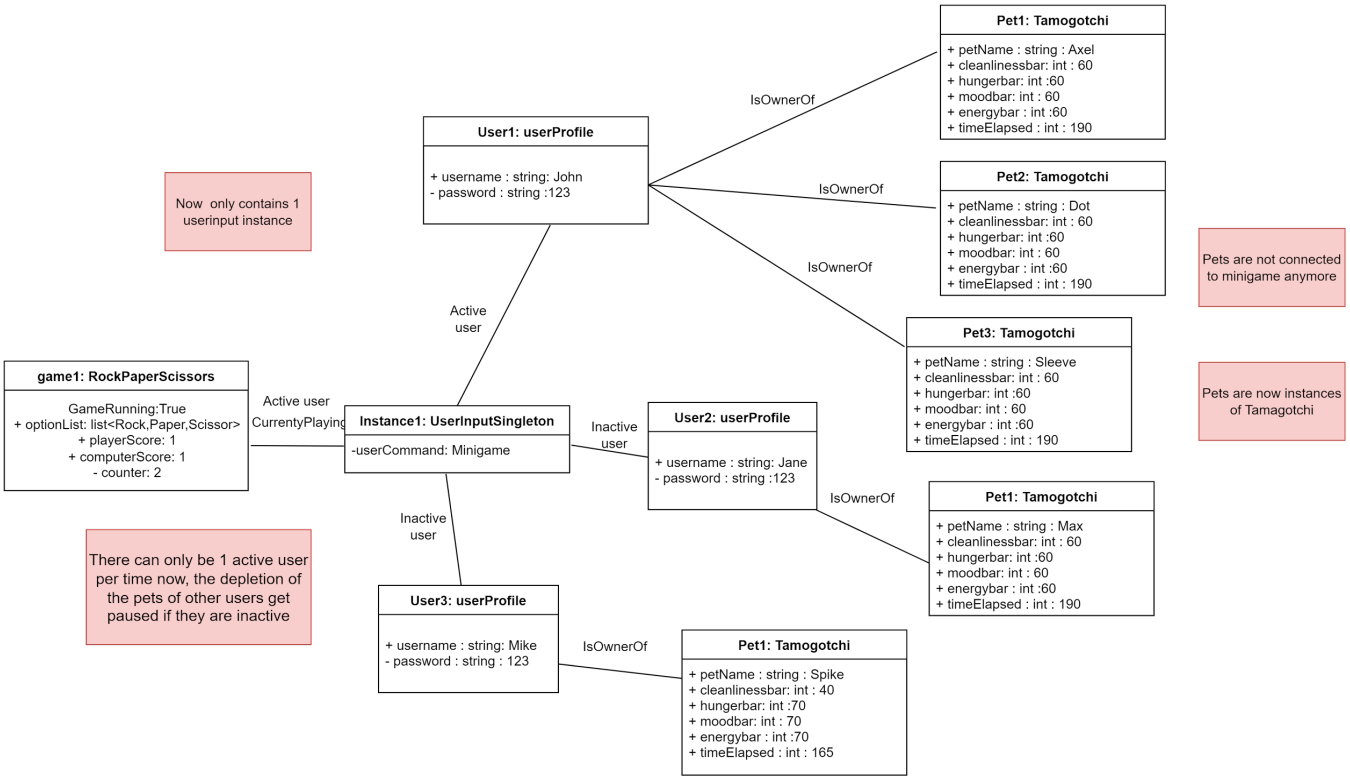
The singleton design pattern is used to ensure that only one instance of a class exists. It contains the attribute instance and the method getInstance() which are class attributes and methods respectively. To elaborate on the above table, it is important for the user distinction to be very accurate. Although our implementation is not on an expanded system, a design pattern like this is very useful when it has to do with user logins and logouts. It is also very useful from a security and safety perspective, to make sure there is only one instance of a class with the necessary storage.

	DP2
Design pattern	Factory
Problem	To make the addition of more minigames more maintainable, we need an easier and more efficient way to initialize the minigames.
Solution	If we use the Factory design pattern, we can call the “new” from within the factory method which makes the objects easier to implement.
Intended use	The abstract minigame class will also serve as a factorypatterndemo, now we only need to create a factory class that initializes one of the subclasses.
Constraints	None
Additional remarks	Using the abstract minigame class as the factorypatterndemo made more sense since it only passes the information of which subclass has to be created. This can be easily done in one line with the information which gets sent to the abstract minigame class anyway.

The factory design pattern is used for the implementation of the minigame. We wanted to make it possible for developers to be able to add in or change their minigames to the implementation as easily as possible. Therefore, we decided to take it a step further and allow them to not only change the minigame but also add multiple minigames. The minigame factory essentially would allow the user to select one of the minigames that were added. Again, our implementation does not have access to multiple minigames as it is a small system. However, this pattern would be very helpful in future more dynamic implementations.

Revised object diagram:

Author(s): Turan



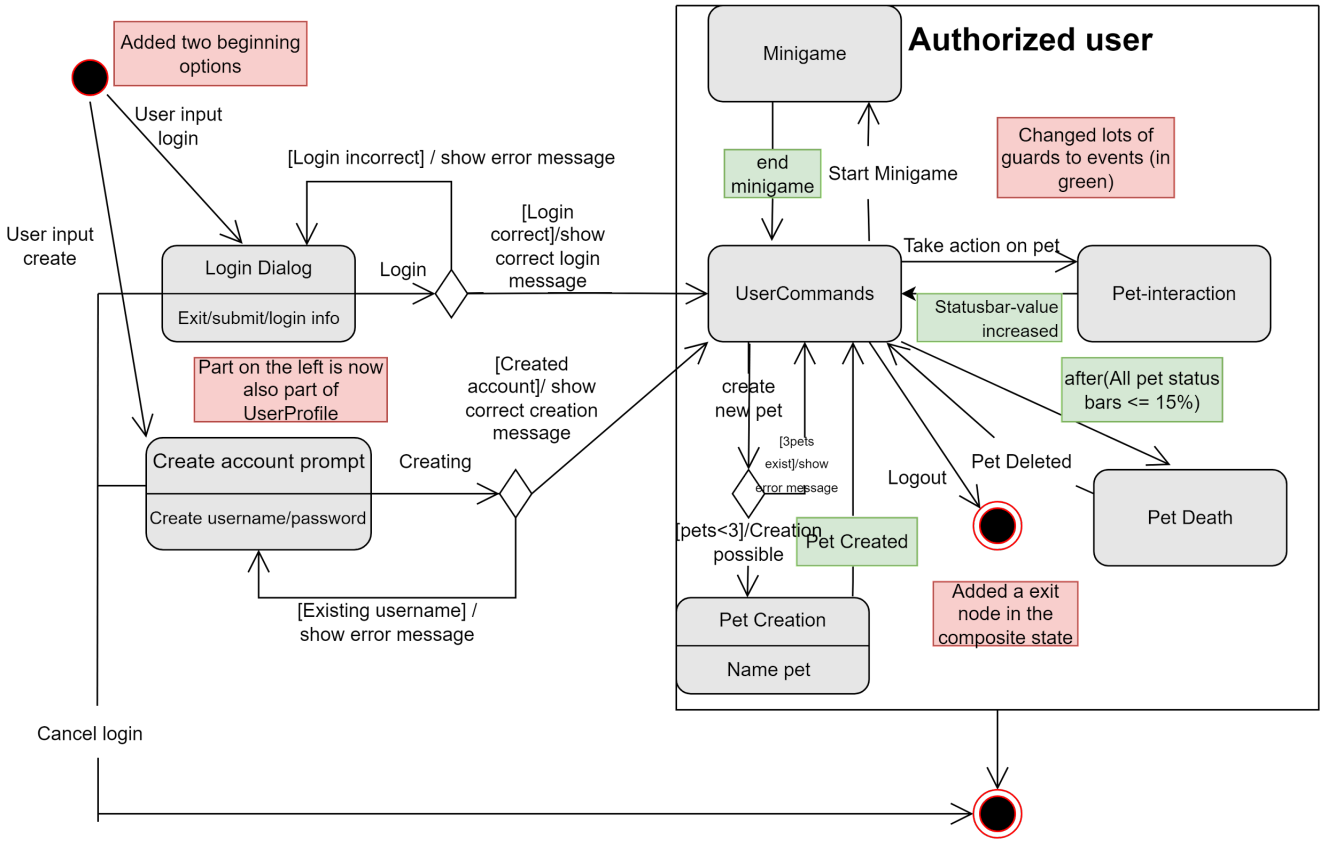
The **UserInput** class represents the system as a whole as it connects all necessary parts of the system. All abstract classes are replaced with their inherited classes as there was a syntax error in the previous diagram. Object names are given to all instances in the diagram. Functions are not shown in the class diagram - the syntax error was corrected. The diagram is consistent with a system using the command line interface.

We have decided on making the **UserInput** a singleton class as this would make the implementation a lot easier. With only one available **UserInput** instance, there can only be one **User** active at a single time. This made it so that we did not have to implement threading which we were unable to implement in time. Many errors were corrected like the connection between **Pet** and **RockPaperScissors** got moved to **UserInput** and **RockPaperScissors. RockPaperScissors** was previously named **Minigame** which was also incorrect since **Minigame** is an abstract class and cannot be initialized. We have also initialized the **Pet** as a **Tamagotchi,** the reasoning behind this is that the **Tamagotchi** and **Pet** classes shared a lot in common, and making the **Pet** an abstract class was more logical. The **Accessibility** class got removed since only 1 **User** can be active at a single time, it is not possible for this **User** to play a minigame and check on his **Pet** at the same time.

Revised state machine diagrams:

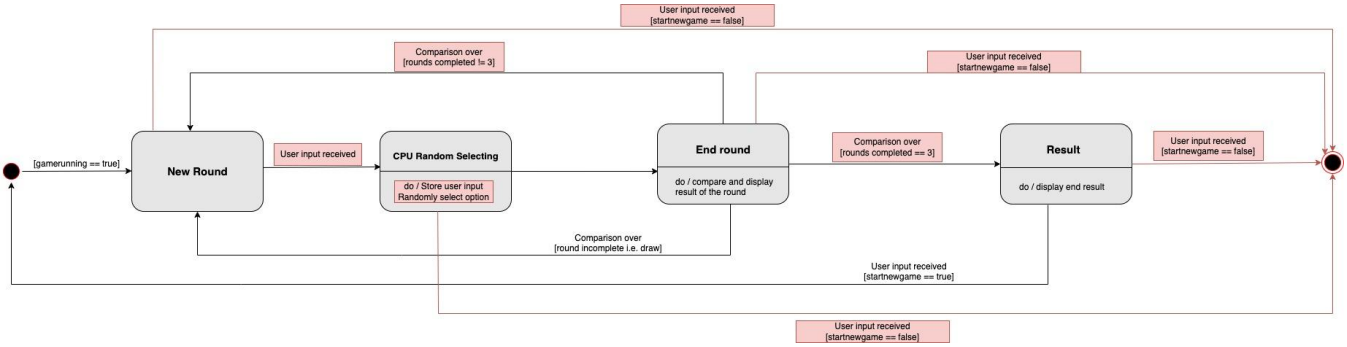
Author(s): Wissal and Shadman

Class Name: UserProfile



Firstly we will go over the mistakes we have fixed. A lot of guards got changed to events, we added an exit node in the composite state and we have changed the classes in such a way that the left part now also belongs to the **UserProfile** class. We now initialize the **UserProfile** when a new account gets created. We've also added two starting options, this was not shown in the previous diagram and that was not intended. We have only added two improvements, the first one is to have the pet status bars checked infinitely since not doing so would allow the pet to be alive even though he was meant to be dead. The second improvement is adding messages after correct login/account creation to improve the completeness of the system.

RockPaperScissors



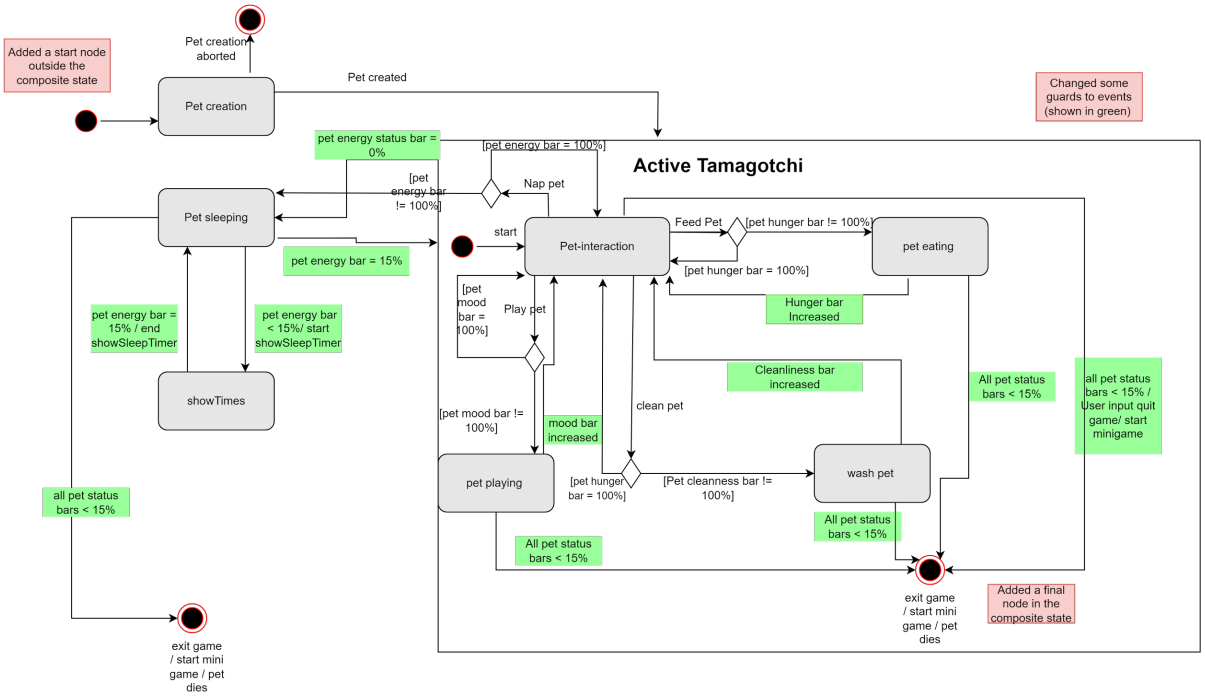
We realised that many of the state transitions did not have enough triggers to cover all possible scenarios. To tackle this issue, we have added events on top of existing guards to our revised state machine diagram. For example, after the end of the first round of rock, paper, scissors, we make sure that the comparison (of user's input and CPU's selection) is completed *and* total rounds completed is below 3. The addition of the "comparison over" event makes sure both conditions are met for the state transition to occur and enables the state to move to a new round.

Three additional state transitions have also been added. These occur in the scenario where we receive user input despite requiring it. Whenever such a scenario arises, we transition to the exit point. For each of the states in our diagram, we have added the transition to the exit point, with an event and a guard triggering it. Receiving user input is taken as the event, and "startnewgame==false" is the guard. The guard condition simply means that the user has *not* started a new game.

There was also no "do" activity specified for our "CPU Random Selecting" state. In our revised diagram, we added two activities that occur while we are in this state - storing user input and randomly selecting an option (from rock, paper, and scissors). The state then transitions to the next state when these activities have been completed.

We went forward with this design because, as stated earlier, we wanted to cover all possible scenarios of state transitions. Using this, we also do not get stuck inside a state as *do* activities have been specified where required. Lastly, the transitions straight to the exit points were added as somewhat of a "fail-safe," cover scenarios for unwanted and unprecedented actions by the user.

Class Name: Pet interaction



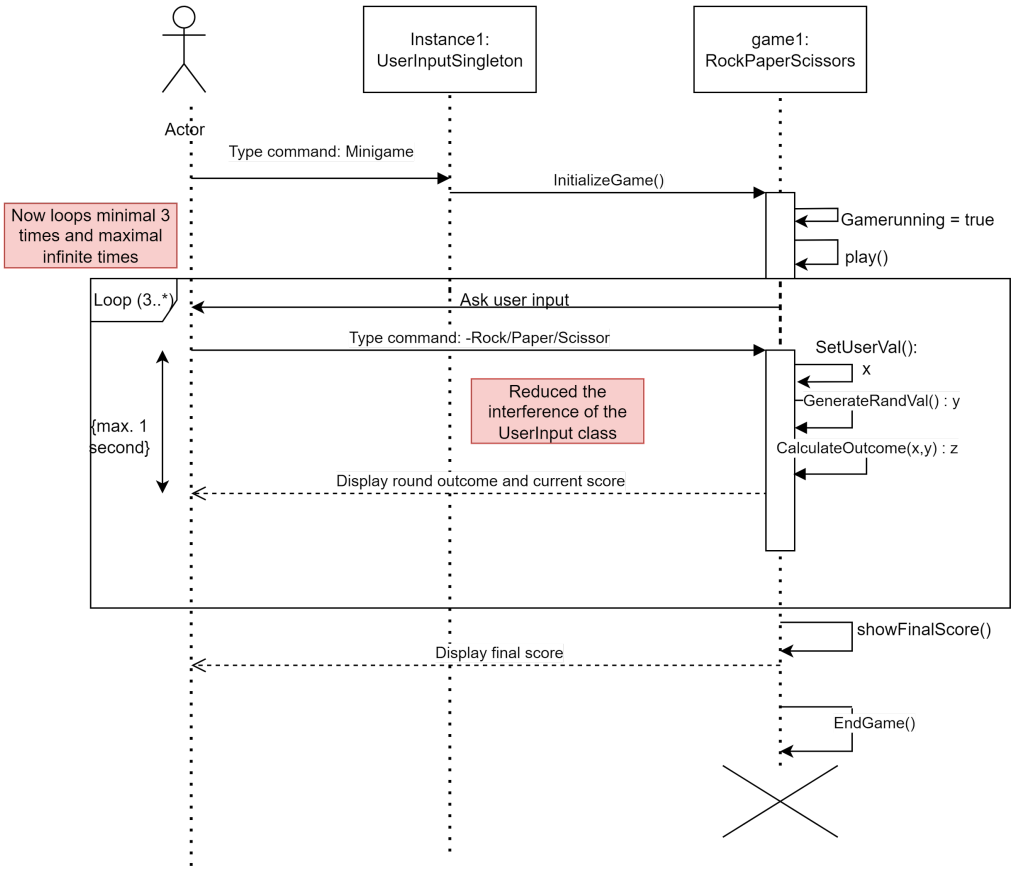
In our revised state machine diagram, we tried to fix all the issues that we had in our previous diagram. One of the improvements we made is, we added a start node outside the composite state. Moreover, a lot of guards got changed to events. Lastly, we had an issue where a transition could technically never be triggered since transitions from a composite state are only activated when the composite state has reached its final node, which we did not have. In order to fix this issue, we added a final node to the composite state.

Another thing is, we added a Pet Creation state. The user starts by creating the pet. If successful, the pet is now active and the user will be able to communicate with their pet(s) by issuing different commands. If not successful, the state is exited, the pet creation is then aborted, and the exit node is reached.

Revised sequence diagrams:

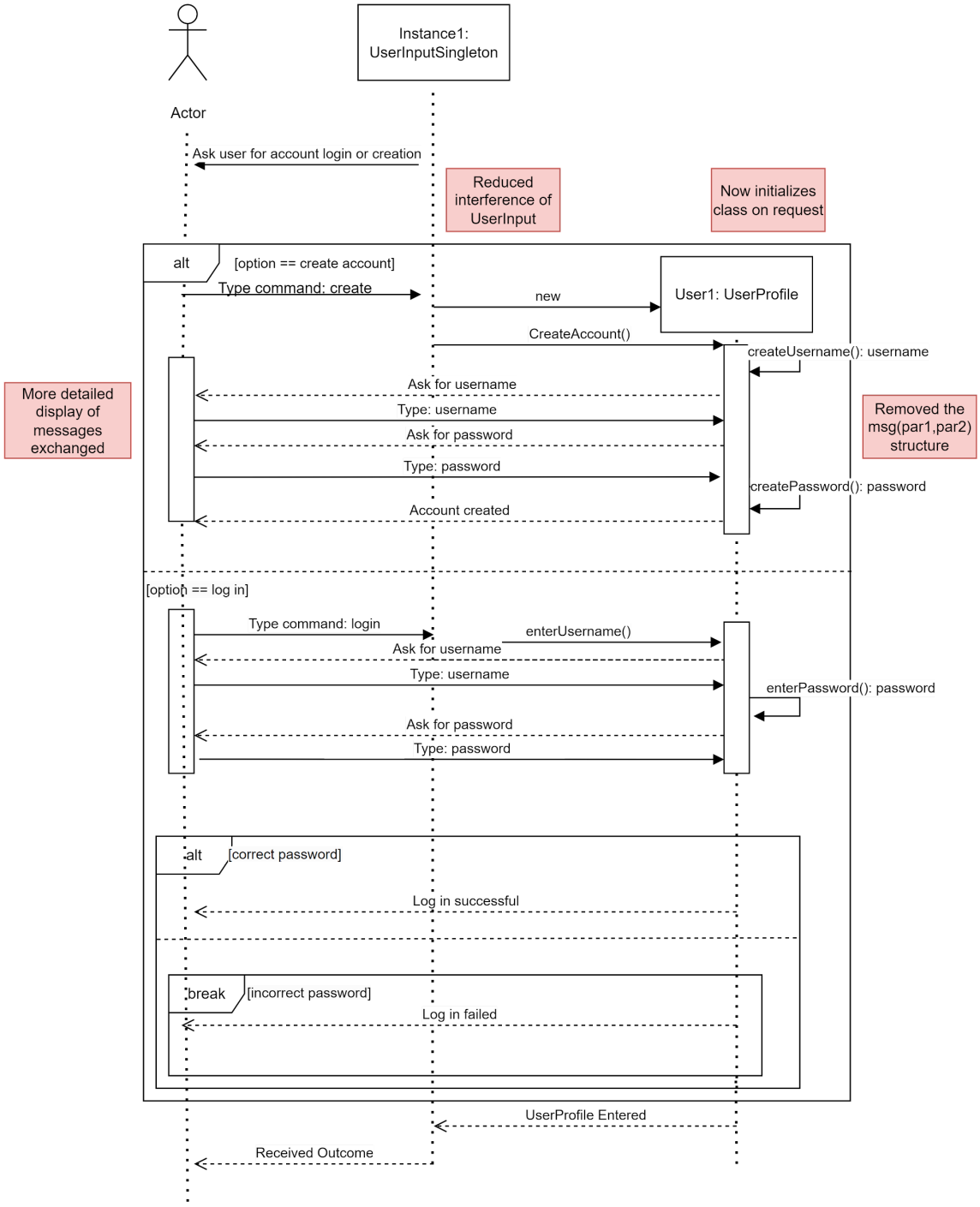
Author(s): Turan

User Interaction with the minigame



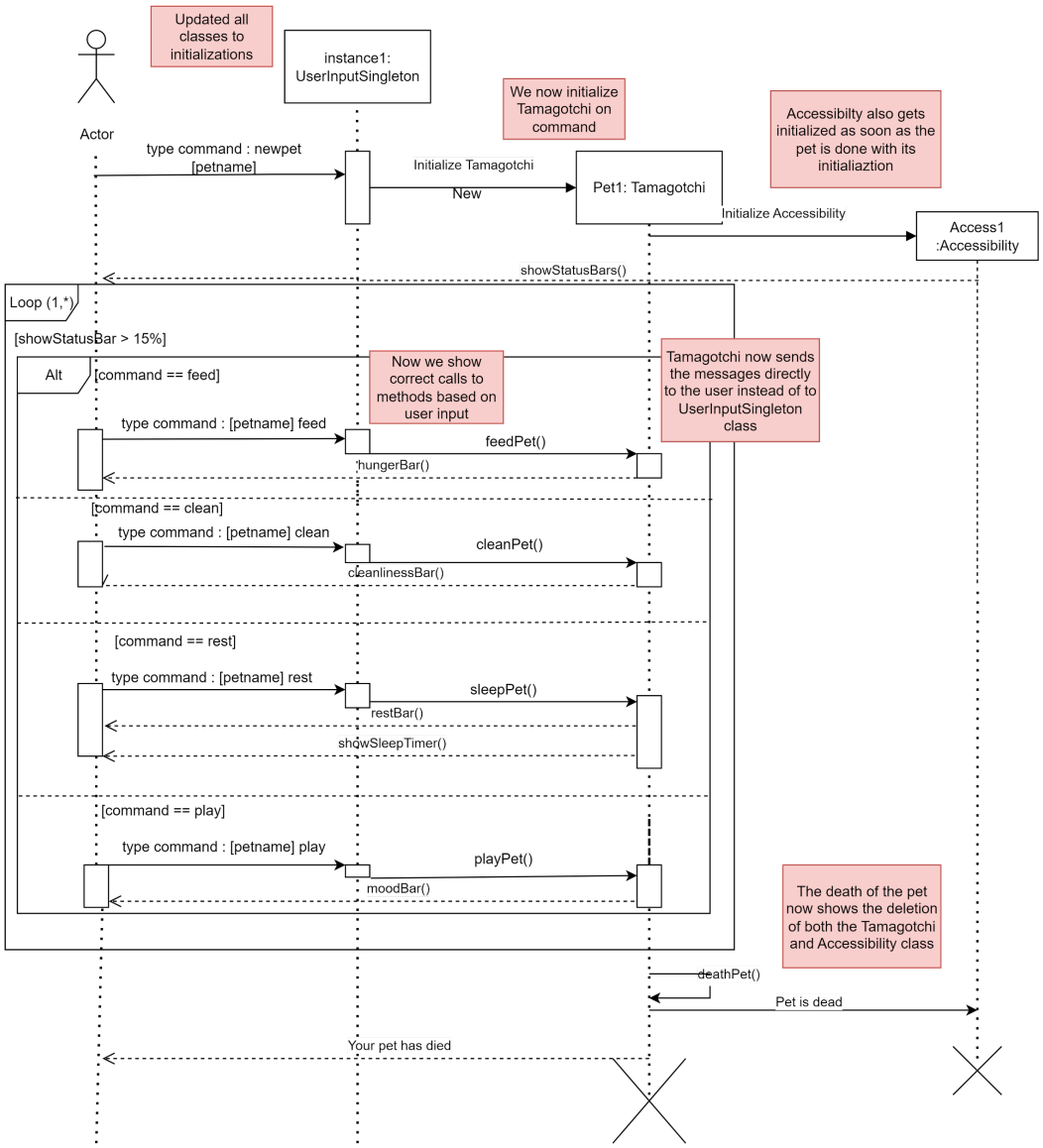
Drastically reduced the usage of the **UserInput** class, now the **minigame** does all the interaction with the **user** on his own. This makes the code on our side more maintainable. We changed the class name from **Minigame** to **RockPaperScissors** since this was an error in the previous diagram. Added instance names as not doing so was also an error in our previous diagram. The loop can now go an infinite amount of times since the game will not count ties as rounds played. Tracking the score of the round is now done inside the *CalculateOutcome* method. The reasoning behind this is that it only took 2 lines to update the variables and creating a method for it seemed unnecessary. Finally, the **Minigame** will delete itself after the game has ended as no variable will be needed again after the game is done.

User Interaction while creating a User Profile



We now initialize the **UserProfile** when a new account gets created. Otherwise, we look at existing initializations to login. We have removed the synchronous messages as this was not intended. We have named the initializations instead of just giving the class name. This clears up the mistakes we made in the previous diagram. The changes we have made are that the **UserInput** class now only acts as an initiator and not as a message exchanger. This makes our code more maintainable. We have also added all the messages that get sent from the **UserProfile** to make the diagram more detailed.

User Interaction with Tamagotchi



We have made multiple changes to this diagram. Firstly we'll go over the many errors that were fixed. The names at the top were at first only classes, this was an error and so we added names to change them to initializations. We also show the initialization of the **Tamagotchi** and **Accessibility** instead of having them initialized beforehand which was not how we wanted to convey our diagram. The *showStatusBars* now gets sent from **Accessibility** now instead of from **Tamagotchi** which was also another mistake. We have also made sure to get the initialization out of the loop which was yet another mistake as it would make no sense to keep initializing the **Tamagotchi** class each time we wanted to call a method. Finally, we put an alt fragment to show that the user gets to choose which method he wants to call. In our previous diagram, we were calling all of the methods in a specific order regardless of user input which was a mistake.

Implementation:

Author(s): Turan

Strategy:

The main strategy that was used was to look at all the UML models and implement the system in a way that sticks to the models that we made. Since the models were made in a way to ensure that the implementation was taken into consideration, it was not very difficult to transition from the UML models to the implementation of the system.

The class diagram served as a very strong base for the implementation of the system. In our opinion, it really is the basis of not only implementing the system but also making other UML diagrams. Hence, the goal of implementing the system was to make sure it reflects the class diagram as much as possible, and the other models would be reflected as a result because they were also heavily dependent on the class diagram.

The class diagram had each class and class type well sorted. We knew what classes were abstract and how the relationships between each class worked. Therefore, building the basic structure of the system was pretty easy. We knew what methods and attributes we wanted to use because of the class diagram as well. When building the class diagrams, we had all the implementation techniques we wanted to use for the actual implementation. Hence, it was easy to refer to the models.

The design patterns resulted in a few changes. Here, the class diagrams and implementation went hand-in-hand to ensure that the patterns were correctly reflected.

In the end, after the entire system was built, the implementation was checked with each UML model to ensure that it stayed consistent throughout.

Key Solutions:

- The UserInput class worked as the basis of the entire system. We wanted to make sure that the main class does not get crowded with too many functions (which was the original plan). Hence, we had a different class called UserInput that dealt with the user commands and was connected to the key parts of the system.

- All the inherited properties and abstract classes were very clearly defined. In fact, it was essential to follow all the UML models which is exactly what we did. Since the relationships were so well-defined it was easy to implement the system.

- The implementation of the design patterns to the specific classes was extremely important. An important aspect was to make the system as dynamic as possible. For example, not only is it easy to detach a minigame from the system and add another, one can add multiple minigames to the system.

- The access to the Tamagotchi was pretty well divided with the Pet, Tamagotchi, and Accessibility classes respectively. This was an important solution in implementing the Tamagotchi system and keeping it separate from the rest of the system.

Location of the main Java class needed for executing the system in the source code:
SoftwareDesign_63/src/main/java/**softwaredesign**//Main.java

Location of the Jar file for directly executing the system:
SoftwareDesign_63\out\artifacts\software_design_vu_2020_jar\software-design-vu-2020.jar

30-seconds YouTube video:
https://youtu.be/M6j1EySLBMM

Time logs:

Team number	63		
Member	**Activity**	**Week number**	**Hours**
Turan Yigit	Application of design patterns	8	2
	Object diagram	7	2
	Sequence diagram	7	2
	Implementation	6	6
Kanushree Jaiswal	Summary of changes	8	1
	Application of design patterns	7	2
	Class diagram	7	3
Wissal	State machine diagrams	8	2
	Summary of changes	8	1
Shadman Sahil Chowdhury	State machine diagrams	8	2
	Summary of changes	8	1