Assignment 3

Team number: 20
Team members

Name	Student Nr.	Email
Andrés Lot Camarena	2724030	a.lotcamarena@student.vu.nl
Stefan Dragosh Vatamanu	2726157	s.vatamanu@student.vu.nl
Marcel Niedzielski	2726454	m.niedzielski@student.vu.nl
Simona Peychinova	2724076	s.peychinova@student.vu.nl

Class
objects
attributes

operations()

associations

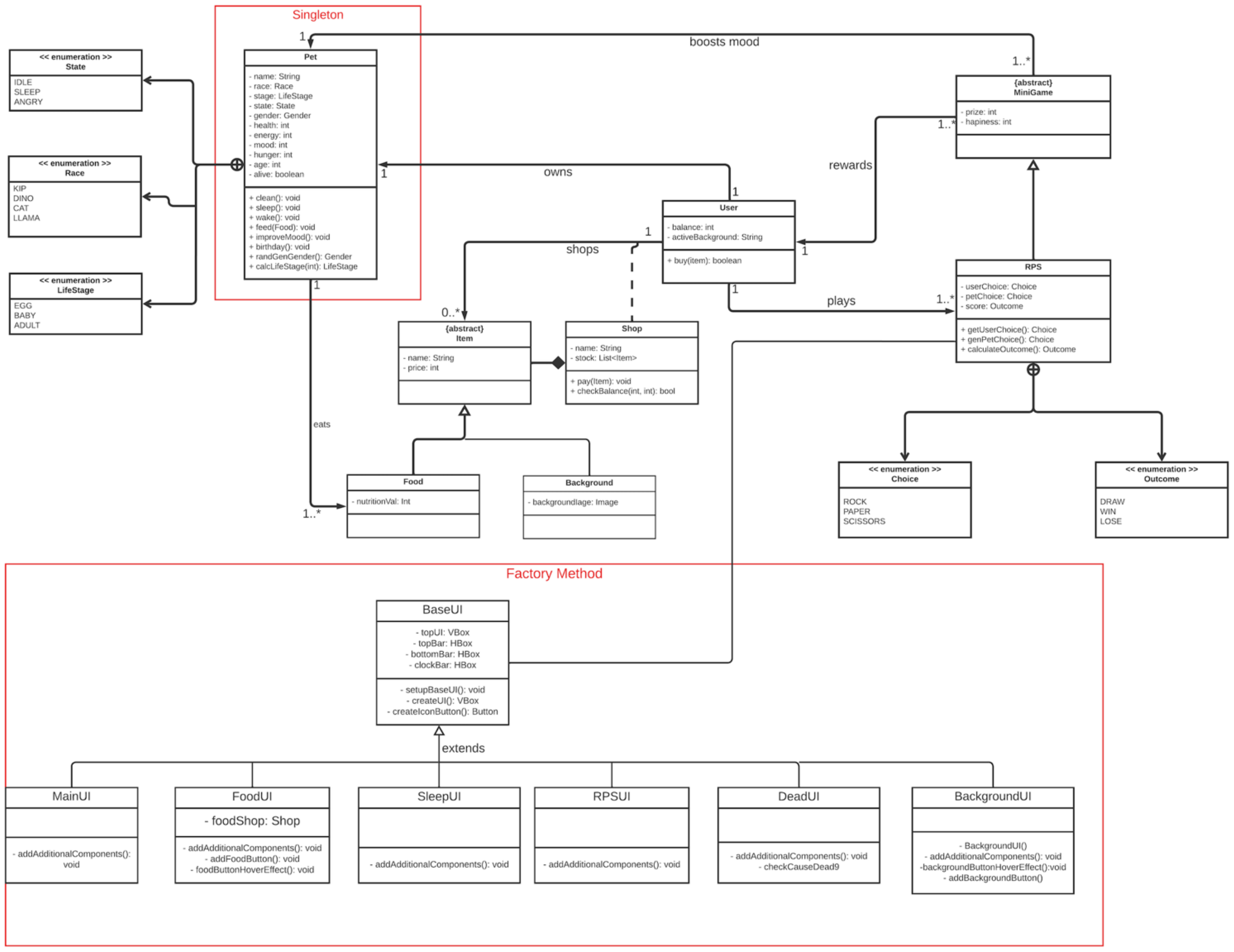
ENUMERATIONS

Summary of changes from Assignment 2

Here is a summary of the changes made to address feedback in Assignment 2:

- Class diagram:
  - We removed play() from the **User** because it is a very complex function.
  - We added new class **BaseUI** with subclasses - **MainUI, FoodUI, SleepUI, RPSUI, DeadUi, BackgroundUI.**
- State machine diagram:
  - We have only 4 states: IDLE, SLEEP, ANGRY and DEAD. Our game state is saving through button. If the player exits the game without saving it by clicking on the button, the game state is not saved and then the next time the player opens the game they will have to start from the beginning.
- Sequence diagram
  - In assignment 2 our first sequence diagram for Process of buying food from Shop was not completely correct as the description did not match the modelled diagram. We changed this and now it is clearly showed that the process is go to the shop, buy food, pet eats and not and "go to the store check the nutrition value of the food and then buy it" as stated in the feedback.

Revised class diagram



With red it is highlighted where the two design patterns are implemented. The first one is Singleton, used for the pet. With this design pattern we ensure that there is only one instance of the **Pet** class present throughout the entire game. The second design pattern is Factory design patter used for UI. It helps us to create easier to moderate code, without duplicating it.

Pet

The **Pet** class represents a Tamagotchi virtual pet with various attributes such as name, health, energy, etc. The most relevant attributes are state, race, and stage, as they are enumeration types that classify the pet's current phase. One of the most relevant operations is birthday(), which is triggered after a fixed amount of time adds up 1 to the age of the pet and boosts its mood as well. Other operations such as clean(), sleep(), play(), and feed() are used to interact with the pet and maintain its overall well-being.

State

The **State** enumeration represents the current state of the pet. It has three values: IDLE, DEAD, SLEEP, and ANGRY.

Race

The **Race** enumeration represents the breed of the pet. It has five values: KIP, DINO, CAT, and LAMA.

LifeStage
The **LifeStage** enumeration represents the life stage of the pet. It has three values: EGG, BABY, and ADULT.

Item
The **Item** class is a superclass that represents the buyable items in the game. It has two attributes: name, and price.
The **Item** class has a composition relationship with the **Shop**, as the **Shop** class contains a list of **Item** objects.

Food
The **Food** class is a subclass of **Item,** which represents the food that the pet can eat. It has one unique attribute nutritionVal, which is an int representing the nutritional value of the food. The **Food** class also has one method, consume(), which is called when the user buys the food.
The class has a many-to-one association with the Pet class, as one pet can eats one or more foods.

Background
The **Background** class is a subclass of **Item** represents the background image of the game. It has one attribute, bgImage, which is an Image representing the background image of the game.

Shop
The **Shop** class is an association class that makes possible for a user to buy an item. It has an attribute called stock which is a list of Item objects that the shop has for sale. The **Shop** has a method called buy() which allows the user to purchase an item from the shop by passing the item as an argument. The **Shop** class relates to the **Item** class through composition, which means that the **Shop** is composed by the items that are for sale. The **User** class is also connected with the **Shop** class, as the user can buy items through a **Shop** object. Overall, the **Shop** class plays a critical role in the game, allowing the user to feed their pet and customize the environment.

MiniGame
The **MiniGame** class is an abstract class that represents mini-games that can be played in the game. It has two attributes, prize and happiness, which represent the prize awarded to the player upon winning the mini-game and the effect of the mini-game on the pet's mood. The class is related to the **User** and **Pet** classes through many-to-one associations, which represent the fact that one or more mini games can reward one user and boost the mood of one pet.

RPS
The **RPS** class represents a Rock-Paper-Scissors mini-game that can be played in the game. It has three attributes, userChoice, petChoice, and score, which represent the user's choice, the pet's choice, and the outcome of the game, respectively. The calcScore() operation is used to calculate the outcome of the game based on the user's and pet's choices.

The **RPS** class is associated with the **Choice** and **Outcome** enumerations, which represent the different options and outcomes of the game.

BaseUI

With a relation to the **RPS** class is the **BaseUI** class. Its method createUI() takes two VBox parameters, topUi and bottomBar, and creates a new VBox with those elements, which is returned. The setupBaseUI() method sets up the basic UI by creating and styling the topUI, topBar, and bottomBar, and adding them to the VBox that represents the **BaseUI**.

This class has 6 subclasses: **MainUI, FoodUI, SleepUI, RPSUI, DeadUi, BackgroundUI**. Each one of them has its own methods. In **FoodUI** class the foodButtonHoverEffect() function is used to give the food buttons a hover effect. The top bar of the user interface is removed when the mouse cursor is placed over a food button. In **BackgoundUI** class the backgroundButtonHoverEffect() method defines a hover effect for a Button that displays information about a background item in the background shop. When the mouse pointer enters the area of the Button, the background of the game scene changes to the background image of the associated Background object.

Choice
The **Choice** class is an enumeration type that represents the possible choices that a user or pet can make in the Rock-Paper-Scissors (RPS) mini-game. The choices are ROCK, PAPER, and SCISSORS. They are used as arguments to the RPS class's calcScore method, which calculates the outcome of the RPS game.

Outcome
The **Outcome** enumeration type represents the possible outcomes of the Rock-Paper-Scissors (RPS) mini-game regarding the user. The possible outcomes are W, D, and L.
This enumeration is used in calcScore method to represent the outcome of a game. The method takes two instances of the **Choice** enumeration as parameters. Based on these, the calcScore method returns an instance of the **Outcome** enumeration, representing the outcome of the game.

User
The **User** class represents the game user. It has one attribute - balance which represents the user's balance of in-game currency. The class is related to **Pet** through a one-to-one association, which represents the fact that one user can own one pet.

Application of design patterns

	DP1
Design pattern	Singleton Design Pattern
Problem	We need to make sure that there is only one version of the pet class running at any moment in the game. Without this design pattern, we might experience problems with resource usage, interaction, and data consistency.

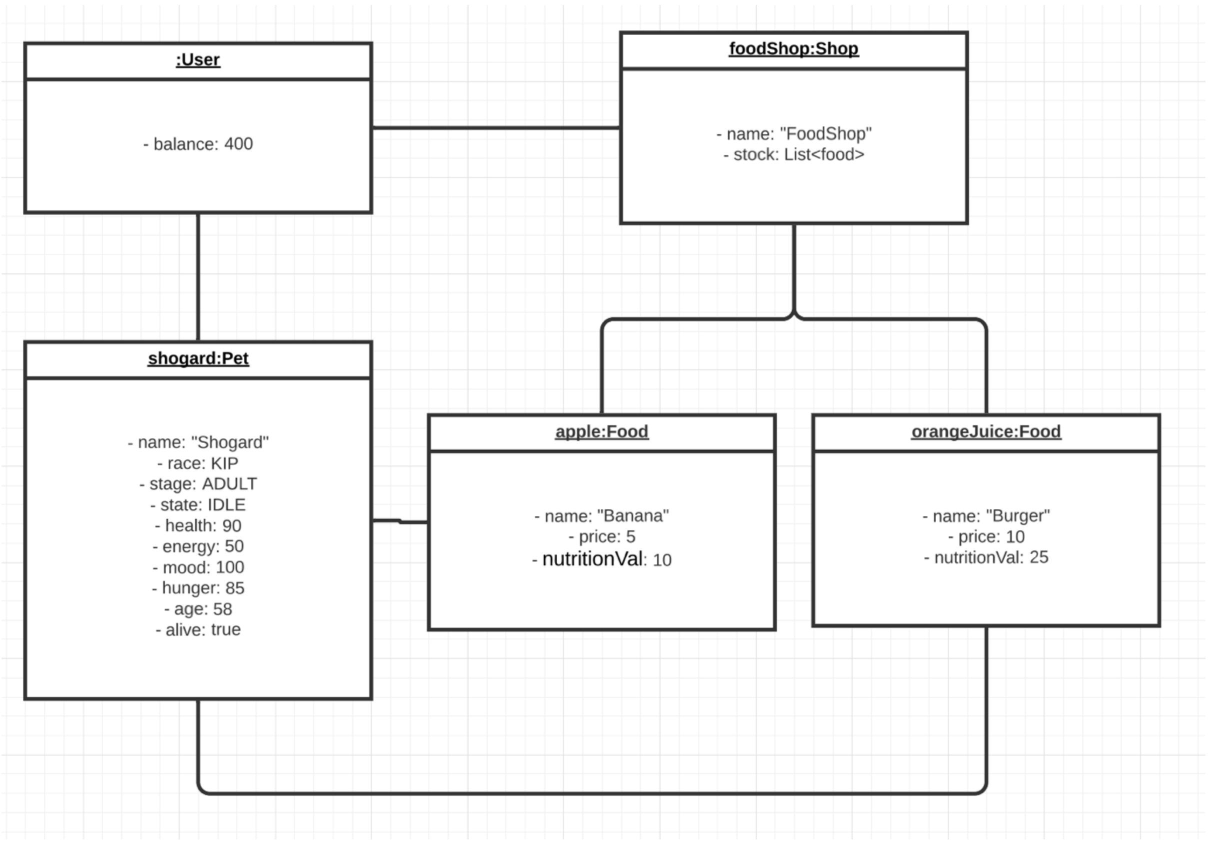
Solution	By implementing this design pattern, we ensure that there is only one instance of the **Pet** class present throughout the entire game. In this way the data is always accurate and there are no synchronization problems when using different examples of the pet class. Moreover, we optimize resource usage by ensuring that we only create an instance of the **Pet** class when it is required.
Intended use	At run-time, we use the Singleton design pattern to make sure that only one instance of the **Pet** class exists at any given time. This is important for preserving the pet's attributes and state throughout the game. The getInstance() function of the **Pet** class will always be called whenever the user needs to interact with the pet. This method ensures that only one instance of the **Pet** class is created and returns that instance whenever it is called. Then the user is able to manipulate the attributes and behaviour of the pet object through different methods available in the **Pet** class, such as sleep(), wake(), improveMood(), and feed().
Constraints	The Singleton design pattern has several drawbacks, one of which is that it can limit the "flexibility" of the code. For instance, it may be challenging to increase the capability of the Pet class without modifying the Singleton class. Furthermore, the Singleton design may introduce hidden dependencies between different parts of the code, making it difficult to test.

	DP2
Design pattern	Factory Design Pattern
Problem	We have different UI for different features, such as food and background. We want to make this UIs more organise and ensure that we do not duplicate code because it can be time-consuming and easy to make mistakes.
Solution	We created an **UIFactory** class. In the **UiFactory** class, we create different types of UI objects based on the specific requirements of each feature. This

	class acts as a central location for creating different types of UIs, and it offers a clear way to instantiate objects. Moreover, in this way the UI code is easier to update.
Intended use	At runtime, the code calls the getUI() method of the **UIFacory** class and pass the wanted UI type as an argument. Based on the input, the getUI() method then creates and returns an instance of the requested UI type.
Constraints	One of the main constrains of the Factory design pattern is that it can make the code more complicated. The reason for this is because requires having a separate factory class.

Revised object diagram

We did not change anything in out object diagram as it is an exact replica of our system. Below is the description.



The UML object diagram represents a snapshot of a moment in the Tamagotchi game where a *user* is buying **Food** objects from the *cafeteria* to feed() their *pet,* therefore filling its hunger attribute. The diagram is composed of four objects: *user, cafeteria, banana,* and *burgere*, and one **Pet** object, *shogard.*

User has one attribute named balance which shows the amount of money the user owns for buying items. The *foodShop* is an object of **Shop**, with two attributes: stock, which is a list of available food items, and name. The stock is represented by a list of **Food** objects, which in this case, includes *banana* and *burger*.

Banana and *burger* have three attributes: name, price and nutritionVal. NutritionVal is the attribute that shows how much the hunger will be filled after the pet is fed up.
Finally, the **Pet** object, *shogard*, has numerous attributes including stage, state, health, alive, etc.

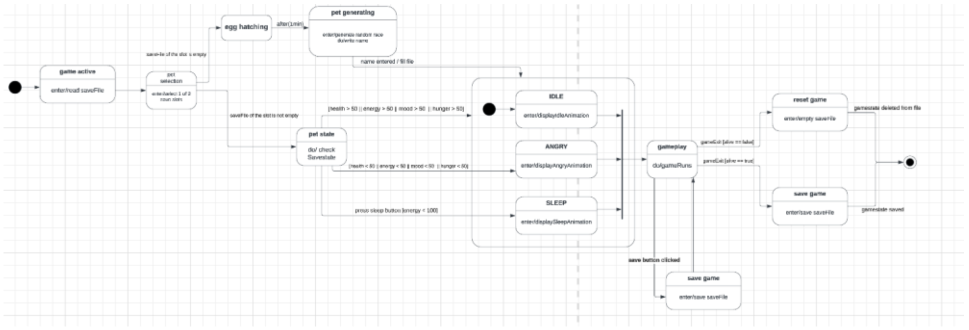
The diagram illustrates how the objects are connected. *User* owns the pet and is also connected to *cafeteria* with a line that indicates a purchase is taking place. The *cafeteria* is then connected to *banana* and *burger*to represent the items that are available for purchase. Finally, *Shogard* is connected to the **Food** objects with a line that represents the feeding process.

Revised state machine diagrams

This chapter contains a revised version of the same UML state machines you modelled in Assignment 2 (with all changes highlighted graphically), together with a textual description of all main improvements and the reasoning behind them.

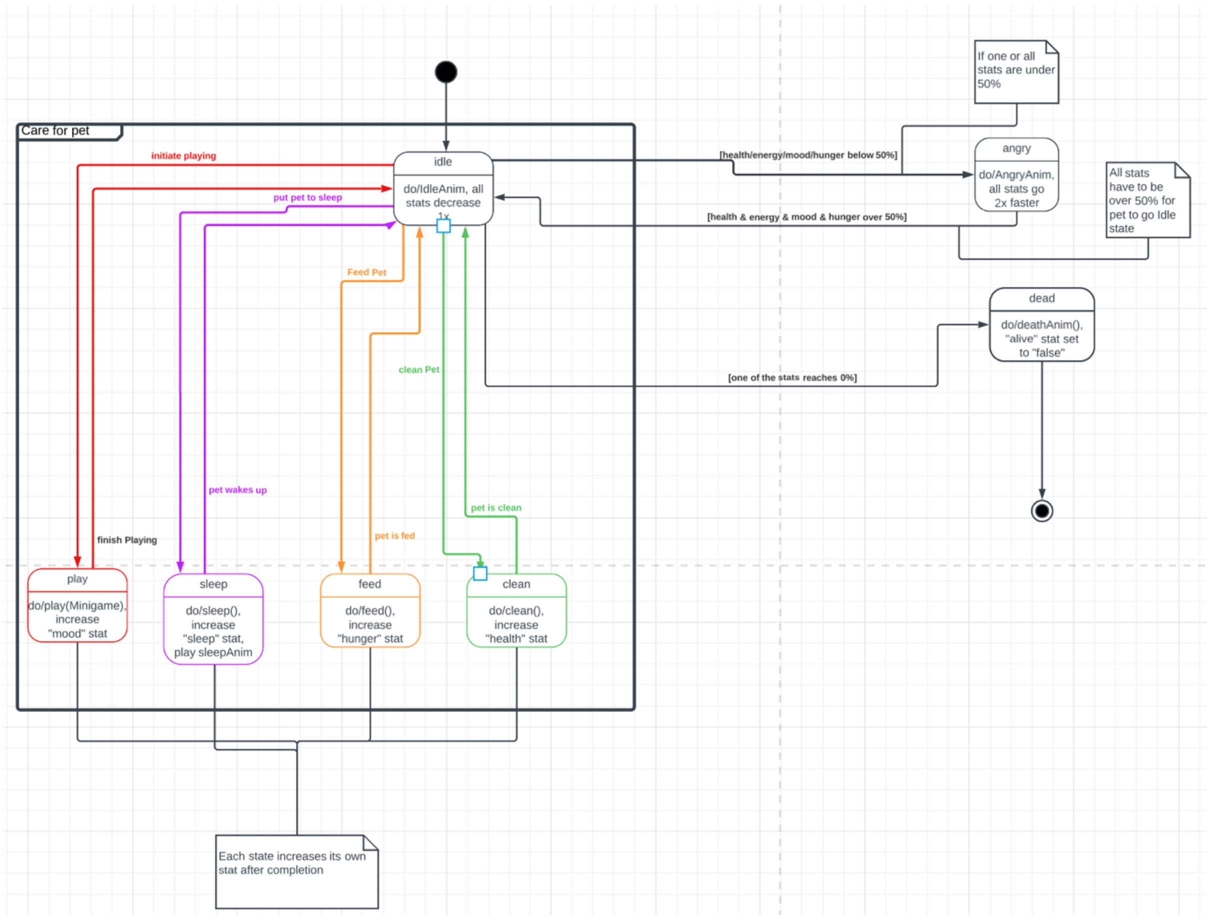
Maximum number of pages for this section: 3

First state machine diagram: Saving game state
(Zoom in to see the details)


game active
enter/read saveFile
pet selection
enter/select 1 of 3 save slots
saveFile of the slot is empty
egg hatching
after(1min)
pet generating
enter/generate random race
do/write name
name entered / fill file
saveFile of the slot is not empty
pet state
do/ check Savestate
[health > 50 || energy > 50 || mood > 50 || hunger > 50]
[health < 50 || energy < 50 || mood < 50 || hunger < 50]
press sleep button [energy < 100]
IDLE
enter/displayIdleAnimation
ANGRY
enter/displayAngryAnimation
SLEEP
enter/displaySleepAnimation
gameplay
do/gameRuns
gameExit [alive == false]
gameExit [alive == true]
reset game
enter/empty saveFile
gamestate deleted from file
save game
enter/save saveFile
gamestate saved
save button clicked
save game
enter/save saveFile

This diagram is a representation of how loading and saving the game state would work. After starting the game, 3 save slots are displayed. After selecting one, it's saveFile is loaded, and checked for content. If it does not contain a saved state, that starts the process of creating a new pet. The egg is hatched, the race and sex are randomly generated, and the user is asked for the desired name of his pet. After the name is entered, the pet enters the idle state, and the actual gameplay starts. If the file contains a saved state after loading, the state of the pet in which the game was exited is checked and triggered, then the gameplay starts. If the user wishes to exit the game, the isAlive boolean is checked. If evaluates to false, the saveFile is emptied and the game exits. If the user does not click the save button and exits the game the saveFile is also emptied. When the user clicks on the save button and the pet isAlive evaluates to true, the saveFile is updated with current stats of the pet. After that the game is exited.

Second state machine diagram: Pet state



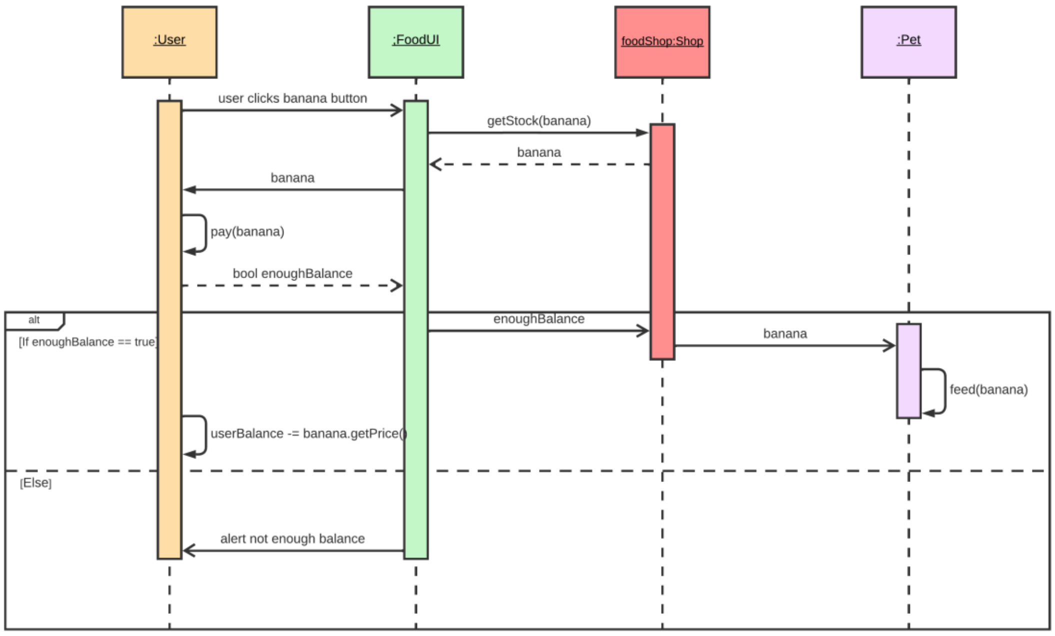
Care for pet
idle
do/IdleAnim, all stats decrease 1x
initiate playing
put pet to sleep
Feed Pet
clean Pet
[health/energy/mood/hunger below 50%]
[health & energy & mood & hunger over 50%]
[one of the stats reaches 0%]
If one or all stats are under 50%
angry
do/AngryAnim, all stats go 2x faster
All stats have to be over 50% for pet to go Idle state
dead
do/deathAnim(), "alive" stat set to "false"
pet wakes up
finish Playing
pet is fed
pet is clean
play
do/play(Minigame), increase "mood" stat
sleep
do/sleep(), increase "sleep" stat, play sleepAnim
feed
do/feed(), increase "hunger" stat
clean
do/clean(), increase "health" stat
Each state increases its own stat after completion

The Pet State diagram represents different states and transitions that will occur within the life of the pet. A pet has 5 states, idle, angry, sleep, and dead. By default, the pet is in his idle state, where the "idleAnim" animation runs. In this state, every stat degrades at the normal rate. If any of the stats goes under 50%, the pet transitions to the angry state, where it will run the "angryAnim" animation, and each state will start degrading twice as fast. It remains in this state until all stats are above 50%.

The rest of the states are triggered by user actions. The pet will enter sleep state when the pet is put to sleep. If any of the stats drops to 0%, the pet will die, transition into Dead State and the user is forced to start over with a new pet.

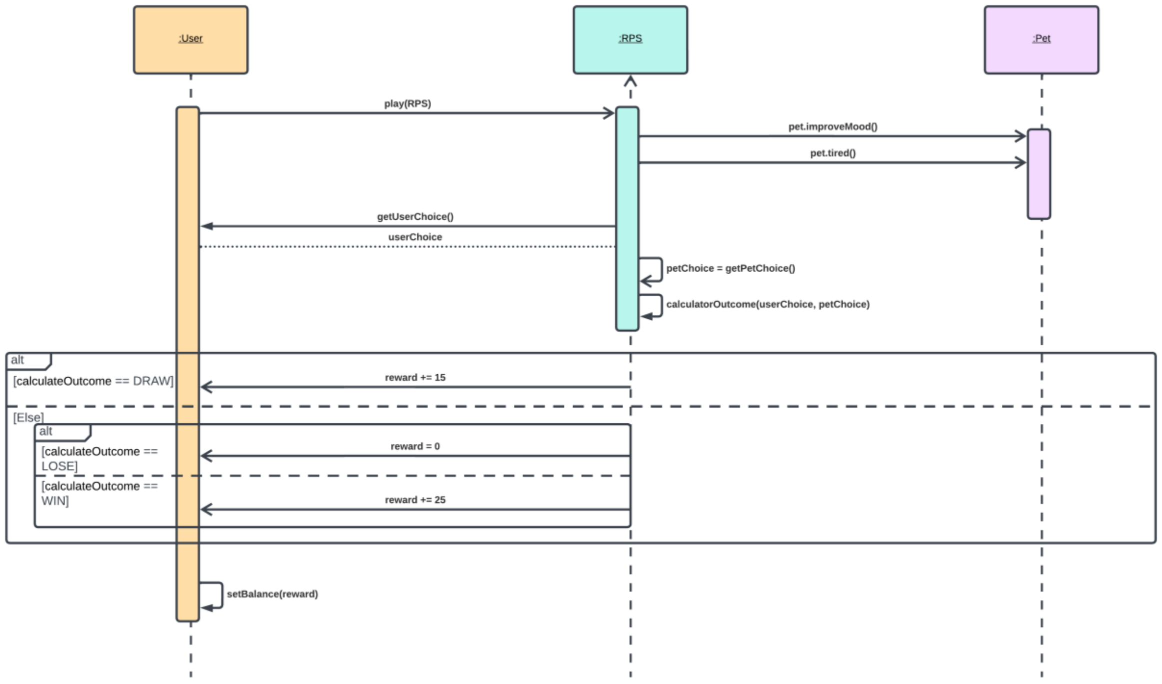
Revised sequence diagrams

First sequence diagram: Process of buying food from Shop



This sequence diagram represents the process when the user buys food to feed the pet. First, the user, clicks a food button (in this case *banana*). After this is checked if the user has enough balance to buy the banana. If yes, the price of the item is subtracted from the user balance and the pet is feed. If the user does not have enough *balance, they receive a proper error message.*

Second sequence diagram: RPS game


:User
:RPS
:Pet
play(RPS)
pet.improveMood()
pet.tired()
getUserChoice()
userChoice
petChoice = getPetChoice()
calculatorOutcome(userChoice, petChoice)
alt
[calculateOutcome == DRAW]
reward += 15
[Else]
alt
[calculateOutcome == LOSE]
reward = 0
[calculateOutcome == WIN]
reward += 25
setBalance(reward)

This sequence diagram shows the process of playing a Rock-Paper-Scissors mini game with the pet. User decides to play the mini game and pet gets the happiness value, no matter the outcome of the game. Then RPS object asks for user input for the mini game and randomly generates one for pet. After having these 2 values it can calculate the outcome. Depending on the outcome of the match the user gets rewarded respectively. If it draws it will get 15 in-game currency added to balance, if he loses, he would not get any money and if he wins, he will get 25in-game currency.

Implementation

In this chapter you will describe the following aspects of your project:

- the strategy that you followed when moving from the UML models to the implementation code.
- the key solutions that you applied when implementing your system (for example, how you implemented the movement of an NPC in a videogame);
- the location of the main Java class needed for executing your system in your source code.
- the location of the Jar file for directly executing your system;
- the 30-seconds video showing the execution of your system (you are encouraged to put the video on YouTube and just link it here).
-

IMPORTANT: remember that your implementation must be consistent with your UML models. Also, your implementation must run without the need of any other external software or tool. Failing to meet this requirement means 0 points for the implementation part of your project.

Maximum number of pages for this section: 4

The strategy we used for moving the UML models we used went as follows: First we implemented all the classes we had in our class diagram. From there we implemented the interactions between them one by one. Starting with creating the **UI**, then **Pet** and **User**. After having those 3 we were able to implement the other interactions.

The system was implemented using the Java FXGL library. The game is based on changing UIs, which have different sets of buttons which allow the user to act. The buttons either open new UIs (for example sleep button opens sleep UI), which have different sets of buttons or allow the user to interact with the pet (shower him, or pick move in the minigame), or get back to the **mainUI**. Once the pet is initialized, it has all the stats are full, and after every 6(for testing purposes) seconds they decrease. If one of the stats falls below 50, the pet gets angry, and the stats start decreasing faster. The goal of the game is to take care of the pet so none of its stats fall to 0. If that happens, the pet dies and the game ends. The user is allowed to quit the game at any time. If the pet isn't in the dead state, user is able to save progress by clicking save button and upon closing and opening the game, the pet from the last run is loaded. Throughout the game the user earns money, which allows him to buy food and backgrounds. In case of a death of the pet, the balance of the user is saved, however the pet will be deleted.

The UIs were implemented using VBoxes and HBoxes which were included in the mentioned library. Each of the UIs has a set of buttons that allow the user to take care of the pet, as well as change backgrounds, play the minigame and save the game (buttons are the only way to take actions).
The class B**aseUI** contains the parts of the code shared between all the UIs. It is extended by specific UI classes, which add all the necessary buttons for the particular UI. This reduces the repetition of code in each of the UIs. Here the Factory Method came in handy for creating the instances of each of the UIs. After passing the name of the UI, the UIFactory returned the newly created instance of the UI. Factory Method was also used for returning

The Pet was implemented using a Singleton design pattern, which allowed its global instance to be passed around the code, without the risk of creating more than one. The same was done for the User class, as it only has one global instance throughout the whole game.

Location of the main Java class needed for executing the system:
src/main/java/softwaredesign/Main.java

Jar file:
out/artifacts/software_design_vu_2020_jar/software-design-vu-2020.jar

Link to the video:
https://youtu.be/QUHlNFsR5oA

Time logs

Team Number: 20			
Member	**Activity**	**Week number**	**Hours**
All team members	Discuss the flow of the project	1	4
Lot	Animations	1 and 2	7
All team members	Draw animations	1	4
Lot, Dragosh, Marcel	UI	1	11
Lot	Background images	1	2
Dragosh	Food class	1	3
Marcel and Dragosh	Item class	1	2
Dragosh	Mini game	1	3
Lot	Pet class	1	7
Marcel	Shop class	2	4
Lot	Sigleton design pattern	2	2
Marcel	Factory design pattern	2	3
Simona	Application of design patterns (in doc)	2	2
Simona	Reversed diagrams	2	7
All team members	Implementation (in doc)	2	3
Simona	Format documment	2	1
		Total:	61