Assignment 3

Team number: 77
Team members

Name	Student Nr.	Email
Yaxin Li	*2781906*	*y.li16@student.vu.nl*
Seoyeon Kim	*2796797*	*s.y.kim2@student.vu.nl*
Hyeonjee Kim	*2796710*	*h.j.kim2@student.vu.nl*
Benjamin de Vries	*2624258*	*b.d.de.vries@student.vu.nl* *Ahn-jaemin@protonmail.com(GitHub)*

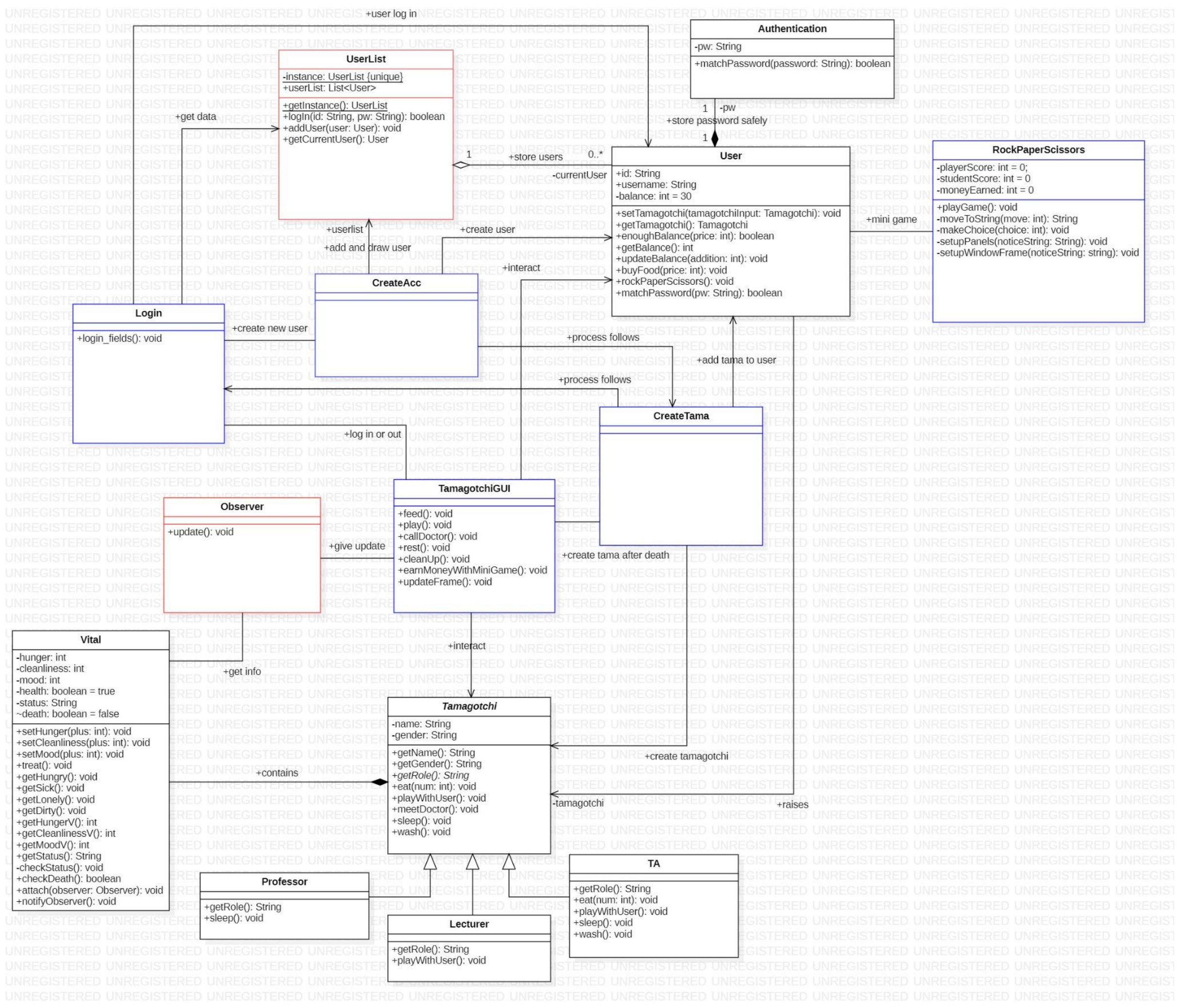
Summary of changes from Assignment 2

Author(s): Hyeonjee Kim, Seoyeon Kim, Yaxin Li

- Reflecting the feedback of Assignment 2, the functions and attributes mentioned in the class diagram and the actual implementation are used in the state machine diagram. By this revision, it will be clearer for the readers to understand how the whole program works and has more consistency with the class diagram. Also I added one starting state and final state for the whole diagram.
- For the state machine diagram, since there were changes in the actual implementation and the sleeping process got much briefer, I changed the second state machine diagram to depict the process of changes in the vitals. The second diagram is about the eating process, and the changes in vitals - as time passes, the vital worsens for the tamagotchi.
- For the first sequence diagram,"play with tamagotchi", we added min/max number of iterations in the "loop" fragment, deletion to the tamagotchiGUI object and changed the name as class/object name.
- With regards to the object diagram, the public/private visibilities have been corrected according to the class diagram, and the UserInterface has been connected to Authentication. All other connections have also been checked and corrected. Finally, the diagram has been updated according to the change of structure we made when actually implementing the program, as opposed to only planning it in UML diagrams.
- In the class diagram, some logical errors and ambiguous expressions like authentication process and creating account process which existed in the former version disappeared. The access modifiers are corrected and justified specifically, and the differentiation between three different tamagotchis is implemented and drawn.

Revised class diagram

Author(s): Hyeonjee Kim



There are some critical changes between assignment 2 and assignment 3. Some of the classes are newly created and the major structure of the interface is totally changed. Also, some details about access modifiers are corrected.

The overall structure consists of User, Authentication, UserList, Tamagotchi, Lecturer, Professor, TA, Vital, Login, CreatAcc, CreatTama, TamagotchiGUI, RockPaperScissors, and Observer.

Two red classes are places where design patterns are applied. Blue classes are classes that extend JFrame, which are totally different from last class diagram.

First of all, there are some classes that have not changed a lot. Player is replaced by **User**, which works almost the same. It has id, username, balance as attributes. Because the balance became private, no other class could modify the balance easily. Also, the tamagotchi attribute became private, too. Thus, User has getBalance() and getTamagotchi() methods to replace direct calling of attributes. User class contains setTamagotchi(Tamagotchi tamagotchi) for raising new tamagotchi, enoughBalance() and buyFood(int price) to feed tamagotchi, rockPaperScissors() to interact with RockPaperScissors class to earn money. Additionally, matchPassword(String pw) is newly created to complete login using the private attribute pw, which is an Authentication object. This change was intended to make password matching not be accessed directly from UI or other irrelevant classes, making the login process secure.

Authentication part has no change.

UserList replaced PlayerCollection. Designing UserList class contains a 'Singleton' design pattern applied. By creating a static instance internally and making the constructor private, an UserList object can be created only once. Then the unique object is accessed by getInstance(). This prevents the user list being scattered all over the place. There is a more detailed description below in 'Application of design patterns'. To help login process and store user data, login(String id, String pw), getCurrentUser(), and addUser(User user) are used.

Tamagotchi has changed a little. Age and color attributes are deleted, and name and gender attributes changed privately so getName() and getGender() are required. Tamagotchi class is abstract and getRole() method is abstract, which would be concrete form in inheriting class.

Professor, Lecturer, and TA classes inherit the Tamagotchi abstract class and implement the getRole() method which returns each role. Differentiation among them is implemented by making their vital change differently.

Vital has some important changes. Status attribute is added, which is used by showing the tamagotchi's status on the GUI page. Status, Hunger, cleanliness, mood, and health became private so that other classes can access those by the getter methods(getStatus, getHungerV, getCleanlinessV, getMoodV). Vital attributes can be changed better by treating methods(setHunger(int plus), setCleanliness(int plus), setMood(int plus), treat()), which are executed by corresponding Tamagotchi methods. As time passes by, Vital attributes become worse by methods(getHungry(), getSick(), getLonely(), getDirty()), which are executed by TamagotchiGUI indirectly. The status of vital is continuously checked by checkStatus(), which is executed by TamagotchiGUI repeatedly. Additionally, to notify an update of the vital, attach(Observer observer) and notifyObserver() methods are used. These would be discussed later.

Now, there are newly created and totally changed classes below. Because we started to implement 'My Tamagotchi' with a real interface, we needed to use JFrame. Therefore, Login, CreateAcc, CreateTama, and TamagotchiGUI became JFrame classes. Because there are so many specific attributes that are not so important like JPanel and JButton in JFrame classes, I didn't add those attributes in my class diagram. Also, JFrame classes work mainly in the constructor, not including many methods. So you can see those JFrame classes have almost empty bodies in this diagram. However, I will explain the connections with other classes, which are just shown as association arcs.

Login is not technically a JFrame class, but the only method is login_fields(), which creates JFrame. In the method, the UserList class is used to get data of the list of users, and User class is used to make appropriate user log in. Otherwise, there is an option of creating a new account, which leads to the construction of CreateAcc class. If the login process is completed normally, the construction of the Tamagotchi class is held.

CreateAcc extends JFrame. To create a new account, User class is used to create new users and UserList class is used to add a new user into the list of users. After creating a new account, CreateTama class construction occurs in progress. If the user wants to go back, Login class construction is held again.

CreateTama extends JFrame. To create a new Tamagotchi for a user, Tamagotchi class is used to create new tamagotchis and User class is used to add tamagotchi to a particular user. After creating an account and tamagotchi, TamagotchiGUI class construction follows. Moreover, if a tamagotchi dies, CreateTama restarts and leads to TamagotchiGUI again.

TamagotchiGUI extends JFrame, and this is the main class where the commands of the player take place. If a user clicks a button of command, the corresponding method(feed(), play(), callDoctor(), rest(), cleanUp(), earnMoneyWithMiniGame()) is executed and affect other classes like User and Tamagotchi.

RockPaperScissors extends JFrame. This class is for earning money by playing a mini game, rock paper scissors. If the playGame() is executed by rockPaperScissors() in User, the panel for the User interface appears by setupPanels(String noticeString) and setupWindowFrame(String noticeString). If a user makes a choice by 'makeChoice(int choice)', this is reflected in the game and the money is earned. The amount of money earned is added to User's balance by updateBalance(int addition) of User.

Observer is not a JFrame class, but it is an additional class by applying design pattern 'Observer'. Using this class, the TamagotchiGUI can get continuous updates from the change of Vital. Observer has vital and tamagotchiGUI instances as attributes, so it can make associations between them by using update(). This process would be explained more specifically below.

Application of design patterns

Author(s): Hyeonjee Kim

1. Applying Singleton design pattern to make UserList object unique

	DP1
Design pattern	Singleton (creational design pattern)
Problem	When I decided to build an interface using JFrame, we designed the program progressing with the Login class, Createacc class, Createtama class, and TamagotchiGUI class using JFrame. However, as the program runs, the classes extending JFrame need to be opened and closed many

	times. This causes a problem of constructing so many objects in a short time. Especially, the Login class and Createacc class are designed to import UserList class objects to load the data list of users or add a new user to the user list. However, if Login class and Createacc class are created a lot of time and keep constructing UserList instances, there will be too many objects, which will scatter the user's information all over the places and make the list data meaningless.
Solution	Singleton design pattern is to make a class have the only instance, which is just what we needed. In the UserList class, we made the constructor private so that any other class cannot access to create a new instance of UserList. Instead, we made a private static instance of UserList internally in UserList class. The instance was intended to be one and only instance of UserList. To give access to the instance to other classes, a public static method 'getInstance()' was created. By this application of Singleton design, the UserList static object was uniquely created and could be accessed by every other class objects like Login object and Createacc object, which are created repeatedly during the process like login. The single object of UserList became the only dataset, which is meaningful by saving everything without missing anything.
Intended use	As a program begins, a Login class object is created and this imports the single instance of UserList to get the list of users(the instance of UserList is created regardless of importing timing). Also, as a Createacc class object is created to create a new account, the object imports the instance of UserList again to add a new user to the list. As the program proceeds, the login process and creating new account process are repeated and the Login objects and Createacc objects are constructed many times. However, because the instance is unique and static, every object uses the same 'getInstance()' method and accesses the identical instance of UserList. Thus, the identical instance stores users and gets new users' information so that the storage of data is protected without missing.
Constraints	This design pattern worked perfectly with our software, so we didn't end up with anything that wasn't designed as intended.
Additional remarks	

2. Applying Observer design pattern to keep tracking of Vital of Tamagotchi

	DP2
Design pattern	Observer (behavioural design pattern)
Problem	In the Tamagotchi game, Tamagotchi's vitality keeps changing and the player should take care of the Tamagotchi to make their vitality better. This is the main idea of this game.

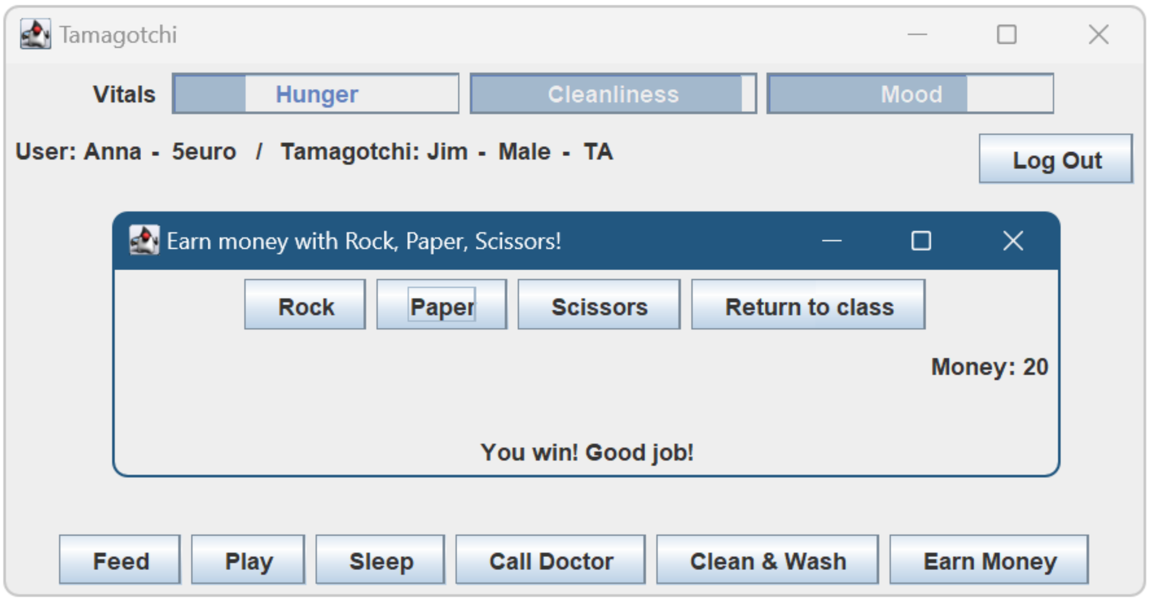
	To implement this property into a program, the vitality of Tamagotchi should announce their changes to other classes which would take the vital information into account. However, it was hard for us to apply that property to our program. Because we designed Tamagotchi class to have a Vital object as an attribute(Vital class and Tamagotchi class is associated with composition association), there wasn't any way to notify the Tamagotchi class or TamagotchiGUI class that the Vital object has been changed. Connecting TamagotchiGUI and Vital directly by making an attribute of TamagotchiGUI in Vital class to create a function to notify the changes didn't seem to make sense.
Solution	Observer design pattern connects two different class objects and makes one object know the other object's change by the notification of the changed object. We made an Observer class, which contains Vital object and TamagotchiGUI object as attributes. By constructing Observer instance with TamagotchiGUI object and Vital object input during the process of TamagotchiGUI method, Observer executes the 'attach(Observer)' method of Vital, which is created to put the observer instance into the vital's attributes. Then, the Vital object can access the observer instance to notify its changes. Vital class contains 'notifyObserver()', Observer class contains 'update()', and TamagotchiGUI class contains 'updateFrame()' so that those methods interact with each other. Consequently, the applying observer to this program solved the problem of notifying changes by connecting TamagotchiGUI and Vital indirectly.
Intended use	As a TamagotchiGUI object is constructed, a new observer object is constructed and stored as an attribute of TamagotchiGUI. At the same time, Observer object stores TamagotchiGUI object and Vital object by constructor and Vital object stores Observer object by the 'attach' method call of Observer construction. Then, if there is any change in Vital, the vital object executes 'notifyObserver()' and the 'update()' method in Observer is called. The 'update()' method calls a method of the TamagotchiGUI object, which is 'updateFrame()'. Then the TamagotchiGUI updates their value sets by getting new values of Vitals via Tamagotchi class.
Constraints	It would have been better if the observer was divided into specific vital change. If so, 'updateFrame' method wouldn't have had to update the unchanged values. However, it was so complicated to implement that because so many classes are required.
Additional remarks	

Revised object diagram

Author(s): Benjamin
Pages: 1 page, omitting images

The object diagram has been updated according to the feedback given on the previous assignments (correct public/private attributes and correct object connections), and the change in structure that came about whilst creating the program.

A snapshot of the system was taken to model an object diagram. At this time during the operation of the program, an account and two tamagotchis have been created, and the game has started. After some time of playing, a snapshot of the system is taken, as shown below:



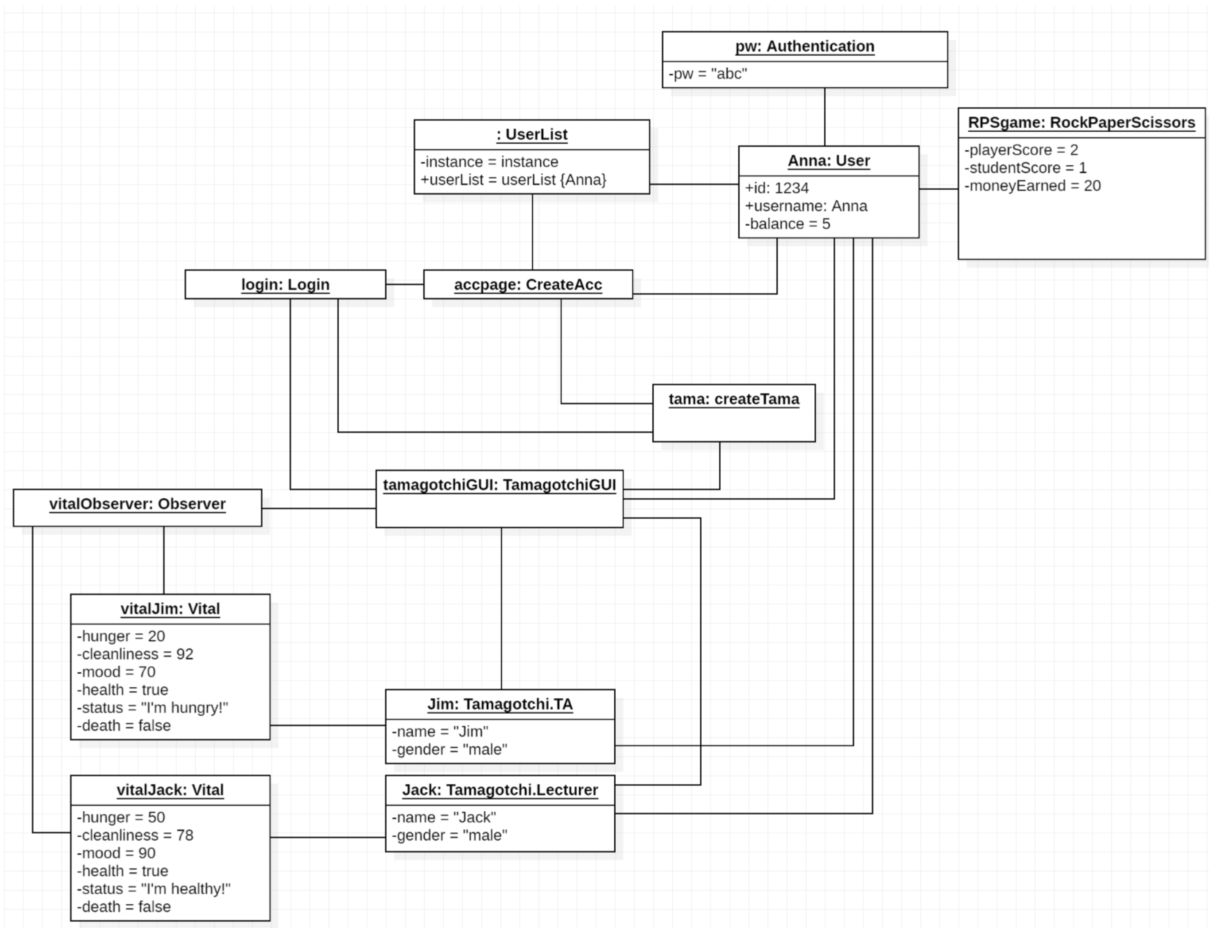
User: Anna - 5euro / Tamagotchi: Jim - Male - TA

Money: 20

You win! Good job!

Some of the information of the objects can be known from the graphical user interface: the user's name is Anna, the current tamagotchi's name is Jim, and he is a male Teaching Assistant. The vitals are shown at the top: because of his teaching work, he has become quite hungry, and his mood is starting to drop. The user doesn't have a lot of balance—only €5—so she decides to let her tamagotchi Jim play some rock, paper, scissors with a student to earn some money. Until now, Jim has earned €20 from successive rounds of playing the game. When the game is finished, the earnings will be added to Anna's balance.

Now let's take a detailed look at the state of the relevant objects at this moment, by studying the object diagram at this moment:



As usual in an object diagram, methods are not shown because of redundancy, so we can focus on the relevant data. Following our observations from the GUI and our development through the game, we can see that a singleton UserList has been created with one instance and a userList called userList with one user: “Anna”. Anna’s username is, of course, “Anna”, and her balance is 5. A password is stored in the Authentication class instantiated as “pw”, with the password itself stored under the attribute pw as “abc”. All the other classes for supporting the tamagotchi creation process and the GUI have been instantiated.

At the bottom we see the current tamagotchi: his name is “Jim” and his gender is “male”. He is a Tamagotchi of the subtype: “TA”. His vitals are connected on the left through the object “vital”. His vitals from the GUI are reflected here. We can also see that another tamagotchi has been created with the name “Jack”. He is of the Tamagotchi subtype Lecturer, and his gender is also “male”. Finally, on the right is the RockPaperScissors game. The current playerScore is 2, the studentScore is 1, and Jim’s moneyEarned is 20.

This reflects all the relevant information for this moment in the game (redundant names are omitted): the graphical user interface (TamagotchiGUI), the classes needed for creating an account (accPage: CreateAcc, UserList), creating a new tamagotchi (tama: createTama), and logging in (Login), the user herself (Anna: User), the tamagotchis (Jim: Tamagotchi.TA, Jack: Tamagotchi.Lecturer), their vitals (vitalJim: Vital, vitalJack: Vital), the observer for updating the GUI (vitalObserver: Observer), and finally the minigame (RPSgame: RockPaperScissors).

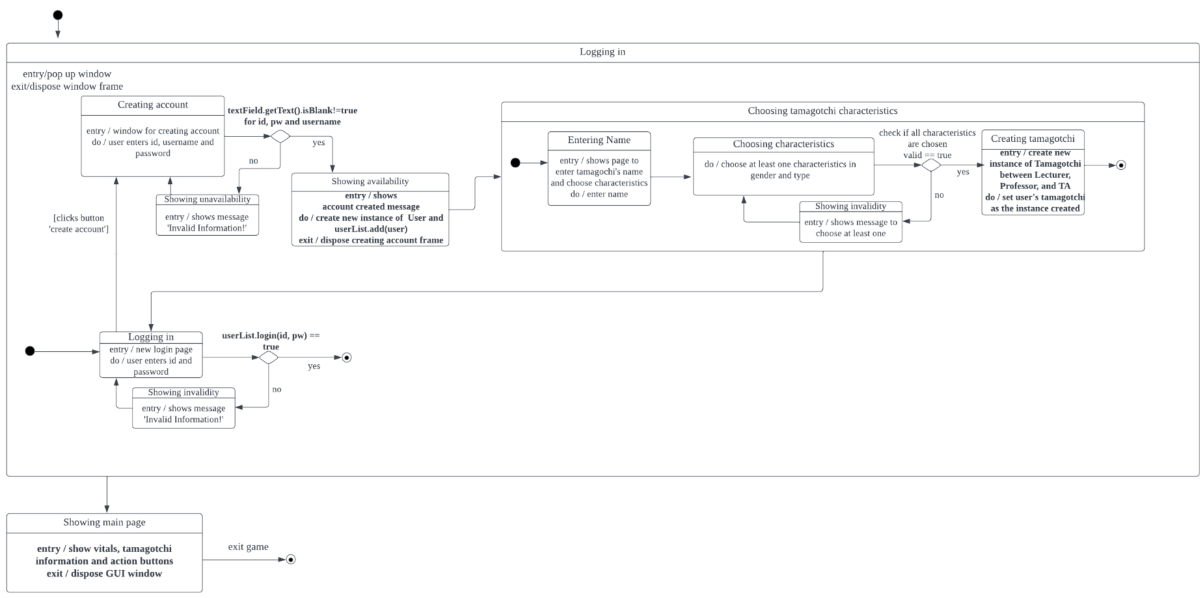
The class and object diagrams reveal that we could have used more singleton classes, because some classes are only ever used once, and that we need to define the structure better, because now there are too many connections between all the classes. Through these assignments we have learned about the value and utility of UML diagrams, such as the fact that the final product can be very different from its initial conception.

Maximum number of pages for this section: 1

Revised state machine diagrams

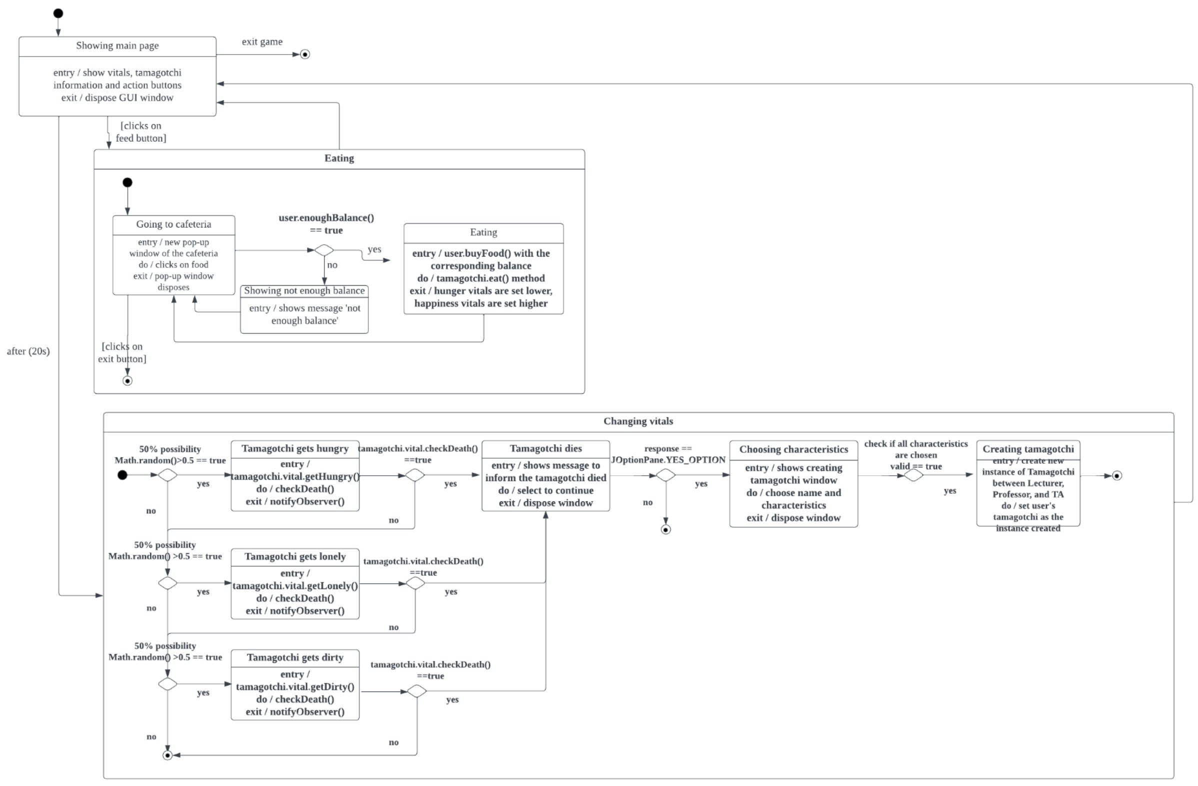
Author(s): Seoyeon Kim

The first state machine diagram is about the logging in and creating the Tamagotchi part. First, it goes to the Logging in state. As it enters the state, a new window is generated and the new login page is created. If the user doesn't have an account and clicks on the button 'create account', a new window for creating the account will appear. The user enters the id, username and password and if the textField is blank, a message showing the unavailability will be displayed. If it is available, it goes to the 'Showing availability' state, where it shows the account created message. A new instance of the User is created and is added to the userList by the method userLIst.add(user). The creating account frame is disposed and it goes to the creating Tamagotchi page. The user enters the name and chooses the characteristics - if they are not all chosen or written, it shows message to choose at least one. If it is valid, a tamagotchi is created based on which type the user has chosen. A new instance - Lecturer, Professor or TA is created and it is set as the user's tamagotchi. It goes back to the logging in state, where there is a new login page. When the user enters the id and password, it checks its validity with userList.login(id, pw) method. If it is valid, the 'Logging in' state terminates and goes to the main page. If it isn't, it shows an error message.


Logging in
entry/pop up window
exit/dispose window frame
Creating account
entry / window for creating account
do / user enters id, username and password
textField.getText().isBlank!=true for id, pw and username
yes
no
Showing unavailability
entry / shows message 'Invalid Information!'
Showing availability
entry / shows account created message
do / create new instance of User and userList.add(user)
exit / dispose creating account frame
[clicks button 'create account']
Choosing tamagotchi characteristics
Entering Name
entry / shows page to enter tamagochi's name and choose characteristics
do / enter name
Choosing characteristics
do / choose at least one characteristics in gender and type
check if all characteristics are chosen valid == true
yes
no
Showing invalidity
entry / shows message to choose at least one
Creating tamagotchi
entry / create new instance of Tamagotchi between Lecturer, Professor, and TA
do / set user's tamagotchi as the instance created
Logging in
entry / new login page
do / user enters id and password
userList.login(id, pw) == true
yes
no
Showing invalidity
entry / shows message 'Invalid Information!'
Showing main page
entry / show vitals, tamagotchi information and action buttons
exit / dispose GUI window
exit game

The second state machine is about the feeding process. The starting point of this diagram is the main page. When the user clicks on ‘feed’ button, it goes to the eating state. First, a new pop-up window is displayed and there are three choices of food and the balance the user need to get the food. The user clicks on the food and if the user.enoughBalance() is true, the user.buyFood() method is executed with the corresponding amount of balance. Then the tamagotchi.eat() method is executed, and then the hunger vitals go lower and the happiness vitals get higher. If there isn’t enough balance, it goes back to the cafeteria state and shows the message ‘not enough balance’. If the user clicks to exit the cafeteria, it goes back to the main page.

Also, the program is designed to make the vitals change with time. For instance, as time passes, the tamagotchi will get hungrier, dirtier and lonelier if you don’t play, feed, or wash them. To make this happen, we made the vitals change every 20 seconds, but not always. When 20 seconds has passed in the main page, it goes to the ‘Changing Vitals’ state. First to start with, with 50% possibility using the Math.random() method the Tamagotchi gets hungry. If it gets hungry, the tamagotchi.vital.getHungry() method is executed and the death of the tamagotchi is checked. Then the observer is notified with the notifyObserver() method to make the changes in the vitals. In the TamagotchiGUI, the vitals are checked again and if tamagotchi.vital.checkDeath() is true, the tamagotchi dies and a message showing that the tamagotchi died is shown. The user can select to continue, or to abort the program. If the response is YES to continue, it goes back to creating a new tamagotchi by entering the name and choosing their characteristics. If the chosen characteristics are valid, a new tamagotchi instance is created and the user’s tamagotchi is set as the new one.
If the tamagotchi didn’t get hungry, or if it got hungry but didn’t die it goes to another state where in 50% possibility the tamagotchi gets lonely. The same thing happens in this method, where the tamagotchi.vital.getLonely() is executed. Repeating it again, if the tamagotchi didn’t get lonely, or if it got lonely but didn’t die it goes to another state where in 50% possibility the tamagotchi gets dirty. After this, it goes back to the main page. Every 20 seconds this process is repeated.


Showing main page
entry / show vitals, tamagotchi information and action buttons
exit / dispose GUI window
exit game
[clicks on feed button]
after (20s)
Eating
Going to cafeteria
entry / new pop-up window of the cafeteria
do / clicks on food
exit / pop-up window disposes
user.enoughBalance() == true
yes
no
Eating
entry / user.buyFood() with the corresponding balance
do / tamagotchi.eat() method
exit / hunger vitals are set lower, happiness vitals are set higher
Showing not enough balance
entry / shows message 'not enough balance'
[clicks on exit button]
Changing vitals
50% possibility Math.random()>0.5 == true
Tamagotchi gets hungry
entry / tamagotchi.vital.getHungry()
do / checkDeath()
exit / notifyObserver()
tamagotchi.vital.checkDeath() ==true
Tamagotchi dies
entry / shows message to inform the tamagotchi died
do / select to continue
exit / dispose window
response == JOptionPane.YES_OPTION
Choosing characteristics
entry / shows creating tamagotchi window
do / choose name and characteristics
exit / dispose window
check if all characteristics are chosen valid == true
Creating tamagotchi
entry / create new instance of Tamagotchi between Lecturer, Professor, and TA
do / set user's tamagotchi as the instance created
50% possibility Math.random() >0.5 == true
Tamagotchi gets lonely
entry / tamagotchi.vital.getLonely()
do / checkDeath()
exit / notifyObserver()
tamagotchi.vital.checkDeath() ==true
50% possibility Math.random() >0.5 == true
Tamagotchi gets dirty
entry / tamagotchi.vital.getDirty()
do / checkDeath()
exit / notifyObserver()
tamagotchi.vital.checkDeath() ==true
yes
no

Revised sequence diagrams

Author(s): Yaxin Li

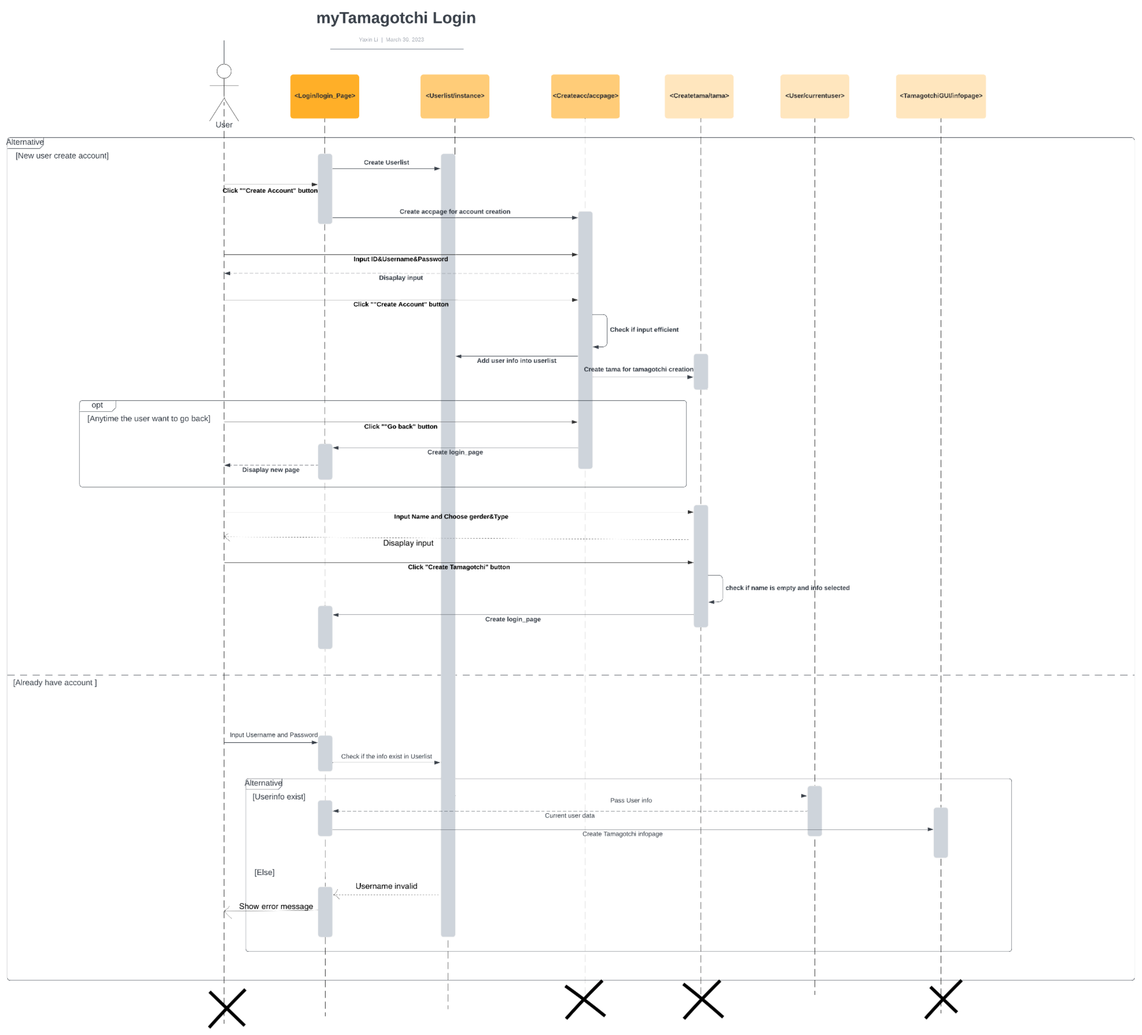

myTamagotchi Login
Yaxin Li | March 30, 2023
User
<Login/login_Page>
<Userlist/instance>
<Createacc/accpage>
<Createtama/tama>
<User/currentuser>
<TamagotchiGUI/infopage>
Alternative
[New user create account]
Create Userlist
Click ""Create Account" button
Create accpage for account creation
Input ID&Username&Password
Disaplay input
Click ""Create Account" button
Check if input efficient
Add user info into userlist
Create tama for tamagotchi creation
opt
[Anytime the user want to go back]
Click ""Go back" button
Create login_page
Disaplay new page
Input Name and Choose gerder&Type
Disaplay input
Click "Create Tamagotchi" button
check if name is empty and info selected
Create login_page
[Already have account]
Input Username and Password
Check if the info exist in Userlist
Alternative
[Userinfo exist]
Pass User info
Current user data
Create Tamagotchi infopage
[Else]
Username invalid
Show error message

Figure: myTamagotchi Login

This diagram about myTamagotchi Login consists of two main parts: one for creating a new account and the other for logging in. The login process is actually very simple, and we have described it clearly in our previous assignments. It has almost no changes. The main part that requires changes is the account creation process, as it requires more user input. In order to make the interface clear and concise, we divided this process into two separate windows, namely "<**Createacc**/*accpage*>" and "<**Createtama**/*tama*>"to complete the process.

When you first start using our tamagotchi, <**Login**/*login_page*> will message <**Userlist**/*instance*> to prepare to save the user's data. As a new user you should begin by clicking "Create Account" on the login screen. <**Login**/*login_page*> dispose of its old frame after creating a new <**Createacc**/*accpage*>. In other words, the user is switched to a new page to fill in more detailed information for creating an account. In this new interface, you will need to input your ID, username, and password, then click “Create account”. If all information is valid, you can proceed to the next step. Otherwise, an error message will be displayed in a pop-up window, and you can continue the operation on the original page after closing the pop-up window and re-entering the information. It is worth noting that our program has a validity check for all input, so it is not possible to proceed to the next step by submitting blank information. As this point is explicitly indicated in the sequence diagram, it will not be further elaborated on later. <**Createacc**/*accpage*> will send valid data to <**Userlist**/*instance*> to create a new object <**Createtama**/*tama*>, and then dispose of itself.

As the under opt element shows, you can return to the login screen by a simple click on the “Go back” button at any time during this process. In the <**Createtama**/*tama*>’s new page, similar to the previous procedure, you will need to input your Tamagotchi name, choose the gender and type for him/her, then click “Create account”. You will be returned to <**Login**/*login_page*> to login when it succeeds.

When the user has an account already and logging in, *The user* should enter the username and password in <**Login**/*login_page*>, that means, a message including the user name and password is passed to <**Login**/*login_page*>, and then <**Login**/*login_page*> transmits this information to <**Userlist**/*instance*> for retrieval. Different from previously described, we added <**User**/*currentuser*> to preserve user’s personal data, enhancing the security of the data.

If the user's information exists in the database, in addition to the user name and password, the information in <**User**/*currentuser*> should also cover the saved player's game information. In this case, <**User**/*currentuser*> will return the information of the game to <**Login**/*login_page*>, and then use it to create <**TamagotchiGUI**/*infopage*>. In the meantime, the user will be redirected to it. At this point, the user can start playing.

In another case, <**Userlist**/*instance*> returns an error message, the user stays in <**Login**/*login_page*> with an option to create an account from here.

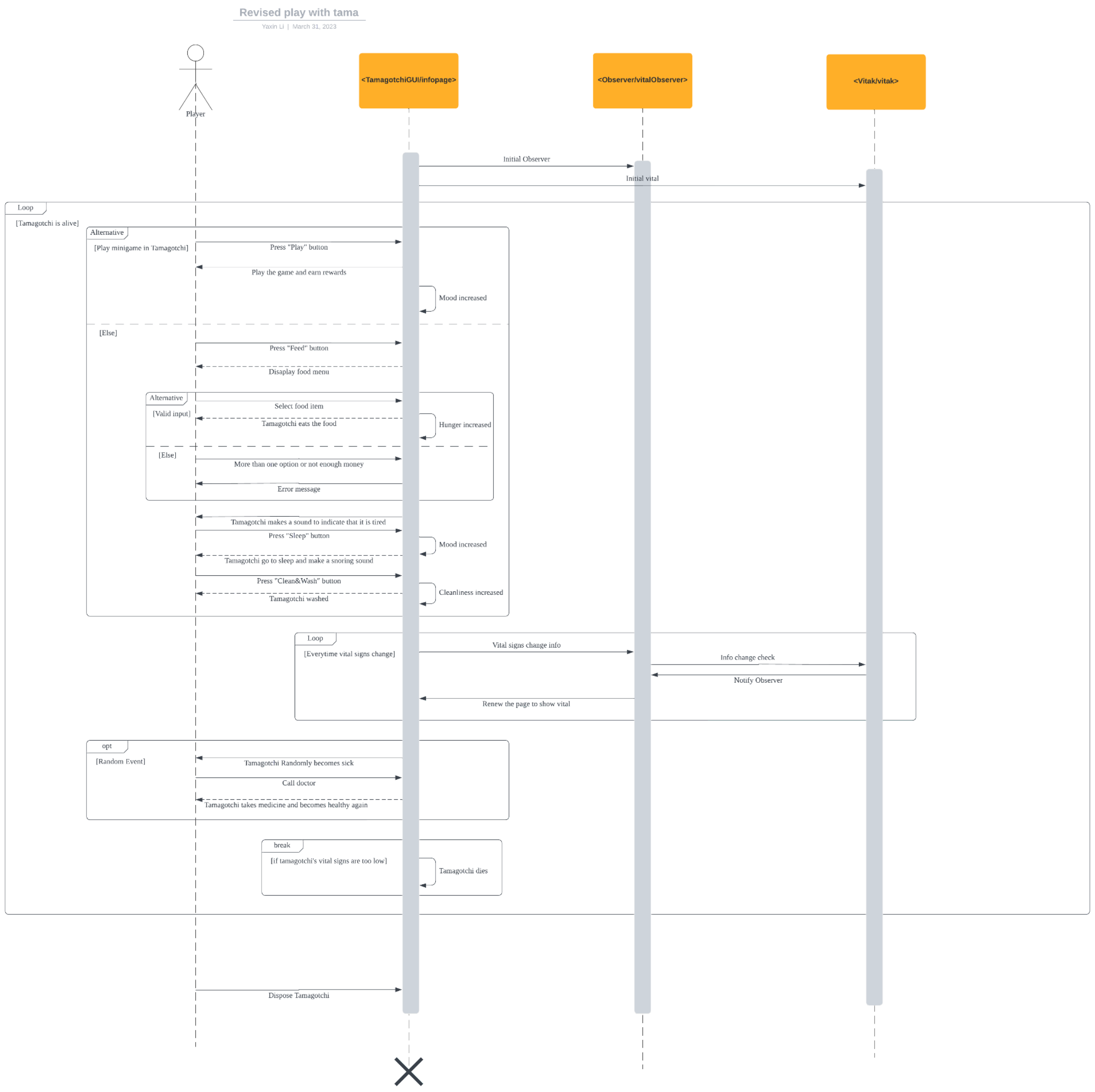


Figure: Revised play with Tama

There haven't been any significant changes to the process of *Player*-<**TamagotchiGUI**/*infopage*> interaction for playing the game, but added <**Observer**/*vitalobserver*> and <**Vital**/*vital*> according the user's actions in the game to control various vital signs' changes. These objects are created together with the <**TamagotchiGUI**/*infopage*> and need to run continuously as long as the Tamagotchi is alive. Once the Tamagotchi dies and its vital signs are reset to zero, these classes are no longer needed.

Implementation

Author(s):Seoyeon Kim, Hyeonjee Kim

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.

1. Login and Creation of account
   As the program is executed, the new Login() page is created. Each window is made with JFrame, with the title, size, and layout set visibly. We made each panel and added them to the frame. The login window consists of the id and password textfield, and a button for creating the account and logging in. Each texts and fields are made with JLabel, JTextField and JButton. If the user enters after putting in the id and the password, first it checks if the fields are not blank, and shows the error message if they are. Then, it checks with the userList.logIn() method to check if the id and password is valid. If not, it shows the error message - if it is valid, the frame is disposed and the new TamagotchiGUI() page is created with the variable currentUser.
   If the user doesn't have an account and clicks on the create account button, the frame is disposed and the new Createacc() page is created. In the Createacc() class, a new frame is created with three textfields for id, username and password. If the user clicks on the create account button, first it checks validity and if it is, it creates a new instance of user and adds the user to the userList. The frame is disposed and the Createtama() page is created.
   In the Createtama() class, a new frame is created with one textfield for the tamagotchi's name, a checkbox for selecting the gender, and a checkbox for selecting the type(Lecturer, Professor, TA). If the user clicks on the button and the the choices are valid, a new instance is created based on the selected type and the user.setTamagotchi() method is used to link the user and the tamagotchi 1:1. The frame is disposed and it goes to the login page.

2. Vitals
   Above the main page, the vitals are shown : hunger, cleanliness and mood. Each are drawn with a JProgressBar from the scale of 0 to 100, and set with the value of tamagotchi.vital.getHungerV(), getCleanlinessV(), and getMoodV(). When the tamagotchi is created, the vitals are set to 70, 20, 20. In the Vital class, it is able to increase or decrease the vitals, get the value of vitals, check the status, and to check the death.
   In the TamagotchiGUI() class, there is a timer that counts every 20 seconds to execute the code. This part is implemented with the Timer() and TimerTask() class.

Every 20 seconds, first, it is checked whether the tamagotchi is not dead - if not, it randomly worsens the vitals. By 50% possibility each, it gets hungrier, dirtier and lonlier. By 30% possibility, it gets sick. When this happens, it goes to the tamagotchi.vital.getHungry(), getLonely(), getDirty() and getSick() method to worsen the vital. After changing the vitals, the death of the tamagotchi and the status is checked, and the changes are notified to the observer to change the vitals in the JProgressBar.
If the vitals were checked and the tamagotchi has came to a death, which means its hunger vitals are bigger than 90, or its mood or cleanliness vitals are lower than 10. In this case the timer first stops with the timer.cancel() function, and it shows a pop-up message to inform that the tamagotchi died and asks the user to continue with a new tamagotchi or exit the game. If the user wants to exit, the program terminates. If the user wants to continue, the frame is disposed and it goes to the new Createtama() page. After creating a new instance of the tamagotchi, it is set as a new tamagotchi of the user and this time, it goes back to the TamagotchiGUI page right after.

3. User, Authentication and User list
   User is the most fundamental component in this software. It contains id and pw for login, and also consists of username, balance, and its own tamagotchi. By constructor User(String id, String username, String password) which Createacc class executed, id and username are stored directly and password is stored as an Authentication format. With an interaction with Createtama class, ‘setTamagotchi’ method is called and new tamagotchi is stored as an attribute. To pass a tamagotchi object to TamagotchiGUI, ‘getTamagotchi’ method is used. TamagotchiGUI’s ‘feed’ method works with User balance by using ‘enoughBalance’ and ‘buyFood’ method to lower the tamagotchi’s hunger level. To show the user’s balance which is a private attribute on the TamagotchiGUI main page, ‘getBalance()’ is used. By executing ‘rockPaperScissors()’ to earn money, the RockPaperScissors class is activated and the game starts.
   The Authentication class has a password string delivered, and the string is saved in a private attribute. Since the Authentication() is called when the account is created, the string contains the password of the user account. The class has a method named matchPassword, which figures if the password that the user entered equals the actual password.
   The User list is a list of the User instances, made with a new ArrayList. It has a few methods for the log in process. The method logIn() receives two strings - the id and password the user has put into - and for each user in the userList, it checks if there is a user with the corresponding id and password. If it is a valid id and password, it returns the true value. Other methods like addUser() and getCurrentUser() adds a user to the list or returns the current user that is logged in.

4. Command buttons in TamagotchiGUI(Feed, Rest, …)
   In the main page(TamagotchiGUI class) there are several buttons : feed, play, sleep, call doctor, clean&wash, earn money. If the user clicks on the ‘feed’ button, a new frame ‘Choose Food’ is created and three checkboxes with the food and needed balance appears. This is implemented with the JCheckbox and has a button underneath to confirm the choice. If only one is selected, the user.enoughBalance()

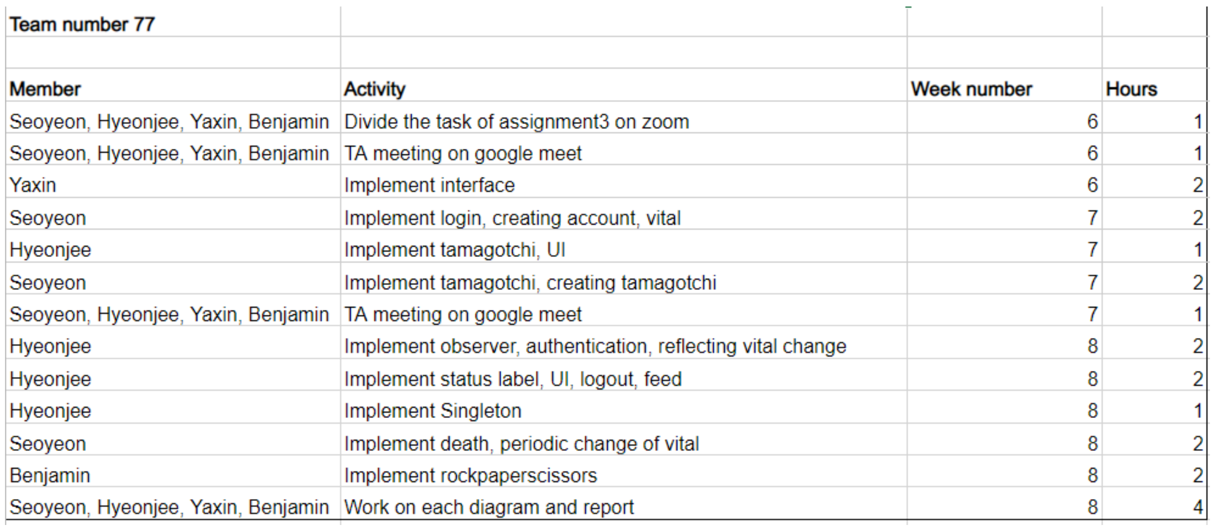
method is used to check if the user has enough money for the food. If the user is short on money, an error message is displayed. If the user has enough money, the user.buyFood(), tamagotchi.eat() function is executed, which decreases the user's balance and improves the tamagotchi's vitals. The frame is then disposed.
If the user clicks on the play, sleep, call doctor, clean&wash, earn money button, for each button the function tamagotchi.playWithUser(), tamagotchi.sleep(), tamagotchi.meetDoctor(), tamagotchi.wash(), user.rockPaperScissors() is executed. Those methods only work with the vitals, by improving the vitals by a certain amount. This part is implemented with the function vital.setMood() and vital.setCleanliness().

Link for the video: https://www.loom.com/share/aed3078877604c5dbcefa96a8705b225

Maximum number of pages for this section: 4

Time logs

<Copy-paste here a screenshot of your time logs - a template for the table is available on Canvas>



Team number 77			
Member	**Activity**	**Week number**	**Hours**
Seoyeon, Hyeonjee, Yaxin, Benjamin	Divide the task of assignment3 on zoom	6	1
Seoyeon, Hyeonjee, Yaxin, Benjamin	TA meeting on google meet	6	1
Yaxin	Implement interface	6	2
Seoyeon	Implement login, creating account, vital	7	2
Hyeonjee	Implement tamagotchi, UI	7	1
Seoyeon	Implement tamagotchi, creating tamagotchi	7	2
Seoyeon, Hyeonjee, Yaxin, Benjamin	TA meeting on google meet	7	1
Hyeonjee	Implement observer, authentication, reflecting vital change	8	2
Hyeonjee	Implement status label, UI, logout, feed	8	2
Hyeonjee	Implement Singleton	8	1
Seoyeon	Implement death, periodic change of vital	8	2
Benjamin	Implement rockpaperscissors	8	2
Seoyeon, Hyeonjee, Yaxin, Benjamin	Work on each diagram and report	8	4