Assignment 3

Team number: 35
Team members:

Name	Student Nr.	Email
Alvaro Pratama Maharto	*2734663*	*alvaropratamamaharto@student.vu.nl*
Mahmoud Asthar	*2696767*	*m.ashtar@student.vu.nl*
Michael Evan Sutanto	*2728578*	*sutanto@student.vu.nl*
Miguel Antonio Sadorra	*2728578*	*m.a.sadorra@student.vu.nl*

Format: The Classes, State Machine, Sequence, and Object names are written in **bold**. All other attributes, methods, activities, events, guards, and other elements seen in the diagrams will be formatted in *italics* in the descriptive texts provided.

Summary of changes from Assignment 2

Author(s): Miguel Sadorra

The following changes have been adapted based on the comments received from Assignment 1. We chose not to fix the comment regarding the proper representation of enumeration for the Button class because we believe it might have been confused with the <<enumeration>> buttons right beside it. The Button Class is a separate class that helps us create objects of the button and is not aimed to represent enumeration. Other comments have been adjusted accordingly as follows:

1. No changes have been made to the state machine diagrams due to receiving positive feedback.
2. Added min and max parameters for our Game Process sequence diagram.
3. Added missing attributes to Object Diagram.
4. Changed navigability with our vital classes. All classes are navigable and visible by health but not the other way around since health looks at all other vitals to update itself.
5. Added a GameEnv class that handles the mini-game and handles changing backgrounds and changing the character's states (i.e. PushUp or PushDown)
6. Changed most public variables and methods to private.
7. Changed relationship of 1 to 1 for Main and Panel since Main only contains 1 panel that enumerates all buttons.
8. Added individual shared aggregations between vital subclasses (e.g. Health, Mood, Hunger, Sleepiness, etc.) as opposed to having just one (1) multiplicity of 1 to 1 to the abstract class Vital.

Revised class diagram

Author(s): Alvaro Maharto, Mahmoud Ashtar, Evan Sutanto, Miguel Sadorra

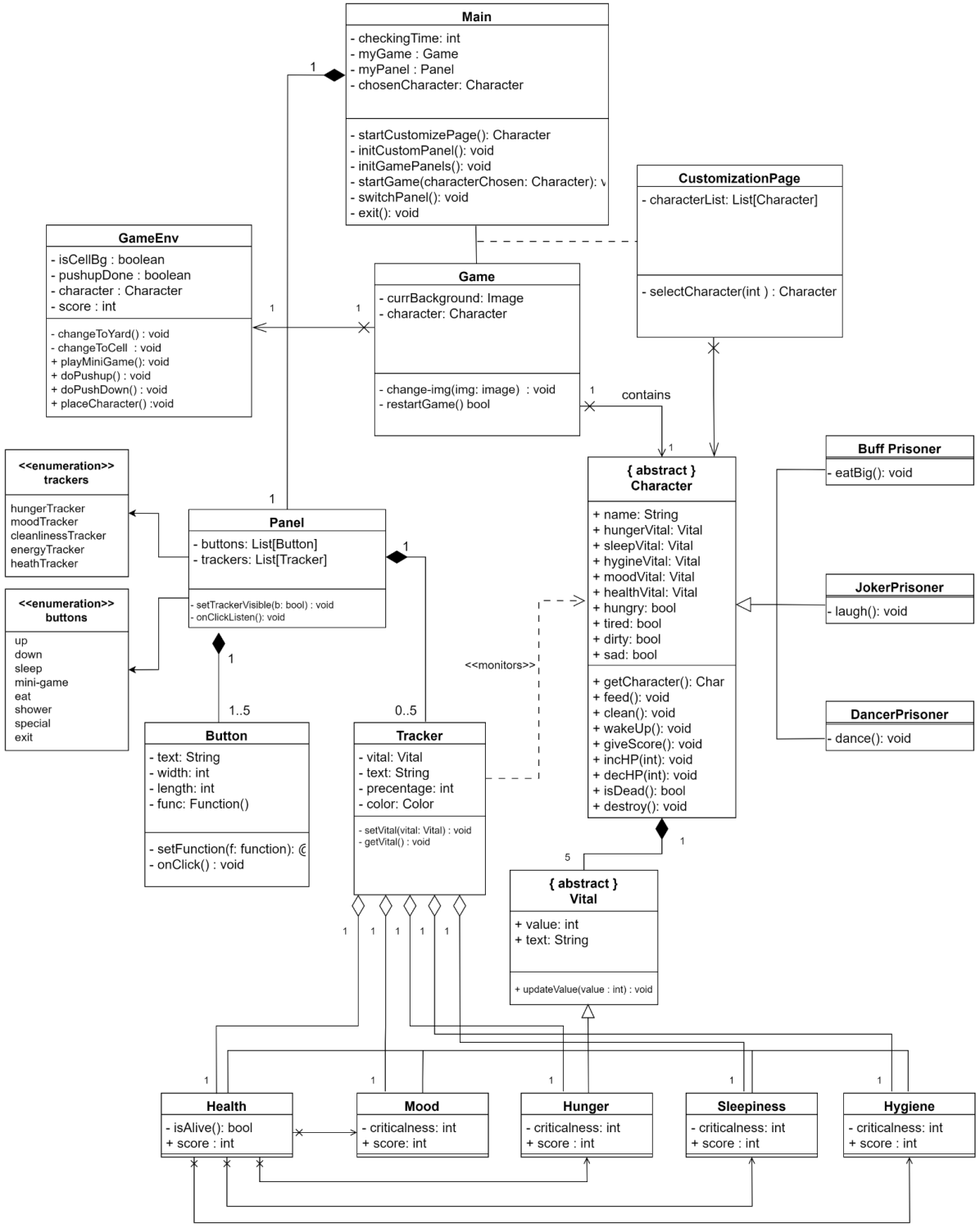


Figure 1.1 Original Complete Class Diagram with Edits based on Assignment 2 Comments

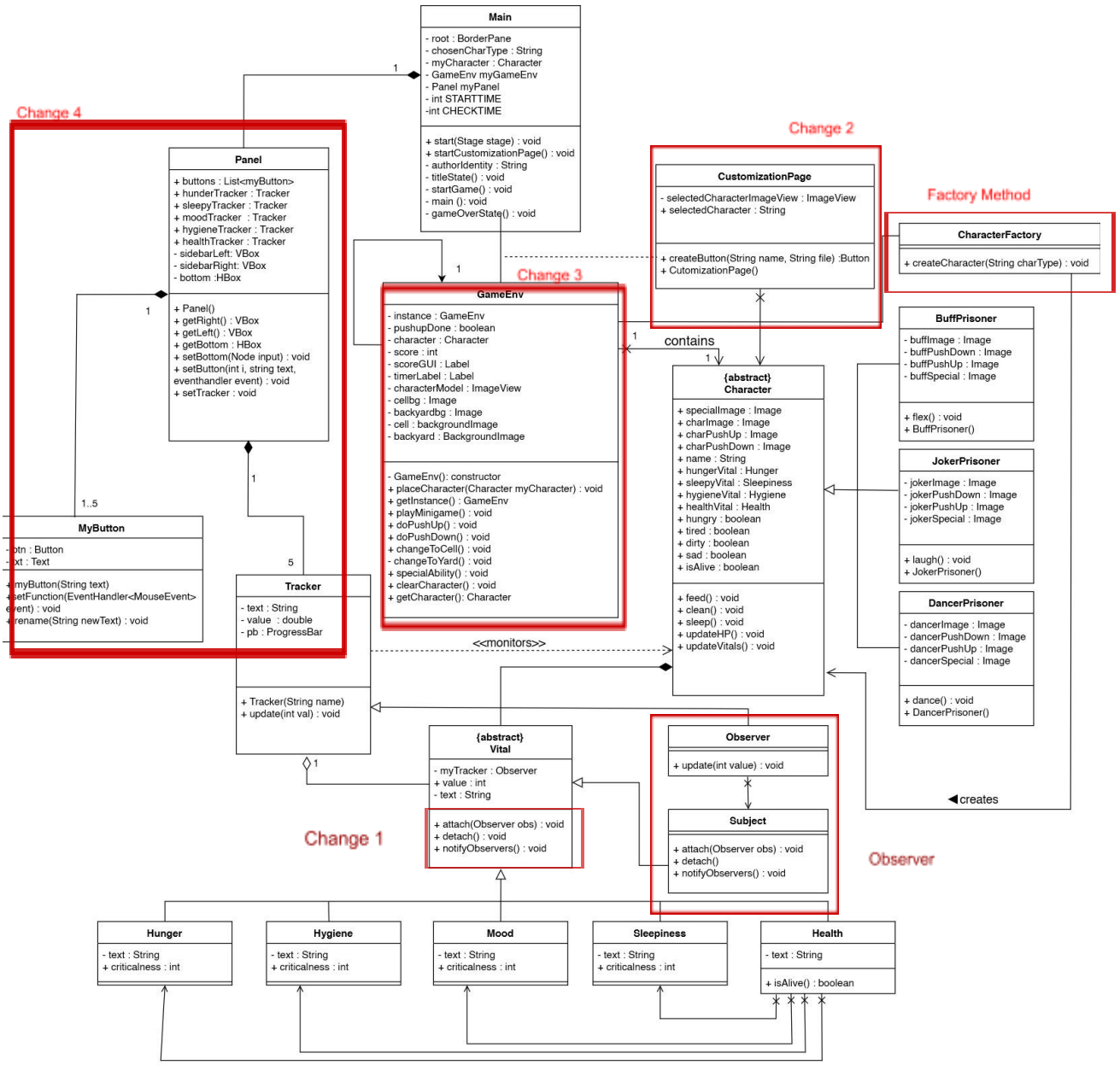


Figure 1.1 Revised Complete Class Diagram

The following has been produced after successfully implementing our game named JailBird. Two design patterns have been evidently shown in our implementation, namely the *Factory Method* and the *Observer Method*. Aside from the addition of the following classes for implementing with our chosen design patterns, the majority of the changes seen are in our variables and methods. Since the initial class diagram was created prior to implementation, the group has seen better variables to use and more methods that could help in building our game.

To begin our discussion of the changes conducted on our class diagram, we begin with the highlight of ID **Observer**. The method aims to observe changes that are seen in the vitals and update the trackers accordingly. Given that the game is a replica of Tamagotchi in a way that it actively monitors a “pet”, we need to always take into consideration the character’s state. In our implementation, we take a look at the following vitals: hunger,

sleepiness, mood, cleanliness, and health. As such, the **Observer** was thought to be the best design pattern to implement as the **Observer** constantly looks at changes in these vitals to update the **Tracker** class. In this case, the trackers are the bars that indicate the current levels of the character's vitals. With this implementation, we get the values inside these bars updated in real-time.

After observing the **Observer** method, we take a look at ID **change 1** which shows some minor differences between the previous methods we have added in our initial class diagram. In our previous implementation, we had a method name *updateValue()*, however, since the implementation of the design pattern, we simply override inherited methods. In this case, no methods are bourne by the class and do less creating more readability rather than updating each time.

The second design pattern we have chosen to implement was the **Factory Method**. This design pattern allows us to create objects based on a specific abstract class, which in this case is the **Character** abstract class. Given that we have a customization page that allows us to select based on three (3) different types of characters, the group decided to create a separate class that could help us choose one of the following options. Apart from the ease of use in selecting the characters, this also allows us to have one primary function that we can call when we want to have an instance of a particular type of character. As such, we see that the **CharacterFactory** class is binary associated with the **CustomizationPage** class upon the selection of a character and then proceeds to have a relationship with the note "creates" as it makes a single character based on the three (3) subclasses that inherit from it (**BuffPrisoner, JokerPrisoner, DancerPrisoner**).

We have also crafted some changes on ID **change 2.** In our initial diagram, we had one variable *characterList : List[Character]* and a single method *selectCharacter*. A major reason for adding numerous variables and methods is that implementation made the group realize more logic behind a character selection method. Particularly considering the CharacterFactory, this means we don't need to create three different Characters and later destroying two of them after choosing. Another reason behind the added variables and methods is the use of JavaFX. The implementation required us to use certain variables such as VBoxes and HBoxes that allow us to place images on the window pane. Additional buttons were also added that are pressable on the game window itself and are not available on the panel for easier character selection.

In ID **change 3**, we see a drastic difference from our **Game** class as seen in the initial class diagram. The initial class only allowed for a *restartGame* and *change-image* method. Since starting implementation, we have realized that the game class acts similarly to a game state method that changes accordingly depending on what part of the game you want to access. Additionally, we only have two states in our game where one is the cell state (prisoner in the prison cell) and the minigame state (prisoner in the backyard). In our initial class diagram, we have a representation of **GameEnv**, which simply handles the minigame. However, the group realized that having only two states, it would be better to implement it in one class. Finally, this shows that GameEnv should only be called once and therefore made into a Singleton. This becomes handy to ensure the buttons can get the instance of the current game environment. They essentially do the same thing with much less code and fewer files to sort through.

Lastly, we take a look at the **Panel** and **Button** class changes where we chose to simply place every single tracker, and eliminated the enumeration class for both of them. Since we will be making use of the trackers all throughout the game process, we decided to have variables readily accessible rather than have to sort through a list and look for a

particular name when we only have five buttons during the lifetime of the game. Similar reasoning goes behind the removal of our button enumeration. We simply have a button class that creates a new instance each time we need to map a new functionality, rather than having a list of buttons that we sift through every state in the game.

Application of design patterns

Author(s): Miguel Sadorra and Alvaro Maharto

	DP1
Design pattern	**Observer**
Problem	Each instance of the **Tracker** Class needs to be updated as the **Vital** Class changes. There needs to be complete accuracy and instant change seen as soon as the values in one class are updated.
Solution	Since we want to update instantaneously, an Observer design pattern would be suitable. Its only job is to check the following classes constantly and update other classes accordingly. Therefore, if one of the **Vitals** changes, the **Tracker** would be updated and shown on the **Panel**.
Intended use	We plan to implement it with 2 separate classes different from our initial implementation. We added a **Subject** class and an **Observer** class. The Tracker will inherit from the **Observer,** therefore, can also act as an observer for the vitals. The **Vital** class will inherit from the **Subject** class and will notify if any of its instances will change. Since Vital is an abstract class, its generalizations namely **Health, Mood, Hunger, Sleepiness, and Hygiene** can also be monitored by our observer class.
Constraints	The **Observer** class only interacts with classes that deal with information regarding the character's vitals and is not accessed by any other parts of the system.
Additional remarks	None

	DP2
Design pattern	**Factory Method**
Problem	The group has three types of characters that it wants to use for the class **CustomizationPage**. We have created an abstract class **Character** and three subclasses namely **BuffPrisoner**, **JokerPrisoner**, and **DancerPrisoner**. We want to have a method that allows us to create a specific instance and keep using it throughout the game.
Solution	In our class diagram, we can see the extra class of **CharacterFacotry**. This allows us to create a character based on other existing classes as specified in the problem above. This also allows us to create a specific **Character** subclass without having to specify a particular class (e.g **BuffPrisoner** or **JokerPrisoner**).
Intended use	The **CharacterFactory** class is an implementation of the **Factory Method** that allows us to create a particular **Character** subclass without specifying which one it is. This creates an easier process when customizing the character as well as having a class that handles this particular type of character. In summary, the class will only be used for handling the creation of a specific character.
Constraints	The **Factory Method** is only seen in all classes relevant to characters such as the **Character** and **CustomizationPage**.
Additional remarks	None

	DP3
Design pattern	**Singleton**
Problem	We want to have only one instance of particular classes. This is logical for classes such as the **GameEnv** class because we should only have one instance of the class in every state of the game.
Solution	Implement a **Singleton** method that is designed to produce one, and only one, instance at a time during any moment of the game. This is represented in the class diagram with an arrow pointing back to itself.
Intended use	The **Singleton** class is used in the **GameEnv** class. The reasoning behind this is that the **GameEnv** handles the current state of the game window and should only have one instance at a time rather than multiple.
Constraints	The class will only be used for classes that should only have one instance at a time. In our case, it is only used for **GameEnv**.
Additional remarks	None

Revised object diagram

Author(s): Miguel Sadorra, Evan Sutanto

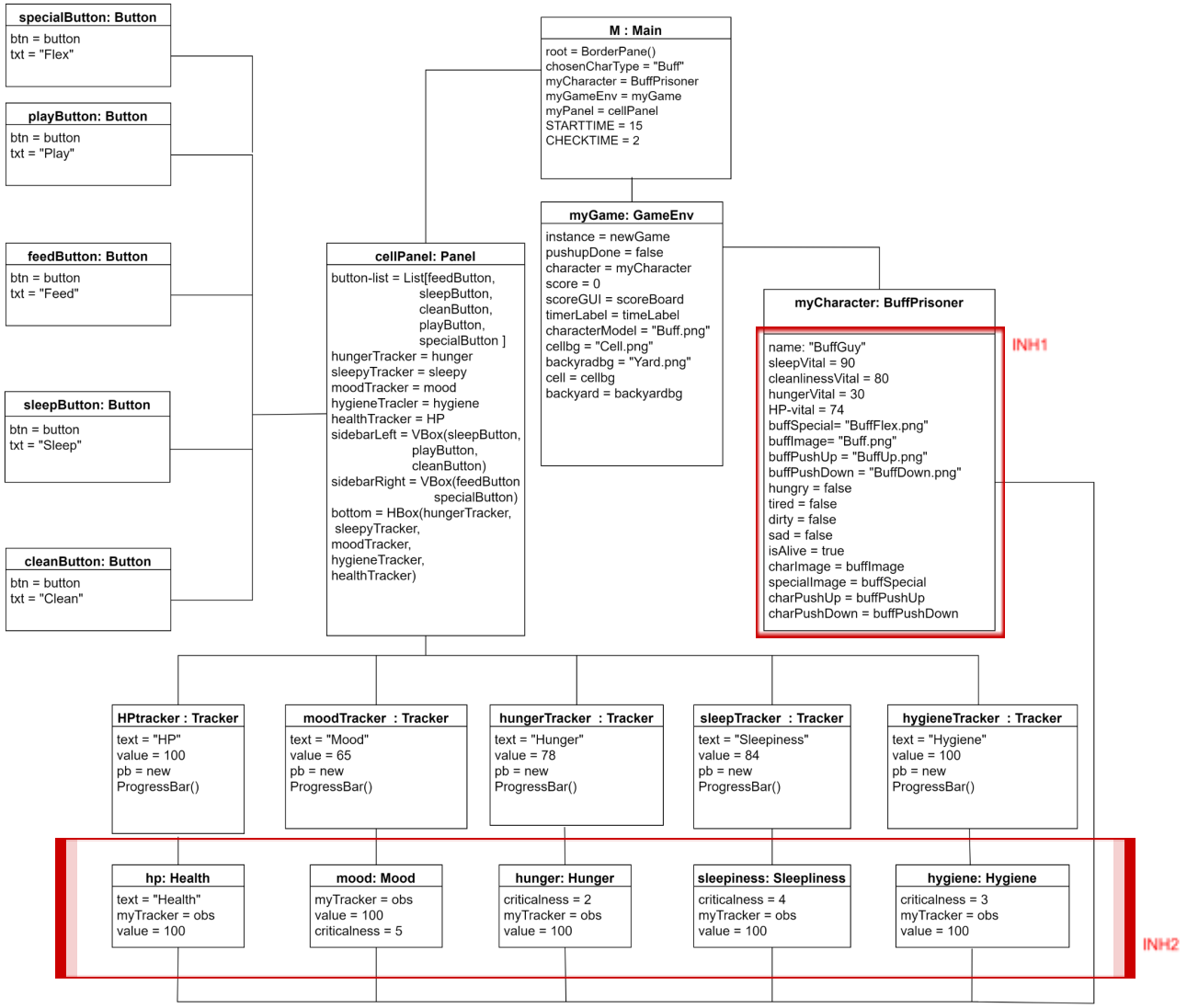


Figure 2.1 Revised Complete Object Diagram

The object diagram depicted above shows not much change from our initial implementation, since the process that the group employed while implementing the game mainly followed the already previous diagrams. The majority of the changes we have added are variable names that we have previously seen in the initial implementation.

Another improvement that we have created for our object diagram is to include the variables inherited from the abstract classes. We see this in both of the highlighted areas above with ID **INH1** and **INH2**. **INH1** in this case inherits variables from the **Character** abstract class and **INH2** inherits from the **Vital** abstract class. The main reason for changing is that this is the right method of representing what the group missed on in the first iteration of the diagram.

Apart from these subtle changes, nothing has changed in the object diagram that still also depicts the state in which the prisoner is in the cell. All relationships stayed the same as well as all the multiplicities with a slight difference in the variables provided that are also seen in our complete class diagram.

Revised state machine diagrams

Author(s): Mahmoud Ashtar and Miguel Sadorra

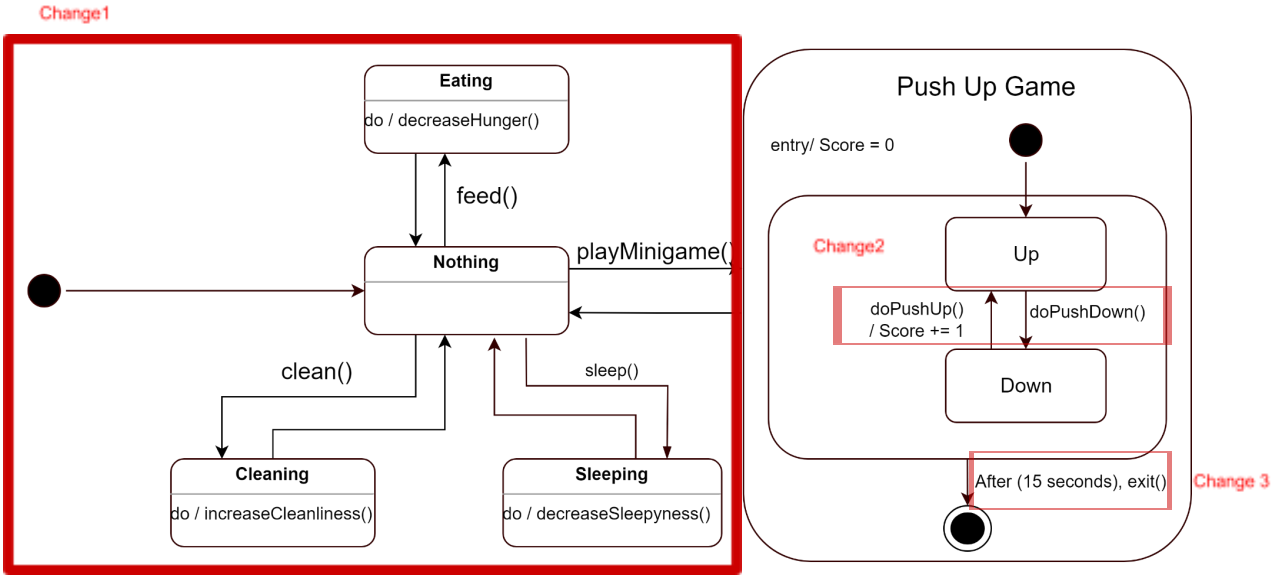


Figure 3.1 Revised State Machine Diagram for the Active Prisoner

A major change that we see in our revised state machine diagram for the Active Prisoner is seen in the highlight above ID'ed **Change1** where the initial state starts from the **Nothing** state as opposed to the initial diagram that started from the **Sleeping** state and would wake up to proceed to the **Nothing** state. After starting the implementation process of the game, we realized that it is better for the character to be starting from the standing position while doing nothing to have a smoother transition from the **CustomizationPage** to the actual game itself. As such, the change has been done.

Another change we see is more minor and just adheres more to the names we have placed in the code as seen in highlight ID **Change2**. From the previous names *pushUp*() and *pushDown*() to *doPushUp*() and *doPushDown*() respectively. There were not many differences seen apart from the two we have highlighted as the implementation process followed the previously placed state machine diagrams.

Lastly, we have changed the **PushUpGame** state where we reduced the time before the exit to 15 seconds. The previous diagram showed 30 seconds. This change is seen in ID **change3** We chose to reduce the time because the time playing a mini-game was too long upon testing the game amongst ourselves.

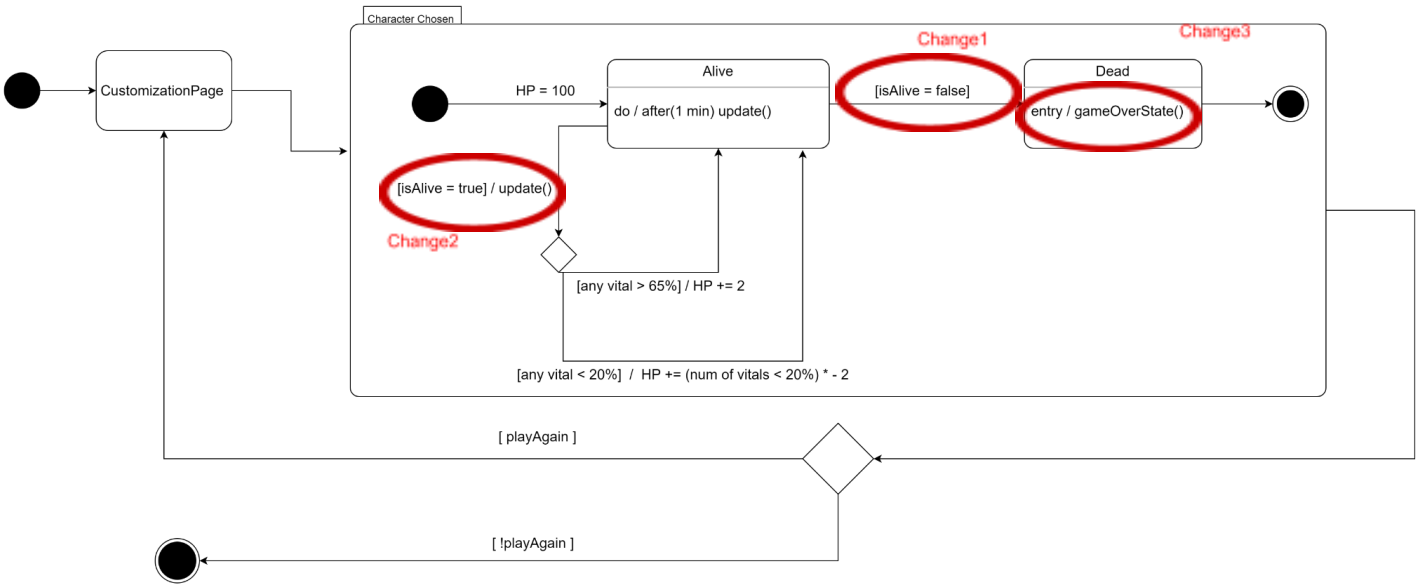


Figure 3.2 Revised State Machine Diagram for the Character Life Cycle

For the state machine diagram presented above, most of the changes have just been adjusting the variable names for our variables. We see in highlights **Change1** and **Change2**, the guard variables inside have been adapted accordingly to our code as *isAlive*, which is a boolean that checks if our character is still alive and checks if the game can continue on. The previous one was named the *check*() method, which was decided by the group not to implement and reduce the amount of code that does the same thing. For **Change3,** we changed the function name from GameOver() to GameOverState() to make it more straightforward.

Apart from this, we only received positive feedback for how we represented our system in our previous assignment and merely adapted the diagram to fit our implementation. As such, not too many revisions have been made, as seen above.

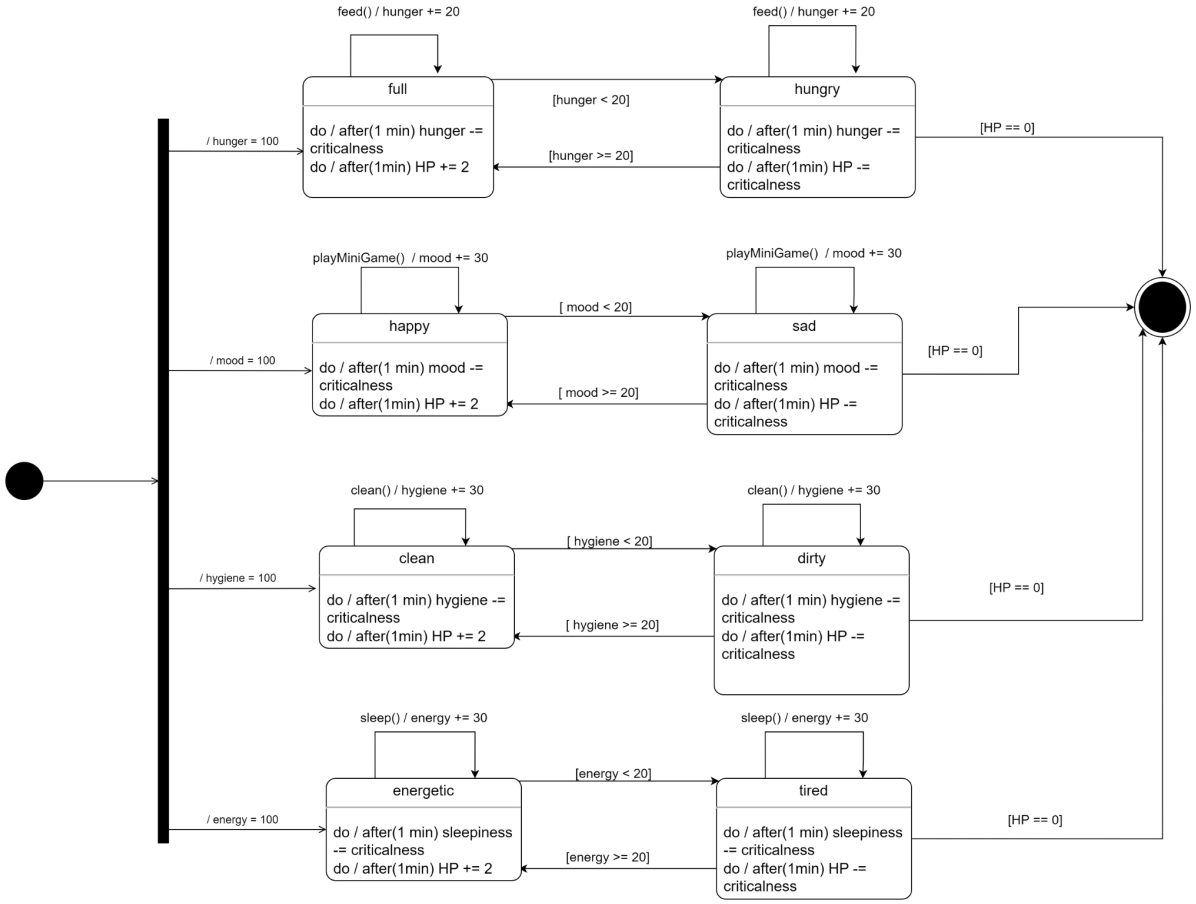


Figure 3.3 Revised State Machine Diagram for the Vitals

The following state machine above received no changes as it was followed exactly during our implementation process. We received mostly positive comments and did not need to change. The final output still follows the exact process when dealing with the vitals. Reduce HP if one of the vitals is below 65 and keep reducing until HP reaches 0. This guard of reaching 0 would then lead to the final state which is the death of the character.

Another interesting part of this diagram is the parallelization that occurs in the beginning. This interacts well with our observer method because the observer can look at all the following events simultaneously and update the tracker accordingly to have real-time updates on the current levels of the character's vitals

Revised sequence diagrams

Author(s): Alvaro Maharto and Mahmoud Ashtar

Game Process

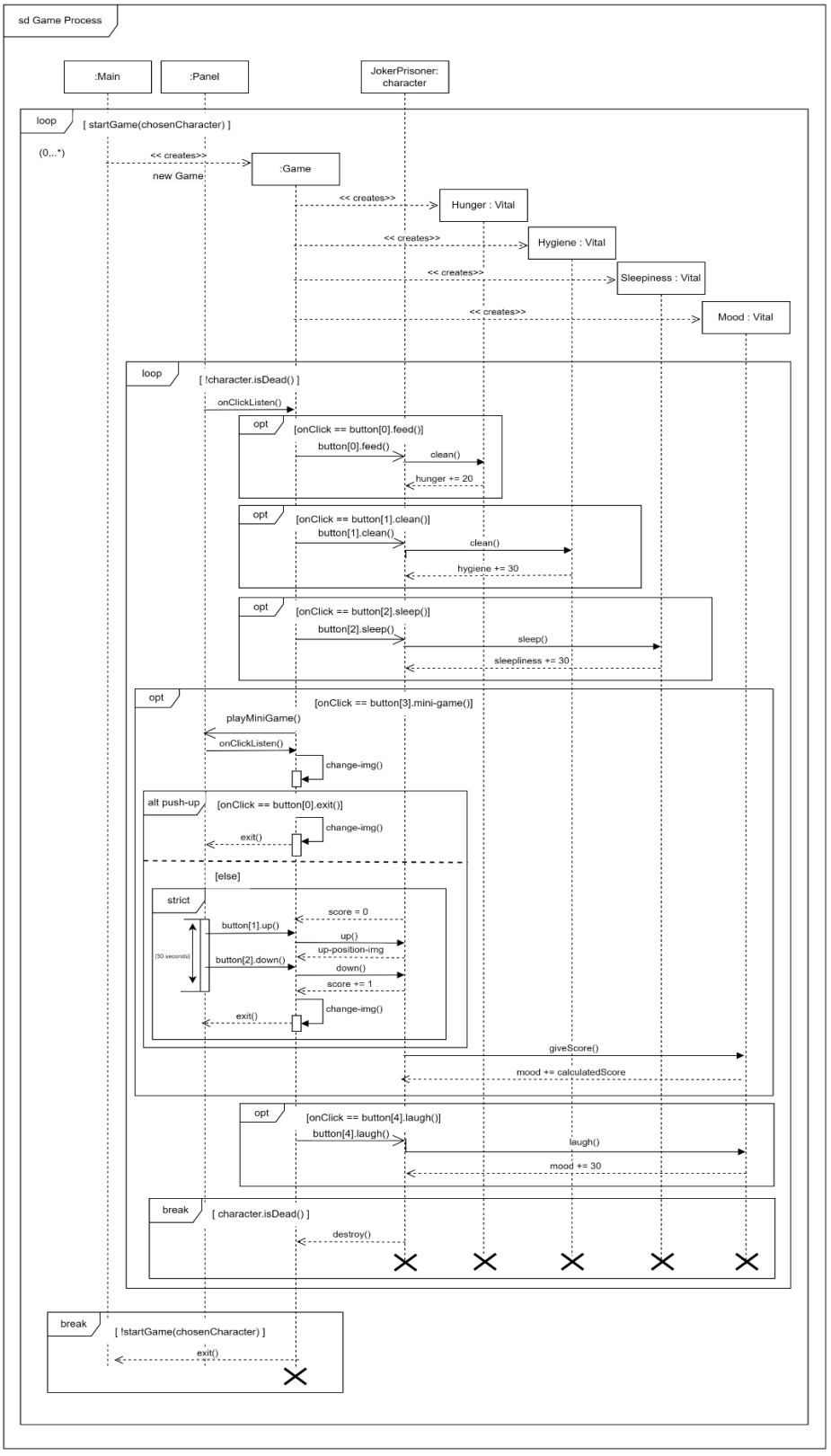


Figure 4.1 Original Complete Game Process Sequence Diagram

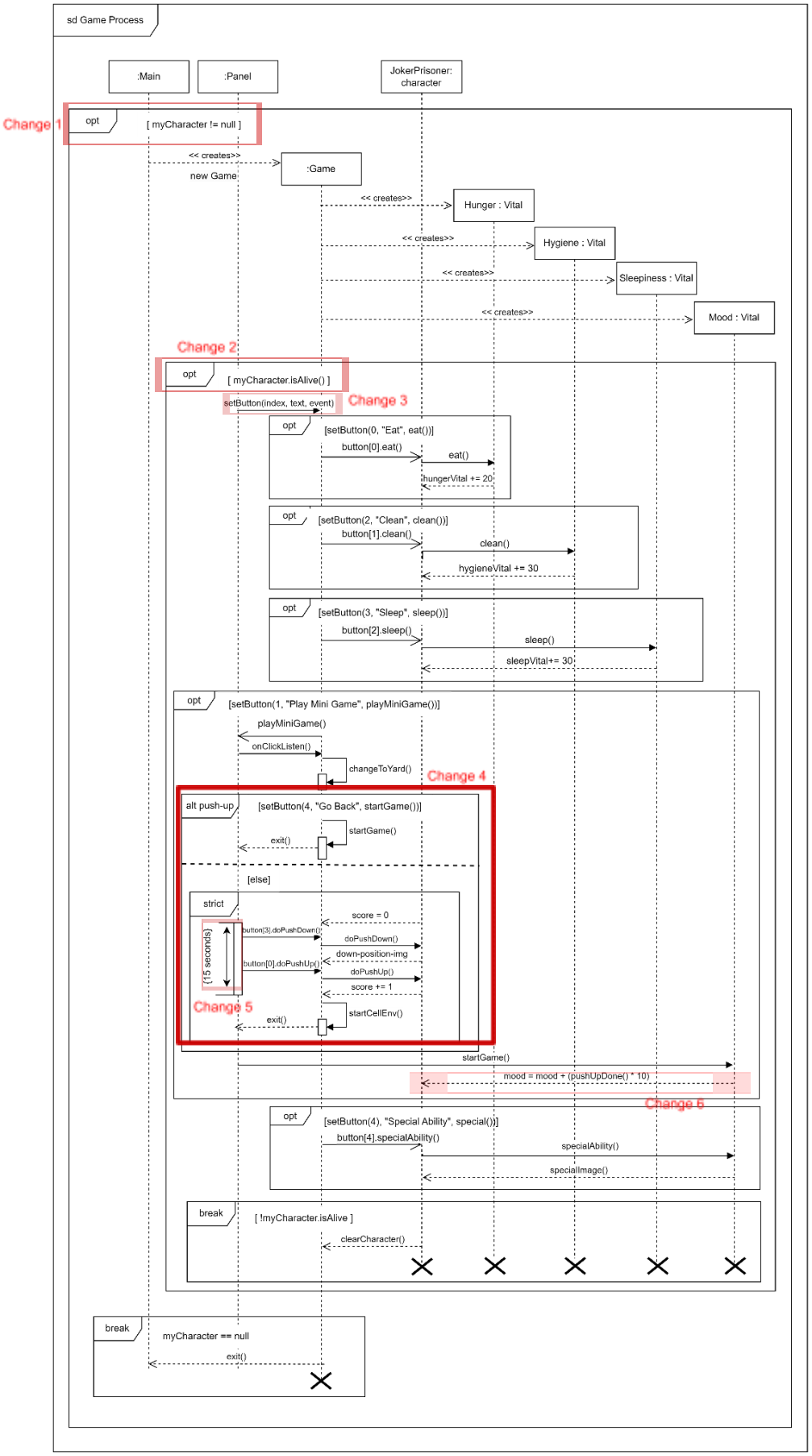


Figure 4.2 Revised Game Process Sequence Diagram

The above diagrams show the original and the revised game process sequence diagrams that illustrate the interactions and flow of information between the different elements of the game.

First, let us take a look at **change 1 and change 2**, which show the change of fragment. We used a loop fragment on the original (old) sequence diagram to start the game. However, when implementing our Tamagotchi application, we preferred to use conditional operations to add the logic inside the application. Hence, we changed from the loop fragment to the opt fragment. We also changed the conditions that checked the condition by checking if **myCharacter** is not null to create a new Game. Then to enter the game, we checked if the **character.IsAlive**. Nevertheless, if (!**character.IsAlive)** the character is dead, the condition will break.

We also made a function “setButton(index, text, event),” which is the **change 3**. This function adds functionality and button names for each button in our application. Therefore, this function can assign buttons for eating, cleaning, sleeping, playing a minigame, and a special ability. Next, for **change 4**, we also used the “setButton” function when the user enters the minigame since the button functionality differs from before and inside the minigame. Furthermore, we changed the button functionality to push up, push down, and go back buttons. For **change 5**, we changed the time of the minigame from 30 seconds to 15 seconds because 30 seconds is too long upon testing the game amongst ourselves. Lastly, after the user finishes playing the minigame, the mood will increase by ten for each push-up done, as we can see in **change 6**.

Game Initialization

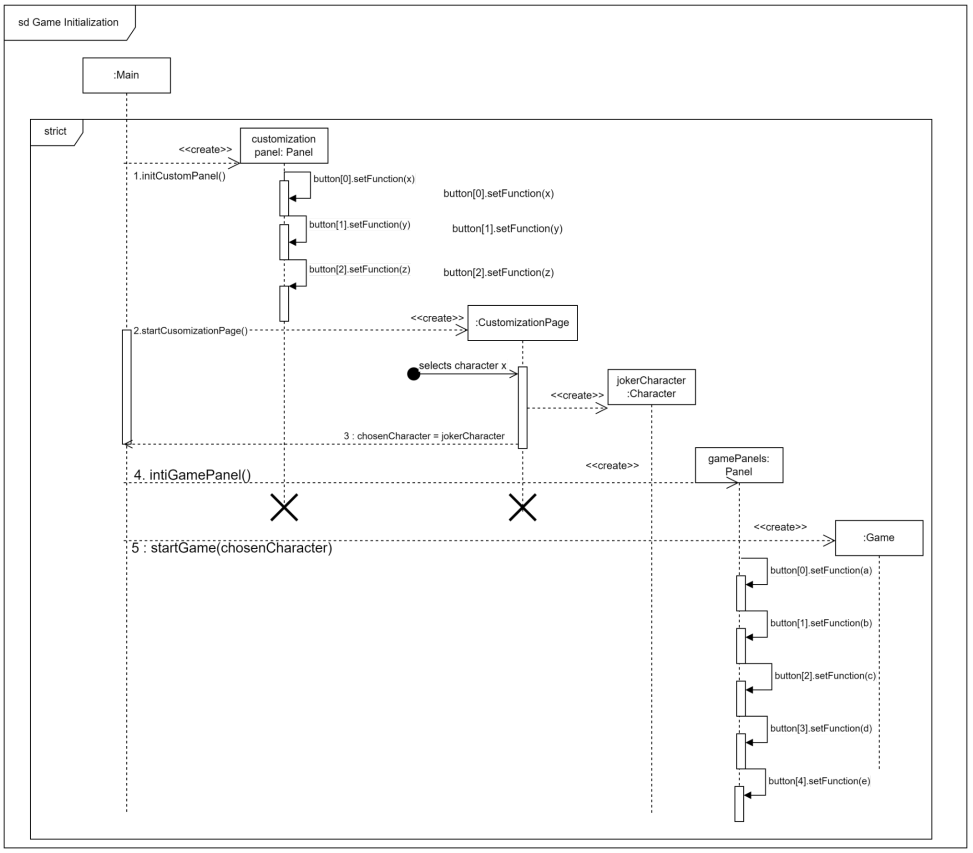


Figure 4.3 Original Sequence Diagram

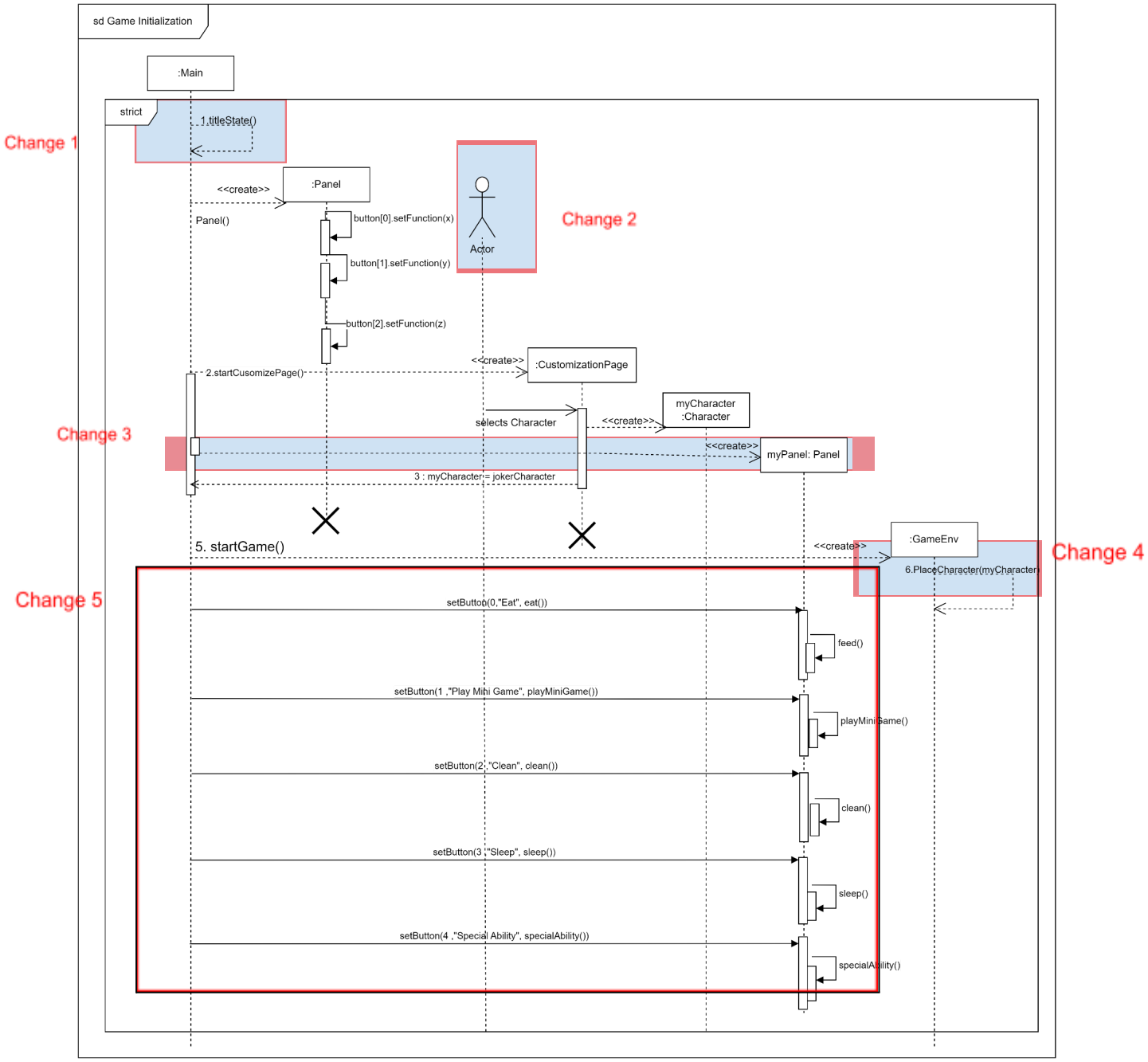


Figure 4.2 Revised Sequence Diagram

Change 1is for the titleState() method. This method is responsible for the starting screen that shows up before the customization page. This method is called from main. **Change 2:** the addition of a user life line. The user must choose a character for the game initialization to go on. **Change 3** is regarding the creation of myPanel, which is the panel for the game. This is created inside the event listener that is set inside the startCustomizPage() method. Panel is only created after selecting a character. **change 4** placeCharacter(myCharacter) is used to put the character inside the GameEnv. Since GameEnv is a singleton we can't initiate the character from inside so we use the placeCharacter(myCharacter). **Change 5** setting the functionalities of the buttons is done from Main

Implementation

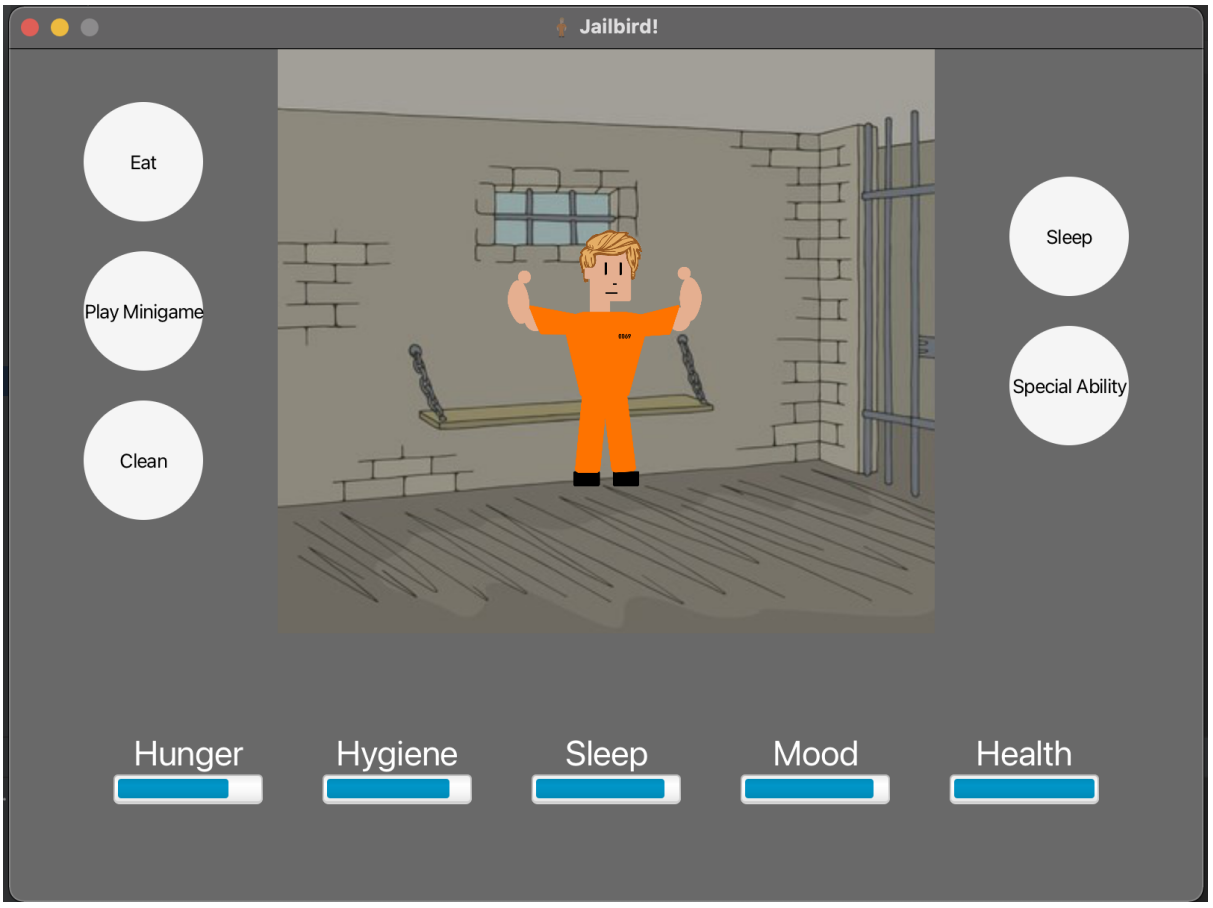
Author(s): Alvaro Maharto, Mahmoud Ashtar, Evan Sutanto, Miguel Sadorra

Main.java File Location:
software-design-35/src/main/java/softwaredesign/Launcher.java/

Jar File Location:
software-design-35/out/artifacts/software_design_vu_2020_jar/

Link to demonstration video:
https://youtu.be/KQbXQ8RzRCo

Snapshot of the system:



Strategy from UML diagrams to code implementation:

Given the limited amount of time to create the actual application itself and produce in writing a formal description of the system, we decided since the beginning of Assignment 2 to assign one of our members to begin coding and be the “champion” of this particular task. This champion was then tasked to assist the other members in getting familiar with the new

language Java and the library we had to use in order to create a game with a User Interface, which was JavaFX. As the champion is familiarizing himself with the new language, the group meticulously designed the UML diagrams so we would have a concrete template as well as a good common idea of what the game would be like for everyone. As such, combining efforts into creating classes and communicating about coding matters was easier for the group.

Since the UML diagrams were good in the group's opinion, we chose to follow all of these diagrams and coded the program as such. Although the diagrams were clear, the group would face a few problems in the syntax. Therefore, we chose to implement first and try our own styles that would compile properly. After having designed a semi-functioning model, we then gathered as a group to adhere to some coding conventions and started applying our design patterns. The implementation of design patterns became easy since the classes we have prepared are ready to be used in the following design patterns such as our abstract classes that could easily use the Factory Method.

The group also decided to make use of Photoshop to design the characters as PNG files that could be easily placed in the resources folder and accessed anywhere in the file. We designed 12 characters and character movements that were toggled depending on which state they are in. For example, the BuffPrisoner would be in a standing position doing nothing in the image "Buff.png" but would have to flex because the prisoner presses the button "Special Ability", then the character image would be toggled to "BuffFlex.png" that made it seem like the characters raised his arms to a flexing position.

Key Solutions:

The primary issue that the group faced, however, was changing states. The changing of the state from CustomizationPage to the GameEnv state, to the gameOverState becomes a tall task that the group would face. The solution we have found is creating objects in our Main class. With the help of object-oriented programming, it was quite an easy fix to the big problem. The logic behind when and how we would change the scenes would be part of the solution. We had a titleState() that showed the title of the game along with all the collaborators' names then smoothly fade into our StartCustomisation() method that allowed for a CustomizationPage for us to select our characters. Creating new objects each time the state changed proved to be vital to solving the problem.

Another tiny yet rather cumbersome problem the group faced was the placing of images onto the Pane. The group solved this issue by placing VBoxes and HBoxes which allowed us to easily add images of the characters onto the window. Also, the method of coding a picture into the window through Java is quite difficult, and we have solved this with Java functions such as getClass().getResource(<name of image>).toExternalForm(). This made it easier for us since our game is mostly based on images and the process became exponentially easier.

Lastly, the creation of our Jar file also proved to be quite a challenge because of the lack of familiarity with the coding environment. The problem we faced would not be actually creating it but rather creating it properly. We would easily create Jar files but would quickly run into error codes. The solution that we found was to look at the error codes and find the

main error, which would be found in the .getResource() function that returns null. We later found out that the Jar files we have been creating would not add the images we are trying to render, therefore the function could not find them and returned an exception. We tinkered around with the IDE and managed a way to add the image files into the jar files and finally produce the final product.

Time logs

Team number **35**

Member	Activity	Week number	Hours
Evan	Create Window with JPanel	5	4
Evan	Change JPanel window with Javafx	5	2
Miguel	Create Character Images	6	2
Mahmoud	Create Character and Vital Class	6	4
Alvaro	Create Game Environment	6	3
Alvaro	Edited Main and Game Classes	6	4
Miguel	Edit Assignment 2	7	2
Alvaro	Create Customization Page	7	4
Mahmoud	Implement Design Patterns	7	3
Miguel	Implement MiniGame	8	3
Evan	Implemented Tracker Class with Vital Interaction	8	4
All	Game Logic	8	15
All	Writing Documentation	8	10
		Total	60