Assignment 2

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 1

Author(s): Miguel Sadorra

The group's submission of Assignment mostly received positive feedback, apart from the recommendations for the group's implementation of the vital tracker and the time log. The following comments have been considered in the adoption of Assignment 2. Apart from the following, no changes have been made.

Changes made:

- Time log based on actual work done and not a planned schedule.
- Added description of how character decreases HP and dies if it reaches zero (0)
- Considered the how vitals would decrease over time.

Class diagram

Author(s): Mahmoud Asthar, Alvaro Pratama Maharto, Michael Evan Sutanto, Miguel Sadorra

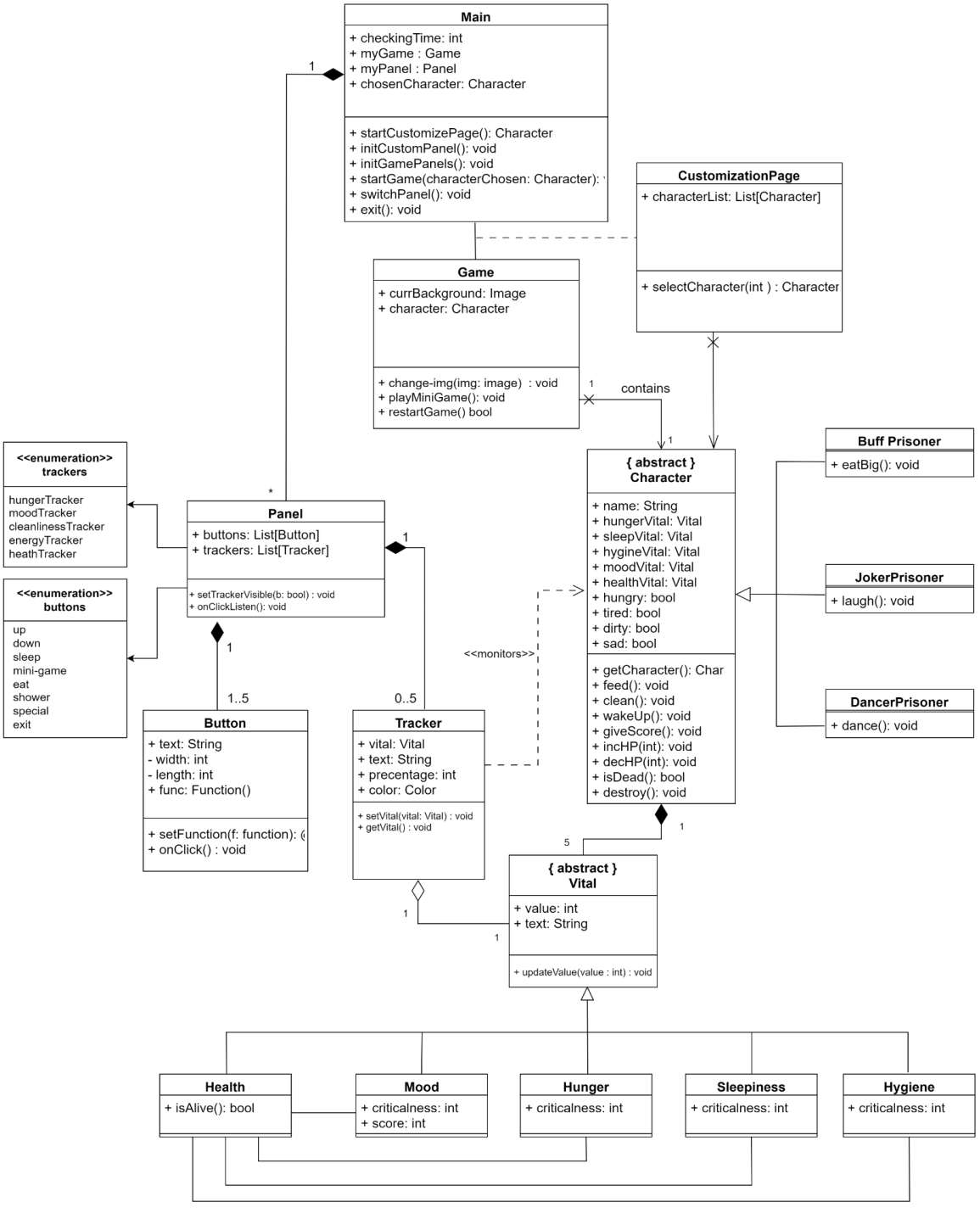


Figure 1.1 Complete Class Diagram

The Class Diagram presented above shows the classes that the group has come up with for the implementation of JailBird, a prisoner care-taking game. The main class of **Game** represents the instance of a game being started and the classes that will work hand-in-hand with it. We have the **Panel** which interfaces with the user as it contains the pressable buttons to the users and the tracker that indicates the current state of their prisoner. The **Character** class in our game is representative of only one (1) character (the

Prisoner) since he is the only realistic character in a care-taking game. We have placed pre-set character types that the user may choose upon based on his/her preference, which all inherit from the abstract class of Character. Lastly, the **Vitals** are an abstract class that indicates the health and well-being of the Character and if it would need action to keep it alive. As such, the diagram above is how we would have envisioned implementation to go.

The following classes have been formulated by the entire group for the implementation phase:

1. **Main**

The **Main** class represents a game engine that handles all the elements in the program. Some key attributes to note are the *myGame* and *checkingTime*. The *checkingTime* attribute specifies the time at which a vital has to increase or decrease. The *myGame*, on the other hand, is an instance of a **Game** for the user to play. The *switchPanel*() method is also a key part of the class, as it allows the game to switch panels and interfaceable buttons. The class has two (2) primary relationships, namely a binary association with the **Game** class, since they both give and take from each other. The second relationship it has is the **Panel** with a composite relationship with multiplicity one (1) to many. It is also worth noting that the relationship between **Main** and **Game** have an intermediary class in between them which is an Association Class called **Customization Page**. This is an added step that **Main** needs to perform before it actually gets to the **Game** class.

2. **Game**

The **Game** class represents the actual ***Game*** session entered by the user. We also added important attributes to this class such as *currBackground* and *character*. The *curr*B*ackground* simply specifies the current location of the main character while the character represents the main character that the user gets to play with. It has two operations with *change-img* and *playMiniGame()*, which are pretty self-explanatory. For instance, the background of the mini-game would differ. The *playMiniGame*() method allows the user to change from game states from normal cell state to play a mini-game with the chosen character.

The class is connected to two classes, one is the abstract class ***Character***, and the other **Main** class with an association class of **CustomizationPage** in between. The ***CustomizationPage*** class is considered an association class, since it adds a layer of operations before the **Game** and **Main** can interact (they first need to select a character to use). On the other hand, the abstract class ***Character*** is considered an association since the ***Game*** class only contains one (1) character which is the main character, which is contained within the **Game**.

3. **CustomizationPage**

The **CustomizationPage** area represents a page where the user can select his/her desired character with special quirk (i.e. **BuffPrisoner**). Its main attributes are *chosenCharacter* and *characterList*, which represent the list of characters it can select and a variable to place which one the user selects. Lastly, we have a method of *selectCharacter*(*int*) which is a method that gives the user an option to pick and returns a **Character**. It has two (2) primary relationships, namely a broken line association with **Main** and **Game**, indicating that the **CustomizationPage** class is an Association Class. The second relationship it has is a one-way association to **Character** since it accesses the type of character that can be used through the game process.

4. **Panel**

The ***Panel*** class represents the user interface, essentially. It shows all the necessary character vitals, as well as the pressable buttons that can be used to interact with the

character. It has two lists as its attributes, the *buttons, and* the *trackers*. The buttons enumerate the number of buttons available to press at the current state of the game, while the *trackers* shows the vital trackers.

The ***Panel*** is directly linked to the ***<<enumeration>> buttons*** **box** and ***<<enumeration>> trackers***, and has a relationship with composition for both ***Button*** and ***Tracker*** classes. Since the *buttons* attribute is a type List, then it receives the enumeration, and similar to the *trackers* attribute which is also a list. Since the panel consists of a set of trackers and buttons, it has a composition as its relationship. The buttons indicate the number of possible actions that can be mapped to a button, while the **Button** and **Tracker** classes indicate common attributes to a user interface. Lastly, it is contained in a composite relationship with **Main,** since **Main** needs access to panel to interface with the user and control other areas of the program.

5. **{abstract} Character**

The **Character** class represents the main character. It is an abstract class in which different types of character types inherit from (i.e. **BuffPrisoner**,**JokerPrisoner**). The vitals are the main attributes placed here which monitor the main character's levels (*health, mood, etc*). The *isDead* attribute is a boolean that ensures the character is alive and the game can go on. The main relationships are a generalization to our character types for player customization, which are **BuffPrisoner**, **JokerPrisoner**, and **DancerPrisoner**. Another relationship is a binary association from Game to Character, signifying the **Game** class contains the **Character**. In addition, we've added a dependency relationship between the **Tracker** class to the **Character,** since the **Tracker** essentially monitors the **Character** throughout the game. Lastly, we have a composition type relationship with the **Vital** abstract class to represent that a **Character** contains vitals. For methods, we have *getCharacter*() method that is worth noting since it retrieves an instance of a specific character to be placed in the game. In addition, additional method such as *feed*() and *clean*() functions are also included that are basically actions that a character may possess.

6. **BuffPrisoner, JokerPrisoner, DancerPrisoner**

The **BuffPrisoner, JokerPrisoner, DancerPrisoner** represents what the names mean. They are pre-set character appearances that give the individual characters possess traits that the user may choose from. For example, **BuffPrisoner** seems more fit and bigger than the other two while **JokerPrisoner** has a laugh function that a user can do at any time of the game. Some methods that are worth mentioning are *eatBig*() for **BuffPrisoner**, *laugh()* for **JokerPrisoner**, and *dance*() for **DancePrisoner**. They give them unique quirks such as the ability to dance and eat more than a usual character. All three classes are part of a generalization relationship with the abstract **Character,** where they inherit all of its attributes and methods.

7. **Button**

The **Button** class represents the buttons that a user may interact with over the course of the game. The **Button** has a composition relationship to the **Panel,** since the panel consists of **Buttons**. The multiplicity of buttons is 1 Panel to 1 to 5 buttons, since there are always 5 buttons available in the **Panel**. However, in certain game modes, some of these buttons are not pressable due to not having limited functionalities in the particular game mode.

Furthermore, some attributes are the *width, length and func*. The *width* and *length* specify the size of the button on the screen, while func specifies the action mapped to the button. The most important method in the class is the *setFunction*() method, since this actually maps

the action to the physical buttons and gives the user an interface to interact with their character.

8. **Tracker**

The **Tracker** class represents the visible bars that indicate the levels of a specific character vital (i.e **Health**). The tracker has a shared aggregation with the **Vital** abstract class, since the class shares certain characteristics that monitor the **Character**. It also has a dependency on the **Character** class, since it monitors the current state of that class to update its own values. Lastly, it is part of a composition of the **Panel** class, which is a user-interactive class. Some key methods worth mentioning are the *setVital()* and *getVital()* which basically initialises the class and gets them for future instances. Lastly, the most important attributes are the percentage and vital, since they are representing the **Character's** current state in terms of the levels of their vitals. There is also an attribute called color that gives each tracker a specific color, so it can be distinguished among other trackers (i.e. **Mood**/**Hunger**).

9. **{abstract} Vital**

The **Vital** class represents the actual vitals themselves as an abstract class. Its main relationships are the **Character** class with a composition relationship since the **Character** has vitals, the **Tracker** with a shared aggregation since they are related to each other, and the five vitals that inherit from it: **Health, Mood, Hunger, Sleepiness, and Hygiene**. The main attribute that is worth noting is the *criticalness* which is the value for the **Character's** vitals, where the value indicates the level specified in the **Tracker** class. In addition, the *updateValue()* method is also important since it makes sure that the vitals are moving in the right way according to the actions of the user.

10. **Health**

The **Health** class represents the health vital and monitors that vital in specific. The only method it contains is an *isAlive()* method that constantly checks if there is HP left to keep the game going. It inherits from the abstract **Vital** and gains access to all its attributes (*value* and *text*) and methods (*updateValue()*). The relationships it has is a generalization from **Vital,** since it inherits from it along other vitals such as **Mood** and **Hunger**. It also has an association relationship with **Mood, Hunger, Sleepiness, and Hygiene** since the **Health** vital is updated based on the other vitals' changes. This will be further explained in our state machine diagrams.

11. **Mood, Hunger, Sleepiness, Hygiene**

The **Mood, Hunger, Sleepiness, and Hygiene** vitals represent the mood, hunger, sleepiness, and hygiene of the character respectively and is updated based on its criticalness. The method *criticalness()* indicates a multiplier effect of how much value will be added or decreased based on the mood that will be later described in the state machines. It has a generalization from abstract **Vital** and can gain access to all of its methods and attributes. It also has a binary association with the Health class, since they work hand-in-hand.

12. **<<enumeration>> trackers and <<enumeration>> buttons**

The trackers and buttons classes are simply list data types that contain the possible elements of the List specified in the **Panel** class. It contains the possible mappable actions to specific buttons during a specific mode in a game (for **buttons**). On the other hand, it contains the *hungerTracker, moodTracker, cleanlinessTracker, energyTracker, and healthTracker* for **trackers** class. The only relationship we have is a direct association with the **Panel** to indicate enumeration.

Object diagram

Author(s): Mahmoud Ashtar and Miguel Sadorra

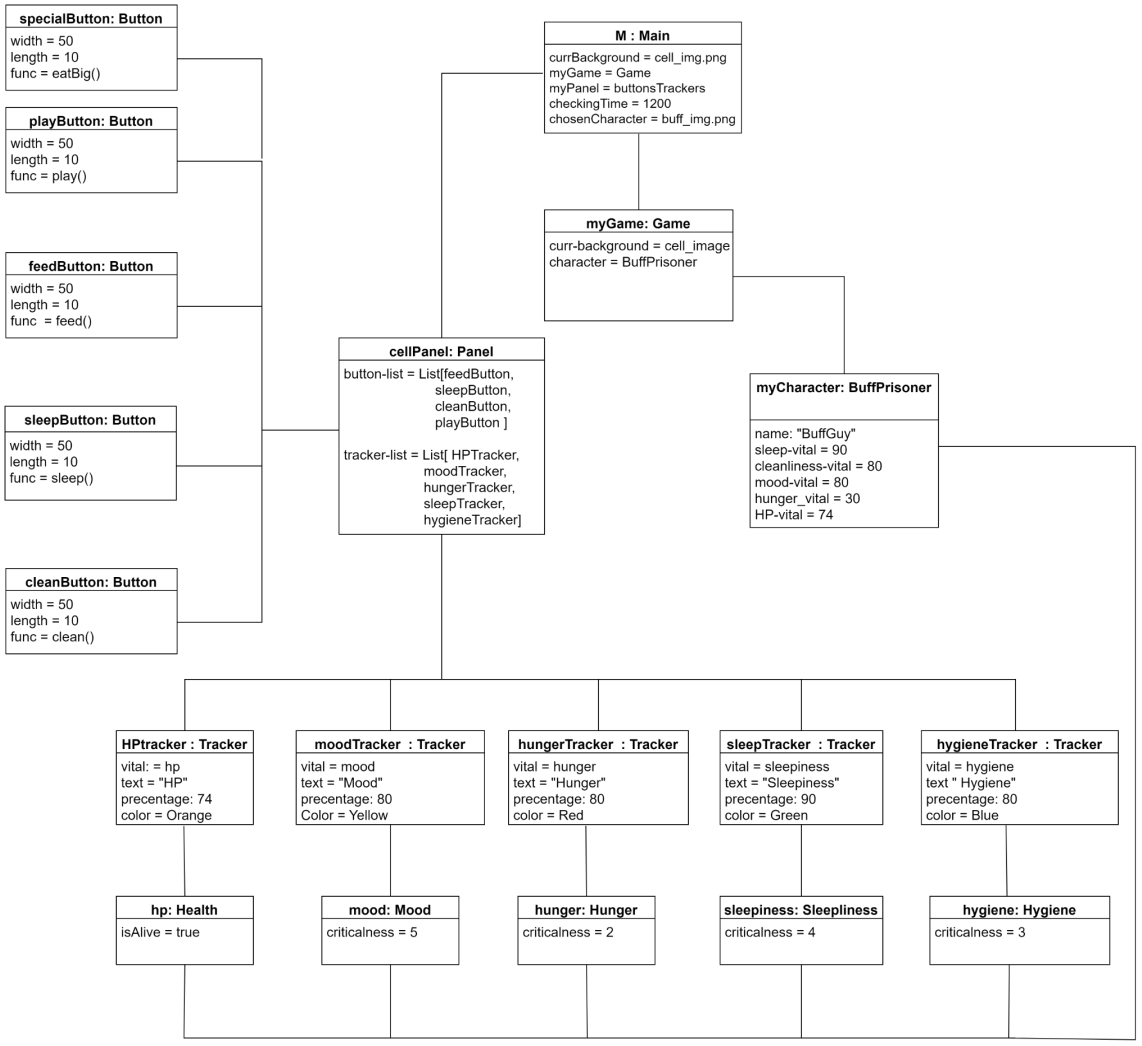


Figure 3.1 Object Diagram in Cell State

The Object Diagram presented above represents the entire system snapshot when the game is in the prison cell (aka *cell_img*) which is when the prisoner is in his primary habitat. The panel shows all five (5) buttons in play with a **specialButton, playButton, feedButton, sleepButton, and cleanButton.** In addition to the buttons, all five (5) trackers are also in the panel with **HPtracker, MoodTracker, HungerTracker, sleepTracker, and cleanlinessTracker**. Lastly, the **Game** object is connected to **myCharacter** which is in a **BuffPrisoner** class as selected by the user. The character is connected to all vitals **sleepiness, hygiene, hunger, mood and hp** where the vitals are representative of the character's current state.

All relationships are as follows: the **Main** object contains the **Panel** object cellPanel with 5 buttons and 5 trackers. The **Main** object also contains the **Game** object, which is a specific instance of a game entered into by the end-user. This class contains a **Character** object **myCharacter** which is an instance of a **BuffCharacter** as selected by the user. Here we see the whole system come into full effect with all of the necessary buttons, trackers, and the game itself.

State machine diagrams

Author(s): Mahmoud Ashtar and Evan Sutanto

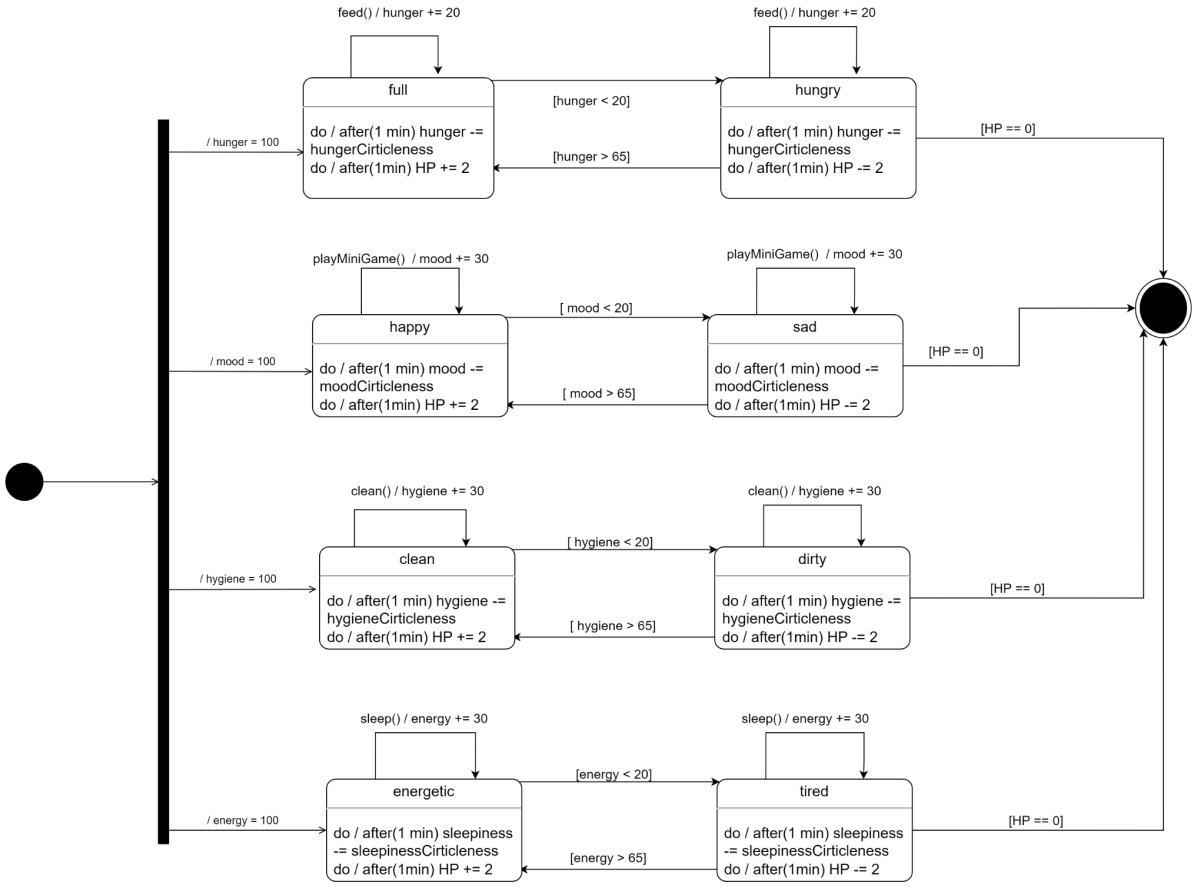


Figure 2.1 State machine diagram for Vitals Internal Behavior

The above state machine diagram depicts the states of the vitals and how the objects under **Vitals** function internally.

As they are all character vitals, they all flow concurrently, and as such, we begin with a parallelization node to indicate concurrency, with all of them having a 100 percent beginning value. Using the ***full*** state as an example, we constantly subtract the *hungerCriticalness*, which is a constant specified in the vital's attributes as an internal activity and then if after 1 minute, the character is still in the ***full*** state, then it will increase *HP* by 2 points. After this state has been done, it checks with a state transition's guard if the *hunger* level is below 20. If it satisfies the condition, then it will move to the other state which is *hungry*. Otherwise, it stays in the full state.

In the event that it switches states to ***hungry***, then it immediately does an activity of subtracting a *hungerCriticalness*, and then does another activity of reducing *HP* by 2 points every minute. If it remains hungry, it will keep on reducing *HP* until no *HP* is left and will end the game, when the character is dead.

Some other events worth noting are the *feed(), playMiniGame(), clean(), and sleep()* triggers. If these triggers are done, it adds to the specific vital and remains in the same state that it previously belonged in. All other concurrent states are similar in essence just like the first vital that we have explained above.

Another note, is that all of our states have two (2) internal activities, and this is because we want constant actions of decreasing the value rather than just decreasing upon entering the state.

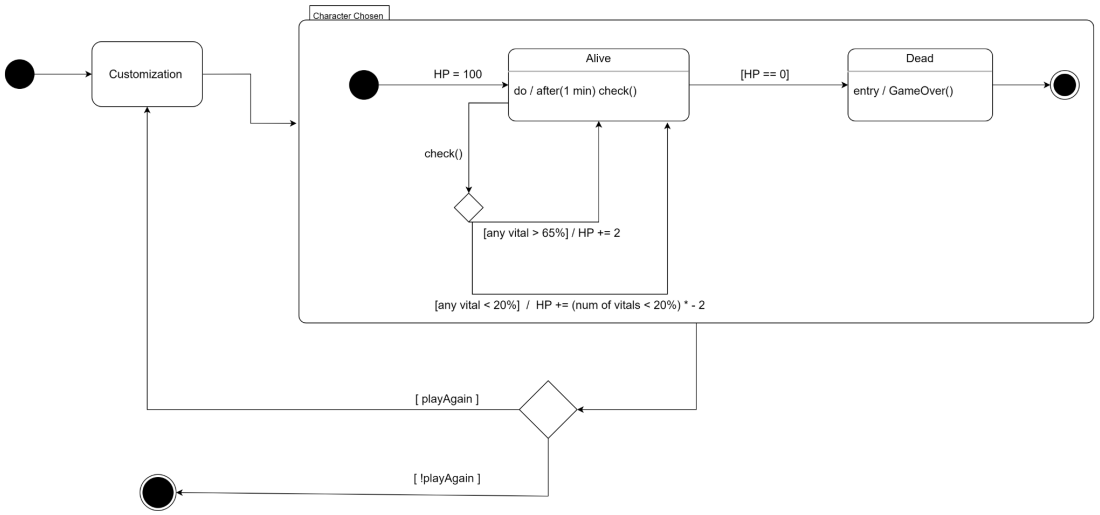


Figure 2.2 State machine diagram for Game Continuity

The state machine diagram presented above represents the **Main** class, where it is decided whether the game should continue or end.

The start state immediately enters into a customization state which then enters a composite state that checks whether the player is still alive. From inside the composite state, we see a start point that specifies a start *HP* of 100. It then enters the **Alive** state where its only activity is to check the character's *HP*. It will then check for the *HP* and adjust the vitals and state accordingly. If any of its vitals is >65% then it will add *HP*, if any of its vitals is <20% then it will adjust with a multiplier of *(num of vitals < 20%) * (-2)*. In addition, it constantly checks whether the HP is at 0 in the guard *[HP == 0]*, and if this guard is met (true), then it will enter the **Dead** state with an entry activity of *GameOver(). The GameOver()* activity is self-explanatory as it will simply end the game. The composite state will then proceed to the final state and exit with no exit activities and proceed to a decision state, where the user can select whether to press *PlayAgain* or not press *PlayAgain*. In the event that the user decides to *PlayAgain*, the state will restart from the customization state and start the game all over again. Otherwise, the game will end, and therefore proceeding to the final state.

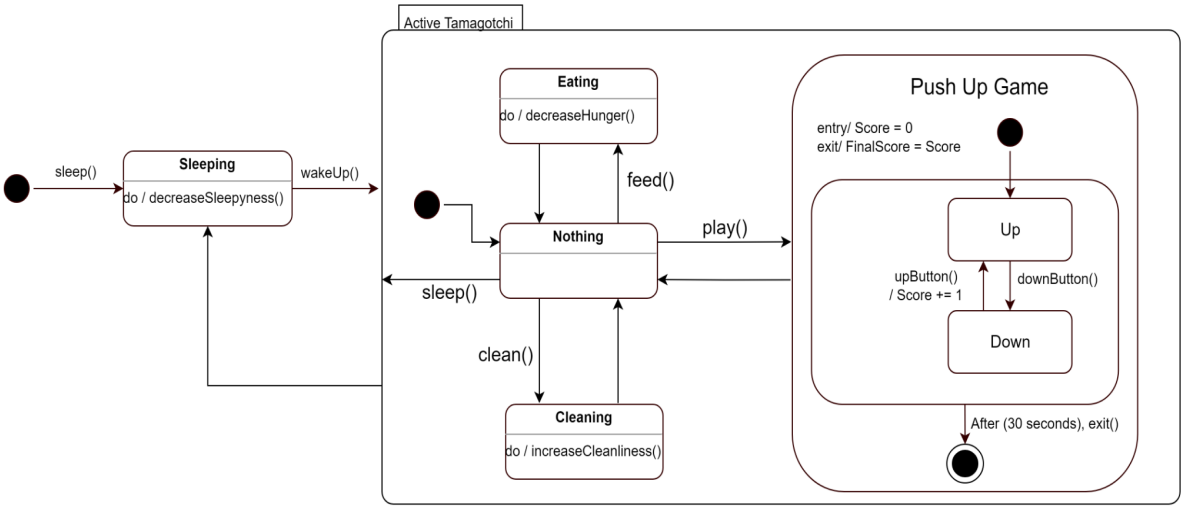


Figure 2.3 State machine diagram for Game Functionality

The State Machine Diagram depicted above represents the **Character** class' internal functions. It shows the character's playable functions such as *feed(), play(), sleep()*, and *clean()*.

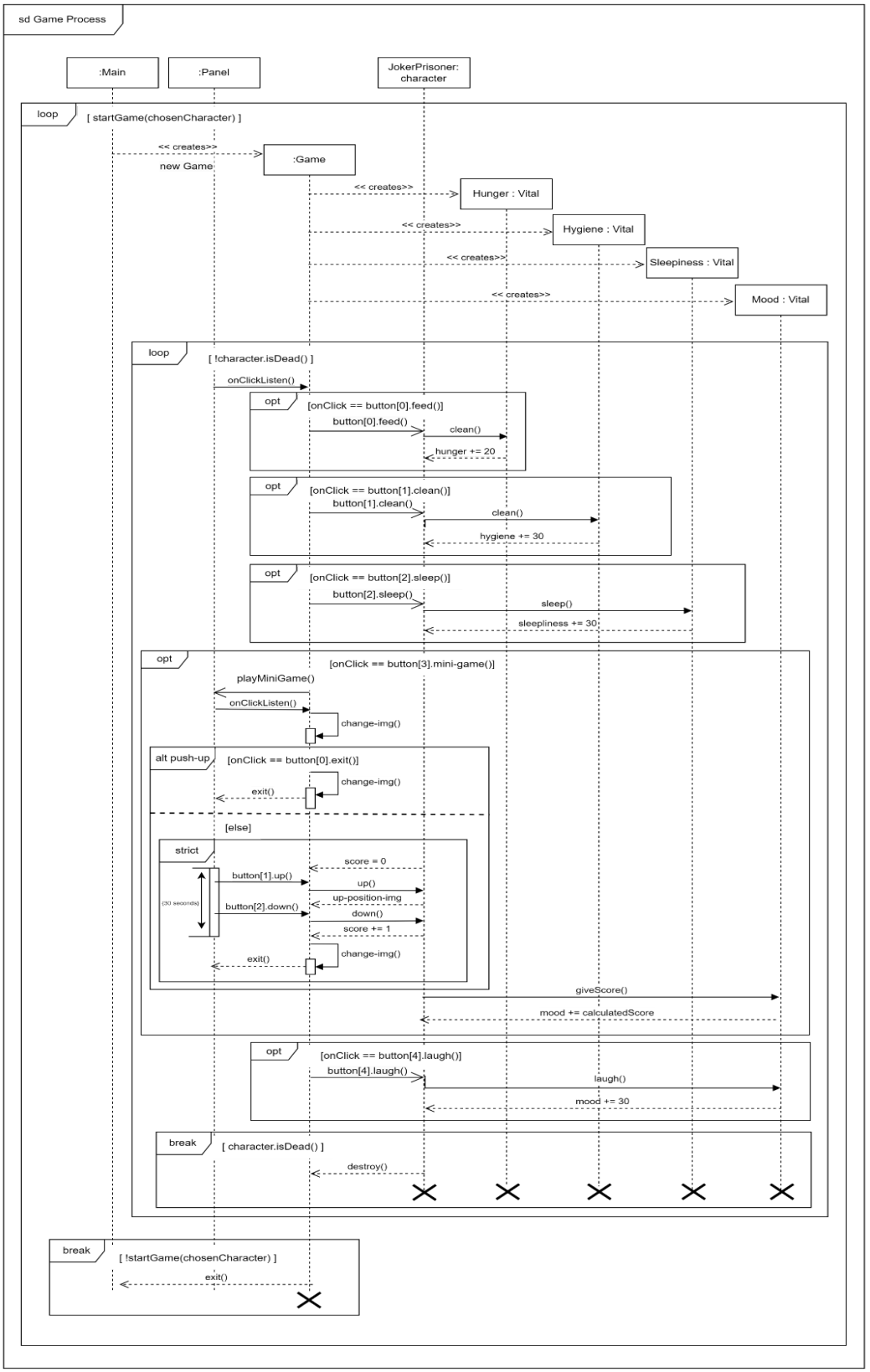
The diagram starts from a **Sleeping** state, which immediately decreases sleepiness as its internal activity and then proceeds to a sequence of events. Inside the state, it begins in a doing **Nothing** state, and subsequently checks for triggers (i.e. *feed(), play(), clean()*, or *sleep()*. We will now describe each state that can be entered from the Nothing state, beginning with the **Eating** state. In the **Eating** state, it is entered when the trigger of a *feed()* activity is done, and within the state, the only internal activity performed is *decreaseHunger()* and subsequently return to nothing. The **Cleaning** state is also triggered by one (1) event, the *clean()* event, and enters with one internal activity in the *do/increaseCleanliness()*. This essentially increases the vital and again return to the **Nothing** state. A big difference in the states is shown in the **Push Up Game** state, where it is represented by another composite state that is nested inside the current state.

In the **Push Up Game** state, we begin in the **Up** state, which begins with the score equating to zero (0), and watches for the event trigger of *downButton()* to transition into the **Down** state. The down state then also waits for the trigger of *upButton()*, and once this is triggered, the action to be done will be to add a point to the *Score* variable. These two states will repeatedly change back and forth until the event of 30 seconds elapsing has occurred. When that event occurs, the **Push Up Game** state will exit and return to the **Nothing** state.

Sequence diagrams

Author(s): Alvaro Pratama Maharto

Game Process



The above sequence diagram shows the game process that illustrates the interactions and flow of information between the different elements of the Game.

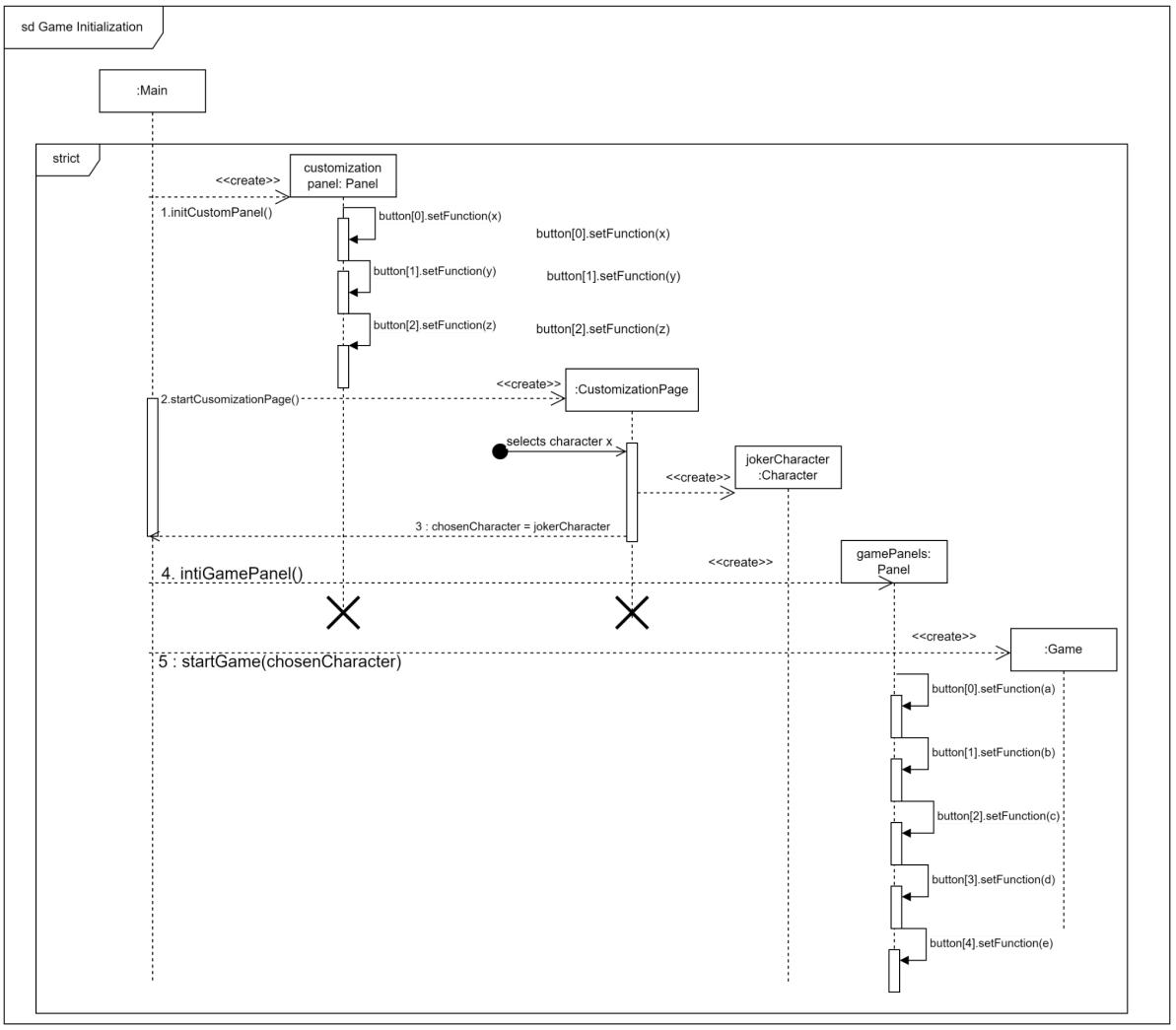
Initially, the: **Main** object enters into a loop and checks if the user has chosen a character. This condition will break if the user is not deciding any character so that it will exit the **Game**. Then, the sequence diagram would show the creation of a new **Game** object by passing the chosen character parameter. Next, the **Tamagotchi** needs a vital's tracker to keep track of their vitals. Hence, the game object creates *hunger, hygiene, sleepiness, and mood* objects for the *vital tracker*.

Once the **Game** and all vitals are created, the **Game** enters into a loop to check that the **Tamagotchi's** character is not dead. The loop will break if the character is dead. The sequence diagram would show this loop involving multiple interactions between the **Tamagotchi** object and other **Game** elements. For the **Tamagotchi** object, we are using an example of the **JokerPrisoner** character. In the **Panel** class, five buttons are clickable for the **Tamagotchi** activity such as *feed(), sleep(), clean(), playMiniGame()*, and their unique ability (in this case, since we are using **JokerPrisoner**, the unique ability is *laugh()*). Therefore, when the user clicks a button, a function "*onClick()*" from the **Panel** is listening and performing an event corresponding to the button pressed. Then, the following cases can occur:

1. When the user clicks the first button, "*feed()*," it will feed the **JokerPrisoner**. After the **JokerPrisoner** eats, the game will increase *hunger()* by 20.
2. When the user clicks the second button, "*clean()*," it will clean the cell of the **JokerPrisoner**. After the **JokerPrisoner** cleans, the game will increase the *hygiene()* by 30.
3. When the user clicks the third button, "*sleep*()," it will make the JokerPrisoner sleep. After the **JokerPrisoner** wakes up, the game will increase *sleepiness*() by 30.
4. When the user clicks the fourth button, "*playMiniGame*()," it will trigger the **Panel** to change the background image to the backyard. Inside this mini-game, the **Panel** has three active buttons such as the *up(), down(), and exit()* buttons. When the user clicks a button, a function "*onClick()*" from the **Panel** listens and performs an event corresponding to the button pressed. Hence, Then, the following cases can occur:
    1.) When the user clicks the exit button, the panel returns to the cell environment.
    2.) Else, we have a strict fragment to make a strict order / fixed sequence of events. The push-up game begins with a score equating to zero (0). After the user presses the *up()* button, the prisoner will have a push-up position. When the user presses the *down()* button, the score will increase by 1. This push-up game will go on for 30 seconds, and after 30 seconds, the game will exit the yard environment.

    After the **JokerPrisoner** has finished playing, it will do the method giveScore() giving the score to the vital and divide the score by 2 inside the vital. From that, it will increase the mood depending on the calculated score.
5. When the user clicks the fifth button, it will show the Tamagotchi Object's unique ability. Users can choose **BuffPrisoner, JokerPrisoner, or DancerPrisoner** on the customization page. For instance, each character has a unique function *eatBig*() for BuffPrisoner, *laugh()* for **JokerPrisoner**, and *dance*() for **DancePrisoner**.

Game Initialization



This sequence diagram explains the first process of the game, which is the intitialisation. Our Main object acts like an engine that manages and regulates the game. It has access to all other classes, so it also inititialises our needed objects. The strict fragment indicates that the initialisation process is always done with the same order of steps in the diagram. When main runs it calls the function “initCustomPanel” which creates the first panel we use to control the customization page. Afterward, three buttons inside the panel are given a functionality to be able to choose the characters that will be shown in the customization page when it is made. Moreover, main uses “startCustomizationPage()” method to create will the first page. The image inside of it will display the possible characters to decide. Then the user clicks one of the buttons and this will trigger the “selectCharacter()” method which in turn will create the character and set the value of chosenCharacter attribute of Main. subsequently, the “initGamePanels()” method will be called to create the different panels for the game. Right after that, we see that lifetime of two classes “customPanel” and “costumizationPage” will end as we don’t use them anymore. Hereafter, The method startGame(chosenCharater) is finally called to create the game. Finally, the buttons of the game panels will be given functionalities. This process will repeat each time the game is restarted.

Time logs

Team number **35**

Member	Activity	Week number	Hours
Alvaro Maharto	Class Diagram	3	4
Evan Sutanto	Class Diagram	3	4
Mahmoud Ashtar	Class Diagram	3	4
Miguel Sadorra	Class Diagram	3	3
Mahmoud Asthar	Object Diagram	4	2
Miguel Sadorra	Object Diagram	4	2
Alvaro Maharto	Sequence Diagram	4	3
Mahmoud Asthar	State Machine Diagram	4	3
Alvaro Maharto	State Machine Diagram	4	2
Evan Sutanto	Basic Code Implementation	5	4
Miguel Sadorra	Changes from A1 Write-up	5	1
Miguel Sadorra	Class Diagram Explanation	5	2
Mahmoud Ashtar	Class Diagram Explanation	5	1
Alvaro Maharto	Sequence Diagram Write-up	5	2
Evan Sutanto	State Machine Write-up	5	2
		TOTAL	39