Assignment 2

Team number: 2

Team members:

Name	**Student Nr.**	**Email**
Shreyas Kumar Parida	2725540	s.k.parida@student.vu.nl
Mohamed Bentaher	2763623	m.bentaher@student.vu.nl
Anass Taher	2743336	a.taher@student.vu.nl
Bilal Derraz	2696011	b.derraz@student.vu.nl

Do NOT describe the obvious in the report (e.g., we know what a `name` or `id` attribute means); focus more on the **key design decisions** and their "**why**," the pros and cons of possible **alternative designs**, etc.

Format:
- The class, state and interactive partner's names are bold
- The attribute, method, activities and messages are in italics

Summary of changes from Assignment 1

Author(s): Full Team

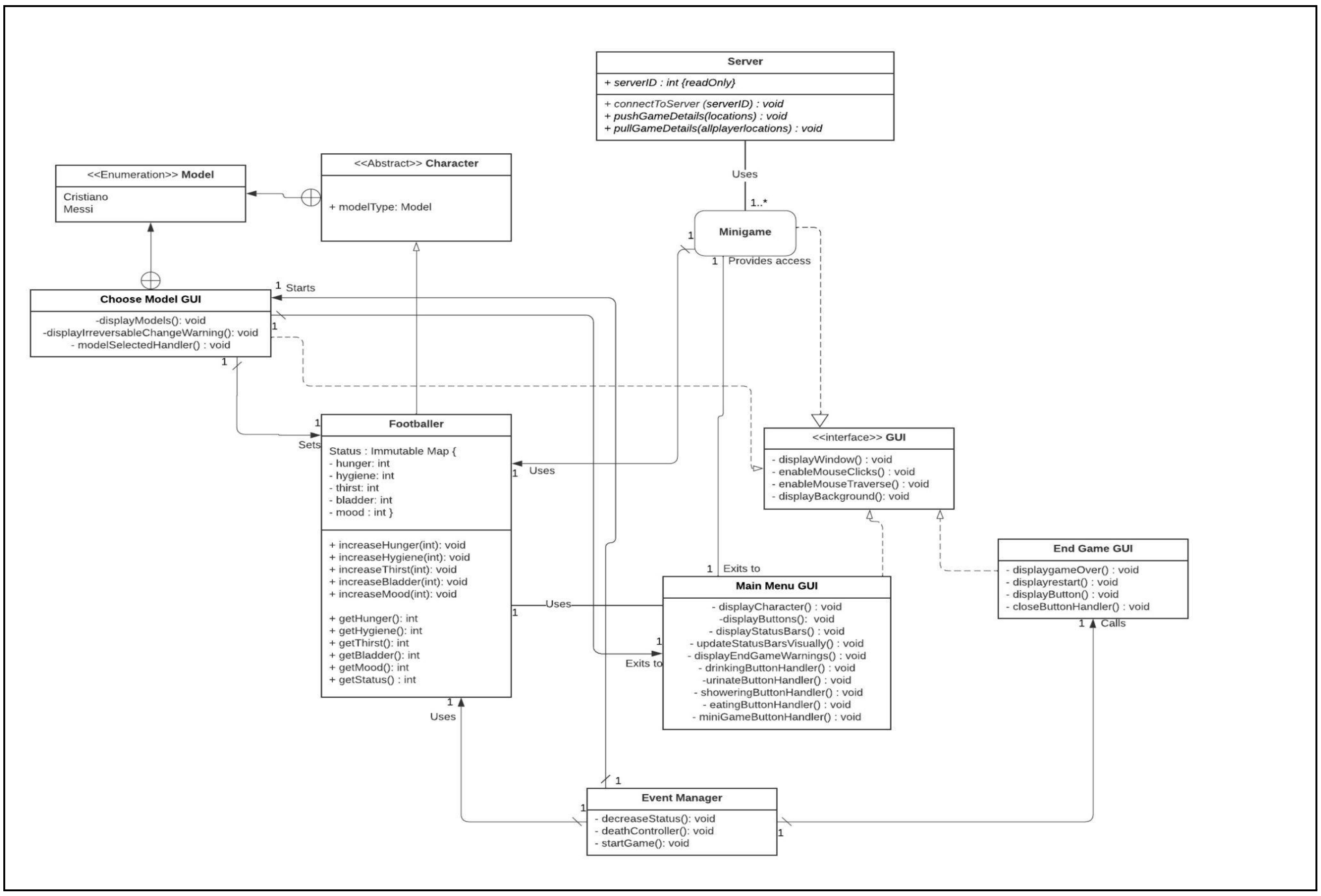
The following changes to assignment one were made:

- We shortened the overview section into a single slide, as requested by the mentor. Furthermore, we rewrote some aspects of the overview section to describe our game more clearly.
- Our mentor suggested we may overcomplicate the development process through too many features. Hence, after discussing with the group and viewing the original Tamagotchi assignment description, we decided to keep only the *essential* features of our game.
- We renamed the quality descriptions and added appropriate tags, such as "reliability" or "usability."

The new version of the changes can be found in the "doc" files in the "Assignment 2" branch on Github.

Class diagram

Author(s): Full Team



Server
+ serverID : int {readOnly}
+ connectToServer (serverID) : void
+ pushGameDetails(locations) : void
+ pullGameDetails(allplayerlocations) : void
Uses
1..*
<<Enumeration>> Model
Cristiano
Messi
<<Abstract>> Character
+ modelType: Model
Minigame
1
1
Provides access
1
Starts
Choose Model GUI
-displayModels(): void
-displayIrreversableChangeWarning(): void
- modelSelectedHandler() : void
1
1
1
Sets
Footballer
Status : Immutable Map {
- hunger: int
- hygiene: int
- thirst: int
- bladder: int
- mood : int }
+ increaseHunger(int): void
+ increaseHygiene(int): void
+ increaseThirst(int): void
+ increaseBladder(int): void
+ increaseMood(int): void
+ getHunger(): int
+ getHygiene(): int
+ getThirst(): int
+ getBladder(): int
+ getMood(): int
+ getStatus() : int
1
Uses
1
Uses
1
Uses
<<interface>> GUI
- displayWindow() : void
- enableMouseClicks() : void
- enableMouseTraverse() : void
- displayBackground(): void
1
Exits to
Main Menu GUI
- displayCharacter() : void
-displayButtons(): void
- displayStatusBars() : void
- updateStatusBarsVisually() : void
- displayEndGameWarnings() : void
- drinkingButtonHandler() : void
-urinateButtonHandler() : void
- showeringButtonHandler() : void
- eatingButtonHandler() : void
- miniGameButtonHandler() : void
1
Exits to
End Game GUI
- displaygameOver() : void
- displayrestart() : void
- displayButton() : void
- closeButtonHandler() : void
1
Calls
1
Event Manager
1
- decreaseStatus(): void
- deathController(): void
- startGame(): void
1

Click Here to view a more detailed version of the class diagram

Below is a detailed description of each class, interface, and enumeration that is present in our class diagram:

<<interface>> **GUI**: In our diagram, **Graphical User Interface (GUI)** is an interface classifier that provides the visual display/interactions for the various functionalities of our game.

The attributes of the **GUI** enable a basic interface that other classes of GUI can further customize. In this diagram, these are **Main Menu GUI, Minigame GUI**, **Choose Model GUI,** and **End Game GUI.** Regarding the attributes of **GUI**, *displayWindow() & displayBackground()* are operations to enable the visual display of the interface, and *enableMouseClick()* & *enableMouseTraverse()* allow for the interactions in the interface.

We acknowledge that introducing this interface may increase the coupling and dependencies across our system, as all GUIs will have to use it. On the other hand, implementing this **GUI** interface will provide a degree of generalization for the core graphic/interaction components across all GUIs - regardless of who develops them. As a result, we expect lower complexity, which is worth the trade-off.

Main Menu GUI: This class is built on the basic functionalities of the **GUI** and aims to provide a visual interface for the main menu of our game.

Main menu GUI has operators that provide displays such as *displayCharacter()* and *displayButtons().* The operator *updateStatusBarVisually()* makes sure to constantly show the most updated status values (i.e., hunger, sleep, etc.). Each button in the GUI also has an operator that handles the consequence of the respective button is pressed. These operators have the word "handler" in their name (Ie. *drinkingButtonHandler*()). Most handlers play a small animation and increase/decrease the relevant status. The operators are private. All operators are handled within this GUI without exposing anything to the user. In such a manner, dependencies are reduced, and if this class were removed, the impact on the overall system would be minimal.

There are multiple association relationships with the **main menu GUI**. Firstly the **main menu GUI** uses data from **footballer** to depict visual content (i.e., the model, the hunger bar level, etc.). It also modifies the data of the footballer when a button is pressed. There is also a 1 to 1 association relation with the **minigame**: the **main menu GUI** starts the **minigame,** while the **minigame** exits to the **main menu GUI**.

End Game GUI: The **end game GUI** is a class triggered when the Tamagotchi perishes due to one of the status values crossing a critical threshold. As described by the operators, *displayGameOver(), displayRestart(),* and *displayButton(),* the GUI will display a game-over indication with a button to nudge the user to close the game. The choice for making all the operators private in this class is the same as in the **main menu GUI** (i.e., to have lower complexity and dependency).

We asked the user to “close” the game rather than “restart” because this may have unnecessarily added extra dependencies between the **end game GUI** and the **event manager.** If the user wants to play again, they can restart the application. We believe the lowered complexity justifies our decision.

Choose Model GUI: The **Choose model GUI** is triggered at the very start of the game. As shown by its operators, *displayModels()* and *displayIrreversableChangeWarning(),* this class provides an interface to choose between the available models in the game. The GUI displays that the choice is irreversible until the game is over. The choice for making the operators private is the same as in the **main menu GUI** (i.e., to have lower complexity and dependency).

The **Choose Model GUI** contains the **Model** type, which is parsed and displayed as a selection option by the GUI (i.e., pictures of Ronaldo and Messi). There is also a uni-directional relationship from this class to the **footballer** class**,** as the **Choose Model GUI** uses the user’s model selection to set it in the **footballer** class. A similar association exists between this class and the **main menu GUI** as the former exits to the latter.

<<Enumeration>> Model: The **Model** datatype represents the various types of footballing models that the user can choose from as their Tamagotchi. At the current moment, there are two available choices: Cristiano Ronaldo and Lionel Messi. This class aims to add more types of models without causing a significant increase in complexity across the rest of the system.

Footballer: The **Footballer** class in this Tamagotchi-style game represents a virtual pet footballer that requires care and attention from the player. The well-being status of the footballer is present in an immutable map called *status,* which has five keys: *hunger, hygiene, thirst, bladder,* and *mood*. Each key allows for values that range from 0 to 100, with 0 representing a low level of the attribute and 100 representing a high level. We chose an immutable map to prevent any issues of escaping references.

Due to the immutable nature of the map, the **Footballer** class also has several public operators that allow the player to interact with and take care of the virtual pet. These methods include increasing the footballer's *hunger, hygiene, thirst, bladder,* and *mood* levels by a given amount. Additionally, there are methods for getting the current values of each attribute.

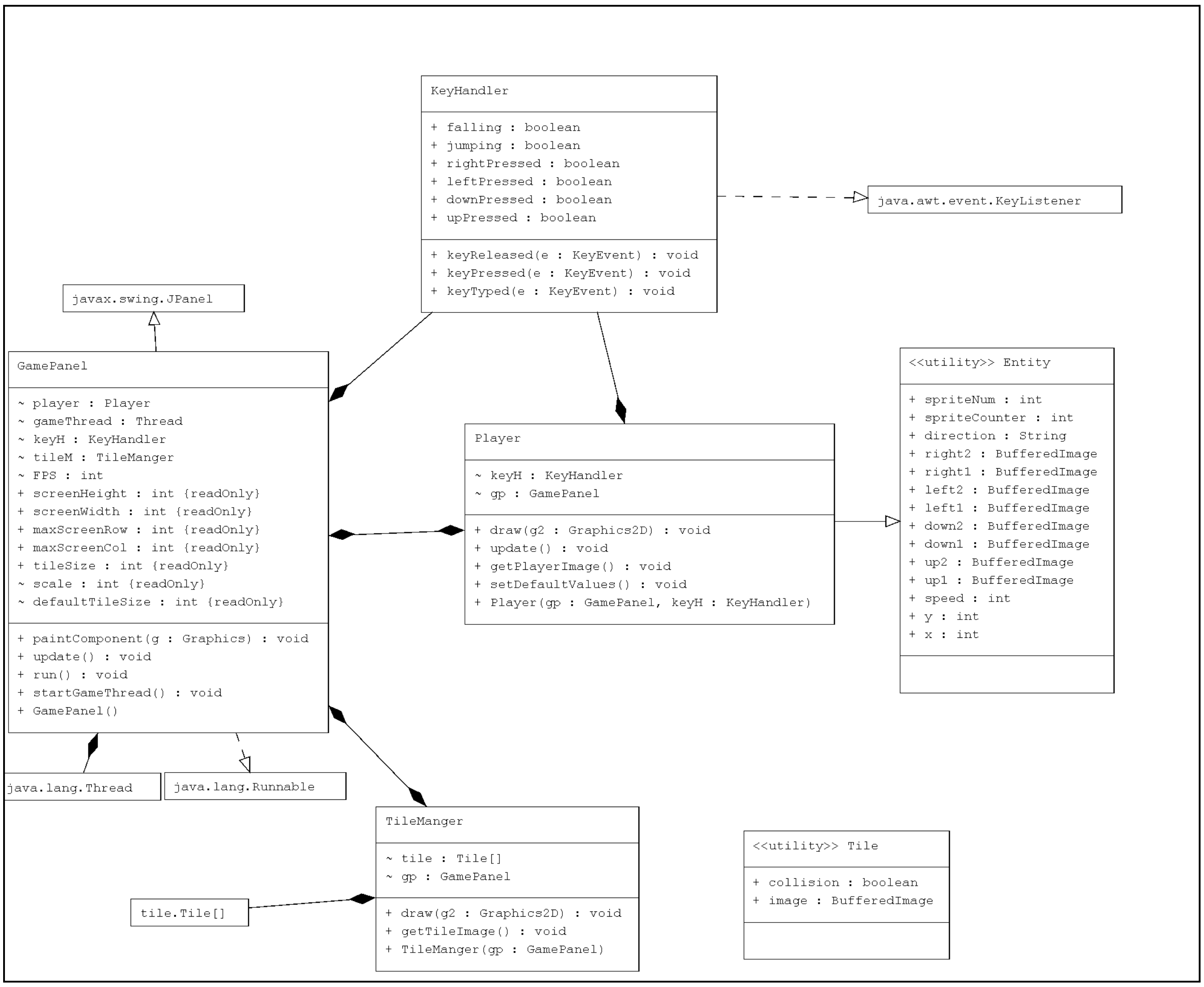
We chose to create this **Footballer class** based on the single responsibility principle. In this current setup, if one of the other classes were to fail, the user’s data would not be lost as it could be recovered from the footballer class. Subsequently, having this class allows us the freedom to modify GUIs or switch to a command-line input without many coupling issues.

Event Manager: The **Event Manager** allows the game to progress. The *decreaseStatus()* method continuously reduces the footballer's status (i.e., hunger, thirst, etc.) to ensure that the player maintains the Tamagotchi. The *deathController()* method checks if the footballer has died due to low stats and triggers the end-game GUI (as shown by the one-way association). The *startGame()* method initializes the game and triggers the start of the **Choose model GUI** (the UML diagram illustrates this via a one-way association). All operators are private, with the reasoning being the same as in the **main menu GUI** (i.e., to have lower complexity and dependency). The reason for creating this class is similar to the footballer class, with the basis heavily influenced by the single responsibility principle.

Server: The **server** class enables the multiplayer aspect of our game. Each server can connect players via connecting their **minigame** object. Each **server** instance has a unique ID (that is read-only). If a minigame wants to connect to the server, it can use the *connectToServer* operator and provide the respective ID. The *pushGameDetails* allows minigames to push the locations of their player in the game**.** Similarly, the *pullGameDetails* will allow the minigame to pull the locations of all the involved players in the game and parse it on the local player's screen. One minigame object can use only one server, while a server can use multiple minigame objects.

Minigame: In the above diagram, the **minigame** class is abstracted. This is done for two reasons: to reduce complexity in our class diagram and to illustrate our focus on having low dependencies across the system. In this current state, it is possible to have any type of minigame and be able to replace that minigame without impacting our overall system. We have added a detailed diagram for our specifically chosen minigame of head soccer below:

UML class diagram specific for the mini-game.



Click Here to view a more detailed version of the class diagram

This UML diagram represents the key classes and interfaces used in the implementation of the game. It shows how they are related and their attributes and functions.

The **javax.swing.JPanel** is a pre-existing class in the Java Swing library. The **GamePanel** class extends the pre-existing **javax.swing.JPanel** class in the Java Swing library. It provides a custom JPanel for our game, defining the visual layout and updating of game graphics.

The **java.awt.event.KeyListener** interface is used to handle keyboard events in Java. It provides methods such as *keyPressed*(e: KeyEvent), *keyReleased*(e: KeyEvent), and *keyTyped*(e: KeyEvent) that can be used to capture different types of keyboard events. **The KeyHandler** class implements the **KeyListener** interface and defines the behavior of the keys that the game should respond to. The boolean attributes defined in the **KeyHandler** class are used to monitor the state of different keyboard inputs, such as whether a key is pressed or not. The falling and jumping attributes are used to control the behavior of the player character, while the *rightPressed*, *leftPressed*, *downPressed*, and *upPressed* attributes are used to monitor the arrow keys on the keyboard.

The **Player** class is a custom class created in the game code to represent the player character. It contains several attributes that represent different properties of the player, such as its position, direction, speed, and sprite image. These attributes are defined within the class and can be accessed and modified by other parts of the game code. The *spriteNum* and *spriteCounter* attributes keep track of the current sprite being displayed for the player character. The direction attribute represents the current direction in which the player is facing. The *right1*, *right2*, *left1*, *left2*, *down1*, *down2*, *up1*, and *up2* attributes hold the different sprite images for the player character, depending on the direction it is facing. The speed attribute determines the rate at which the player character moves, and the x and y attributes represent its current position on the game screen. The class also contains several functions that handle the player character's movements and sprite display. The *draw(g2: Graphics2D)* function uses the provided *Graphics2D* object to draw the player character's current sprite image onto the game screen. The *update()* function updates the player character's position and sprite image according to its current direction and speed. The *getPlayerImage()* function returns the current sprite image for the player character. The *setDefaultValues()* function resets the player character's attributes to their default values.

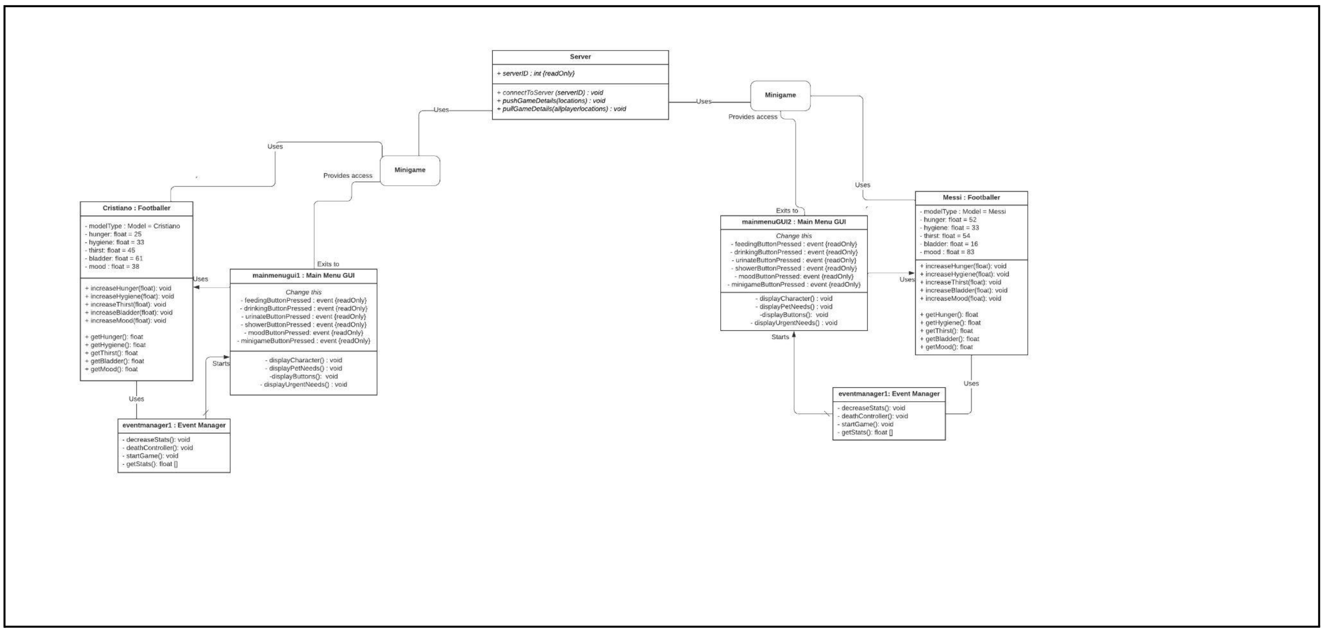
The **Thread** class in Java is a pre-existing class in the Java library that provides methods to create and control threads. When an instance of the **Thread** class is created, it creates a new thread of execution that can run in parallel with the main thread of the program. The **gameThread** in **GamePanel** is an instance of the Thread class that is responsible for running the game loop in a separate thread of execution. This represents the game loop and is a critical part of our game, responsible for updating the game state, rendering graphics, and handling user input. By running the game loop in a separate thread of execution, the program can continue to respond to user input and render graphics smoothly, even while the game state is being updated in the background.

TileManger is a class that manages the different tiles of the game. It has attributes such as *tile*, *gp*, and *FPS* that represent the tile graphics, the game panel, and the frames per second at which the game runs. The class has functions such as *draw(g2: Graphics2D)* and *getTileImage()* to handle tile graphics display.

The **Tile** class represents the individual tiles that make up the game environment. It has attributes such as collision and image, which represent the tile's collision status and graphic respectively. The class has no methods and is solely used to define the attributes of individual tiles.

Object diagram

Author(s): Full team


Server
+ serverID : int {readOnly}
+ connectToServer (serverID) : void
+ pushGameDetails(locations) : void
+ pullGameDetails(allplayerlocations) : void
Uses
Uses
Minigame
Provides access
Uses
Minigame
Provides access
Uses
Cristiano : Footballer
- modelType : Model = Cristiano
- hunger: float = 25
- hygiene: float = 33
- thirst: float = 45
- bladder: float = 61
- mood : float = 38
+ increaseHunger(float): void
+ increaseHygiene(float): void
+ increaseThirst(float): void
+ increaseBladder(float): void
+ increaseMood(float): void
+ getHunger(): float
+ getHygiene(): float
+ getThirst(): float
+ getBladder(): float
+ getMood(): float
Uses
Exits to
mainmenugui1 : Main Menu GUI
Change this
- feedingButtonPressed : event {readOnly}
- drinkingButtonPressed : event {readOnly}
- urinateButtonPressed : event {readOnly}
- showerButtonPressed : event {readOnly}
- moodButtonPressed: event {readOnly}
- minigameButtonPressed : event {readOnly}
- displayCharacter() : void
- displayPetNeeds() : void
-displayButtons(): void
- displayUrgentNeeds() : void
Starts
Uses
eventmanager1 : Event Manager
- decreaseStats(): void
- deathController(): void
- startGame(): void
- getStats(): float []
Exits to
mainmenuGUI2 : Main Menu GUI
Change this
- feedingButtonPressed : event {readOnly}
- drinkingButtonPressed : event {readOnly}
- urinateButtonPressed : event {readOnly}
- showerButtonPressed : event {readOnly}
- moodButtonPressed: event {readOnly}
- minigameButtonPressed : event {readOnly}
- displayCharacter() : void
- displayPetNeeds() : void
-displayButtons(): void
- displayUrgentNeeds() : void
Starts
Uses
Messi : Footballer
- modelType : Model = Messi
- hunger: float = 52
- hygiene: float = 33
- thirst: float = 54
- bladder: float = 16
- mood : float = 83
+ increaseHunger(float): void
+ increaseHygiene(float): void
+ increaseThirst(float): void
+ increaseBladder(float): void
+ increaseMood(float): void
+ getHunger(): float
+ getHygiene(): float
+ getThirst(): float
+ getBladder(): float
+ getMood(): float
Uses
eventmanager1: Event Manager
- decreaseStats(): void
- deathController(): void
- startGame(): void
- getStats(): float []

Click Here to view the diagram in more detail

This object diagram represents a snapshot of the multiplayer status of our system during its execution.

Currently, there are two players in the system. One player uses the Tamagotchi of Cristiano Ronaldo, instantiating the **Cristiano** object from the **Footballer** class. The other player has chosen to use the Tamagotchi of Lionel Messi, instantiating the **Messi** object.

Subsequently, each above-mentioned object also has its instances of gameplay objects. For example, **Cristiano** has its event manager (**eventmanager1**), main menu GUI (**mainmenuGUI1**), and minigame (**minigame1**). A similar situation is apparent in the **Messi** class.

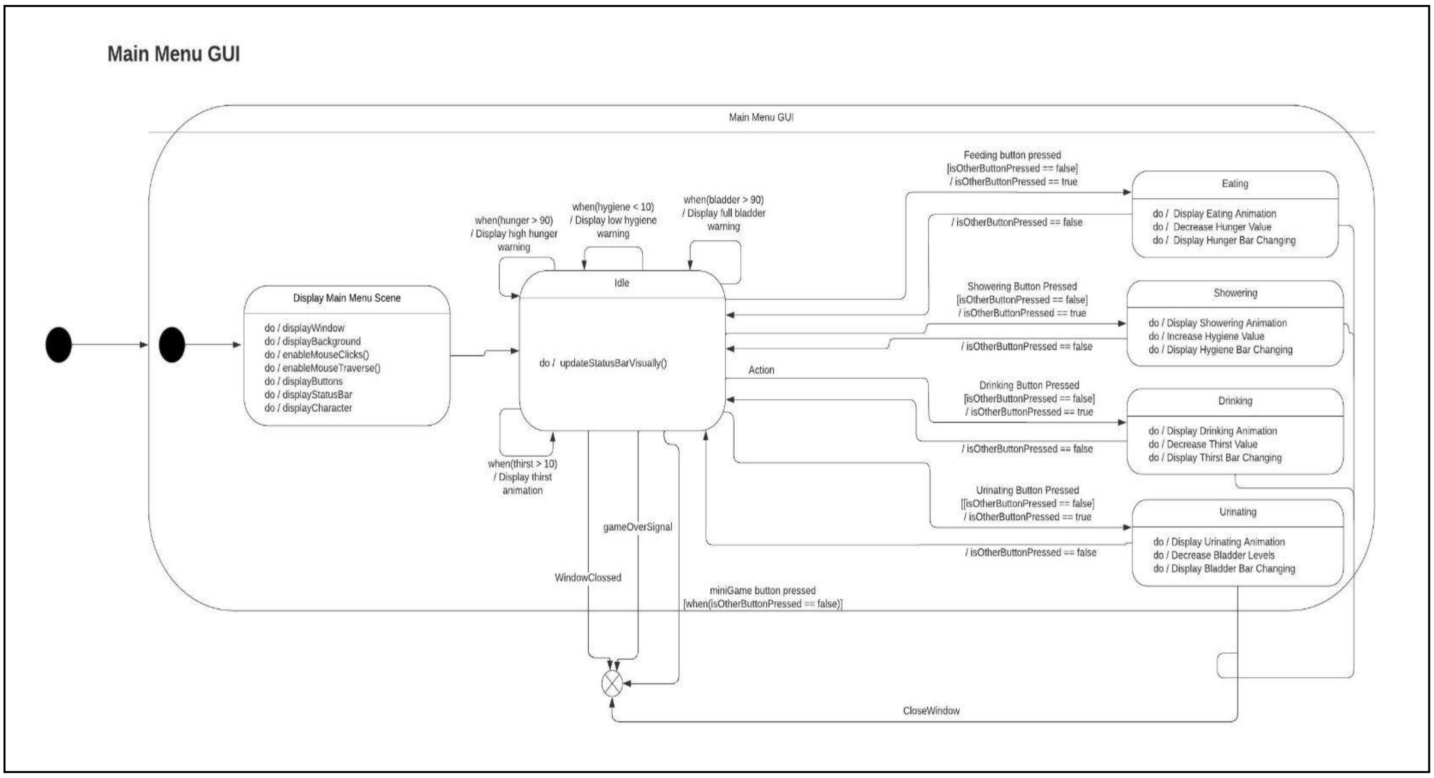
It may be argued that it is possible to merge all of the object instances that belong to **Cristiano** to create a deeper-level class. However, we have chosen not to follow this path. Rather we have built our objects on the single responsibility rule. This means that each object only has one responsibility. The benefit of this approach is that if one of the classes were to fail, the complete system would not be destroyed. For example, in this current situation, if **eventmanager1** were to fail, the user would still be able to retrieve data from their **Cristiano** object.

Consequently, as observed in the diagram, **server1** represents an instance of the **server** class. To facilitate multiplayer, **server1** connects the minigame objects from **Cristiano** and **Messi.** This is done by both minigame objects using the function *connectToServer(19)*, with the *serverID* - 19 - of **server1**. Through this manner, both users can play against each other.

State machine diagrams

Author(s): Full Team

Class: Main Menu GUI



Main Menu GUI
Main Menu GUI
Display Main Menu Scene
do / displayWindow
do / displayBackground
do / enableMouseClicks()
do / enableMouseTraverse()
do / displayButtons
do / displayStatusBar
do / displayCharacter
Idle
do / updateStatusBarVisually()
when(hunger > 90) / Display high hunger warning
when(hygiene < 10) / Display low hygiene warning
when(bladder > 90) / Display full bladder warning
when(thirst > 10) / Display thirst animation
Feeding button pressed [isOtherButtonPressed == false] / isOtherButtonPressed == true
/ isOtherButtonPressed == false
Eating
do / Display Eating Animation
do / Decrease Hunger Value
do / Display Hunger Bar Changing
Showering Button Pressed [isOtherButtonPressed == false] / isOtherButtonPressed == true
/ isOtherButtonPressed == false
Showering
do / Display Showering Animation
do / Increase Hygiene Value
do / Display Hygiene Bar Changing
Action
Drinking Button Pressed [isOtherButtonPressed == false] / isOtherButtonPressed == true
/ isOtherButtonPressed == false
Drinking
do / Display Drinking Animation
do / Decrease Thirst Value
do / Display Thirst Bar Changing
Urinating Button Pressed [[isOtherButtonPressed == false] / isOtherButtonPressed == true
/ isOtherButtonPressed == false
Urinating
do / Display Urinating Animation
do / Decrease Bladder Levels
do / Display Bladder Bar Changing
gameOverSignal
WindowClossed
miniGame button pressed [when(isOtherButtonPressed == false)]
CloseWindow

Click Here to view the diagram in more details

The state machine for this class starts with entering the **Display Main Menu Scene** state. Inside this nested state, seven internal activities allow for the eventual interactive display of the main menu. We believe that the names of these activities can provide an apt representation of their function without requiring additional details.

When these activities are completed, the machine reaches an **idle** state. The only activity occurring in this state is *updateStatusBarVisually,* which continuously provides visual updates to all the status bars (hunger, hygiene, etc.). As shown in the diagram, if any status value reaches a critical threshold, a warning sign (ie. *Display high hunger warning*) is shown to the user to take care of their footballer urgently. This warning sign remains to display until the user takes care of their Tamagotchi.

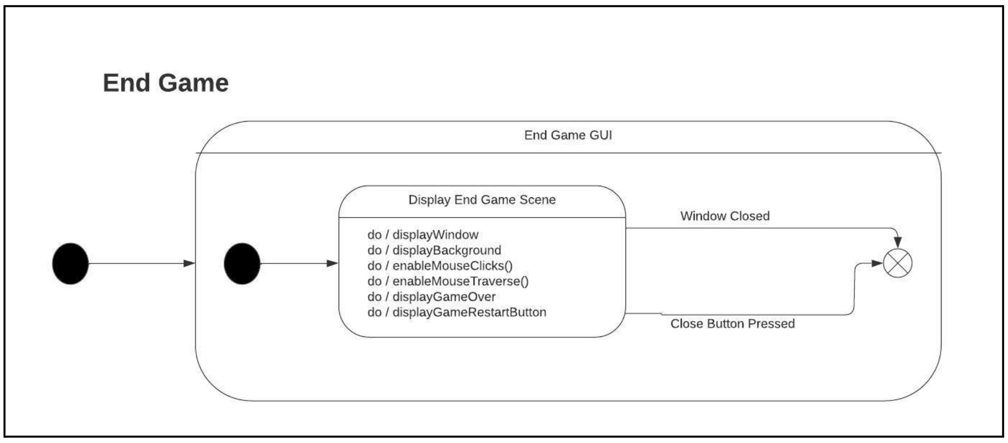
Suppose the user were to not interact with the system. In that case, we will remain in the **idle** state until the **event manager** sends a game-over signal, hence initiating the *gameOverSignal* event. When this occurs, the complete state machine will be terminated. Another way of terminating the machine state is via the event *Window Closed* (closing the window).

If users want to feed their Tamagotchi, they can press the feeding button, triggering an event to transition to the **Eating** state. The event is guarded by a boolean, *isOtherButtonPressed,* that checks if any other button has been pressed. If no button has been pressed, then *isOtherButtonPressed* is set to “True.” This means that if the user clicks on any other button during this phase, it will yield no change. We added this function to prevent users from double spamming on buttons or any other similar cheating tricks.

Once the guard passes, the state transition happens, and we enter the **Eating** state. The **Eating** state is temporary and consists of the following activities in sequential order: displaying an eating animation, decreasing the hunger status, and showing a change in the hunger bar. Once the activities are completed, the state returns to **idle**, while *isOtherButtonPressed* is false. The same pattern as **Eating** occurs for **Showering**, **Drinking,** and **Urinating**. It is important to observe that in all these states, it is possible to altogether terminate the machine state by closing the window.

Finally, pressing the mini-game button will also trigger a state transition to terminate the machine state on the guard condition that *isOtherButtonPressed* is false.

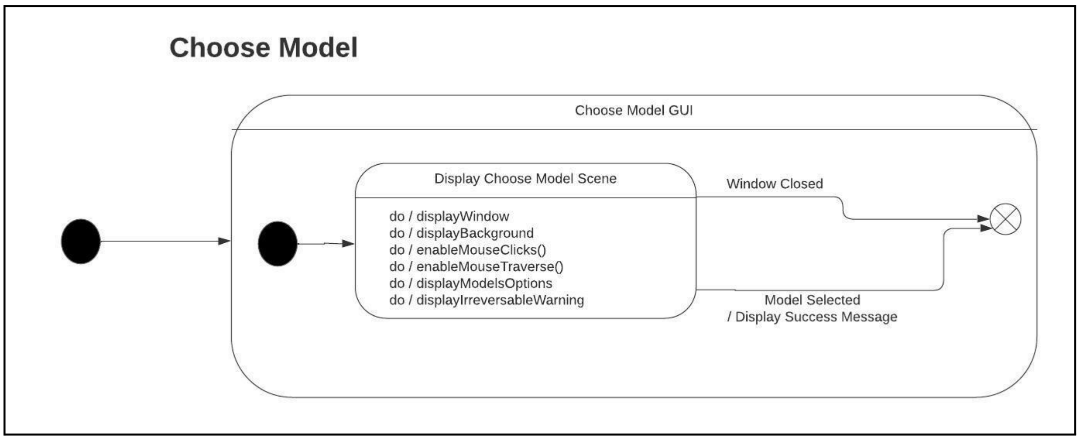
Class: End Game GUI



Click Here to view the diagram in more details

For this class, we first enter the nested state, **Display End Game Scene.** This state lets the user know the game is over through its visual display. Also, this state displays a button in the interface asking the user to close the game. If the button is pressed, the state machine is terminated. Another way the state machine can be terminated is by closing the display window.

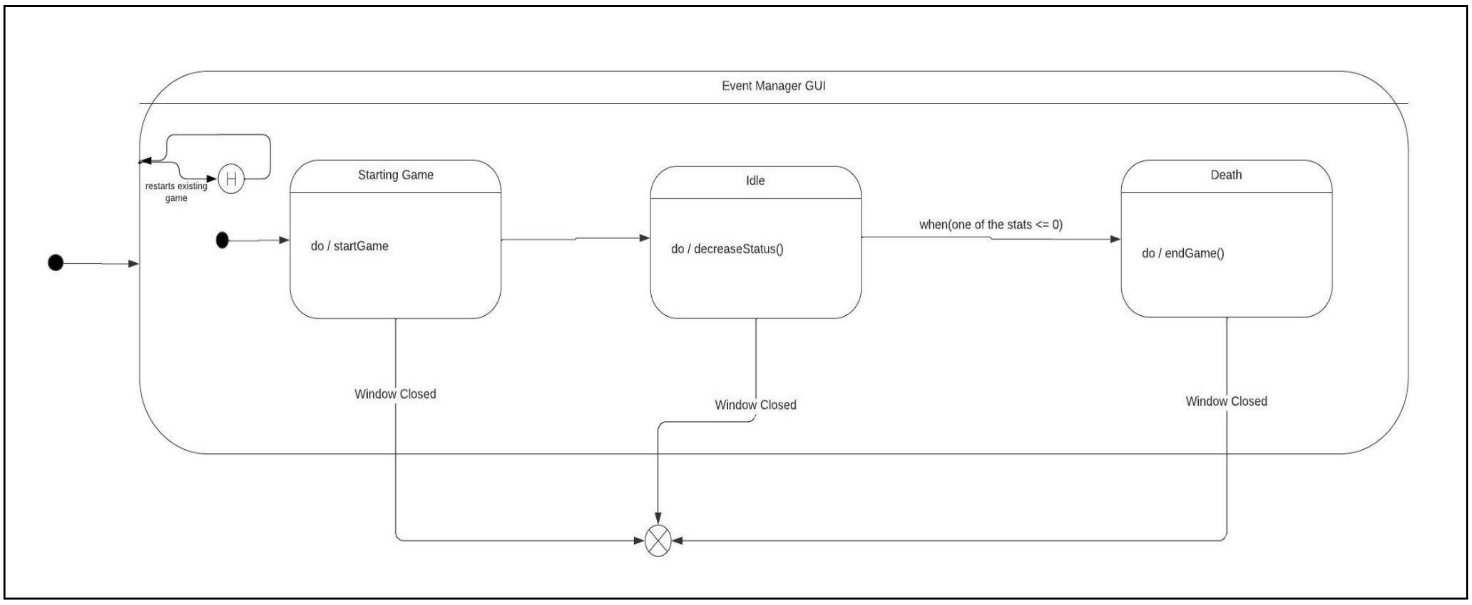
Class: Choose Model GUI


Choose Model
Choose Model GUI
Display Choose Model Scene
do / displayWindow
do / displayBackground
do / enableMouseClicks()
do / enableMouseTraverse()
do / displayModelsOptions
do / displayIrreversableWarning
Window Closed
Model Selected
/ Display Success Message

Click Here to view the diagram in more details

For this class, we also use a nested state similar to the above**.** However, the **Display Choose Model Scene** displays the varying models used for this game and requests a choice from the user. It also warns the user that this choice is irreversible. Once the choice is made by pressing the “select” button, the player receives a “success” message on their screen, and the state machine is terminated. Another way the state machine can be terminated is by closing the display window.

Class: Event Manager GUI



Click Here to view the diagram in more details

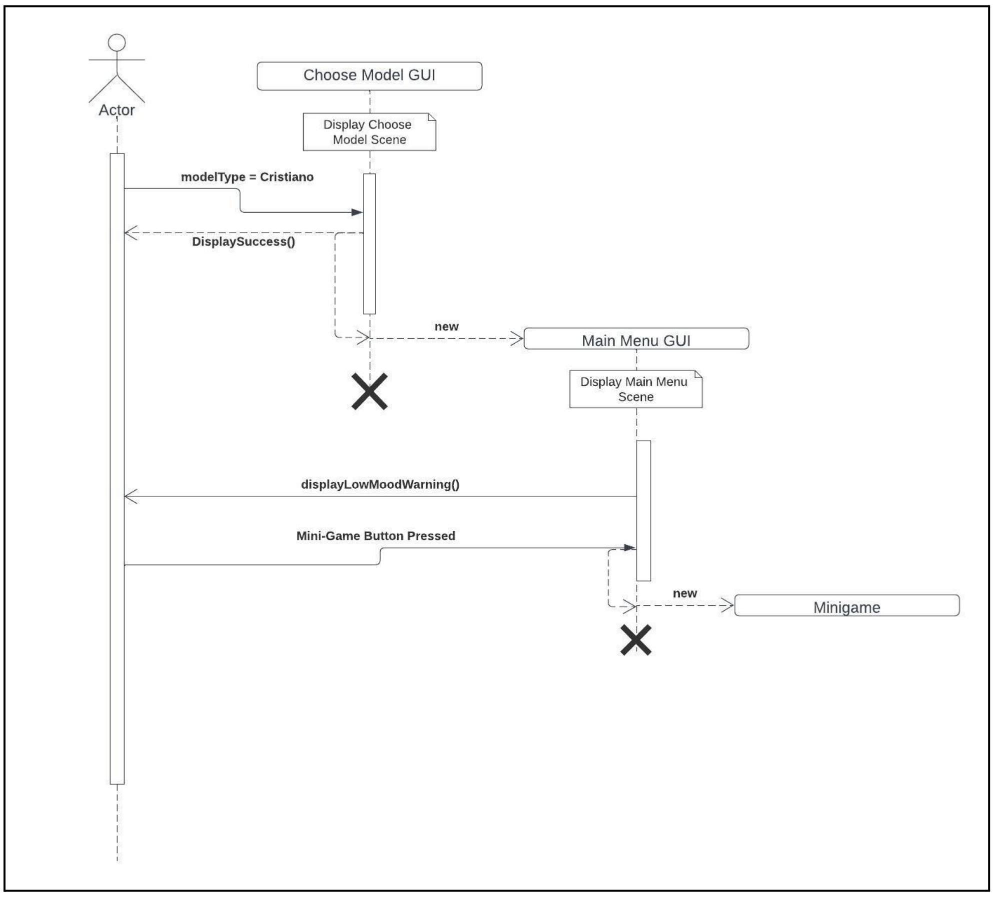
The event “*restart existing game”* occurs when the user restarts a previous version of their game. In such a scenario, the history state is triggered. This means that suppose the state machine had last terminated at the **idle** state, then the event manager will restart at that **idle** state.

If there is no such history, the first state is the **starting game** state. In this state, the game starts by launching the GUI associated with choosing the model. After this, the event manager will go into the **Idle** state and continuously perform the *decreaseStatus* activity, which reduces the status values (i.e., hunger, thirst, etc.) of the Tamagotchi. When one of the status values is below or equal to 0, the event manager will switch to the **Death** state, which handles the death mechanic of the player by exiting any running GUIs and launching the **Game Over GUI**. The **event manager** state machine can only be terminated when the player quits the game by closing the window.

Sequence diagrams

Author(s): Full Team

Player's interaction from starting our game till choosing to start the mini-game.



Click Here to view the diagram in more details

The above diagram represents the player starting our game till the point at which they start the min-game.

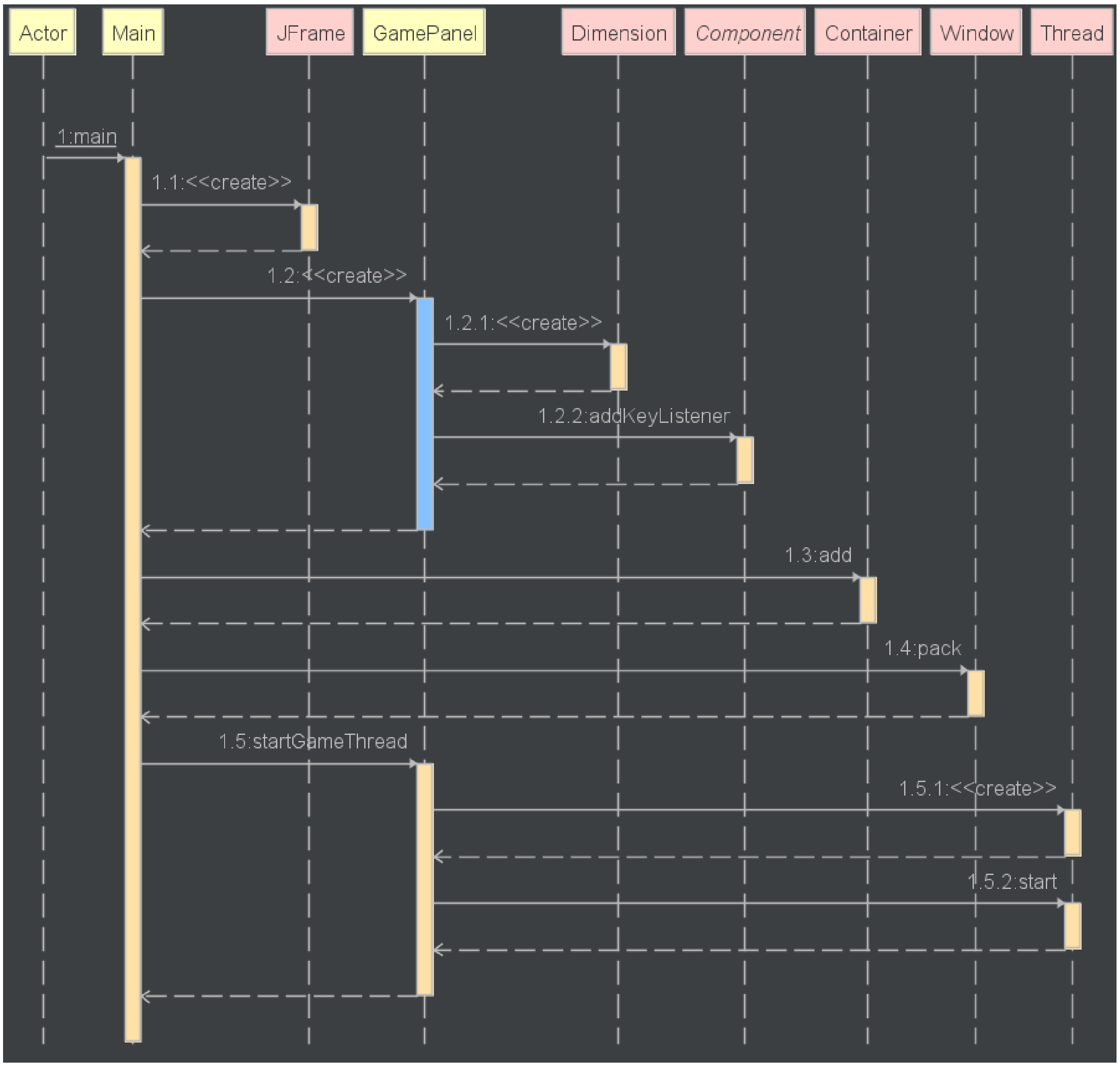
When the player - modeled via the **actor** partner - first starts our game, they must interact with the **Choose Model GUI** partner. In the lifetime of the **Choose model GUI**, it first goes through the state invariant, **"display choose model scene**." We believe the title of this invariant aptly describes its functions. Nevertheless, more information can be found in the above section.

The player must now carefully look at the scene and choose the model they see fit. In this example, the **actor** chooses Cristiano Ronaldo by clicking on the picture of him displayed in the GUI. This sends a relevant message to the **Choose Model GUI** partner - as shown by the message, *modelType = Crisitano*. In response, the GUI provides visual confirmation to the player. Another response by this interaction partner is that it creates a new partner, **Main Menu GUI.** Consequently, after this creation, the **Choose Model GUI** terminates.

The **Main Menu GUI** first goes through a state invariant of **Display Main Menu Scene.** This state is described in more detail in the section above. After this, the user can look at the scene and decide what action to take. In this diagram, the planner initially takes no action. As a result, the mood of the Tamagotchi decreases to a critical level. This triggers the GUI to send an asynchronous warning to the player, to which the player can continue without responding to the message.

Suppose the warning concerns the player; they can then press the “Minigame button,” which will then send a signal to the **main menu GUI**. The response will be to start the minigame and terminate itself instantly. In such a way, the player goes from the start of the game to the minigame.

The process of starting the mini-game.



The above sequence diagram shows the interactions between components and partners involved in the game-starting process. The main **actor** partner first triggers the **main** partner as represented by *1:main*. For now, this acts as the driver function of our game. Consequently, the **main** creates the **JFrame** partner, whose function is described in detail in the class diagram. The **main** partner also creates a **GamePanel** partner, as represented by *1.2<create>>*. This **GamePanel** partner provides the display scene for the minigame.

To establish the **GamePanel** box, a **Dimension** partner is created via *1.2.1:<<create>>* that models the dimension of the scene. The **Component** box (responsible for showing various components such as the button) is subsequently connected to the **GamePanel** box through *1.2.2:addKeyListener*.

The **Container** box, which provides vital debugging support by containing data instances, is added to the **Main** partner through the add function represented by *1.3:add.* This way, the **main** partner can provide more accurate debugging detail. The **Window** box (ie. provides the option to close and minimize the scene) is also added to the **GamePanel** box through the same add function.

Then, the **startGameThread** represented by *1.5:startGameThread* is initiated. This creates a new **Thread** partner shown by *1.5.1:<<create>>*. **Thread** is responsible for all subsequent logical and visual functionings of the game. The choice of deviating to a thread is so that if the thread fails, we can restart it in the abovementioned process. On the other hand, if everything was inside the main, a failure of it could result in the game crashing.

Time logs

SD: Time Logs