Assignment 3

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

Format:

- The class, state and interactive partner are **bold**
- The attribute, method, activities, and messages are in *italics*

Summary of changes from Assignment 2

Author(s): Entire Team

We made the following changes based on the feedback given from Assignment 2:

Class diagram:

- Added multiplicity and labels to associations that were missing.
- For some associations (i.e. containment), we added some descriptions about the multiplicity.
- For some types of classes (i.e. coarse-grained), we provided some more descriptions about them.
- We re-structured our class diagram to more accurately depict our implementation of the system.
- We added design patterns and technical implementations to our class diagrams.
- We added a note at the beginning talking about some of our modeling language choices to prevent any miscommunication.

Object diagram:

- We provided more detailed descriptions of the integration of the class diagram and object diagram.
- We have a more comprehensive description of the state of the object diagram.
- We added design patterns and technical implementations to our class diagrams.
- We re-designed our object diagram to more accurately depict a snapshot of the implementation of our system.
- The object diagram represents design patterns and technical implementations.
- We added a note at the end talking about some of our modeling language choices to prevent any miscommunication.

State-machine diagram:

- Removed one state-machine diagram.
- Removed the initial state inside the composite state diagram.
- Made the diagram more representative of our implementation and design pattern.

Sequence diagram:

- Re-designed the structure of the sequence diagram.
- Elaborated on the mini-game interactive partner.
- Fixed some logic errors that were mentioned by the TA.
- Added various fragments (loops, opt, etc.).

Overall, we took note of the comments raised and adjusted the diagrams as needed.

Application of design patterns

Author(s): Entire Team

Design patterns provide a blueprint for solving typical problems that occur in code. While implementing the code, it was our intention to use our understanding of design patterns to reduce the complexity of our code and improve performance.

	DP1
Design pattern	Command
Problem	- The GUI components and their respective behavior were strongly coupled - Each button in the GUI implemented its own behavior. This posed the following problems: - *Increased complexity*: Any modification to the style of a button would expose clients to its corresponding behavior and vice versa. This increases complexity by exposing the clients to unnecessary information. - *Reduced Modularity:* Deleting the button would delete the corresponding behaviour. This was a problem, especially if we wanted to keep the behavior, but enable it via another GUI component (i.e. text input). - *Increased redundancy*: There were a lot of redundancies. In our GUI, pressing most buttons performed a similar behavior with minor variation (i.e., changing one of the status bars). As such, changing the behavior would require us to modify it through all buttons individually.
Solution - How the DP solves the problems	- *Reduces complexity:* The command pattern allows us to separate the invoker/executer (i.e. UI buttons) from the underlying behavior (i.e. increaseStatusBars). In this manner, clients looking to change the style of the UI buttons will not be exposed to unnecessary code about the behavior. - *Increased Modularity:* This is also in accordance with the single responsibility principle (a module should be responsible to one and only one actor). We can now delete the buttons, and the behavior continues to exist. In such a way, we can now add another manner for the behavior to enable itself (i.e. text input) without much hassle. - *Improves maintainability and reduces redundancy:* We aggregated many buttons with similar behavior to a single command. As a result, redundancy is reduced, and modification of the respective behavior requires little change.
Intended use	For each interactable GUI component (i.e. buttons), an appropriate command object is instantiated and paired with the object. The pairing is done via the "*actionListener*" method of the GUI component, such that in the event of the component being clicked, it calls the "*execute*" method of the command object.
Constraints	- *Increased Performance Overhead*: Since each GUI component object also is paired with its command object, there is increased performance overhead. However, since there are few GUI components, we believe the benefits outweigh this cost.

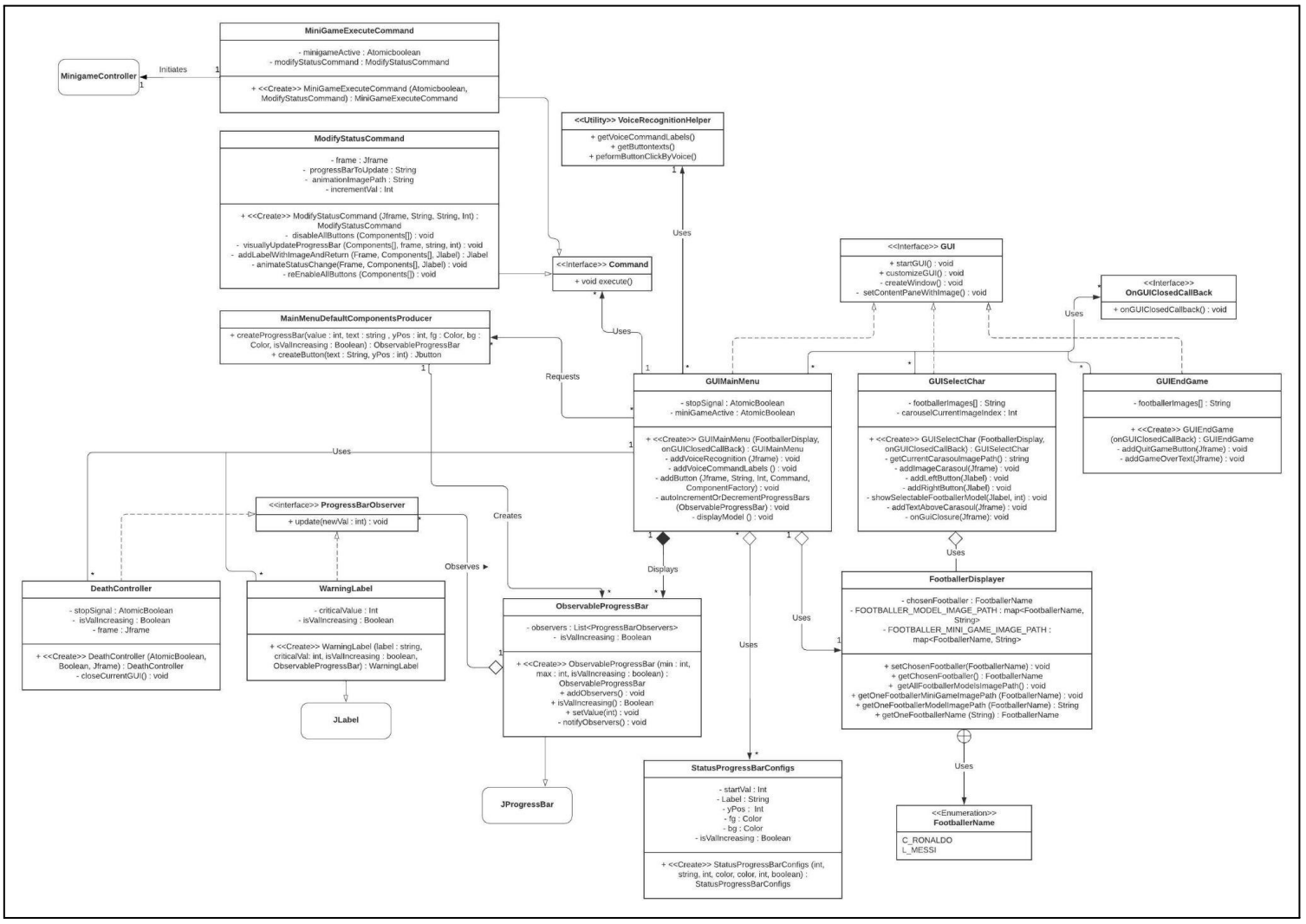
	DP2
Design pattern	Observer
Problem	- We wanted to add a mechanism such that when our progress bars that conveyed status (i.e. hunger, mood, etc.) eclipsed a critical value, a warning sign would appear on the screen. Furthermore, the game would be over if the user ignored this warning and the progress bar reached a certain "game-over" value. - Our initial solution was to let the **GUIMainMenu** be responsible for periodically pulling the values of the progress bar and when appropriate, adding the warning label or ending the game. This implementation had multiple issues: - *Violation of Single Responsibility Rule:* The **GUIMainMenu** should only be responsible for one thing; adding the aforementioned behavior violates these rules, making the code more complex, less modular, and harder to maintain. - *Tight Coupling:* Due to the increased dependencies between the progress bar and **GUIMainMenu**. if the progress bar is changed, it would require substantial changes to all these components. - *Scalability concerns:* The **GUIMainMenu** has to poll the ProgressBars constantly. As such, if the number of progress bars was to increase, bottleneck situations in performance could occur.
Solution	- The observer design pattern solves these issues as follows: - *Adherence To Single Responsibility Rule:* Separate observer classes are now

	responsible for monitoring the progress bars and taking appropriate actions. The **GUIMainMenu** is now mainly responsible for setting up the GUI components, adhering to the single principle rule, and making the code more modular. - *Lower Dependencies:* As long as the progress bar implements the required interface for this design pattern, it can modify itself without any repercussion to the observers or to the **GUIMainMenu**. A similar situation is true for modifications to the observers. - *Ease of scalability:* Whenever one of the progress bars is modified, its observers are notified and can handle the respective change. As such, there is no need for constant polling, and we can scale up our features easily.
Intended use	Whenever a progress bar object is created for the GUI, observers' objects - one for the warning label and the other for ending the game - are instantiated and attached to the progress bar object. The observers are notified whenever the value of the progress bar changes. Consequently, the observers take their respective actions based on the notification.
Constraints	- *Performance overheads:* Similar to the command design pattern, it is a possibility that the instanciation of new objects for each progress bar could lead to the possibility of increased performance overhead. Nevertheless, we believe the possibility of this happening in the current GUI context is low as there are not that many progress bars. Furthermore, a greater performance risk is bottlenecks caused by the initial implementation. Hence, we believe the benefits outweigh its costs. - *Memory leaks*: Since the progress bars keep track of the observers by reference, it is a possibility that memory may be leaked. However, this constraint can be overcome through careful management in the addition and removal of observers.

Revised class diagram

Author(s): Entire Team

Due to the addition of the design patterns, we have significantly restructured our class diagram in contrast to assignment 2. Since we have changed a majority of the structure of the diagram, we have opted not to highlight the changes made as that would affect legibility:



Note: The tiny dots throughout the diagram are multiplicities. We have used multiplicities in all associations. Please view this link to see the diagram in more detail and zoom in to see the **"*"**, if required.

Before explaining each class, below are some explanations on some common design conventions we adopted:

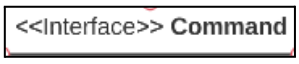
- The constructor is denoted by a *<<Create>>* modifier as per the conventions of UML.
- In the specific situation where the constructor accepts arguments used to assign all the attributes in a class, we have omitted the exact variable names from the constructor's parameters.
- 

: A white rectangle with rounded corners represents a coarse-grained class. This is illustrated to represent a granular level of detail of the class. They are used as conventions to show libraries, and as done in assignment 2, we will not elaborate significantly on each library represented with a coarse-grained class. However, whenever we use such notation for other purposes, we will describe why we used them as such.
- 

: We have omitted the multiplicity from the containment relationship as per the rules of UML; these relationships are strictly 1 to 1.
- The **Main** class has been omitted from the UML diagram as instructed by the course coordinator. Furthermore, responsibilities are extremely limited to invoking one GUI after the other.
- 

: The small black triangle next to the label indicates the direction the label can be read as.

To prevent repetition, we will not explain the reasoning behind **Command, ModifyStatusCommand,** and **MiniGameExecuteCommand** classes as we have explained in the previous section about design patterns.



: The **command** interface requires that the *execute()* behavior be implemented by all its implementors. Each **command** object is used by **GUIMainMenu**, which pairs it up with it's GUI component. Hence the reason for the directional association from that object to this one.

: The **ModifyStatusCommand** class implements the command interface. As it's name suggests, this command modifies the status of the Tamagotchi by increasing/decreasing the status-related progress bars. It's constructor is used to assign values to all the attributes. The key method, *visuallyUpdateProgressBar* works via scouring the *frame* for the *progressBarToUpdate.* When the bar is found, it adds *incrementVal* to it. During the process of progress bar modification, three other methods are also called *disableAllButtons(), reEnableAllButtons(),* and *animateStatusChange()*.

: The **MiniGameExecuteCommand** also implements the command interface. This command aims to execute the minigame and update the "mood" status bar. Its *execute()* method consists of initializing the Minigame, which is consequently depicted in the classes' 1-1 association with **Minigame**. In addition, this class stops the auto changing of the status progress bars by modifying the atomic boolean, *isMiniGameActive,* which - due to the nature of the atomic boolean - informs all processes that share this boolean.

: The **MiniGameController** is responsible for controlling everything to do with the minigame. It is coarse-grained to illustrate our focus on having low dependencies across the system. In our current system, it is possible to have any type of minigame and replace the MiniGameController with another controller without impacting our overall system. Furthermore, we have done this to reduce the complexity in the above diagram as we describe the minigame system in further detail later.

: The **VoiceRecognitionHelper** is a utility class that contains a few methods that are used to aid in any class using voice recognition. It is currently used by the **GUIMainMenu** as indicated by **GUIMainMenu's** directed association with this class. It's creation is justified as it increases modularity and reduces the amount of code present in **GUIMainMenu,** hence reducing the overall complexity.

: The **MainMenuDefaultComponentsProducer** will produce a templated main menu component (i.e. **ObservableProgressBar**) that is of a default style. This producer is necessary to ensure consistent style across all components and provide greater modularity to the code. A single producer can create multiple **ObservableProgressBar** as illustrated by its one-many directed association.

The following classes **ObservableProgressBar, DeathController**, **WarningLabel, ProgressbarObservers** are manifestations of the Observer design pattern. As such, their reasoning has not been described.

: The **observableProgressBar** inherits the normal properties of the Java progress bar, **JProgressBar**; however, it also implements the methods and attributes of the generic "*subject abstract class*" that is present in this observer design pattern - (i.e. *notifyObservers(), addObservers(), etc.)*. It is important to note that we did not create the subject class. Our type of implementation of this design pattern is called the "interface" implementation and is done to bypass the multi-inheritance restriction in Java.

setValue() is a method inherited from its superclass; however, we have modified it to *notifyObserbers()* when this value is set changed. The class is a composite of the **GUIMain**, while the **GUIMain** can be associated with multiple progress bars. It is important to note that **observableProgressBar** is just one component of many GUI components. It is explicitly modeled in this diagram, and the object diagram as significant modifications have occurred to it from the standard **JProgressBar**. In all other scenarios, components with heavy associations to external libraries are not explicitly modeled without reason, as they unnecessarily clutter up this system.

: The **ProgressBarObserver** is an interface that observes the progress bar. Its *update()* method has a parameter that requires the new value set in the **observableProgressBar.** It has an aggregate relationship with the **observableProgressBar,** as is the norm in this design pattern. The multiplicity of this relationship illustrates that one **observableProgressBar** can have multiple observers.

: The first type of concrete observer is responsible for displaying a warning sign when it observes the value of the **observableProgressBar** crossing over a *criticalValue.* It inherits the Jlabel and uses the **GUIMainMenu** to display the warning. The **GUIMainMenu** also uses the WarningLabel to attach it to **observableProgressBar.** Hence a *uses* relationship is depicted between the two objects. The *label* in the Constructor Parameter is used to set the warning label, and the respective **observableProgressBar** is required to craft the label stylistically.

DeathController: The **DeathController** is the second type of concrete observer. Depending on *isValIncreasing,* the end game values are either 0 or 100. So, when the controller observes these values have been reached, it triggers the *closeCurrentGUI*() method that disposes of the current frame and set's the *stopSignal* to true. As this is of type atomic boolean that is shared with other processes associated with the **GUIMainMenu,** all of them terminate themselves.

StatusProgressBarConfigs: The **StatusProgressBarConfigs** was created to provide greater modularity It also allows decoupling as the configurations of the status bar can easily be used by other GUIComponents if required. We will not delve into the attributes and methods of this object as we believe it is self-explanatory. This class has an aggregate relationship with the **GUIMainMenu,** which uses the attributes of this class to request **MainMenuDefaultComponentsProducer** to produce an **observableProgressBar.**

<<Enumeration>>
FootballerName: This enum is the same as the model class in assignment 2. We have renamed it and it's enumerations to provide greater clarity. To prevent repetition, we will not further elaborate on this enum.

FootballerDisplayer: The **FootballerDisplay** class keeps track of the chosen footballer and displays any images that relate to the footballer. Its various methods help GUIs with displaying the footballer models. The class ultimately decouples the models from the system, hence allowing for extremely easy extensibility. To add more footballing models to this class, we have to modify just this class, and the rest of the system will adjust accordingly. More specially, the two main attributes are *FOOTBALLER_MODEL_IMAGE_PATH* and *FOOTBALLER_MINI_GAME_IMAGE_PATH.* These attributes keep track of all the possible models for the various modes, and it is from these attributes that much of displaying occurs. To add more models, users can simply modify these attributes.

<<Interface>>
OnGUIClosedCallBack This interface has a simple default method that the invoker of the GUIs can implement and consequently pass into the constructors of GUIs. Consequently, this method is used as a callback function to inform the invoker that the respective GUI has closed. As observed by the directed association towards this class, all GUIs in this diagram use this callback function.

For the **GUI, GUIMain, GUISelectChar,** and **GUIEndGame**, remain structurally the same as assignment 2; however, with minor changes that have been described below.

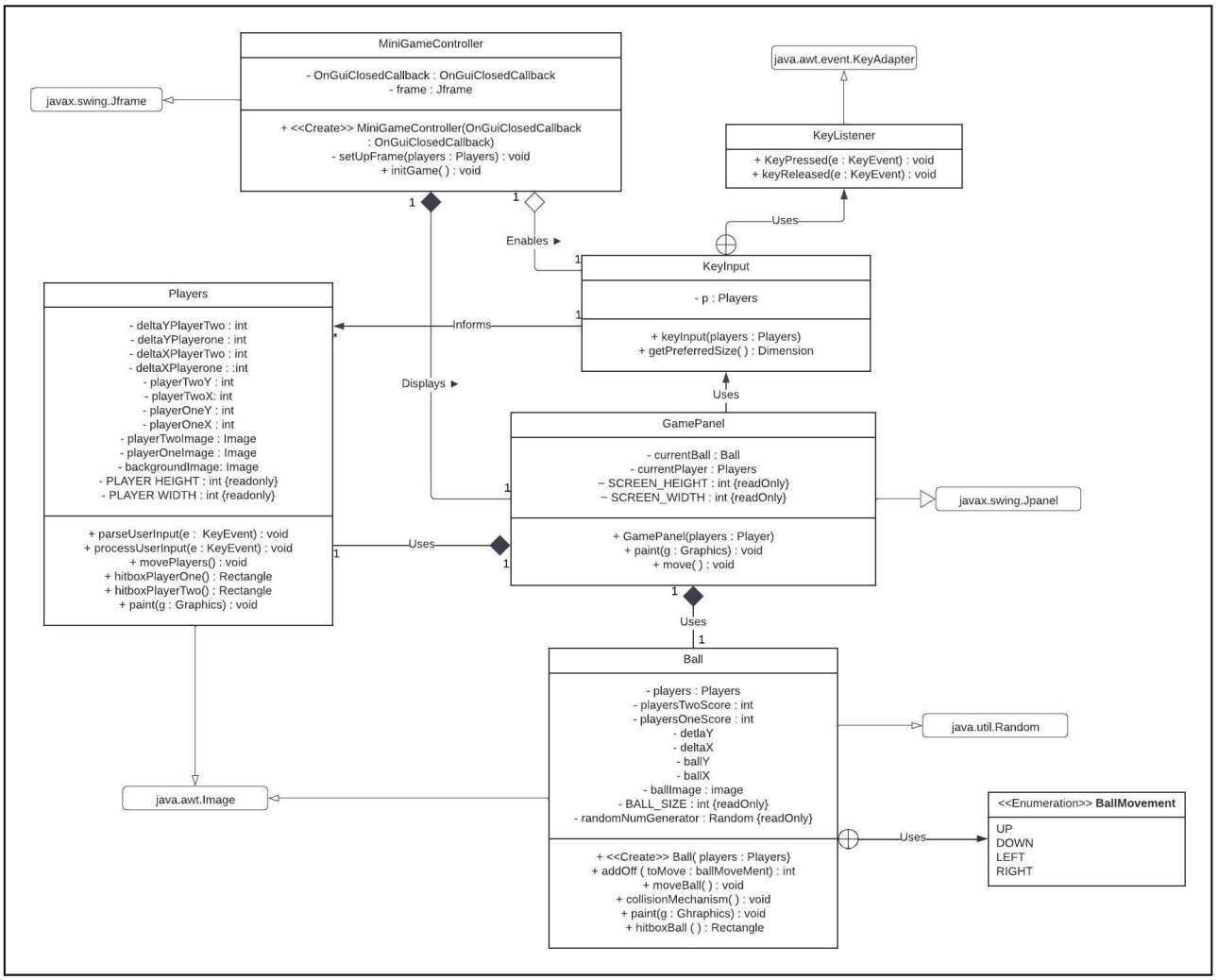
<<interface>> GUI: The interface now contains a default method, *startGUI()*, which is made to be the only function required to start and run GUIs. It contains code to setup up a basic window with a background image alongside the *customizeGUI()*. The *customizeGUI()* requires implementation. For each GUI class, clients can put all their code inside *customizeGUI()*, which will be run to produce the customized GUI. All other functions in the GUI can just be helper functions invoked by *customizeGUI().* Consequently, since *customizeGUI()* is part of *startGUI()*, when *startGUI()* is executed, it launches the full GUI in accordance with our initial intention.

This way of implementation reduces complexity, provides great modularity, and increases scalability. Furthermore, since the window and background are fixed, there is now a possibility for a universal manner in which GUIs can start up and present themselves. The downside is that GUIs, such as that of the minigame that wants to stand alone (as they need more dynamic visuals), will be unable to implement this interface. Nevertheless, we envision that this interface will make extending other Tamagotchi functionalities (i.e., shop) easier.

GUIMainMenu: The purpose and implementation of this class remain the same as Assignment 2. However, the attributes and methods are slightly modified in the technical sense to fulfill the GUI purpose, as the diagrams of assignment 2 were more theoretical. One method that is noticeable in its addition and was not there in the previous assignment is the *addVoiceRecognition().* To prevent overcrowding in the GUIMainMenu class, we have omitted two methods from the diagram that are present as a utility to *addVoiceRecognition(),* but cannot be moved to a class of their own due to the nature of the voice recognition object.

GUISelectChar GUIEndGame: Both classes have the same purpose and implementation as in Assignment 2. However, the attributes and methods are slightly modified in the technical sense to fulfill the GUI purpose, as the diagrams of assignment 2 were more theoretical.

Revised class diagram for the MiniGame:



Please click here to view the image in more details

MiniGameController: The **MiniGameController** class, which serves as the game's principal entry point, is at the system's heart. It creates the game objects, such as the **KeyInput**, and **GamePanel**, and sets up the game window using a **JFrame** object (using *setUpFrame())*. As a parameter, the MiniGameController class accepts an **OnGuiClosedCallback** function, which allows a callback to be performed when the game window is closed. This is a solid design decision since it allows the game to be canceled cleanly, any resources to be correctly disposed of, and the game to return to the correct state before the commencement of the minigame.

Players: The **Players** class represents the game's two players and manages their movement and detection collisions (*collisionMechanism()*) with the ball. It uses **Image** objects to portray the players and the game's background. The class is informed on events by the **KeyInput** class, which processes user input. This relationship is shown through the directional association between the two classes. Separating the user input processing logic from the movement and collision detection logic is an excellent design decision here since it provides better separation of concerns and makes the code more modular and easier to maintain.

KeyInput: The **KeyInput** class is in charge of handling key input. It is intended to parse key events. It uses the **KeyListener** to detect the inputs (hence the directional association with it) and consequently processes them to be informed to **Player**.

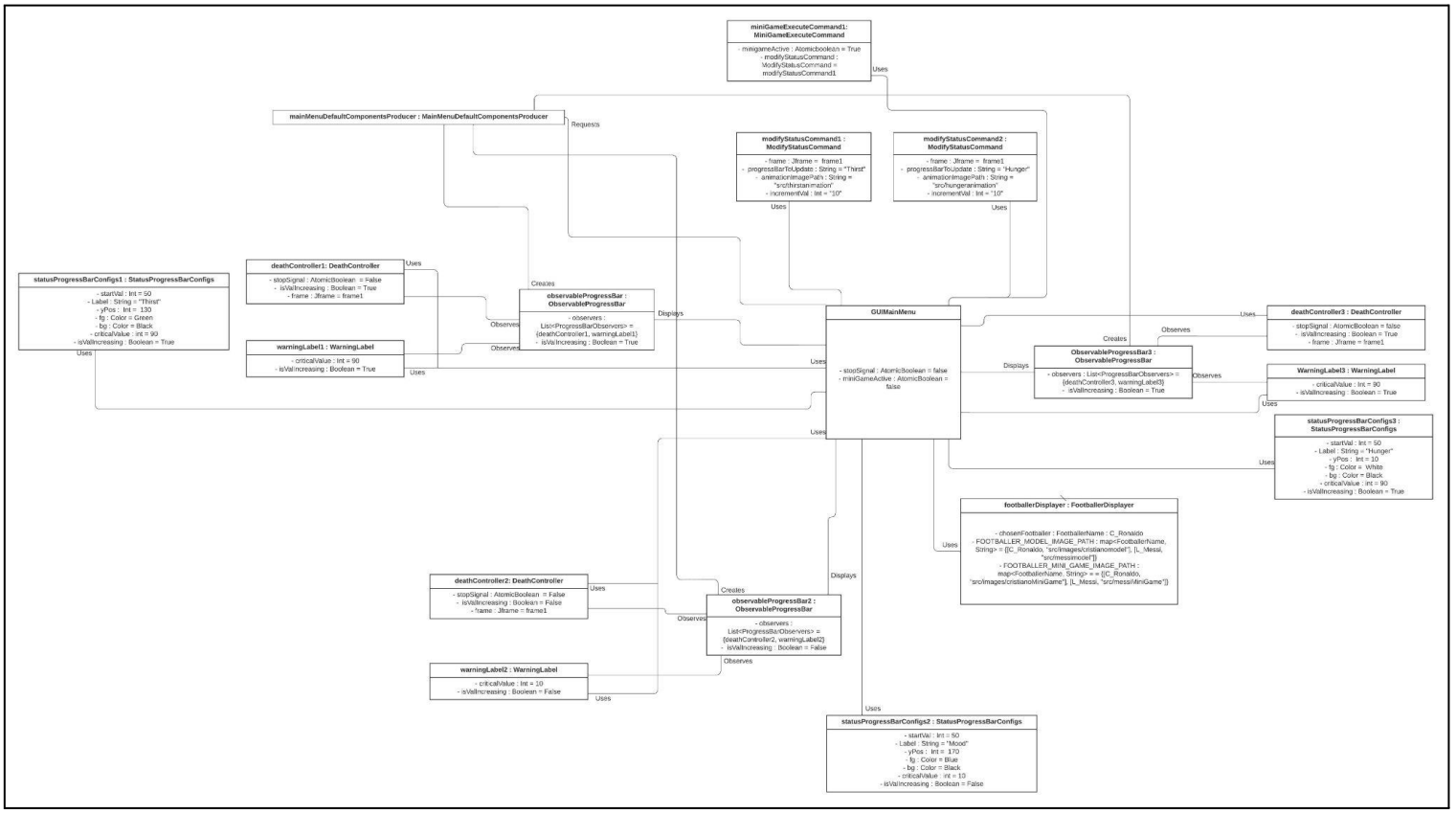
GamePanel The **GamePanel** class is in charge of rendering game items on the screen and handling the game loop. Through *paint()* and *move()* It renders the players and ball using the **Players** and **Ball** objects, respectively. A timer and a delay value are used to implement the game loop. This is an excellent design decision because it ensures that the game plays smoothly and is not influenced by system performance changes.

Ball: The **Ball** class represents the ball in the game and manages its movement, and detects collisions with the players. It checks for collision detection using the **Players** object *(collisionMechanism())*. Encapsulating the ball movement logic within the Ball class itself is a great design decision here, as it means that the code is modular and easier to maintain.

Revised object diagram

Author(s): Entire Team

Due to the addition of the design patterns, we have significantly restructured our object diagram in contrast to assignment 2. Since we have changed a majority of the structure of the diagram, we have opted not to highlight the changes made as that would affect legibility:



Please click [this link] to view the image in more detail

The above object diagram is a snapshot of the **GUIMainMenu** in its idle state.

The **GUIMainMenu** instance uses the **footballerDisplayer** object to visually represent the footballer (i.e. FootballerName) chosen by the user.

There are three instances of **StatusProgressBarConfigs.** Each instance has different attributes as they represent different statuses (i.e. varying labels, Ypos, fg, bg, etc.). The attributes of the **StatusProgressBarConfigs** objects are used by the **GUIMainMenu** object to request **mainMenuDefaultComponentsProducer** to create three respective **ObservableProgressBar**. In reality, the **GUIMainMenu** should have five such instances; however, we have not modeled all five instances due to limited space and to avoid repetition.

Consequently, each instance of the **ObservableProgressBar** contains a list of its observers. In this situation, all instances have two observers, a **deathController** object and a **warningLabel** object.

The **GUIMainMenu** object also uses objects of the **ModifyStatusCommand** and **MiniGameExecuteCommand.** Its use has been described in greater detail in the class diagrams. As observed, each instance of **ModifyStatusCommand** varies slightly in its attributes (i.e. *animationPath* and *progressBarToUpdate*). There should be four instances of **ModifyStatusCommand**; however, two have been omitted due to space optimization.

As described in the class diagram, the **MiniGameExecuteCommand** also instantiates the **MinigameController.** However, as mentioned in the feedback, including this in the object diagram does not add much value as it is a coarse-grained class. Furthermore, **MinigameController** does not fit the context of this object diagram snapshot very well.

Finally, below are some explanations of some common design conventions we adopted:

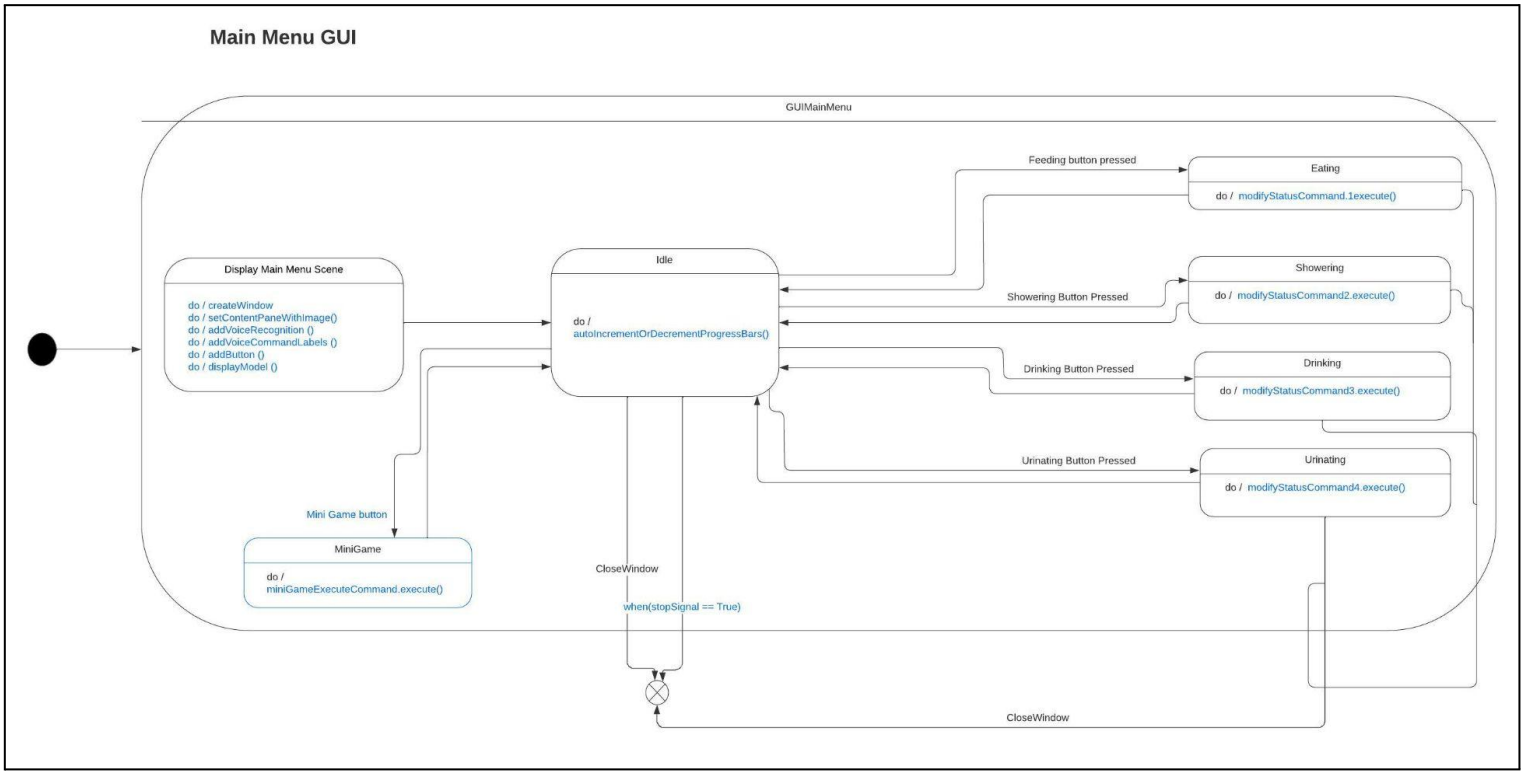
- We removed the methods in the object diagram and depicted them using a straight rectangular box. This was done in accordance with the professor's examples (Week 2, Slide 5).
- All associations were converted to links. The labels present in each association of the class diagram were transferred to the object diagram similarly to Assignment 2, as approved by the TA.
- Since VoiceRecognitionHelper is a utility function, it has not been implemented as it goes against the principle of instantiations that is present in the object diagram.

Revised state machine diagrams

Author(s): Entire Team

We have highlighted the changes made to the state diagram in blue.

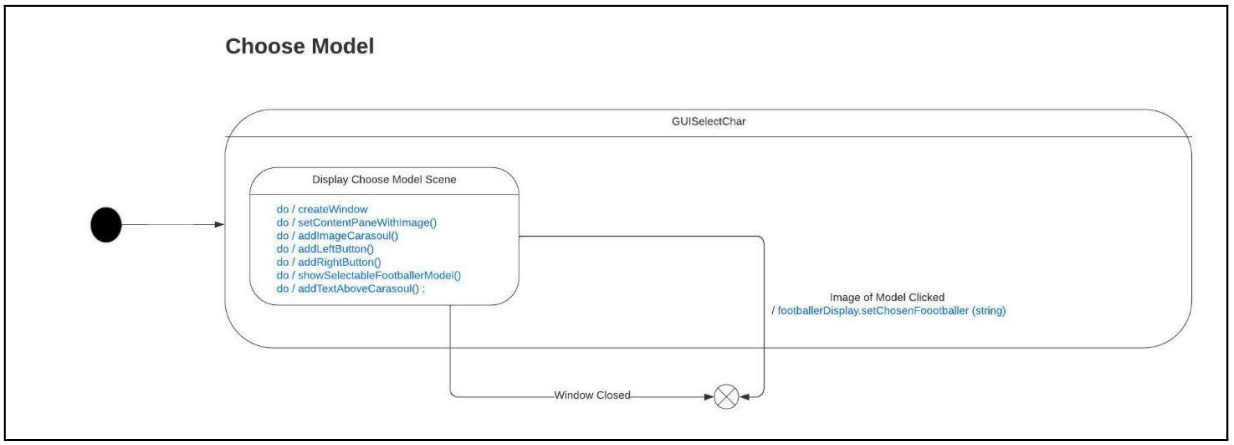
Class: GUISelectChar



Please click this link to view the diagram in more detail.

The initial state from within in the composite of **GUISelectChar** has been removed. The rest of the diagram remains similar apart from minor changes. The names of the internal activities of the **Display Choose Model Scene** have been renamed to the functions that are currently used to display the respective scene. As is the convention with state machine diagrams, we have removed the parameters and type of the functions; to solely display the behaviors carried out in the state. Finally, if the user clicks on the model's image, the activity during the transition is renamed in the technical sense to the function executed in our code.

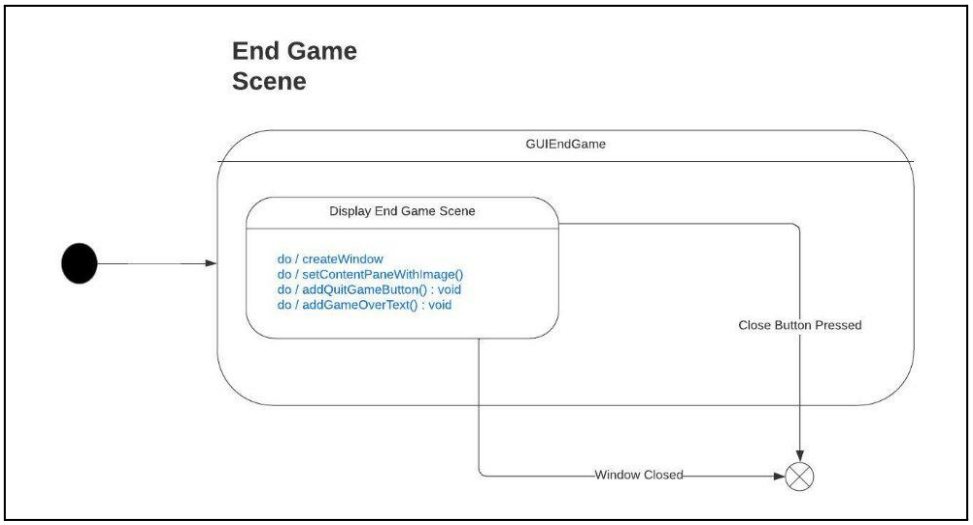
Class: GUIEndGame



Please click this link to view the diagram in more detail.

The initial state from within in the composite of **GUIEndGame** has been removed. All the names of the internal activities of the **Display End Game Scene** have been renamed to the functions that are currently used to display the respective scene. This is the only change from the previous diagram.

Class: GUIMainMenu



Please click this link to view the diagram in more detail.

Initial state from within in the composite of the **GUIMainMenu** has been removed. The only big change for the **Display Main Menu Scene** and **Idle states** was renaming the internal activities towards the functions currently executing the behaviors. We have excluded the active voice recognition tool from this state machine diagram as the tool pairs with external libraries and runs in concurrency with the **GUIMainMenu**. As such we believe it is not necessary to portray it in this diagram as its active running has not much to do with this class.

The addition of the observer design pattern has decoupled the previous behavior of the **GUIMainMenu** of polling the status bars and providing warning updates. This behavior now belongs to the observers; as such, it has been removed from our code (see the DP section for the reasoning).

In the previous version of this diagram, certain activities during the transition would be carried out upon pressing any button. Consequently, each state of the button press had its behavior. Using the command pattern, all these individual behaviors have been decoupled from the **GUIMainMenu.** They are now encapsulated into their own command (see the DP section for the reasoning).

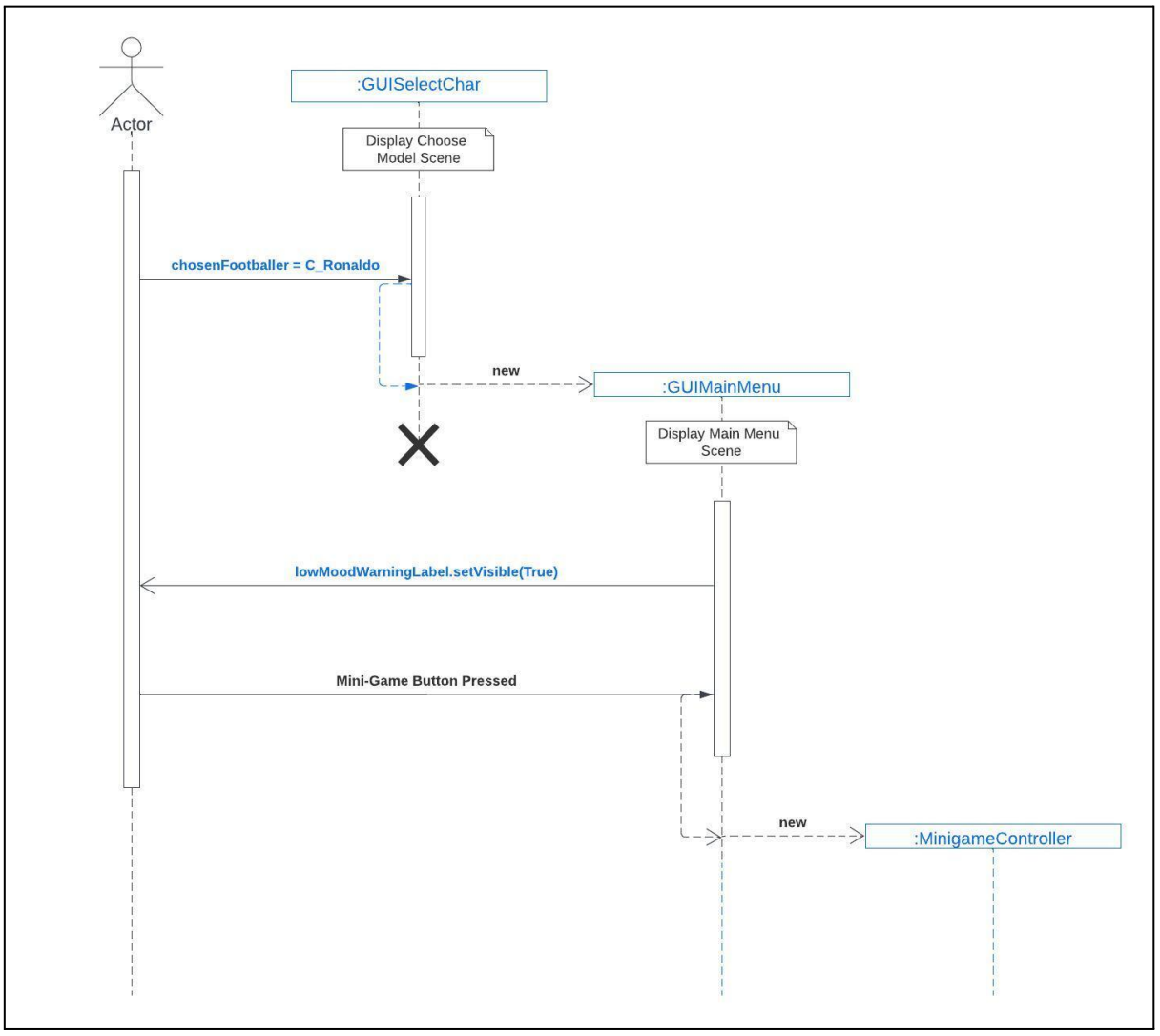
As observed pressing any of the feedings, showering, drinking, or urinating button, directly leads to a state upon which an instance of the **ModifyStatusCommand** is executed. As mentioned in previous sections, the command for each status is similar in premises but varies slightly, hence the need for different command objects. Once the command is executed, we return back to the **idle** state.

Furthermore, in line with the command pattern, it is now the case that pressing the Mini game Button leads to a state that is responsible for executing an instance of **MiniGameExecuteCommand.** Once the execution is finished, we return to the **idle** state. The final change made from Assignment 2 is the renaming of the event that causes the termination of the state machine (from Game Over Event to when(stopSignal = True)).

Revised sequence diagrams

Author(s): Entire Team

Player's interaction from starting our game till choosing to start the mini-game.
The changes have been illustrated in blue.



Please have a look at the link here to view the diagram in more detail

The primary change has occurred in renaming the messages between interaction partners more appropriately toward what are similar names in our code. We have also changed the layout (we now use straight rectangles) and presentation of the interaction partners, as shown by their blue outline and text color.

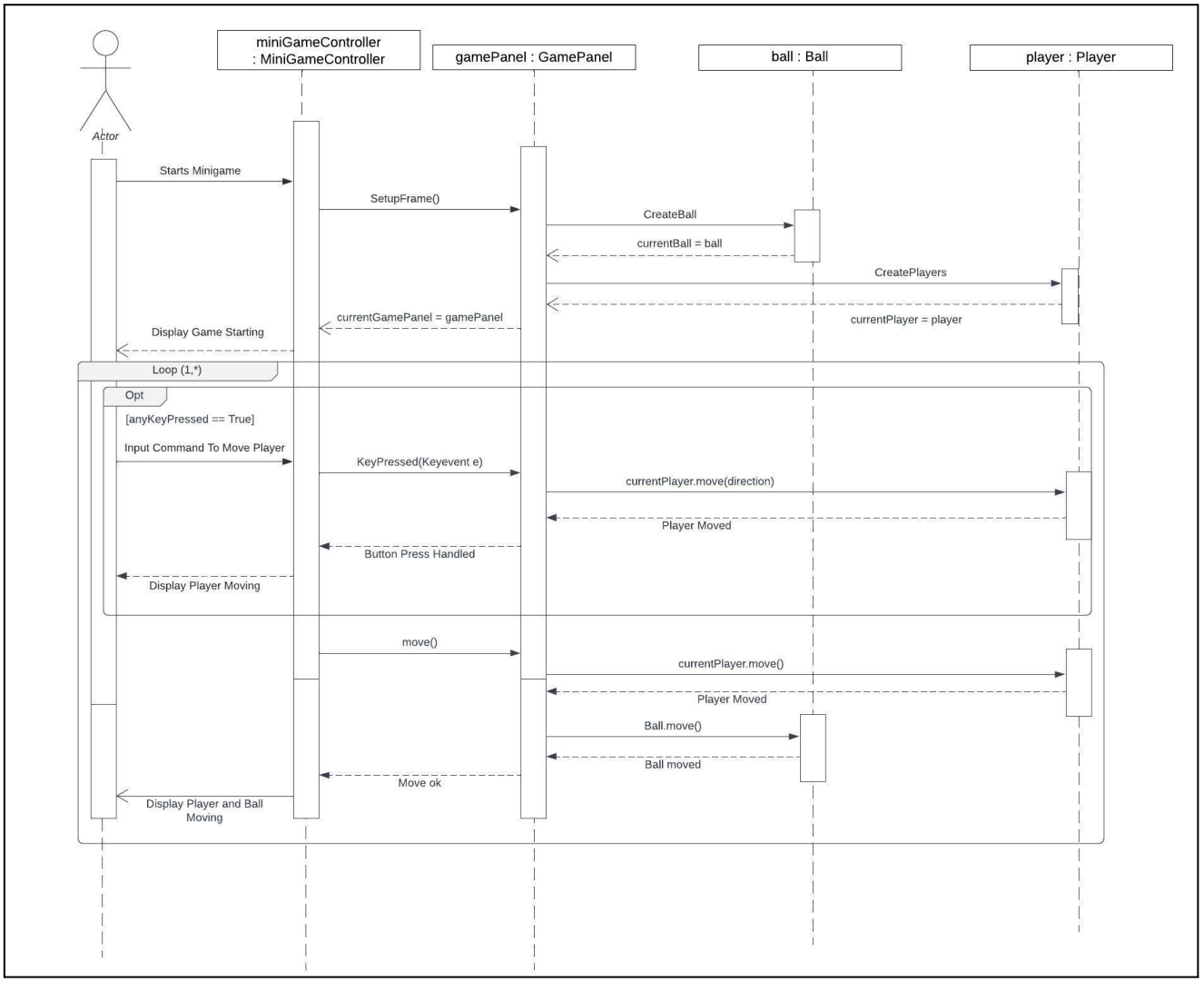
Furthermore, due to the high-paced nature of the game, we have removed a display of success when the **actor** chooses their footballer. On the contrary, the window just closes, and the consequent window of the main menu opens up to the **actor**.

Also, when the user presses the mine-game button in the updated sequence diagram, the **GUIMainMenu** does not terminate as was initially the case. On the contrary, it invokes the mini-game and then runs in the background. The reason for this was purely performance-based, as the new setup is substantially quicker.

Finally, the feedback from assignment 2 was to show a continuation of the diagram once the **MinigameController** started. As such, we have extended the lifelines of the interactive partners by adding the extra dotted lines in blue. In addition, we did not place any activations on these extended lifelines. By convention, this illustrates that although the interactive partner (i.e., **actor**, **Minigamecontroller**, and **mainmenuGUI**) continue to exist, their subsequent interactions are not captured in this diagram. This is indeed the intention of this sequence diagram, with the aim being to only show until the point of starting the mini-game.

Players interact with starting the mini-game and playing it.

Due to the addition of the design patterns, we have significantly restructured this sequence diagram in contrast to assignment 2. Since we have changed a majority of the structure of the diagram, we have opted not to highlight the changes made as that would affect legibility:



Please have a look at the link here to view the diagram in more detail

It is important to note that the GUIMainMenu is also running in the background during this whole process. However, we have omitted it as quite irrelevant to this scene and does not affect it whatsoever.

Once the actor starts the game, the **MiniGameController** object initiates the **GamePanel**, which initiates the **Ball** and Player **objects**. The minigame then launches the **GamePanel**, informing the **actor** that the minigame has started via the display.

After starting the game, we enter the “running phase” of the game by entering a loop fragment. The label "1,*" of the loop indicates that the **actor’s** interactions with the game will continue indefinitely until a specified condition is fulfilled, such as the **actor** abandoning the game or reaching a certain score; however, these “end-game” conditions are not illustrated in the of this diagram as it is outside the scope that we have chosen to depict.

During the loop, **MiniGameController** calls the **GamePanel** object to move() the game. Which, in turn, invokes the move() function in the **Player** object to make it move automatically (without the actor’s input). Similarly, the **Ball** object's move () causes its position to change depending on its current direction and velocity. During this process, the **Ball** object checks for collisions with the walls and the **Player** object and responds accordingly. The **GamePanel** then confirms the respective movements, and the **MiniGameController** displays these changes to the **actor** by making them visible through the **GamePanel**. This is an asynchronous communication, as the **actor** did not specifically request this action. The incorporation of collision detection and handling in the sequence diagram is a key design decision since it enables for a more realistic and dynamic gameplay experience. It also necessitates careful consideration of the game's object position and movement and the logic for handling collisions and updating the score.

An opt (optional) label within the loop reflects the user's decision to input a key command to move the player. If the user enters a command, the **MiniGameController** captures it and presents it to the **GamePanel**, that parses it and tells the **Player** to move in a certain direction. Confirmation responses are then cascaded back to the **MiniGameController** object. The **MiniGameController** displays the video game player moving to the actor by making them visible through the **GamePanel**.

Implementation

Author(s): Entire Team

- As we moved from UML models to the actual implementation of our code, we did it in a methodical and comprehensive manner. The stages required in this tactic were as follows:
  - ➢ **Examine UML diagrams**: In order to better understand the structure and behavior of the system, we thoroughly analyzed all the UML diagrams, including class diagrams, sequence diagrams, and other pertinent ones. The initial stage served as our development process's building block.
  - ➢ **Recognize classes and methods**: By thoroughly comprehending the UML diagrams, we were able to identify the classes and methods required to build the system. In this stage, classes and methods that matched those shown in the UML diagrams had to be created.
  - ➢ **Implement the classes and methods**: After identifying the classes and methods, we started putting them into practice in our code. In this procedure, the necessary constructors, accessor and mutator methods, instance variables, and other key methods were created.
  - ➢ **Use proper design patterns:** We used the right design patterns to enable scalable and maintainable interactions between classes. For instance, to create a one-to-many relationship between objects, we used the Observer pattern. The **ObservableProgressBar** class and its observers, **DeathController** and **WarningLabel**, are clear examples of this pattern in our code.
  - ➢ Using this proactive approach, we were able to go smoothly from the conceptual stage, which was represented by UML diagrams, to the actual implementation of our code, creating a system that is cohesive and extendable.
- Below are the key solutions that we applied when implementing our system:
  - For both the *pauseAutoUpdateProgressBar* and the *stopSignal*, we used AtomicBoolean variables. We ensured operations on boolean values were atomic (thread-safe) by utilizing the AtomicBoolean class from the java.util.concurrent.atomic package. This method eliminates the possibility of race situations or other concurrency-related problems by allowing safe access to and change of these variables by different threads.
  - The CMU Sphinx library, an open-source speech recognition system that enables real-time processing of speech, was used to create voice recognition. We started the **LiveSpeechRecognizer** object with the required parameters and used a ScheduledExecutorService to run the speech recognition process simultaneously with the main thread, guaranteeing smooth performance. When a voice command is recognized by our system, it is compared to the available commands (button texts) and, if a match is made, the associated action is carried out. Users may now engage with the program using their voice, making it more user-friendly and entertaining.
  - We can manage, show and extend several footballers thanks to the **FootballerDisplayer** class. In order to show the chosen footballer model or mini-game picture, it keeps track of footballer names and the image paths that correlate to those names.
    Any footballer may be added to this class by simply adding their name to the FootballerName enum and updating the maps for the *FOOTBALLER MODEL IMAGE PATH* and *FOOTBALLER MINI GAME IMAGE PATH* with the matching image files. This modular approach allows for the smooth integration of new footballers, allowing the mainMenuGUI to show them without requiring any substantial modifications to the current code.

- The location of the main Java class needed for executing your system in our source code is as follows:
  - "src/main/java/softwaredesign/Main.java"

- The location of the Jar file for directly executing your system:
  - "out/software_design_vu.jar"
  - The out folder also includes an "images" folder containing all the images needed by the jar file

- Youtube link:
  - https://youtu.be/N5BQY9ajWvM

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

Maximum number of pages for this section: 4

Time logs

https://docs.google.com/spreadsheets/d/1htiVsp2OwhPHPZsh4VCSY6qxfuasAF0i66pce39OsG0/edit#gid=1523319035