University of Liège

INFO2051-1 : Object-oriented programming on mobile devices

Bob Jump

Authors :
Nadir BOUNAR s193980
Louis HOGGE s192814
Raphaël ROMOLI s182115

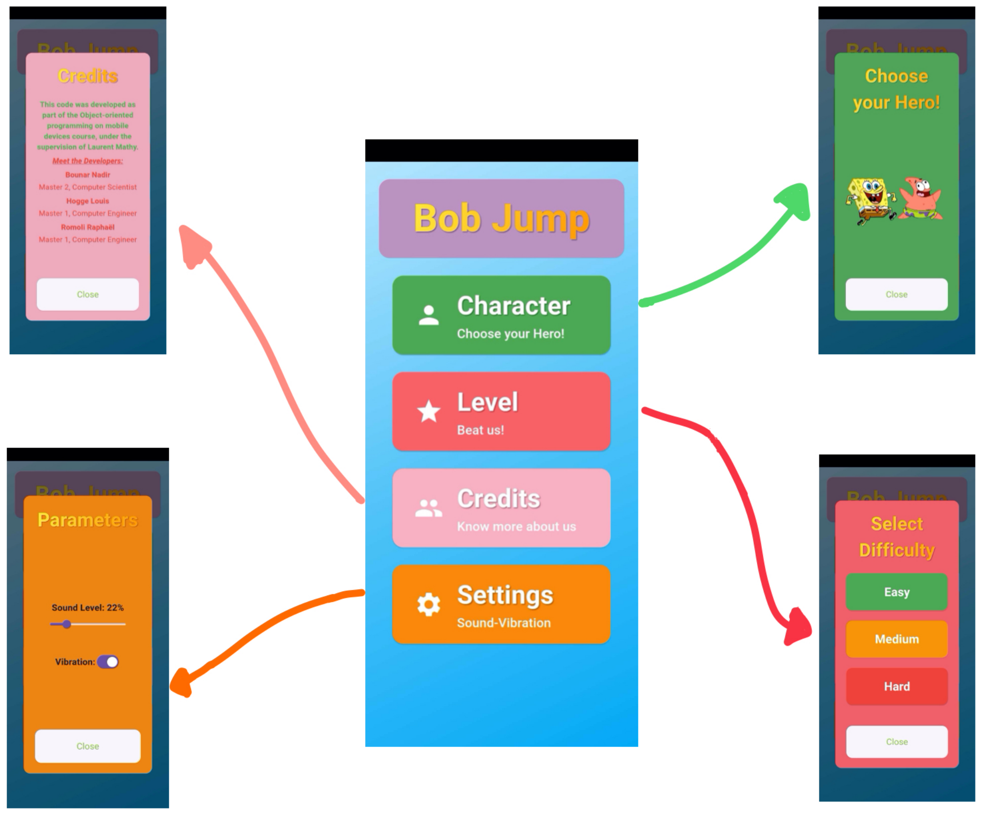
Professor : L. Mathy
Year : 2023-2024

Contents

1 App Description

Bob Jump, formerly known as Flutter Jump, is a mobile game developed in Flutter inspired from the famous game Doodle Jump and the universe of SpongeBob SquarePants.

1.1 Menu


This code was developed as part of the Object-oriented programming on mobile devices course, under the supervision of Laurent Mathy.

Meet the Developers:

Bounar Nadir
Master 2, Computer Scientist

Hogge Louis
Master 1, Computer Engineer

Romoli Raphaël
Master 1, Computer Engineer

Sound Level: 22%

Vibration: ●

FIGURE 1 – Menu Screen

The application features a menu with four buttons :

1. **Character Selection :** The user can choose between two characters, Bob or Patrick. To enhance the user experience, a simple animation shows the user that the choice has been saved.
2. **Level Selection :** Offers 3 levels of difficulty (Easy, Medium and Difficult). This choice influences the scenarios the user encounters during the game, the speed of play and certain properties of the entities making up these scenarios. These entities include :
   - **Platforms :** The green ones are fixed, the brown ones are bait, the white ones disappear on rebound, the pink ones move vertically, and the orange ones move horizontally.
   - **Power-ups :** Flyer for a strong boost, trampoline for a moderate boost, spring shoes for a light boost over 3 uses and a shield for protection.
   - **Monsters :** There is only one type of monster, but they sometimes appear as Mrs. Puff or Plankton.
3. **Credits :** Displays some credits about the game.
4. **Parameters :** Allows users to set sound levels and activate/deactivate vibrations.

1.2 Game

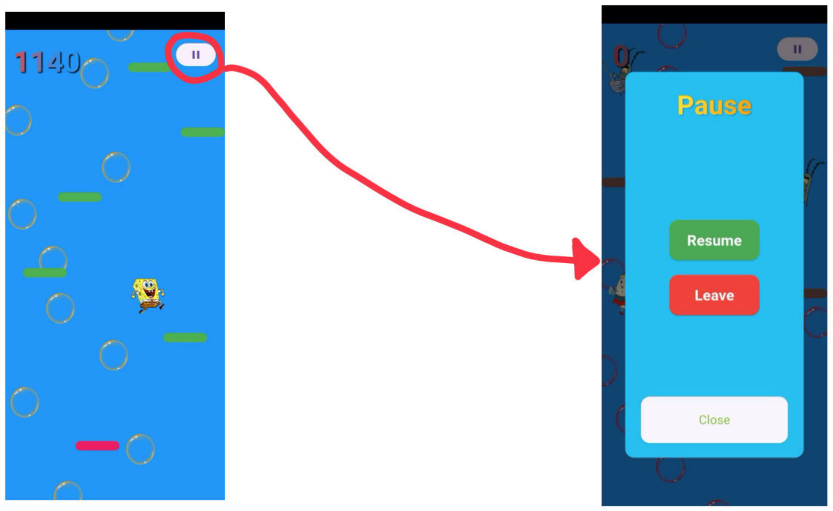


FIGURE 2 – Game Screen

The application offers a game screen with several features :

1. **Pause button :** Displays a pop-up window offering three choices : Resume the game, close the dialog and observe the paused game, or exit the game to return to the menu screen.
2. **Score :** The user's current score is displayed.
3. **Play :** The user can play the game by tilting its phone to one side or the other, and try to bounce off platforms, collect power-ups and avoid monsters.

1.3 Game Over

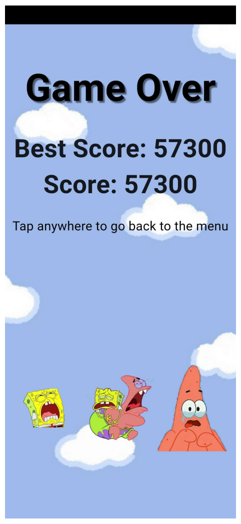


FIGURE 3 – Game Over Screen

The game over screen displays the score the user has reached in this game, his best score and allows the user to return to the main menu by tapping anywhere on the screen.

1.4 Route Structure

Thanks to the use of dialogs for the various buttons, the application presents only 3 routes :

1. `/homeMenu`: This is the route to the main menu. This is the application's starting route. It is used to move from the game over screen to the main menu screen, and from the pause pop-up to the main menu screen.
2. `/game`: This is the route to the game. It is selected when the difficulty level is chosen from the main menu.
3. `/gameOver`: This is the route to the game over screen. This is triggered automatically when the user loses the game.

1.5 Overall

Some features transcend the different sections of the application. For example :

- **Soundtrack :** Different soundtracks are played throughout the app's use
- **Responsive Design :** The design of the entire app is responsive and adapts to all types of device, from the smallest phones to the largest tablets
- **Preferences :** User preferences such as the chosen hero, sound volume or vibration activation are remembered
- etc.

1.6 Libraries

Several libraries have been imported to ensure the proper functioning of the code :

- `'package:flutter/material.dart'`: This package is essential in Flutter app development. It contains a comprehensive set of pre-built widgets (UI elements) and tools for creating visually appealing and responsive user interfaces. It provides components for creating buttons, lists, navigation bars, flexible layouts, etc that we actually used to make the game responsive.
- `'package:provider/provider.dart'`: Provider is a commonly used package in Flutter for managing application state. It facilitates the management of the application's global state and passing data through widgets. It uses the "Provider" design pattern to provide data to different parts of the application.
- `'package:vibration/vibration.dart'`: This package allows control over device vibration in mobile applications. It provides functionality to trigger vibrations at specific times during the application's execution.
- `'dart:async'`: This statement imports the core library for working with asynchronous operations in Dart. It offers functionalities for working with futures, streams, and timers, which are often useful for handling asynchronous tasks in applications.
- `'package:sensors_plus/sensors_plus.dart'`: This package grants access to sensors available on mobile devices, including the accelerometer. It allows retrieving data from the device's sensors, such as acceleration, orientation, etc. In our case we use the accelerometer to allow the user to lead the hero.
- `'dart:math'`: Dart provides a built-in library for mathematical operations, including generating random numbers. Importing 'dart :math' allows using functions to generate random numbers, perform mathematical operations, etc. In our case we used it for the random operation in the model.

2 Code Structure

2.1 Patterns

2.1.1 MVC Pattern

In this project, the implementation of the Model-View-Controller (MVC) pattern adheres to a structured approach, ensuring a clear separation of concerns. The application's core functionality revolves around interactions facilitated exclusively by the controller. Users interact only with the controller, which acts as an intermediary between the user interface (view) and data processing (model).

Each application component is separately categorized as a model, view or controller :

- **Models** are responsible for managing the application's data, logic and rules. They represent the application's dynamic data structure, independent of the user interface.
- **Views**, on the other hand, are the visual representation of models, presenting data in a specific format dictated by the controller. The separation between view and model means that the user interface can be modified without altering the underlying business logic.
- **Controllers** act as the interface between the Model and View components, handling all business logic and incoming requests, manipulating data using the Model component and interacting with Views to obtain the final result.

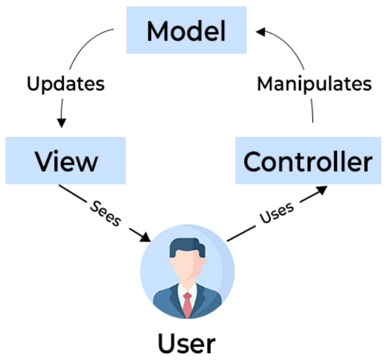


Figure 4 – MVC Pattern

In our project, **Model** is made up of classes that implement our game's logic, generation, states, physics, entities, and so on. *Game_ model* acts as the orchestra conductor, while *game_ state*, *game_ generation* and *physics_ engine*, as their names suggest, deal respectively with the state, generation and physics of the game. All *entity* children represent objects on the game screen, such as *bubble* or *my_ hero*, while *component* children represent components of the game environment, such as *my_ monster* monsters, *power_ up* boosts and *my_ platform* platforms.

Then, **View** is used to display all the elements of our application on the screen. This includes all the entities previously seen in Model, not forgetting the display of the background, score or pause button on the game screen. It also includes all menu components, buttons, pop-ups, the game over screen, etc.

Finally, **Controller** is filled with classes that manage the user's inputs as they move through the application, such as *menu_ controller*, *pause_ controller*, *custom_ dialog_ controller* and *game_ over_ controller*. When launching a game of Bob Jump, it is the *accelerometer_ controller* class that lets the user tilt his phone to move the hero, and the *game_ controller* class that lets the user start, pause or lose the game. Following the user's navigation through the app, the *sound_ controller* singleton manages the playing and stopping of the right music at the right time. *Sharedpreferences_ controller* allows past user inputs such as the user's score, character choice, sound level or vibration preferences, to be retained by the app.

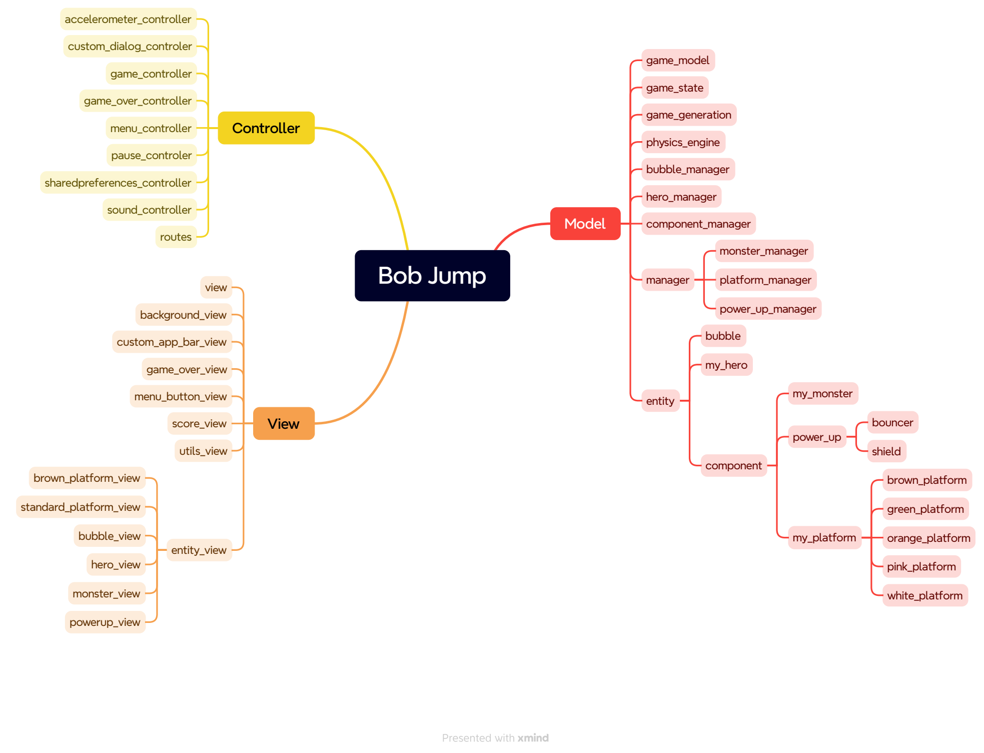


Figure 5 – Bob Jump MVC Pattern

2.1.2 Provider Pattern

When we're on the game screen, the display update is managed by a Provider pattern. Using *pauseGame()*, *resumeGame()* and *updateGameState()*, *game_ controller* acts as **provider**, while *view* acts as **listener**. So, when we decide to pause or resume the game, the view is rebuilt accordingly. Then, when we play the game, as long as we don't lose it, at each *tick()* of*game_ model*, *updateGameState()* notifies *view* that it needs to build the score, the required entities, etc., so as to stay up-to-date on the game that the classes from Model decide to generate.

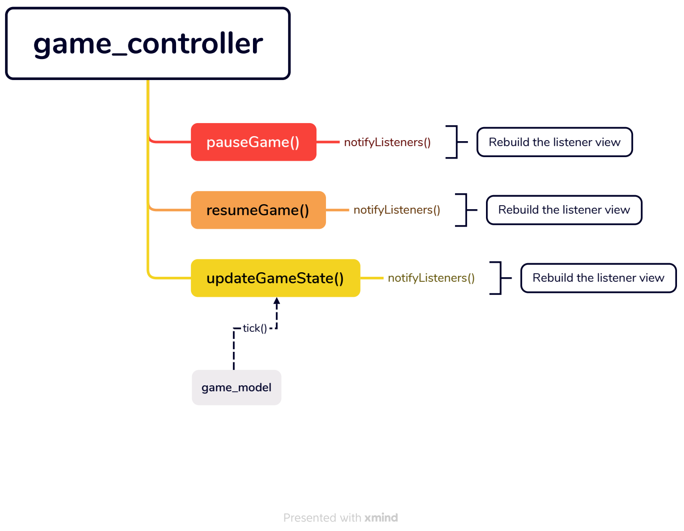


Figure 6 – Bob Jump Provider Pattern

2.1.3 Stateful Widget Pattern

In our project, we use 3 stateful widgets : MainMenu, GameScreen and GameOverScreen. At first, these widgets worked fine in stateless mode, but when it came to managing the application's music, we could not find any other way than to make them stateful in order to play and stop the soundtrack at the right moment. These offer the ability to manage internal state, which is particularly useful when addressing issues related to sound. They enable reacting to state changes and altering the appearance or behavior of the interface accordingly.

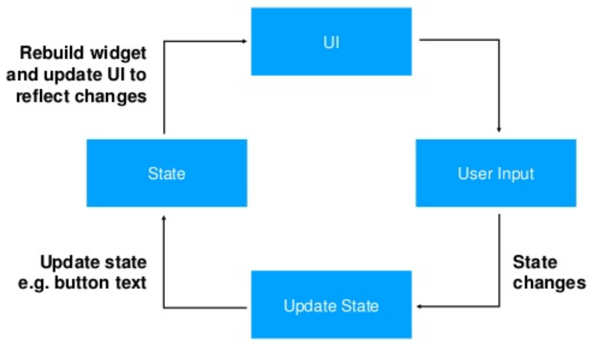

UI
User Input
State changes
Update State
Update state e.g. button text
State
Rebuild widget and update UI to reflect changes

FIGURE 7 – Stateful Widget Pattern

2.2 Widgets

In our code, we extensively used **StatelessWidget**, which are components in Flutter devoid of internal state management. They are perfect for representing static user interface elements, as they do not change in response to events or data modifications.

Additionally, we integrated **WidgetBuilders**, which is a function that takes a context as a parameter and returns a widget. This function is often used to define the structure of widgets in Flutter, particularly when defining routes within an application.

Lastly, we leveraged **WidgetBindingObservers** to handle specific functionalities within our application, notably for the PreferenceService and audio management. WidgetBindingObservers are objects that listen to and react to changes in the lifecycle of widgets, making them ideal for coordinating tasks based on system events such as initialization, suspension, or resurgence of the application. In our case, we used them to manage user preferences via the PreferenceService and to handle audio-related features.

3 OO & UML

- **UML Structure Overview** : Our UML diagram is segmented into three main parts : Model, View, and Controller. Each part encapsulates specific classes and functionalities relevant to its domain.
- **Naming Convention** : In our diagram, when a class belongs to a specific part but is related to another, we prefix the class name with the part it belongs to. For instance, a class within the Model part that is related to another part is named with the prefix 'Model : :'. This indicates the class's primary association.
- **Example** : The class 'GameModel', which is defined in the Model part of our system, is represented outside of the Model as 'Model : :GameModel' in the UML diagram. This naming convention clearly indicates that 'GameModel' resides within the 'Model' segment, providing a quick reference to its location and relationships within the overall system architecture.

3.1 Model

Below is a detailed text version of the Model UML. You can find the diagram version by clicking here.

- **Abstract Classes**
  - **Entity :** This abstract class serves as the foundation for all game entities, encompassing attributes defining position and size, along with methods handling movement and teleportation.
  - **Component :** Another abstract class representing entities with scores that can be destroyed. It encapsulates fields defining scores and methods for component destruction.
  - **Platform :** Serving as an abstract base for game platforms, it includes fields defining platform types and abstract methods to retrieve platform colors.
  - **PowerUp :** An abstract class for power-ups in the game, encapsulating fields defining their status, collection methods, and abstract functions to determine power-up types.
  - **Manager :** This abstract class handles specific game managers, containing lists of components, screen size, and stored scores. It provides methods for component management and score handling.
- **Inheritance Hierarchy**
  - **Entity** serves as the parent class for Bubble, Component, and MyHero.
  - **Component** is the parent class for Platform, Monster, and Power Up.
  - **Platform** serves as the parent class for all platform variations.
  - **PowerUp** acts as the parent class for Bouncer and Shield.
  - **Manager** is the parent class for PlatformsManager, MonsterManager, and PowerUpsManager.
- **Aggregation and Composition**
  - **GameModel :** Composed of several classes—BubbleManager, PhysicsEngine, GameGeneration, GameState, HeroManager, and ComponentManager. As the central class, GameModel orchestrates interactions among these entities.
  - **PhysicsEngine :** Aggregated with BubbleManager, ComponentManager, and MyHero, applying gravity and collision detection to all entities.
  - **HeroManager :** Composed with MyHero, manages hero movements.
  - **GameState :** Aggregates with MyHero to monitor the hero's position and determine game status.
  - **GameGeneration :** Aggregates with PhysicsEngine, BubbleManager, and ComponentManager to dynamically modify components and physics based on difficulty levels.
  - **ComponentManager :** Aggregates with MyHero for the addition of power-ups to the hero.
  - **PlatformsManager, MonsterManager, and PowerUpsManager :** Respectively aggregate with Platform, Monster, and PowerUp, managing these specific game elements.
  - **Entity :** Each entity aggregates with its corresponding EntityView for display purposes.

3.2 View

Below is a detailed text version of the View UML. You can find the diagram version by clicking here.

- **Abstract Class**
  - **EntityView :** An abstract base class responsible for rendering entities as stateless widgets constructed based on an entity. It encapsulates fields defining the entity and methods for rendering.
- **Inheritance Hierarchy**
  - **EntityView** acts as the parent class for different entity-specific views : BrownPlatformView, BubbleView, HeroView, MonsterView, PowerUpView, and StandardPlatformView. Notably, BrownPlatformView stands out due to its distinct animation compared to other platform views.
- **Aggregation and Composition**
  - **Viewed :** The primary class responsible for the view layer, composed of ScoreView, BackgroundView, and all specific EntityView classes. It orchestrates the display of these components, forming the user interface.
  - **Entity View Aggregation :** Each EntityView class aggregates with its respective Entity in the model, ensuring a seamless representation of model elements in the view layer.
  - **MenuController Composition :** Comprising CustomAppBar and MainMenuButtonView, integral components of the MenuController responsible for handling user interactions in the menu section.
  - **GameOverController Composition :** Involving GameOverView, a crucial component in displaying the game over page within the GameOverController.
  - **UtilsView :** Operates as a singleton and serves utility functions utilized in various parts of the controller for diverse functionalities, aiding in different operations within the system.

3.3 Controller

Below is a detailed text version of the Controller UML. You can find the diagram version by clicking here.

- **Agregation and composition** :
  - **MenuController** : Composed of CustomAppBar and MainMenuButtonView and in aggregation CustomDialog,PreferencesServices and AudioPlayerManager. This class is used to handle user interactions in the menu.
  - **GameScreen** : GameScreen is composed by GameControllerprovider.
  - **GameControllerprovider Aggregation** : Includes PreferencesService, AudioPlayerManager, AccelSubscription and GameModel (part of the model).
  - **GameOverController** : He is in composition with GameOverView and in aggregation with PreferencesServices and AudioPlayerManager.

4 Technical challenges

- **Gravity Simulation** : A primary challenge was simulating gravity in our game. We aimed for the player to jump a distance equal to one-third of the screen's height. To achieve this, we started with the gravitational acceleration equation $y'' = -g$, where g is the acceleration due to gravity, and y is the vertical position.
  From this, we integrated to find the velocity equation $y' = -gt + v_0$, with v_0 being the initial velocity. To calculate displacement, we multiplied this velocity by a time increment.
  Our goal was to determine the initial speed v_0 for a desired jump size (jumpSize), set to $0.3 \times$ screenHeight. We calculated the time to reach maximum jump height (t) from $y' = 0$, leading to $t = v_0/g$.
  Substituting this into the position equation $y = -\frac{1}{2}gt^2 + v_0t + x_0$, where x_0 is the initial position, and setting y to $0.3\times$screenHeight, we derived the initial speed as $v_0 = \sqrt{0.3 \times \text{screenHeight} \times 2g}$.

- **Screen Center Boundary Handling** : A significant challenge was ensuring the player could not move above the screen's center. Our solution involved maintaining a hypothetical vertical position for the player, termed the "supposedYAxis." This represents where the player would be if not constrained by the screen's center. When the player reaches the center, their movement is restricted, but we continue updating their position on the supposedYAxis. Concurrently, other game elements move downward, creating the illusion that the player is ascending. Once the player's supposedYAxis descends below the screen's midpoint, we revert to updating their actual position on the true Y-axis, and the game proceeds as normal.

- **Game Generation :** Game generation is a key element of our project. The first difficulty was to actually generate the entities of a game. Once this was possible, we opted for a basic random generation algorithm. Several negative points emerged : the game wasn't very fun to play, entity generation seemed messy and made no sense, difficulty was completely random and progression wasn't always possible. A first solution was to invent a random-generation algorithm to ensure that progression was always possible, that difficulty increased gradually and that the game was fun to play, but this seemed highly complex. A second solution, the one we decided to follow, was to set up a custom scenario creation system to ensure feasibility of progression, management of game difficulty and fun. The next step was to build around this a level system and, based on this, methods for increasing difficulty, an algorithm for selecting scenarios and a method for building them.

- **Sound Management :** Managing the sound turned out to be somewhat complex. The original idea was to play different music at each screen transition, whether a menu screen, a game screen or a game over screen. This required the ability to play and stop music at the right moment. A first idea was to use the dispose() method for this purpose. However, this method, located in each screen file, could not be called outside these files. As a result, audio tracks were not stopped when they should have been, and overlapped. The sound also continued when the user returned to the phone's home screen. Later we also learned that it wasn't good practice to use this method ourselves, and that we should rather let flutter call itself when it destroys a widget. So we had to look for a solution, which was this : a singleton in *sound_ controller*, a single instance of an audio player with which we would play and stop the appropriate music on the appropriate screen.

- **Responsive Design :** Making the game responsive on all types of support was a problem we hadn't anticipated as beginners. Correcting this took a lot of time, as we had to refactor a large part of the code already implemented, and test and validate the changes.

- **Code Modularity :** Because dart/flutter was totally new to us, the first step in the project was to understand how it worked by gradually adding complexity. Then, once we had a better grasp of the language, we set about refactoring the project to redefine its structure using the appropriate OOP principles. This is obviously time-consuming and sometimes complex, given that certain choices were made earlier without any knowledge of their future impact.

- **IOS :** Although certain implementation choices worked perfectly well on Android, these led to some exceptions for the IOS version. These prompted us to rethink certain choices and rework them so that the project works on both Android and IOS.

5 What could be improved ?

- **Vibration :** We took the decision, using the "vibration" package, to add a vibration when the user loses his game and, although it works equally well on Android and IOS, we observed that the package used is now obsolete. A non-obsolete alternative should therefore be found. We've already tried using the Flutter services' `HapticFeedback.lightImpact()` function and the "vibrate" package, but without success.

- **Hitbox :** We have encountered some difficulties with monster hitboxes. Some of them don't have a perfect rectangular shape, leading to situations where the main character can appear to die without direct contact with the monster concerned. An improvement could be to rework the monster images so that their shape is as close to a rectangle as possible. This would improve the user experience.

- **Pause Pop-up :** When refactoring the code, we inadvertently forgot to separate the pause pop-up into its Controller and View parts. We should do so.

- **my_hero & hero_manager :** When refactoring the code, we inadvertently forgot to transfer the interactions with *my_hero* to its *hero_manager*. We should do so.

- **component_manager & game_generation :** When refactoring the code, we inadvertently forgot to use *component_manager* instead of each of its children in *game_generation*. We should do so.

- **Types :** Small errors in variables and function types are still present in the code. These should be corrected. Maybe we should also have identified more static classes and singletons.

- **Manager Classes :** Perhaps in the ideal project, each entity would have been merged with its manager. Or more abstraction would have been present between the manager classes.

6 UML

6.1 Model

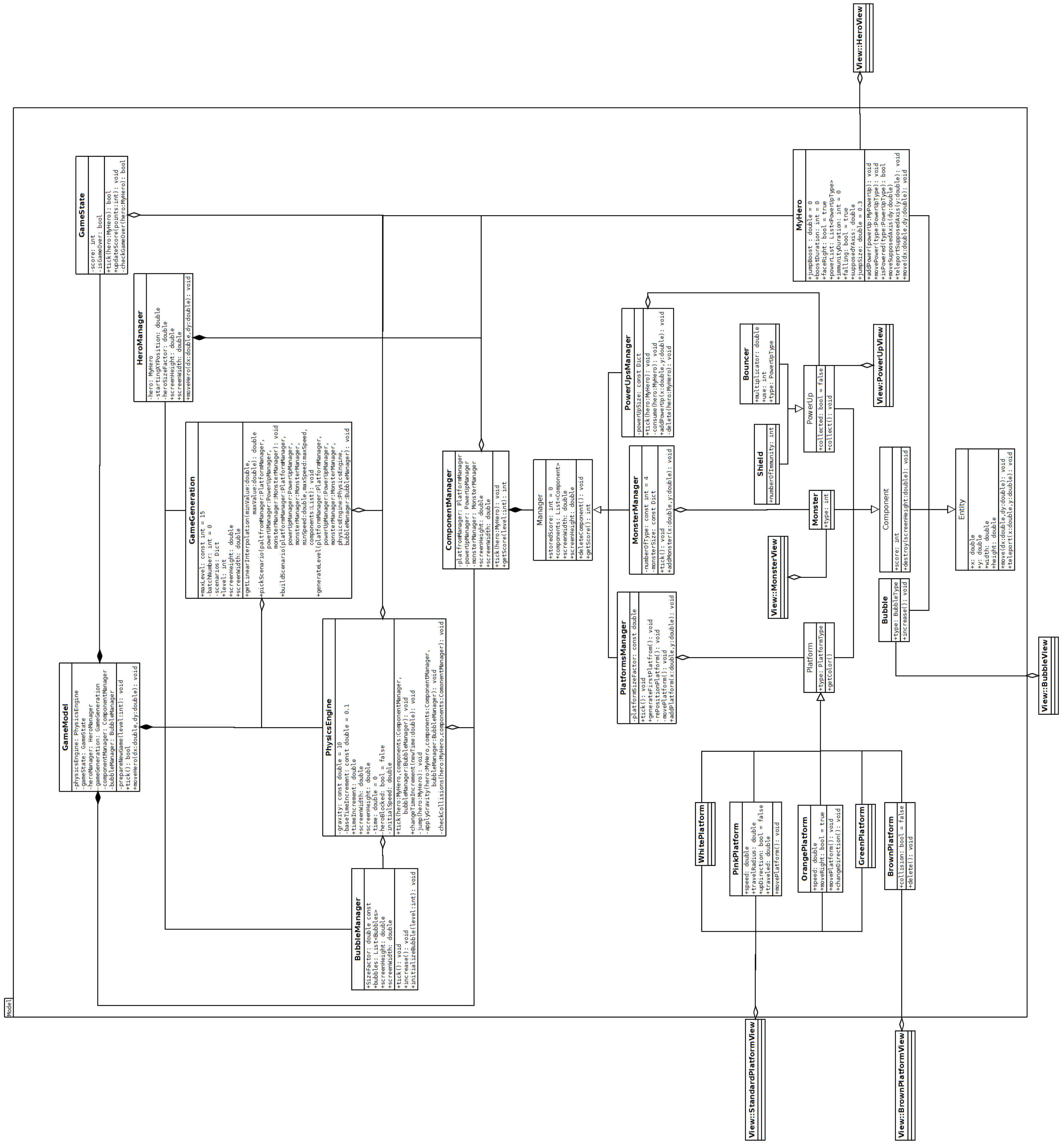


FIGURE 8 – Model UML

6.2 View

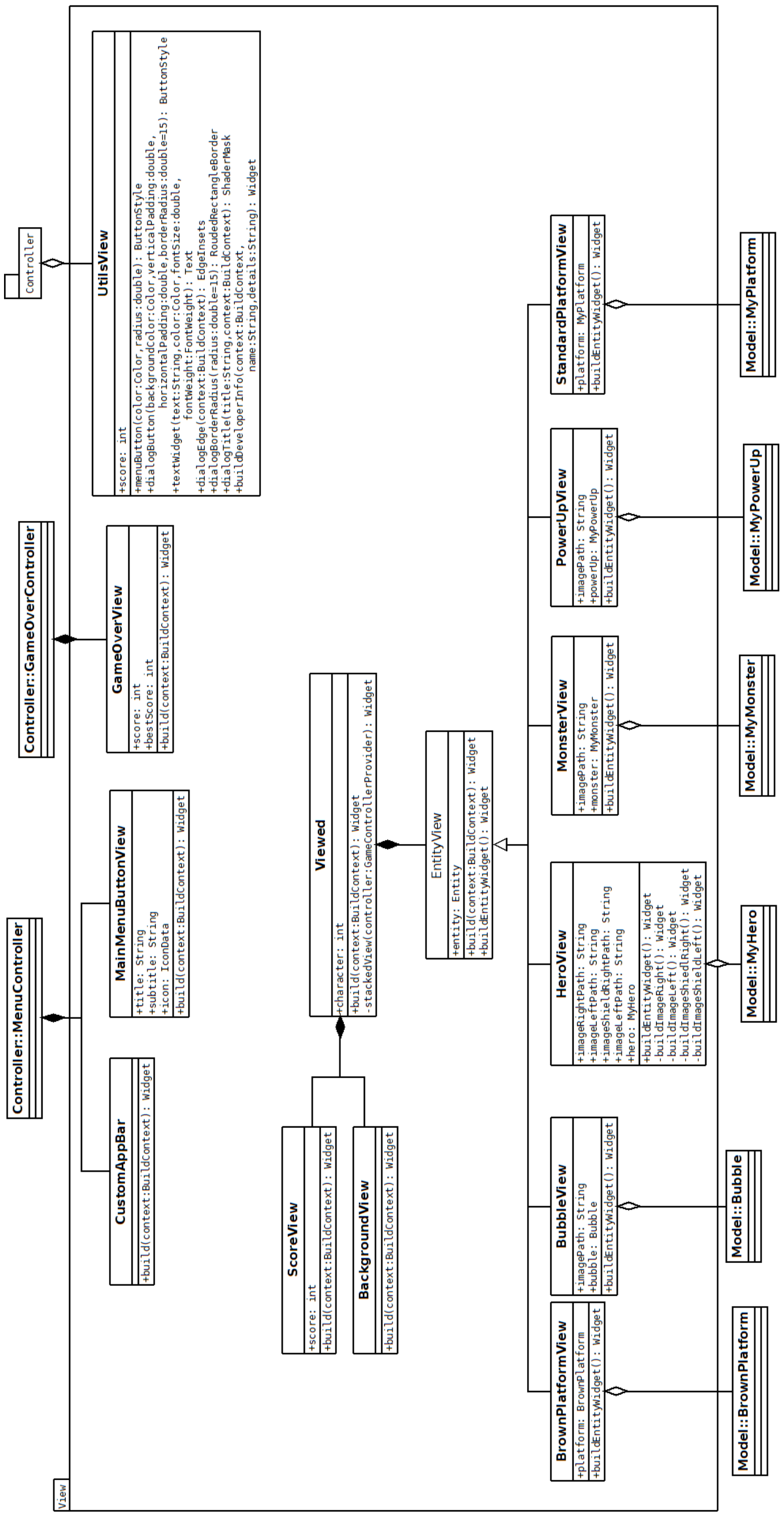

View
Controller
UtilsView
+score: int
+menuButton(color:Color,radius:double): ButtonStyle
+dialogButton(backgroundColor:Color,verticalPadding:double, horizontalPadding:double,borderRadius:double=15): ButtonStyle
+textWidget(text:String,color:Color,fontSize:double, fontWeight:FontWeight): Text
+dialogEdge(context:BuildContext): EdgeInsets
+dialogBorderRadius(radius:double=15): RoudedRectangleBorder
+dialogTitle(title:String,context:BuildContext): ShaderMask
+buildDeveloperInfo(context:BuildContext, name:String,details:String): Widget
Controller::GameOverController
GameOverView
+score: int
+bestScore: int
+build(context:BuildContext): Widget
Controller::MenuController
MainMenuButtonView
+title: String
+subtitle: String
+icon: IconData
+build(context:BuildContext): Widget
CustomAppBar
+build(context:BuildContext): Widget
Viewed
+character: int
+build(context:BuildContext): Widget
-stackedView(controller:GameControllerProvider): Widget
EntityView
+entity: Entity
+build(context:BuildContext): Widget
+buildEntityWidget(): Widget
ScoreView
+score: int
+build(context:BuildContext): Widget
BackgroundView
+build(context:BuildContext): Widget
StandardPlatformView
+platform: MyPlatform
+buildEntityWidget(): Widget
PowerUpView
+imagePath: String
+powerUp: MyPowerUp
+buildEntityWidget(): Widget
MonsterView
+imagePath: String
+monster: MyMonster
+buildEntityWidget(): Widget
HeroView
+imageRightPath: String
+imageLeftPath: String
+imageShieldRightPath: String
+imageLeftPath: String
+hero: MyHero
+buildEntityWidget(): Widget
-buildImageRight(): Widget
-buildImageLeft(): Widget
-buildImageShiedldRight(): Widget
-buildImageShieldLeft(): Widget
BubbleView
+imagePath: String
+bubble: Bubble
+buildEntityWidget(): Widget
BrownPlatformView
+platform: BrownPlatform
+buildEntityWidget(): Widget
Model::MyPlatform
Model::MyPowerUp
Model::MyMonster
Model::MyHero
Model::Bubble
Model::BrownPlatform

FIGURE 9 – View UML

6.3 Controller

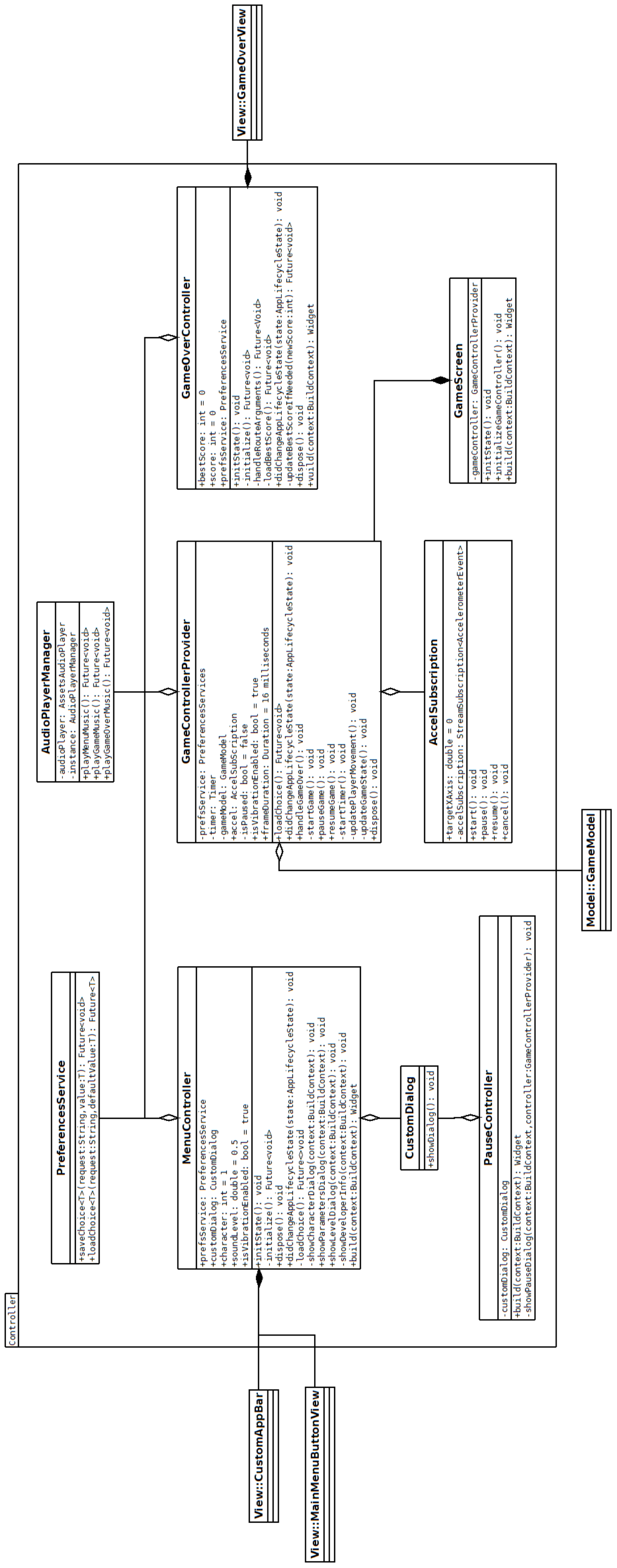

View::GameOverView
GameOverController
+bestScore: int = 0
+score: int = 0
+prefsService: PreferencesService
+initState(): void
-initialize(): Future<void>
-handleRouteArguments(): Future<Void>
-loadBestScore(): Future<void>
+didChangeAppLifecycleState(state:AppLifecycleState): void
-updateBestScoreIfNeeded(newScore:int): Future<void>
+dispose(): void
+vuild(context:BuildContext): Widget
GameScreen
-gameController: GameControllerProvider
+initState(): void
+initializeGameController(): void
+build(context:BuildContext): Widget
AudioPlayerManager
-audioPlayer: AssetsAudioPlayer
-instance: AudioPlayerManager
+playMenuMusic(): Future<void>
+playGameMusic(): Future<void>
+playGameOverMusic(): Future<void>
GameControllerProvider
-prefsService: PreferenceServices
-timer: Timer
-gameModel: GameModel
+accel: AccelSubScription
-isPaused: bool = false
+isVibrationEnabled: bool = true
+frameDuration: Duration = 16 milliseconds
+loadChoice(): Future<void>
+didChangeAppLifecycleState(state:AppLifecycleState): void
+handleGameOver(): void
-startGame(): void
+pauseGame(): void
+resumeGame(): void
-startTimer(): void
-updatePlayerMovement(): void
-updateGameState(): void
+dispose(): void
AccelSubscription
+targetXAxis: double = 0
-accelSubscription: StreamSubscription<AccelerometerEvent>
+start(): void
+pause(): void
+resume(): void
+cancel(): void
Model::GameModel
PreferencesService
+saveChoice<T>(request:String,value:T): Future<void>
+loadChoice<T>(request:String,defaultValue:T): Future<T>
MenuController
+prefsService: PreferencesService
+customDialog: CustomDialog
+character: int = 1
+soundLevel: double = 0.5
+isVibrationEnabled: bool = true
+initState(): void
-initialize(): Future<void>
+dispose(): void
+didChangeAppLifecycleState(state:AppLifecycleState): void
-loadChoice(): Future<void>
-showCharacterDialog(context:BuildContext): void
+showParametersDialog(context:BuildContext): void
+showLevelDialog(context:BuildContext): void
-showDeveloperInfo(context:BuildContext): void
+build(context:BuildContext): Widget
CustomDialog
+showDialog(): void
PauseController
-customDialog: CustomDialog
+build(context:BuildContext): Widget
-showPauseDialog(context:BuildContext,controller:GameControllerProvider): void
Controller
View::CustomAppBar
View::MainMenuButtonView

Figure 10 – Controller UML

7 Image Sources

- Figure 4 : Shrey Sharma, "MVC Architecture in Salesforce," accessed 12/12/2023, `https://shreysharma.com/mvc-architecture/`.
- Figure 7 : "StatefulWidget, StatelessWidget and InheritedWidget" accessed 13/12/2023, `https://wna.gitbook.io/flutter-learning-path/ji-ben-gai-nian/stateful-and-stateless-widget`.