Flappy Bird - OOP Final Project

Team member

- b07902047 資工三 羅啟帆
- b07902064 資工三 蔡銘軒
- b07902131 資工三 陳冠宇
- b07902139 資工三 鄭豫澤

Design

Program design

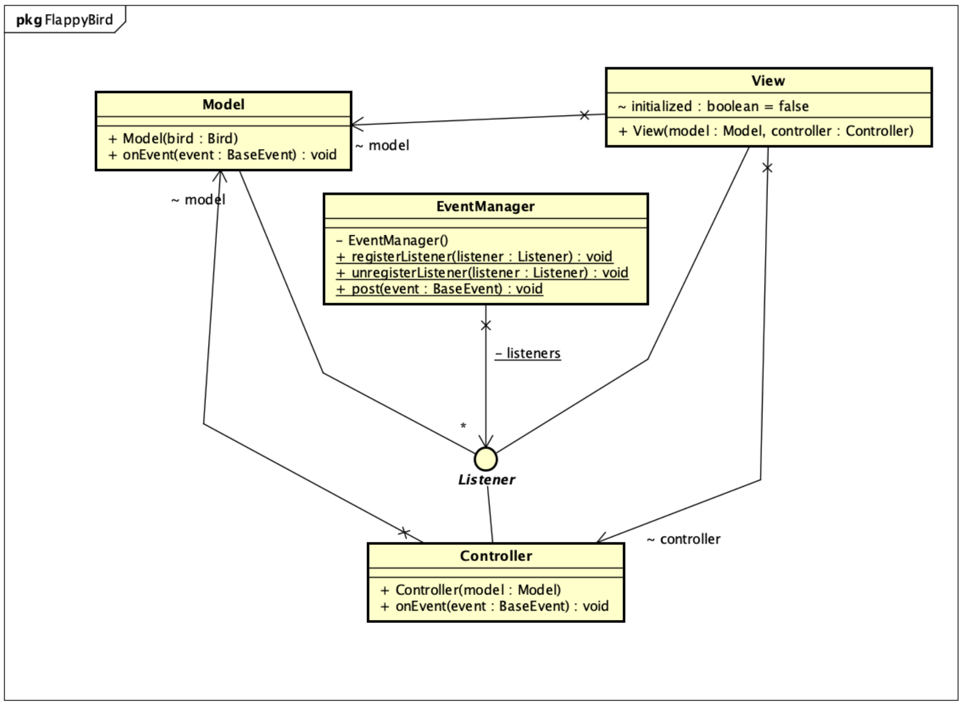
Model-View-Controller (MVC) + Event

We adopted the well-known MVC structure to design our game, which divides the program into three main parts:

- Model: Implement game logic and store all the data.
- View: Render the screen using data stored in model.
- Controller: Receive user input and notify the model

Furthermore, we introduced another component:

- Event: Coordinate messages between the three components above.
    - We created a class `EventManager` responsible for coordination.



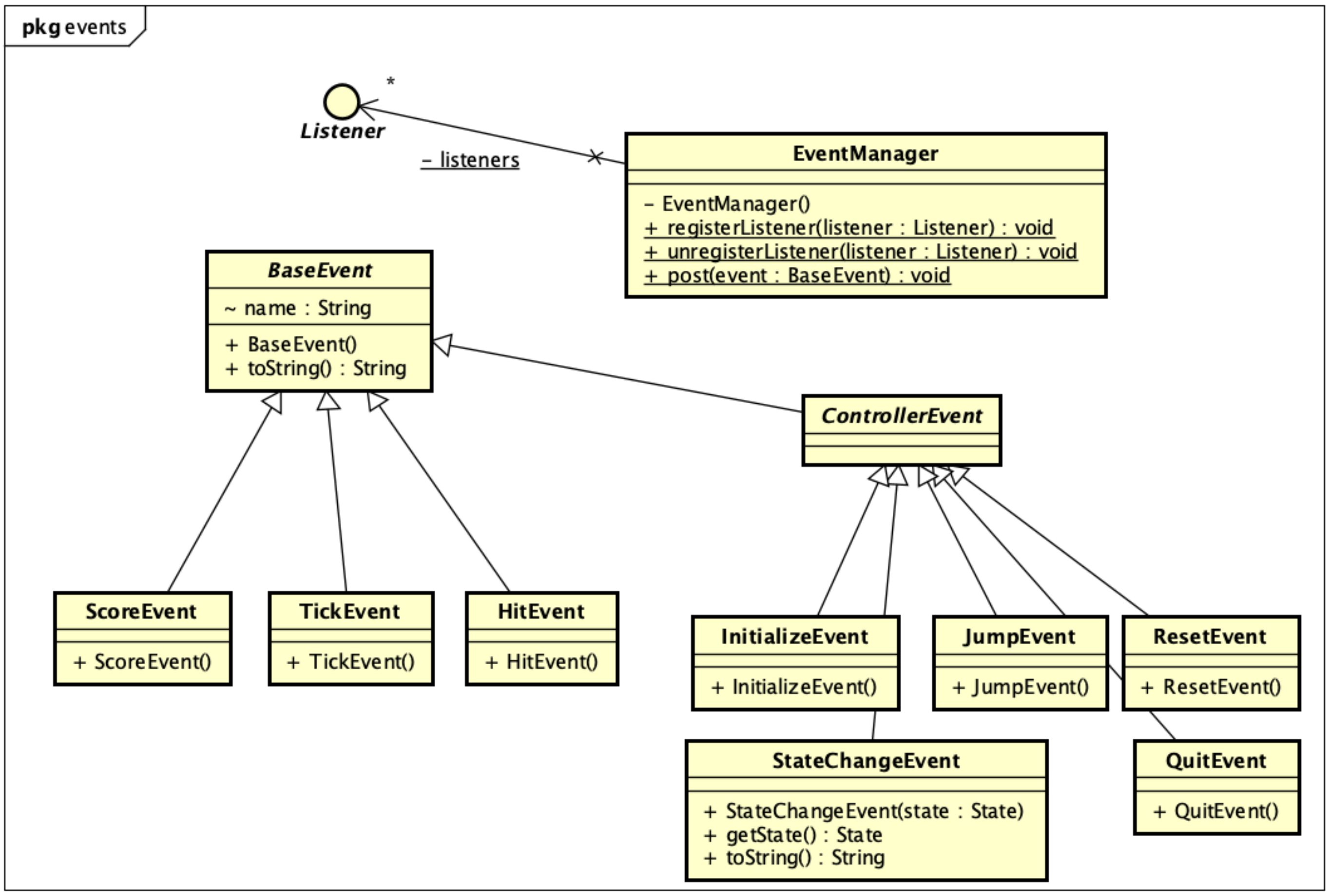
pkg FlappyBird
Model
+ Model(bird : Bird)
+ onEvent(event : BaseEvent) : void
View
~ initialized : boolean = false
+ View(model : Model, controller : Controller)
~ model
~ model
EventManager
- EventManager()
+ registerListener(listener : Listener) : void
+ unregisterListener(listener : Listener) : void
+ post(event : BaseEvent) : void
- listeners
*
Listener
Controller
+ Controller(model : Model)
+ onEvent(event : BaseEvent) : void
~ controller

Assumption: We assume the player controlling **one bird** is the core of this game. So there is no OCP on the bird. (Controlling multiple birds would change the nature of this game)

Event

We implemented three important components:

- `BaseEvent`: An abstract class for all other events
- `EventManager`: An event manager that broadcasts events to all registered listeners.
- `Listener`: An interface that all listener should implement (Model, View, Controller)
    - it contains a function `void onEvent(BaseEvent event)` which specify the listener's action when receiving events.

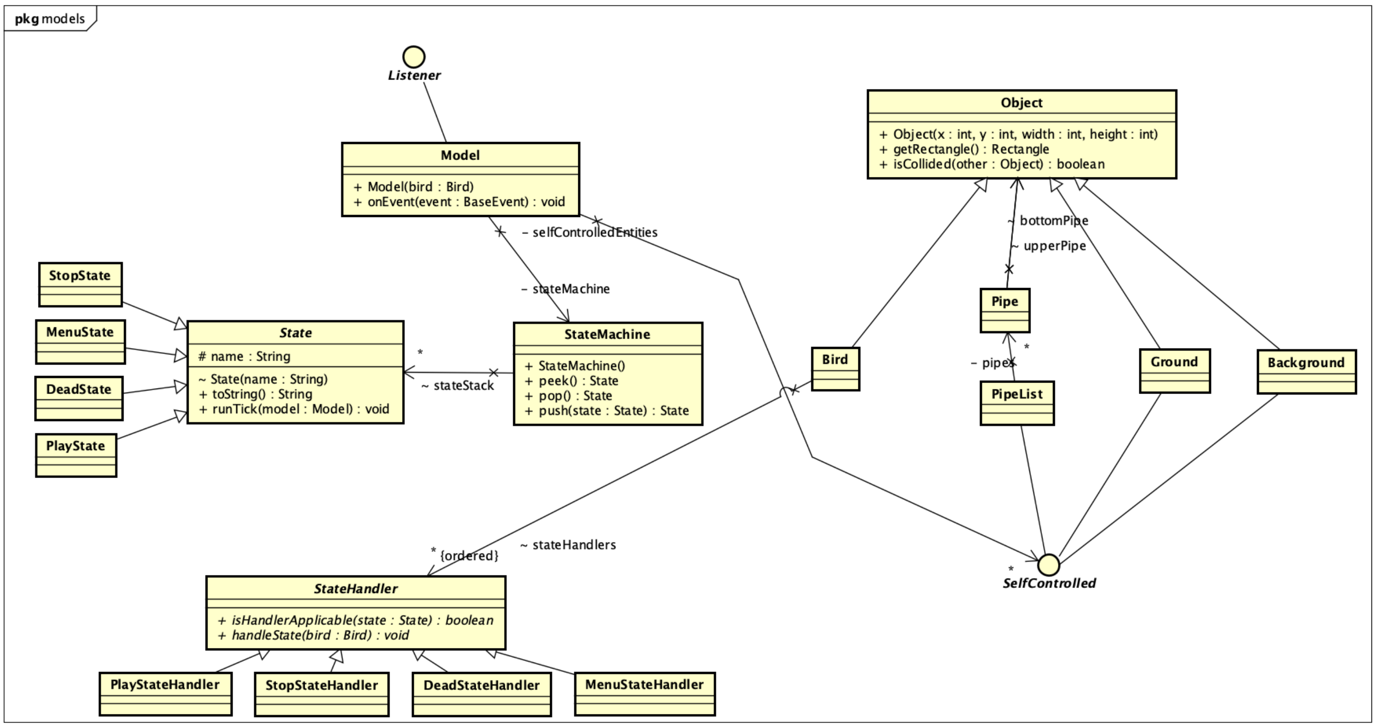


Model

`Model` is the entry point of the model component. It holds the bird, the pipes, and the ground. `Model` responds to each event, and triggers updates of the objects accordingly. It also provides many getters and status checker for the view component to get the information that it needs for rendering.

In the model component, there is a list of `state`. Each `state` describes the state that the bird is in. For each `state`, the bird has a corresponding `StateHandler` to update the bird's position.

Objects other than the bird implement the `SelfControlled` interface. They are responsible for their own positions.



View

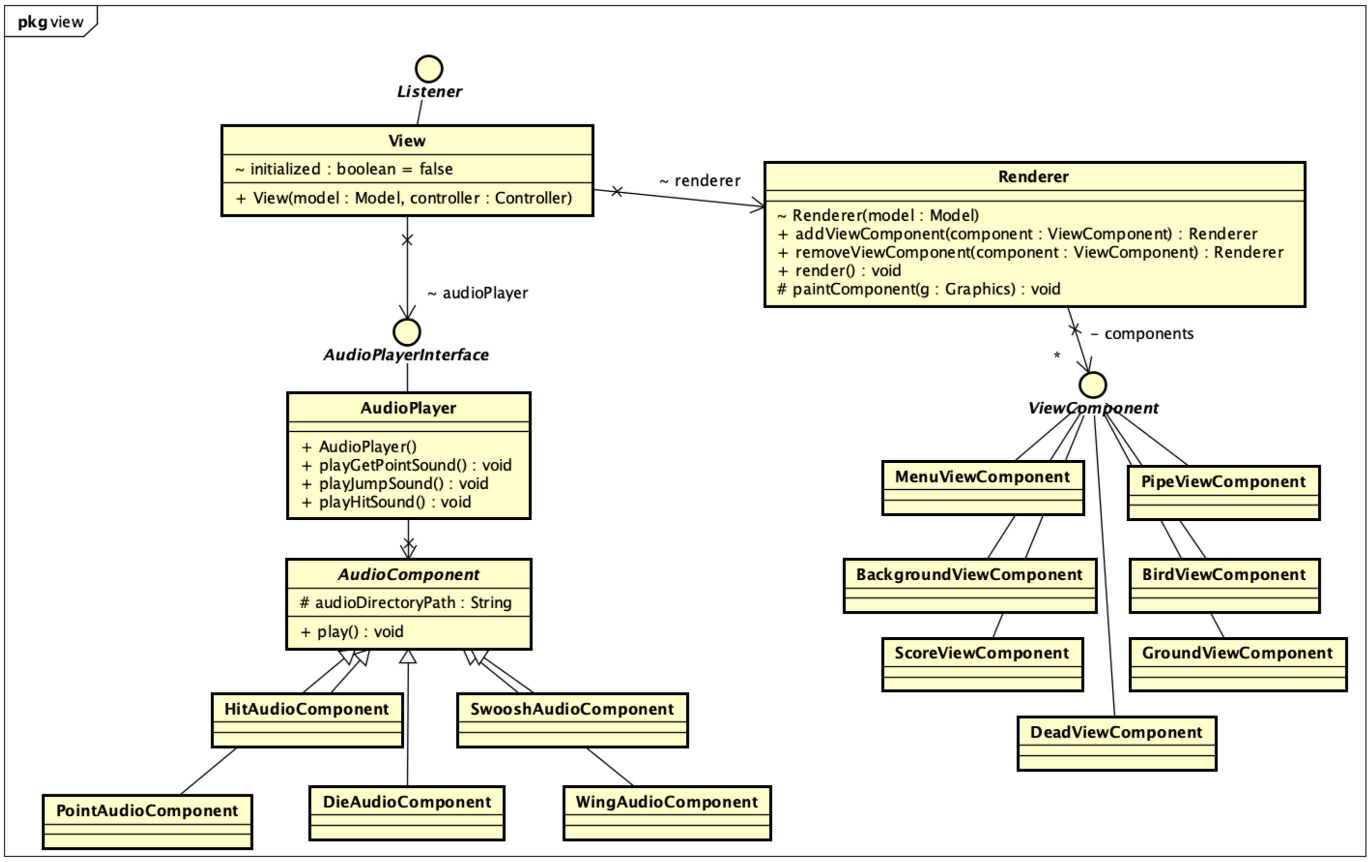
`View` is the entry point of the view component. It store the `jFrame` object (as the window), and also responsible for updating according to the event.

In `Renderer`, this is our main `JPanel`. We store a list of `ViewComponent` in renderer, and call their `paint(Graphic)` function when in `paintComponent`. Each `ViewComponent` will use the `Graphic` object to draw itself according to the model informations.

Each `ViewComponent` will receive `Model` (or `Object`s it need) in construction. It will load image when constructed, and display to the window depends on model's data.

In this design, if we want to draw more components, we only need to create the component and add it into renderer.

The structure for `AudioPlayer` is similar. We will pass the current state and incoming event to player, the each player component will decide whether to play itself.

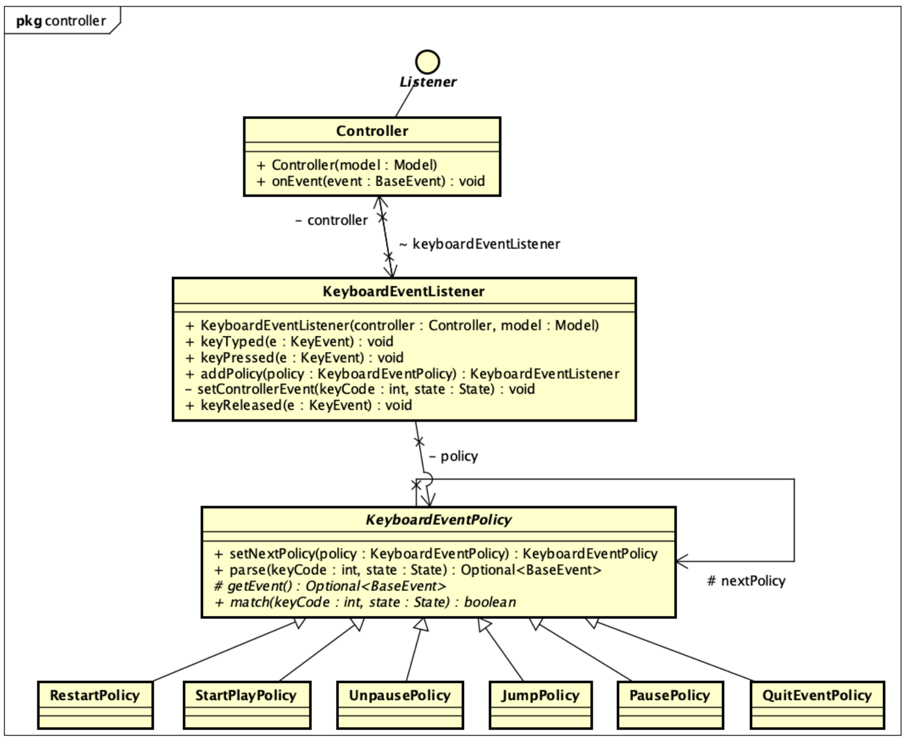


Controller

There is a `KeyboardEventListener` that will record all the key press events, and use `policies` to parse the event.

`KeyboardEventPolicy` defines which event should be emitted based on the key pressed and current state. These policies forms a chain and will check if each policy is satisfied one by one, and return the corresponding event once there are a match.

`Controller` will add policy into `KeyboardEventListener`. When receive a `TickEvent`, controller will send all buffered event. These event will not be sent right after it is created.


pkg controller
Listener
Controller
+ Controller(model : Model)
+ onEvent(event : BaseEvent) : void
- controller
~ keyboardEventListener
KeyboardEventListener
+ KeyboardEventListener(controller : Controller, model : Model)
+ keyTyped(e : KeyEvent) : void
+ keyPressed(e : KeyEvent) : void
+ addPolicy(policy : KeyboardEventPolicy) : KeyboardEventListener
- setControllerEvent(keyCode : int, state : State) : void
+ keyReleased(e : KeyEvent) : void
- policy
KeyboardEventPolicy
+ setNextPolicy(policy : KeyboardEventPolicy) : KeyboardEventPolicy
+ parse(keyCode : int, state : State) : Optional<BaseEvent>
# getEvent() : Optional<BaseEvent>
+ match(keyCode : int, state : State) : boolean
# nextPolicy
RestartPolicy
StartPlayPolicy
UnpausePolicy
JumpPolicy
PausePolicy
QuitEventPolicy

Advantage

- MVC structure enables clear work division between classes, which is not only beneficial to our model design but also great for dividing works between group members and work on them in parallel.
- We try to make it easy to add new "gaming components" in the future by considering OCP in all parts of this project. Adding a sound / image / object in the game are a piece of cake in this project.

Disadvantage

- Using MVC structure in such a small game might be recognized as an overkill.
- If we want to add a user-controllable object, we need to modify `Model` class.

External packages used

- Java Swing

How to play

- The main goal of this game is to control the bird to pass through as most pipe as possible. Passing each pipe earns you one point. If you hit the pipe or the ground, you die.
- How to control?
  - Space: (re)start game / bird jump
  - P: pause while playing
  - esc: quit game

Work disivion

- b07902047 資工三 羅啟帆：MVC design and model design and implementation
- b07902064 資工三 蔡銘軒：Model design and implementation

- b07902131 資工三 陳冠宇：View & Controller design + implementation
- b07902139 資工三 鄭豫澤：View & Controller design and implementation