Report
“MTSK-GAME”

Pietro Olivi
Leonardo Tassinari
Lorenzo Dalmonte
Alessio Paoloni

February 13, 2023

Contents

Chapter 1

Analysis

The software commissioned by the professors of the Object Oriented Programming course aims to create an application designed to enhance psychomotor skills through a gaming experience based on multitasking.
The term *Multitasking* refers to the ability of a person or a product to do more than one thing at a time.

1.1 Requirements

Functional

- Upon starting, the software will display a simple minigame.
  A minigame is a potentially-never-ending challenge that requires simple actions from the player in order to keep the game going. After a short amount of time a new minigame will appear and so on until all four minigames are shown.

- The player's goal is to survive as long as possible. After failing a minigame the application will display the final result, therefore the software has to keep track of how long the player has lasted in the match in order to calculate the score.

Non functional

- The application shall sustain high framerate (around 120 fps) in all the sections of the gameplay, even on older hardware[1].

[1]e.g. Intel Core i3 (fourth generation), 4Gb of RAM.

- It shall be possible to easily develop and swap the minigames among the ones that best fit the training purposes of the user on top of those already provided.

1.2 Domain analysis

MTSK-Game must exhibit some *minigames*, the ones supplied by us are:

- *WhacAMole*: where the goal is to crush all the moles that emerge from the dug holes, before they re-enter them.
- *DodgeATriangle*: in which the player has to slide a *rectangle* up and down switching lanes, aiming to avoid hitting moving *triangles*.
- *CatchTheSquare*: where the user should collect *squares* running over them with a *circle* before they disappear.
- *FlappyBirdAlike*: where the user needs to control a *cursor* leading it to fit between *obstacles* that will come towards it.

For the game to end, and the *score* to appear, it'll be sufficient losing in only one of the currently displayed mingames.

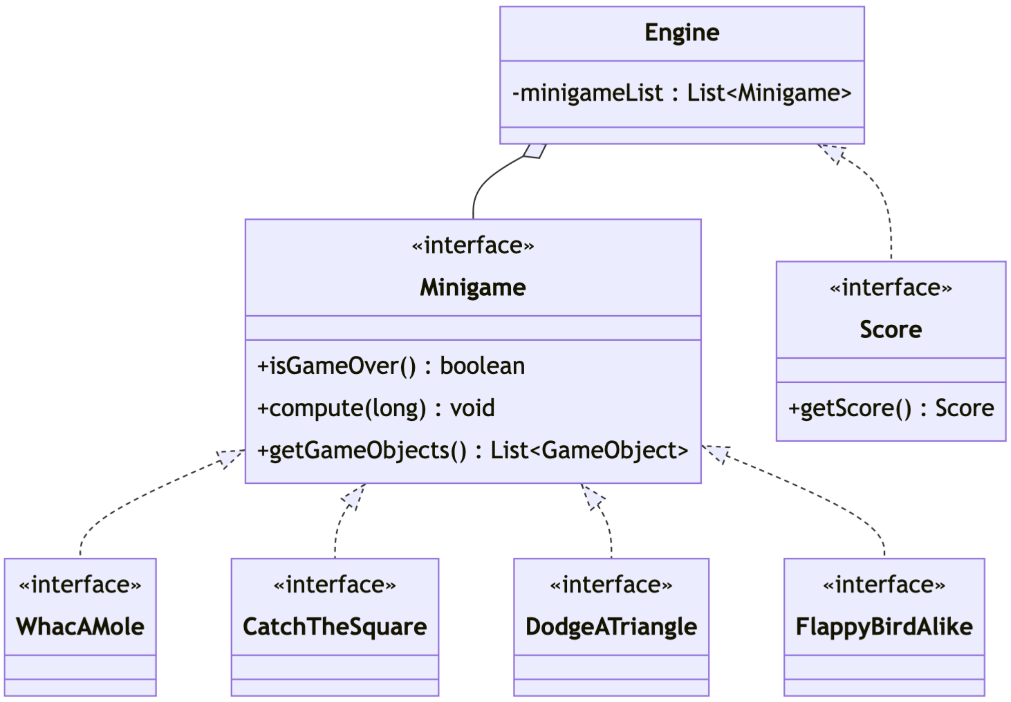

Figure 1.1: UML diagram of the domain analysis

Chapter 2

Design

2.1 Architecture

The project is built on the "model-view-controller" architecture: the **Engine** class represents the controller, hence the entry point of the program. Inside the controller we create an instance of the **View**, SwingView in our case, which takes care of intercepting the user's input, then forwarded to the Engine, and deals with the graphical representation of each minigame. The engine also contains a list of minigames, each of which provides its own implementation of the **Minigame** interface, that forms the logic. Once the input has been received, the Engine is responsible for communicating it to each minigame: these in turn will update the state of all the GameObjects they contain, i.e. every single entity in the various playing fields. The Model part and the View part have an indirect link: the Drawings class. In fact, each GameObject is characterized by a series of specifications (Drawings), contained in the AspectModel, which the View is able to interpret to draw the object in the GUI.

2.2 Detailed design

Each minigame is composed of **GameObject**s: those items use a component pattern, thanks to which we get a full separation of concerns based on domains (allow a single entity to span multiple domains each other [1]).

[1]From GPP, CH 14

- *PhysicsModel*: Interface that deals with the physical state of a Game Object, moving it according to its speed, considering the environment in which it is located (edges of the field) and the other objects it interacts with (collision with obstacles).
- *AspectModel*: It is the interface that orders the updating of the graphics of the single object to the View, specifying which of the instructions already contained in SwingView need to be used.
- *InputModel*: Interface related to a single GameObject that reads the input stored in the engine and applies it, if the object recognizes it as its own command, changing its specifications (e.g. coordinates, speed).

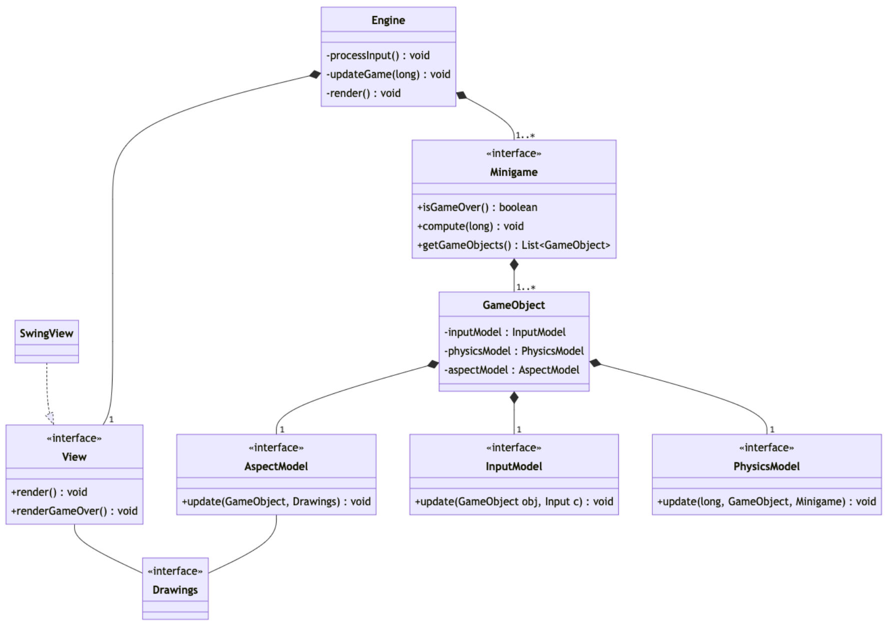

Figure 2.1: UML diagram of the architecture