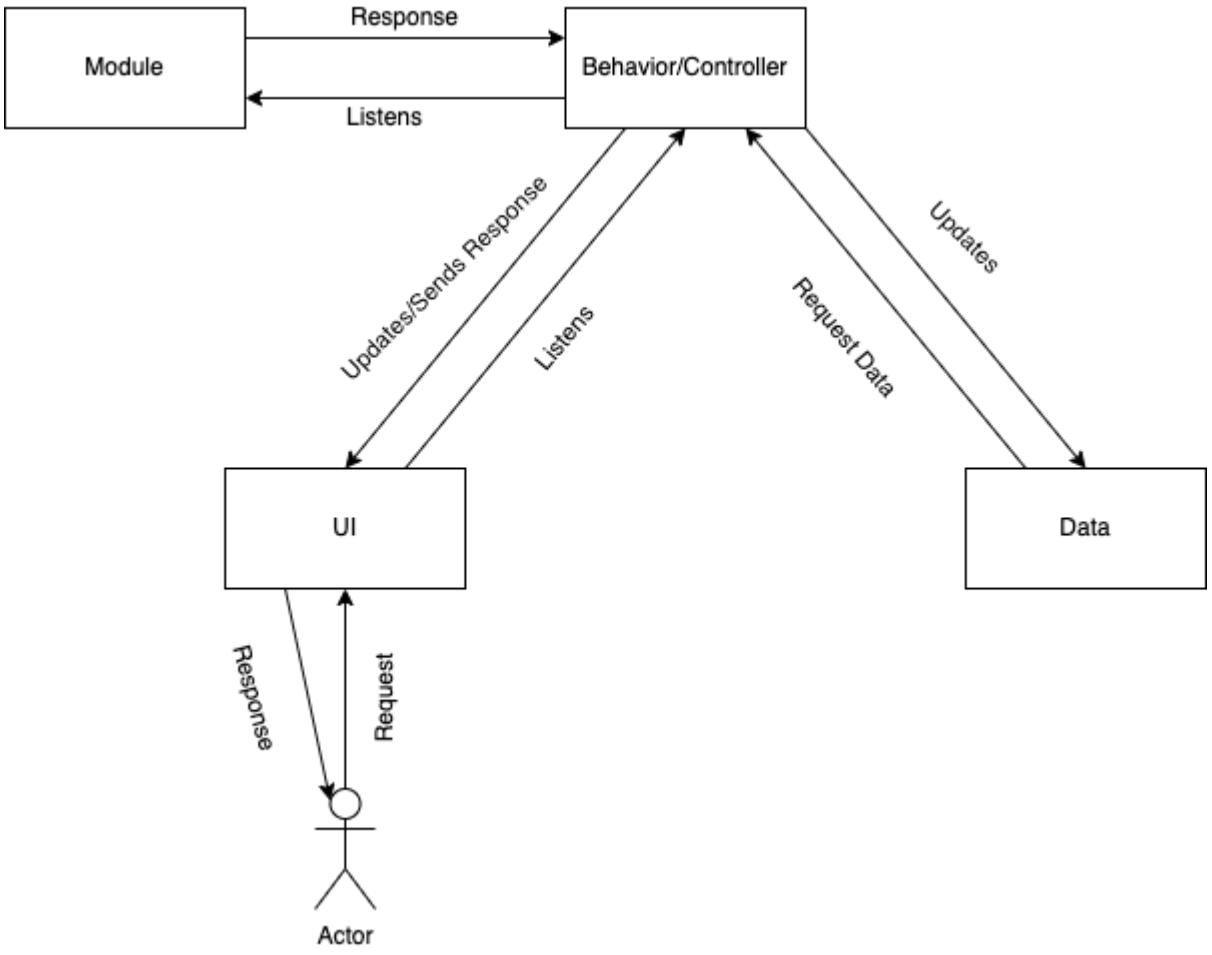
Scape From Mon Design Document

1. Design Structure

As it can be seen in the diagram, the design of the “Scape From Mon” implementation is based on the Model View Controller Principle where Behavior and Controller classes represent the Controller, UI represents the View and Data represents the Model. Behavior and Controller classes keep business logic inside them; actually only these classes are responsible for maintaining business logic. In the implementation, there are objects which represent the real 3D objects inside the game, classes are placed in these objects and these classes don’t have any logical operation inside them. UI keeps the views belonging to the objects and Data keeps the data related to the objects. For example; when the player is haunted by an obstacle, the amount of energy it has must be decreased. Controller gathers the current energy level of the player from the Data, makes some logical operation and says Data to update the energy level of the player. Data updates the energy level data. Also the Controller sends a request for UI to show the new energy level of the player.



2. Subsystems

2.1. Main Program

This system keeps the game specifical issues inside it and uses other subsystems. It gives commands to other systems like load this level, open these panels, etc. Behavior and controller classes belong to this subsystem. It is also responsible for deciding which data will be updated when, so it is the center of the logic in the game.

2.2. Game Cycle System

This system gives the game the ability of loading levels, moving to the next level and keeping the data of the player's last played level.

2.3. UI Manager System

This system is responsible for the capability of managing pop-ups and panels and deciding the issues related to them like which will be displayed next, which will be closed etc.

2.4. Collectibles System

This system acts like an inventory manager. It keeps the items that are collected inside the game in an inventory.

2.5. Player Preferences System

This system acts like a database and it keeps the data that is not session-based. This data, for example, can be the player’s selected options from the settings pop-up.

3. Patterns

3.1. Template Method

Overview
This pattern is used with a motivation of decreasing the change that may be needed in the large portion of the code and prohibiting the possibility of duplicate code in future. In the game, there may be situations where the player needs to contact the non-player characters to solve the puzzles. In these situations, we may need to override the methods according to the type of the non-player characters. To achieve this, the Template Method pattern was used.

Structure
Each non-player character will need a main logic conversation so the same method will be used; at this point the Template Pattern will get on the stage.

Behavior
Different non-player characters will be in the game with different animations and to trigger these animations, the startConversation method needs to be overridden and in each non-player character, its specific animation will be called that way.

3.2. State

Overview
State pattern was used in the game because a situation occurred where the behavior of an object changes when its internal state changes. Instead of using the State pattern, if conditions may be used, this would create a bigger burden that is hard to manage.

Structure
Classes that represent the Non-player characters, an abstract class which represents the base state class and two derived classes that are Chatting and Idle state classes are the participants of this pattern. When behavior classes are initiated, a state array is set up which has two elements: a Chatting class instance and an Idle class instance in which they represent the two states that an NPC can be in. When a state is updated in a behavior class, one of these two states is selected and kept in the CurrentState variable.

Behavior
Things that need to be carried out inside the update method of NPC will change depending on the value of the currentState variable.

3.3. Observer

Overview
This pattern is used to set up a subscription mechanism between behavior/controller classes and the modules, also between the view classes and the behavior/controller classes. We applied this pattern to get rid of the mess of updating the publisher's code when a new subscriber class is added.

Structure
Participants of this pattern are the modules, the classes inside these modules, UI classes and the controller/behavior classes. The relation between them is represented in the diagram presented in the first section.

Behavior
When a player collects a key in one stage, this key will be shown in the panel that is called an in-game panel.

3.4 Mediator

Overview
To reduce the dependency between the classes and to prohibit the possibility of having chaotic relations between these classes. So we enforce the object to collaborate with other objects through a mediator class.

Structure
In the Game Manager class, there is a method called awake. Instead of giving awake to each of the classes and losing the control because of that, each class calls the awake method inside this class.

Behavior
The Awake method inside the Game Manager class is the one who initializes and controls everyone. When the Game Manager ensures that everyone is ready then the game is ready to be played. So the Game Manager class acts like an entry point.

3.5. Singleton

Overview
The Singleton pattern is used to create a global access point to the UI Manager class and prevent the possibility of having multiple manager classes that are responsible for the UI.

Structure
UI Manager class and behavior/controller classes are the participants of this pattern.

Behavior
The UI Manager class has a unique instance that controls everything related to the view of the game.

4. Use-Case Realizations