Scape From Mon
Architecture Notebook

1. Purpose

This document is here to serve as a description of the architectural aspects of project Scape From Mon. It intends to clarify the points related to the chosen architecture and the reasons behind this selection. From Section 2 to 5, details related to the motivation behind the architecture, problems that it aims to solve and dependencies that drive the architecture can be found. In the rest of the sections, explanations will be given to describe the abstractions that are used in the architecture and the diagrams will be presented to resolve the architectural views of the system.

2. Architectural Goals and Philosophy

Game developments are very prone to having spaghetti code at the end of the day since because of the dependencies between features. Let's make it more concrete with an example from "Scape From Mon". In the game, there is a character which walks and collects objects. These two functionalities can't exist without one another. This makes development phase of these two features dependent on each other. To get rid of this side effect, in the development of "Scape From Mon", we applied more modular architecture where developers can work individually and risk of having code chaos in the middle of the development may reduce significantly.

Another aspect that drives modular architecture is having reusable code pieces/components. In game developments, it is very likely to have one feature in multiple games. To make this an advantage for us, by means of the modular architecture, we have been able to use structs/components from other game developments and also keep the developments that are made for "Scape From Mon" for usage in another game.

3. Assumptions and Dependencies

Following assumptions and dependencies that drove using highly modular architecture:

- Functionalities in "Scape From Mon" are very cohesive which creates dependency between team members in development phase.
- High rate of design pattern usage in the game that may create chaos and spaghetti code at the end of the day.
- Time is limited; this creates a need for having reusable code which is possible with modular architecture.

4. Architecturally Significant Requirements

Following requirements from the System Wide Requirements document need to be implemented to realize the architecture:

- The application collects data on the user's game performance and transfers them to BI tool. The sending application data is in JSON format.
- The game will support players to choose a level to start or start a new game.
- Players will be able to delete their progress.
- The game will be able to be completed starting from the Intro section in one run, without having any errors such as crashes or problems that may cause the player to not be able to progress.
- After the first scaling of the game, new levels will be added with 1 month length update iterations.
- Main character is customized with a name and skin.
- Graphics will be supported through sounds.
- Sound label with on and off buttons.
- Language label with Turkish and English buttons.
- The application shall be given to three levels of gameLevel1/Level2/Level3.
- The game shall be automatically loaded to the next level if the current one ends successfully.
- The game automatically ends, and the current session is lost.

5. Decisions, Constraints, and Justifications

- Since in the game development, Model View Controller architectural pattern will be used, it is not allowed for developers to include Business Logic other than the Controller classes and the classes/modules will be developed by obeying Single Responsibility Principle.
- External packages used are not protected so instead of modifying these packages, modify behavior classes that are using them to obtain desired action. This restriction is here to prevent damaging the functionality of the packages.
- Follow SOLID principles.
- Don't call the *"awake"* method everywhere, instead call the *"initialize"* method. Awake will be called only from the *"Game Manager"* class.
- Application will be created on Unity platform and the development will be carried out using C#.
- Application will be available on Android Store.
- To test the application, a Windows version of the game will be constructed, too.

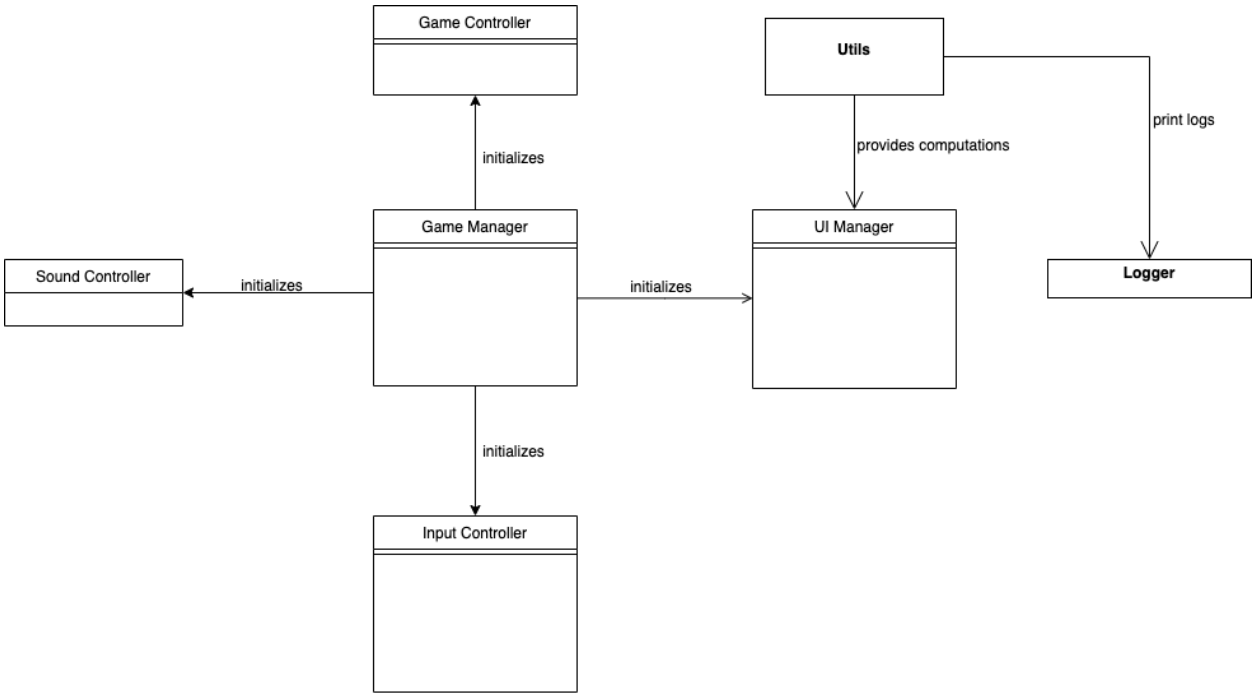
6. Architectural Mechanisms

- To find problematic points that propagate the development of the "Scape From Mon", elements that will support debugging and error management will be used.
- Localization will be applied to support users' shift between English and Turkish for the scripts in the game.
- When a crash happens in the game, a crash report will be created. This report will be sent to the admin. (Note that this mechanism will be available in final iterations.)

7. Key Abstractions

- *Logger:* Responsible for printing logs depending on the *"Utils"* module request.
- *Game Manager (Game Cycle):* Responsible for functionalities related to the *"Play Game"* use case like level loading, start or end level. Also, this module keeps business logic related to the main menu.
- UI Manager: Responsible for providing user interface elements.
- Utils: Responsible for encapsulating the tools that will not be related to the game functionality directly, and will act more like a helper.
- Game Controller: Responsible for level loading. When a level ends, it loads the next level by keeping references for all levels.
- Sound Controller: Responsible for handling all sound related aspects.
- Input Controller: Responsible for perceiving all inputs that exist other than UI. Wraps Unity's input mechanism.

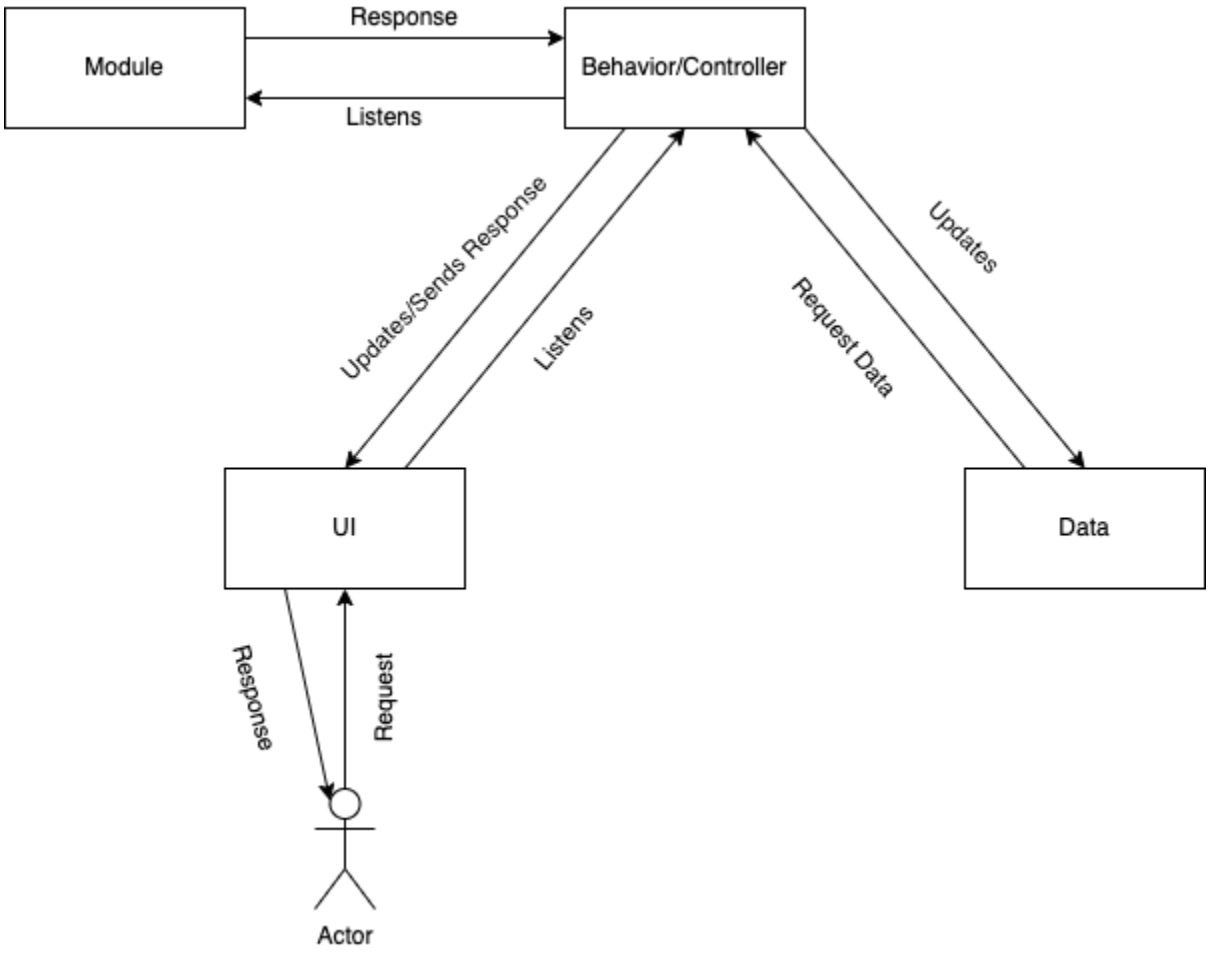
Class Diagram:



8. Layers or architectural framework

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

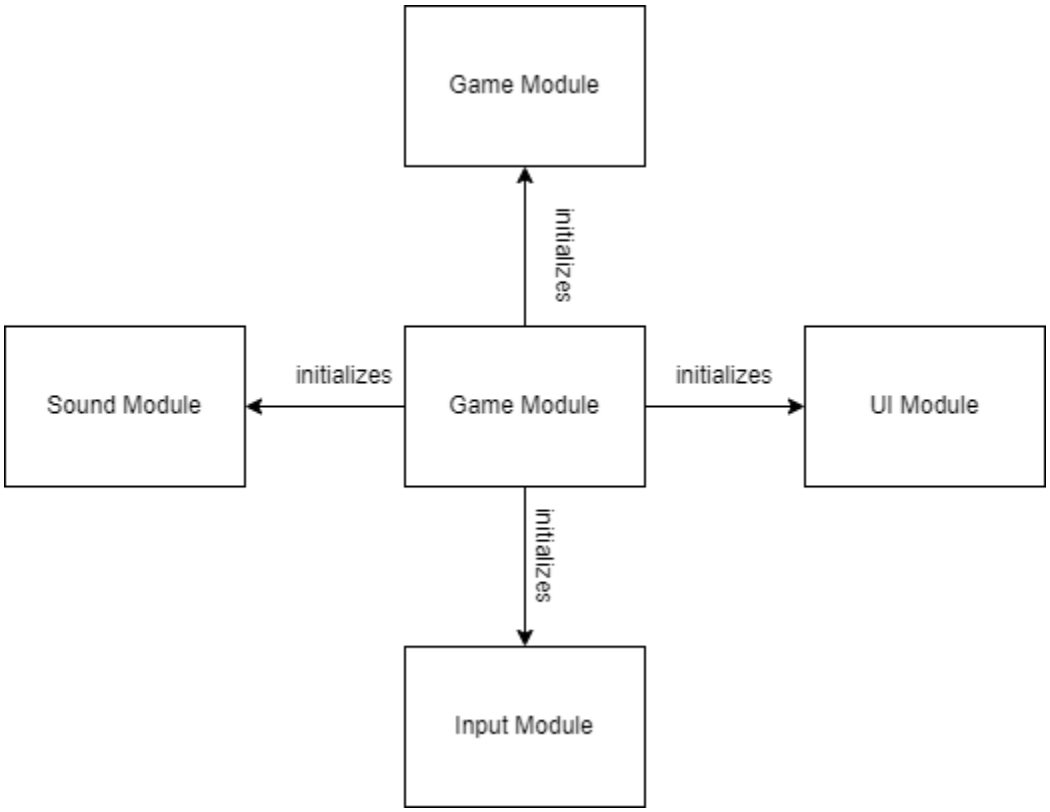
implementation, there are objects which represent the real 3D objects inside the game, classes are placed in these objects and these classes don't have any logical operation inside them. UI keeps the views belonging to the objects and Data keeps the data related to the objects. For example; when the 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.



9. Architectural views

Logical View

Package Diagram:



Deployment View

Deployment Diagram: