**Software Design Document**

**for**

**TohoGame**

Team: teamidk

Project: TohoGame

Team Members:   
*Joe Steeb*

*Hugo Aponte*

*Joshua Long*

*Ben Kaufman*

*Dahlia Xhe*

**Last Updated: [3/25/2023 4:51:38 PM]**

# **Table of Contents**

[Table of Contents 3](#_17dp8vu)

[Document Revision History 4](#_3rdcrjn)

[List of Figures 5](#_26in1rg)

[List of Tables 6](#_lnxbz9)

[1. Introduction 7](#_35nkun2)

[1.1](#_tyjcwt) Architectural Design Goals 7

[1.2](#_1ksv4uv) Scope 7

[2. Software Architecture 8](#_44sinio)

[2.1 Overview 8](#_2jxsxqh)

[2.2 Subsystem Decomposition 8](#_z337ya)

[3. Subsystem Services 10](#_2s8eyo1)

# **Document Revision History**

| **Revision Number** | **Revision Date** | **Description** | **Rationale** |
| --- | --- | --- | --- |
| **1** | 3/20 | Document Formatting | Setting up the document. |
| **2** | 3/24 | Architecture Design Development | Detailed architecture design in the proper subsection. |
| **3** | 3/25 | Software Architecture Development | Detailed software architecture (MVC) and incorporated diagrams. |
| **4** | 3/26 | Subsystem Decomposition and Services/Interfaces Development | Detailed subsystem decomposition, services/interfaces required/provided, and incorporated more diagrams |
| **5** | 3/26 | Final formatting changes | Reviewed document for submission |

# **List of Figures**

**Figure # Title Page #**

1-1 Use Case Diagram #

1-2 Model Class Diagram #

1-3 View Class Diagram #

1-4 Controller Class Diagram #

1-5 Subsystem Composition Diagram #

# **List of Tables**

**Figure # Title Page #**

# 1. Introduction

TohoGame will consist of a player evading moving enemies and their bullets, as they move and shoot bullets themselves. There will be various, customizable game stages the player must get through to win and update the TohoGame best score.

## **Architectural Design Goals**

The TohoGame system emphasizes a couple non-functional requirements: Performance and Testability. The tactics associated with the Performance non-functional requirement are managing sampling rate, limiting event response, prioritizing events, reducing overhead, bounding execution times, and increasing resource efficiency. Our program seeks to implement limiting event response, prioritizing events, and increasing resource efficiency tactics. The tactics associated with the Testability non-functional requirement are black box testing, white box testing, and other forms of testing. We have mainly used whitebox testing with the help of TohoGame debug attributes (such as debug, god mode, no damage, etc) which alter the state of the game to help us target certain functionality more efficiently. We are still developing tests to target unit level functionality in the form of executable assertions. In the future, these will help us maximize Testability for the TohoGame system. In addition to Performance and Testability, we believe Modifiability is important as well. The TohoGame system will aim to be flexible to changing requirements and interfaces, which will require loose coupling and high cohesion amongst components and subsystems. In turn, applying tactics to strengthen modifiability will also strengthen testability, through loose coupling, high cohesion, and separation of concerns.

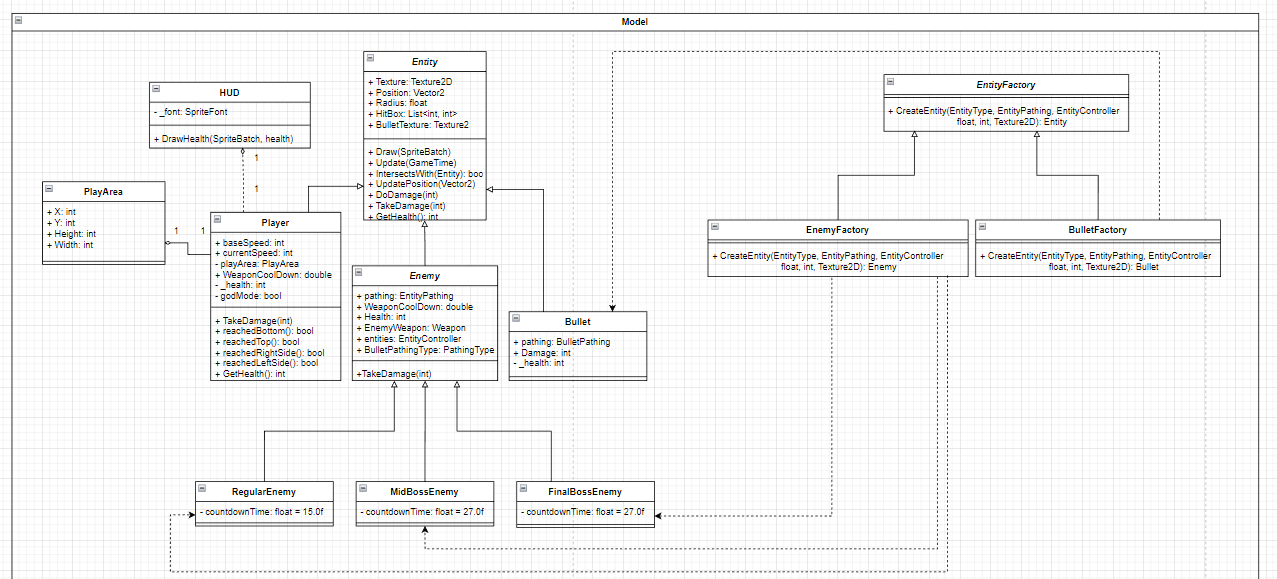
# 2. Software Architecture

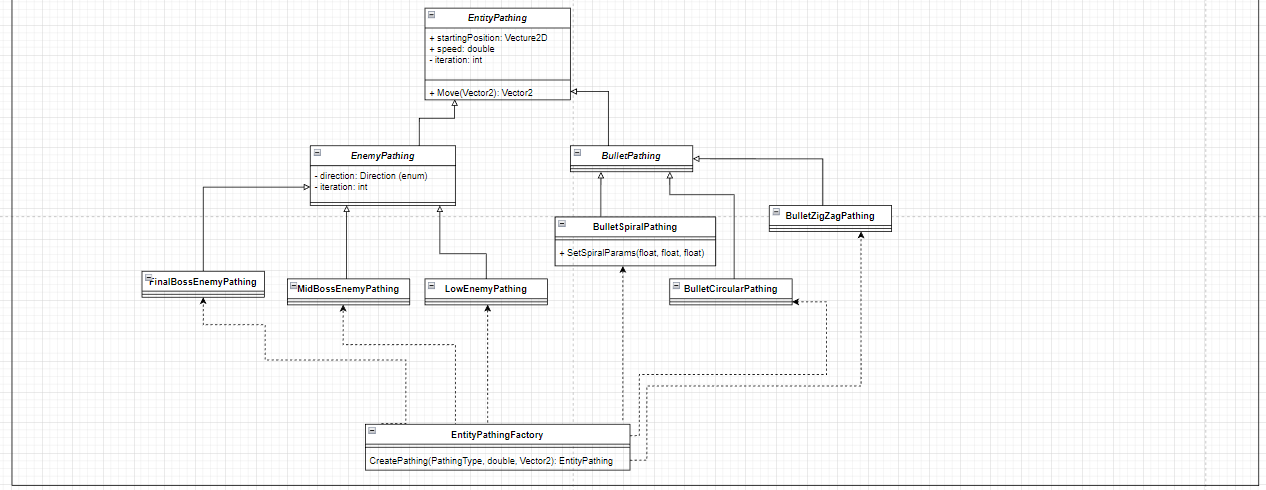
**Figure 1-1 Use Case Diagram.**

## Sequence Diagrams were not completed.

## **2.1 Overview**

TohoGame utilizes the Model-View-Controller architectural design pattern to orient design decisions and patterns amongst subsystems. The Model consists of the data being stored and manipulated by the View and Controller subsystems; it supplies proper interfaces to the View subsystem to Load Content, Draw, and Update; and it supplies proper interfaces to the Controller subsystem to manage the interactions between objects represented by the Model, such as FireBullet, HandlePlayerMovement, and DetectCollision. The ultimate goal was to satisfy important software design principles like Liskov’s substitution principle in the form of substitutable entities, paths, and associated tendencies. Another case is the Single Responsibility Principle in the form of specific entity and pathing factories, as well as StageManager and EntityController. In essence, our subsystems seek to provide and require satisfactory interfaces to facilitate data manipulation, display, and high performance.





**Figure 1-2 - Model Class Diagram.** Class diagram depicting the Model subsystem. We have utilized the Factory Design Pattern multiple times to create Entities and EntityPathing objects. These objects also utilize inheritance to loosen coupling amongst objects; this also amplifies our focus on Liskov’s substitution principle.

## 

## 

## 

## Figure 1-3 View Class Diagram. Class diagram depicting View subsystem. We utilized the Composite design pattern for the EntityController, where it is made up of Entities like Player, Enemies (different types), and Bullets. These objects have different functionalities but similar interfaces to facilitate View functionalities such as Draw, Update, LoadContent.

## 

**Figure 1-4 Controller Class Diagram.** Class diagram depicting Controller subsystem. Here we have also depicted the EntityController’s composite design as well as Entity and EntityPathing Factory design patterns.

## **2.2 Subsystem Decomposition**

### 2.2.1 Model Subsystem

The Model subsystem contains two main components; Entity/Pathing and EntityFactories. The Entity/Pathing component is composed of the models themselves such as Enemy, Pathing, and Bullets. On the other hand, the EntityFactories component is composed of the different factories used for instantiating the models.

### 2.2.2 Controller Subsystem

The Controller Subsystem also contains two main components. The StageManager takes a JSON Script input and parses the information into the phases of the game. This is where instantiation of the models takes place. The EntityController component handles updating the models and is used for communication between the other two subsystems.

### 2.2.3 View Subsystem

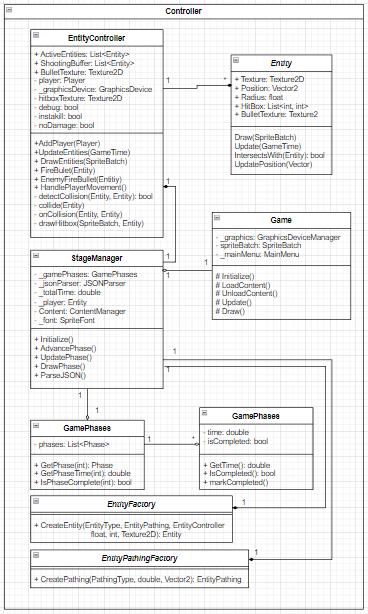
The View Subsystem contains the user interface for the system. This subsystem handles the visual aspects of the game along with collision of the sprites. When collision is detected, requests are sent to the Controller subsystem for interpretation and updates are received from the Model subsystem for drawing to the user interface.

### 2.2.4 Design Patterns

The main creational pattern used for instantiating entities is the Factory Pattern. Since there are many variations of each of the entities (Pathing, Enemies, Bullets), it is important to properly instantiate based on said variation. Instead of using a Factory Method that could be altered by subclasses, our system uses Factory classes themselves to handle instantiation. This way, an external class (StageManager) can cleanly handle the creation of entities.

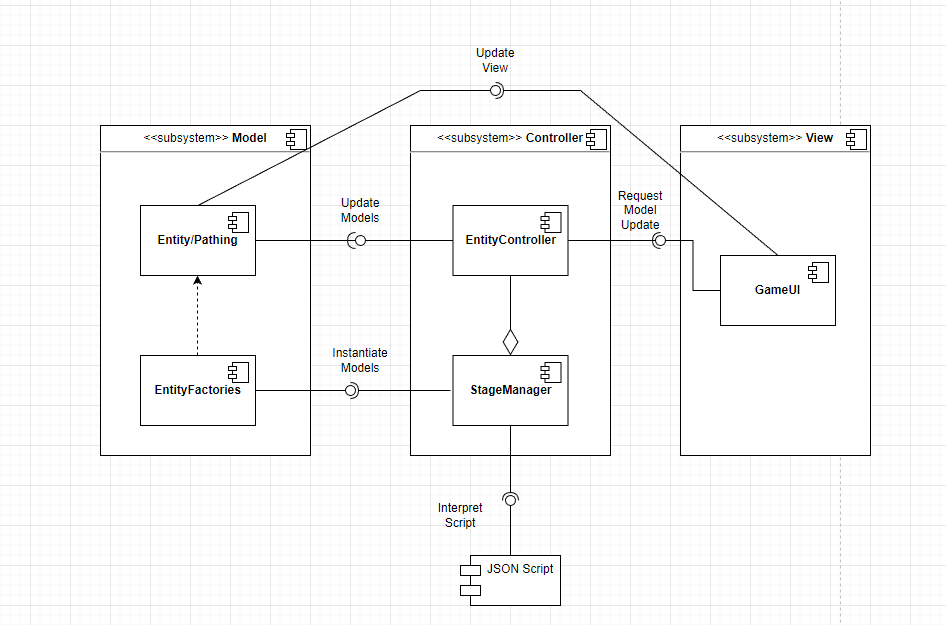
When it comes to handling the different types of entities, the Composite Pattern was utilized. EntityController contains many types of entities like Player, Enemies (different types), and Bullets. These objects have different functionalities but similar interfaces to facilitate View functionalities such as Draw, Update, LoadContent.

The overall architectural design pattern followed was the Model-View-Controller Pattern. As stated before, this pattern separates the system into three main subsystems to avoid coupling.



**Figure 1-4 Controller Class Diagram.** Here we have also depicted the EntityController’s composite design as well as Entity and EntityPathing Factory design patterns.

# 3. Subsystem Services



**Figure 1-5 Subsystem Component Diagram.** Required and provided interfaces amongst subsystems.

Interpret Script: StageManager is given a JSON Script which determines the timing and entity types that each phase of the game script contains.

Instantiate Models: StageManager uses factory methods from the different entity factories to instantiate entities.

Update Models: Entity Controller updates the models of the Entities/Pathings in accordance to what is happening in the game.

Request Models Update: The UI of the game requests model updates via the EntityController. These are updates which are view-driven such as collision.

Update View: The GameUI utilizes the Update() and Draw() methods within the Entity and Pathing models to update the game visually.