**Software Design Document**

**for**

**BulleitHell**

Team: BulleitHell

Project: BulleitHell

Team Members:   
*Bobby Belter*

*Cong Trinh*

*Garrett Rudisill*

*Jeff Walsh*

*Mike Berger*

*Slater Weinstock*

**Last Updated: [3/2/2019 12:28:48 PM]**

# **Table of Contents**

[**Table of Contents**](#_gjdgxs) **1**

[**Document Revision History**](#_oyzltv7v17sr) **2**

[**List of Figures**](#_4wz428l80nss) **3**

[**1. Introduction**](#_n7iw3rqnlz6m) **4**

[Architectural Design Goals](#_o3g6r9h8g7hq) 4

[**2. Software Architecture**](#_wmnav9ondwbd) **7**

[2.1 Overview](#_l1phderfvvh8) 7

[2.2 Subsystem Decomposition](#_q1eot2ir0cg6) 9

[**3. Subsystem Services**](#_37wbmccxz1v6) **12**

# 

# **Document Revision History**

|  |  |  |  |
| --- | --- | --- | --- |
| **Revision Number** | **Revision Date** | **Description** | **Rationale** |
| **1.0** | 3-10-19 | Baseline Document | N/A |
|  |  |  |  |

**List of Figures**

**Figure # Title Page #**

1 Use Case 1: Create Enemies Sequence Diagram 5

2 Use Case 2: Check Collisions Sequence Diagram 6

3 Use Case 3: Update Positions Sequence Diagram 6

4 Architecture Overview 8

5 Abstract Factory UML 10

6 Factory UML 11

7 Game Properties Singleton 11

8 Ball-Socket UML 13

(etc.)

# 

# 

# **1. Introduction**

Our project is a bullet hell game with a strong sense of humor to it. Rather than traditional themes often inspired by anime, the bullet hell shooter will feature alcoholic theming, tying into some of the team member's love for Bulleit whiskey. Sprites will have themes such as varying brands, bottles, shot glasses, and the like.

## **Architectural Design Goals**

The two primary design qualities we are trying to imbue in our software are performance and availability. These are both key in the design of any video game. Performance is first and foremost in any game, as any perceived latency can be extremely frustrating to the player. This applies exponentially to any sort of competitive or fast past game, such as a bullet hell shooter with many fast moving enemies and projectiles.

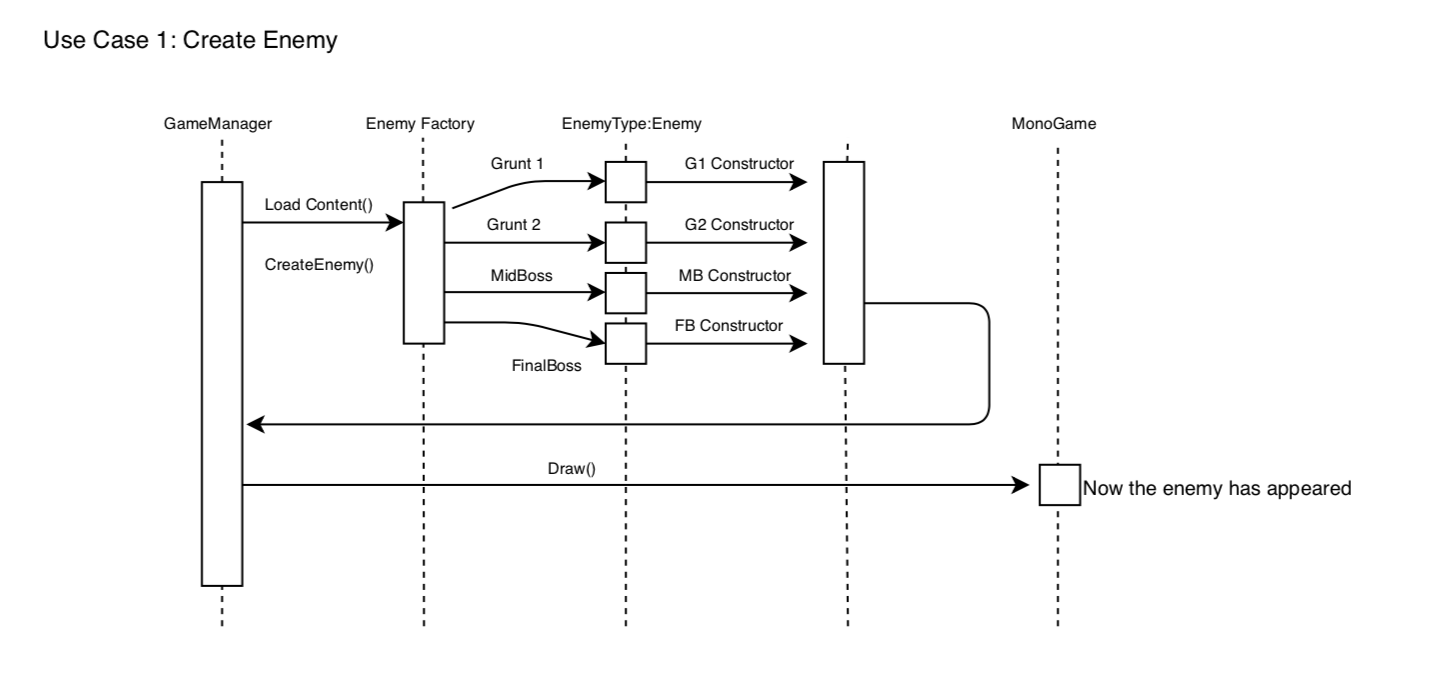
Secondly, we desired availability. The only thing more frustrating than lag in video games are game breaking glitches, crashes, and unintended undesirable behavior. The last thing a player wants is to crash on the final boss of a rogue-like run, be unable to reload in a shooter, or be unable to move to dodge projectiles. The game must perform well, but also perform without fault.

# 

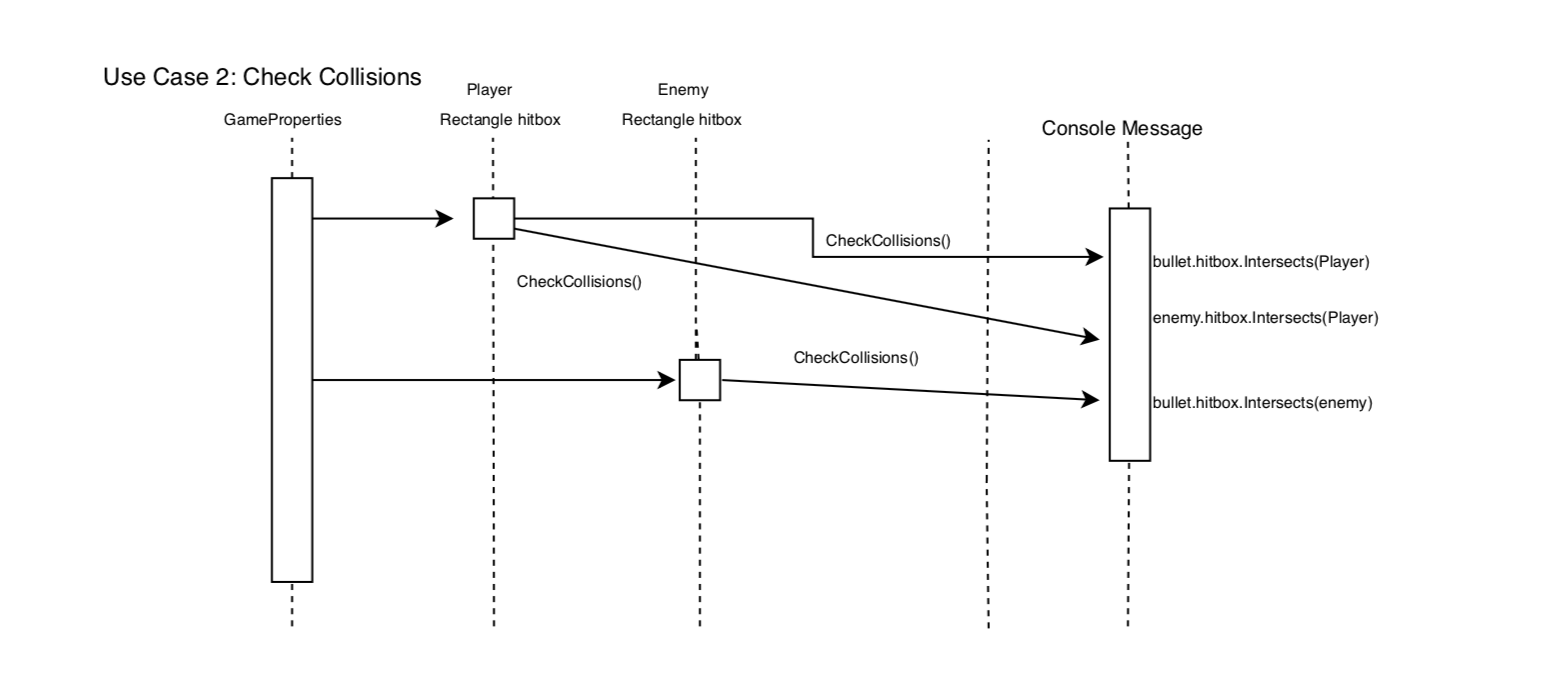
# 2. Software Architecture

There are many use cases within the overall software architecture. For this section we will consider a couple of the main use cases.

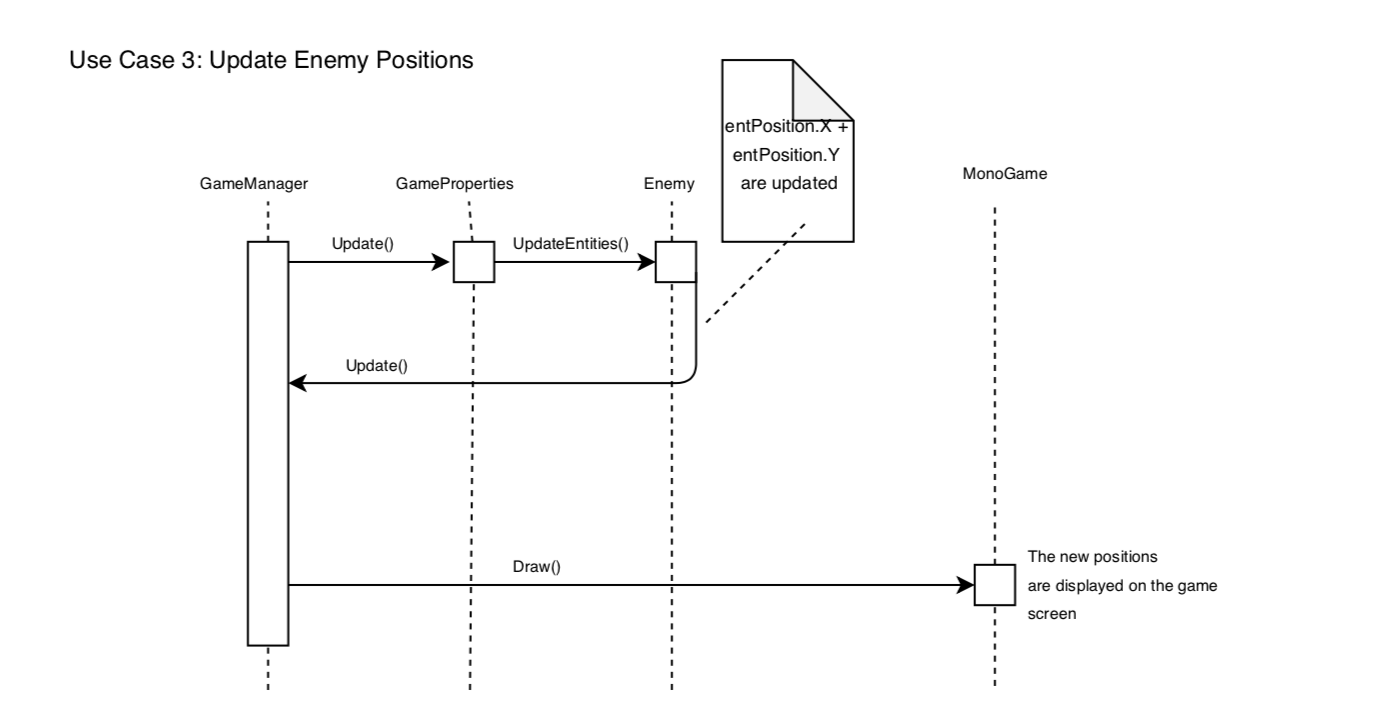
Use Case 1: Create enemy



Use Case 2: Check hitboxes for collisions



Use Case 3: Update enemy positions



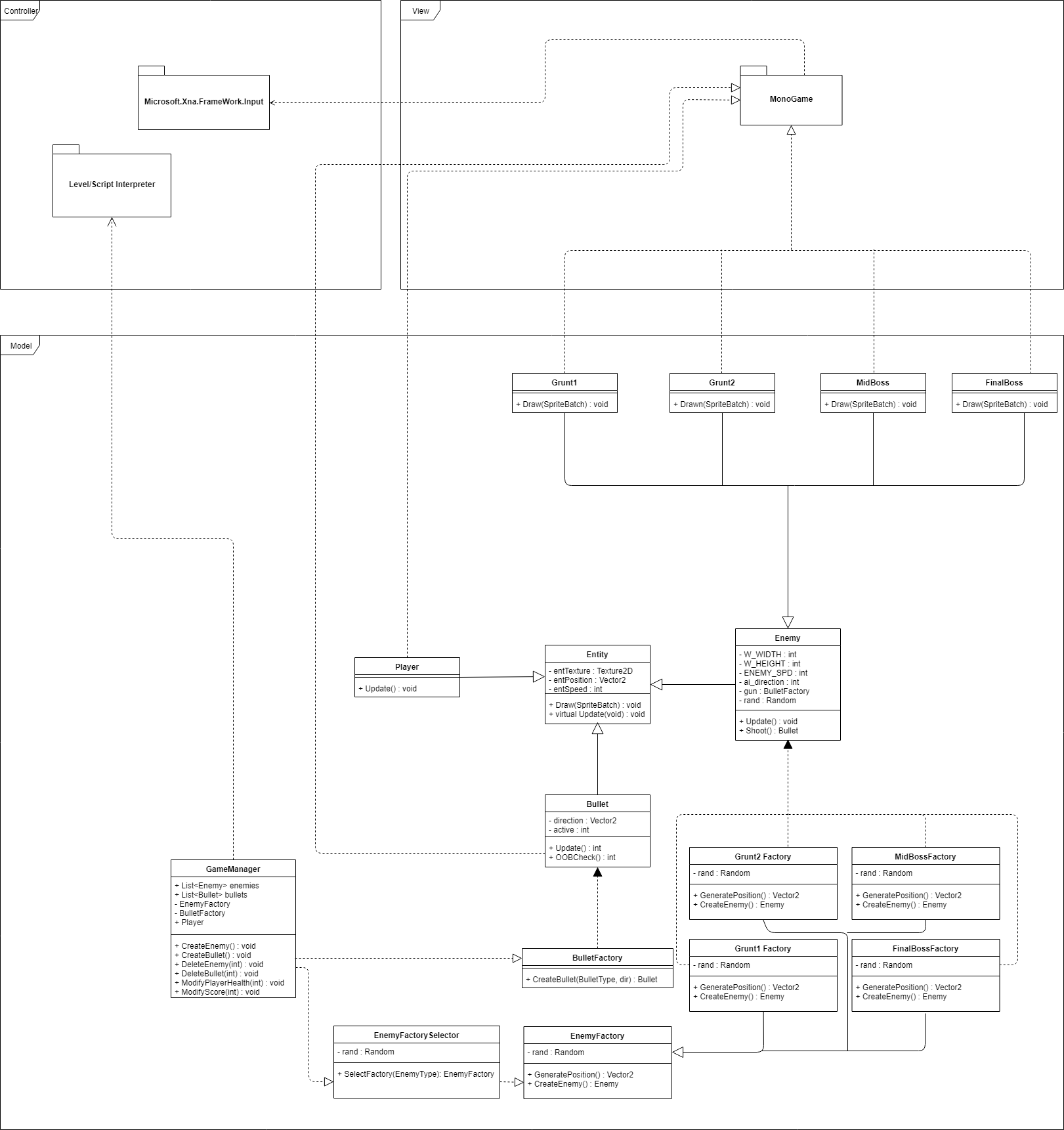
## 

## 

## **2.1 Overview**

Our game uses the MVC architecture. We chose the MVC architecture because of its simplicity in relation to the scale of this project. Being a small game with few components, there is no need to come up with a complex architecture design that would ultimately make this project more difficult than it needs to be. In addition, Monogame provides us with a lot of prebuilt functionality that covers a majority of the View, and Controller subsystems of the architecture.

In our implementation of MVC, the model is represented by our game data, such as enemies, score, lives and related info. The view is the visual representation, updated on each draw for the player. Finally, the controller handles input from the player to update both the model and view respectively.



## **2.2 Subsystem Decomposition**

### 2.2.1 Model

The model is a collection of all data and data control that exists within the game. The model is composed of the GameManager, a singleton that holds all of the information that changes at runtime. This includes all players and bullets seen on screen as well as the appropriate methods to modify, add, and remove these objects from the playable area.

Additionally, All of the player, enemy and bullet definitions can be found inside the model. This includes all necessary factories, and abstracts involved in making the object interaction clean and easy.

### 2.2.2 View

The view is a very simple package that consists only of the monogame framework for drawing objects to the screen. This framework allows us to easily output objects to a well defined screen space and frequently update any changes of the model for the user to see.

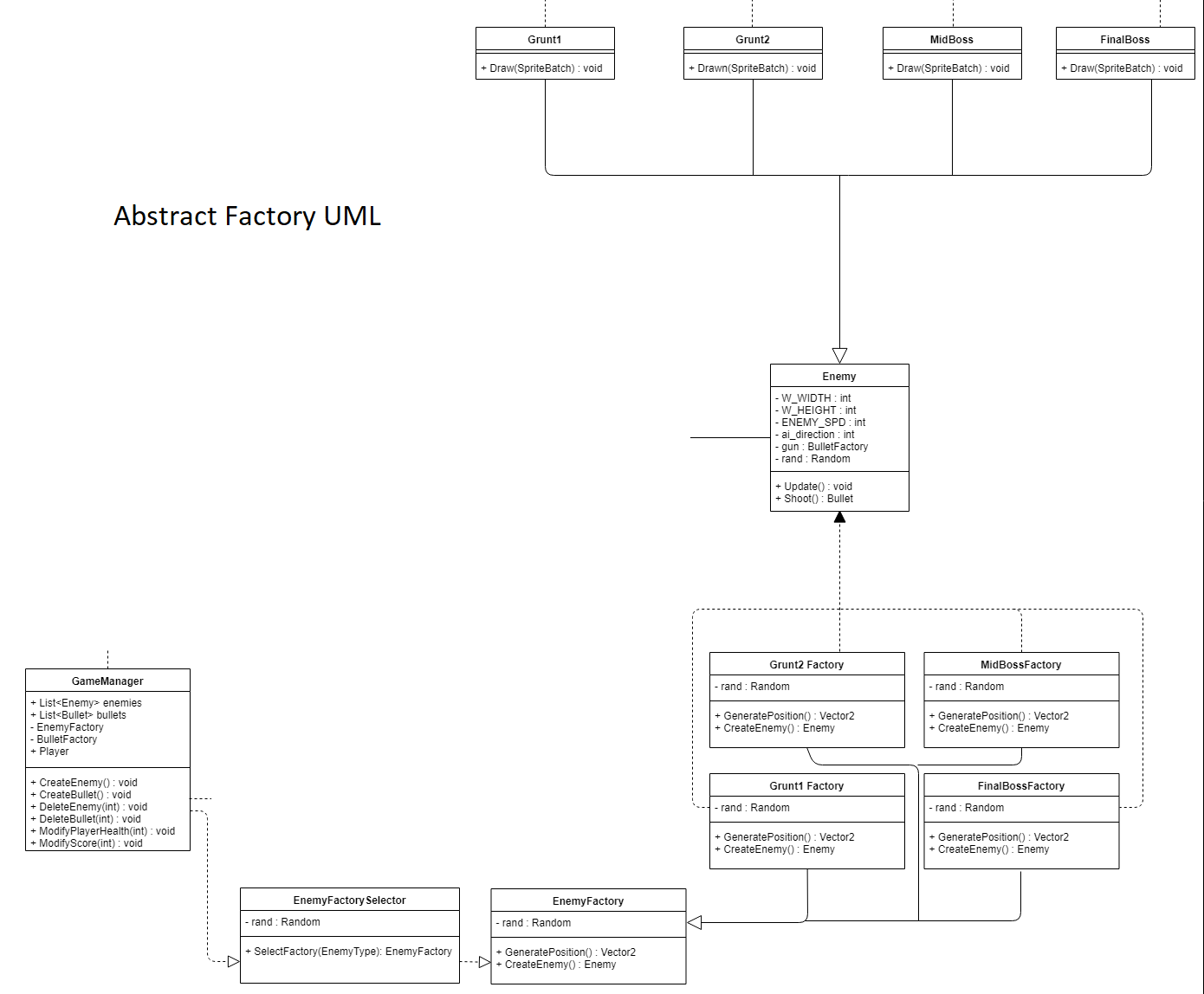
### 2.2.3 Controller

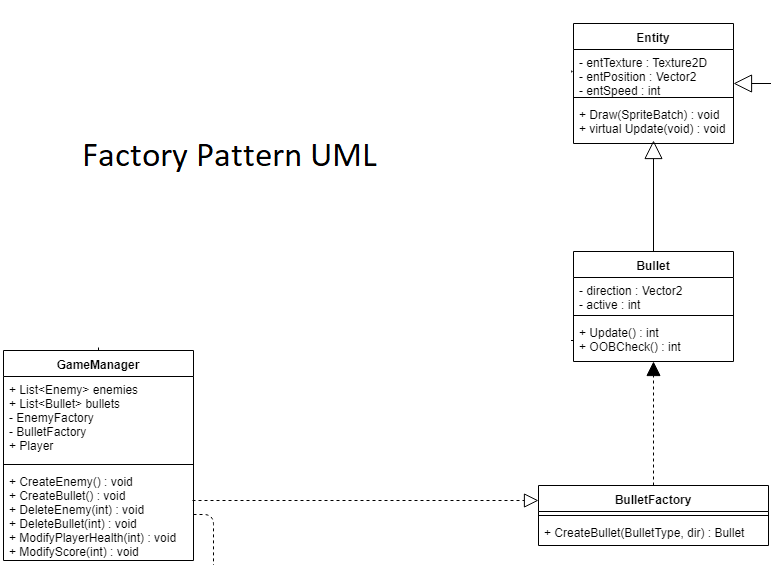
The controller is a very simple package consisting of two elements. The XNAFramework for input and the level/script interpreter for running game levels. The XNAFramework for input is what allows for a variety of different user inputs ranging from controllers to keyboards and mice and many more. The level/script interpreter is code we have written to take a level script as input to control the enemies in a scripted manner.

### 2.2.4 Design Patterns

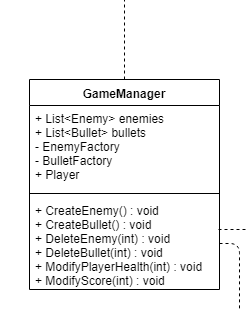
We heavily use the following design patterns within our application: Factory, Singleton, and Abstract Factory.

Enemy factory: All enemies are created using an abstract factory interface. This allows for an easier use of the enemy factory interface when creating a variety of different enemies at a given time.



Bullet Factory: This is a basic factory from which all bullet objects are generated. These are concrete objects, which inherit from the entity class. Any time the bullet factory is called to create a bullet, it is added to the game properties singleton, which keeps a list of bullets to manage.

Game Property Singleton:



# 3. Subsystem Services

**Model:** The model is represented by the Game Properties singleton, which contains the significant majority of data necessary to update the view.

Game Properties Singleton provides:

* LoadSprites() - Loads all sprites into the content manager struct of MonoGame
* CheckCollisions() - performs all hitbox calculations
* DrawEntities() - calls draw function on all active non player entities
* UpdateEntites() - calls update functions on all active non player entities
* UpdateScore() - incremements score value

**View:** The view is handled by MonoGame framework. This subsystem handles updating the data as needed, and drawing so that it is visually represented for the player to interact with.

Monogame Framework provides some of the following:

* GraphicsDeviceManager - sets width and height of screen. Allows access to SpriteBatch which is necessary for draw calls and representing objects visually
* Draw() - adds the new information to the frame buffer to be drawn on call of Update()
* Update() - draws a new frame to the screen, representing any changes that have been made to the frame buffer

**Controller:** The controller handles all user input which is needed by the model to signal when and how to update the view. The controller portion is also handled by MonoGame Framework, but can effectively be treated as a separate system.

The Controller portion of MonoGame Framework provides:

* KeyboardState - allows us to check which keys are pressed/not pressed
* IsKeyUp() - check for a key released state
* IsKeyDown() - check for key pressed state

The Level Interpeter provides the following services:

