Software Design Document
for
Bravens Bullet Hell Shooter

Team: Bravens

Project: The Mono Game

Team Members:
Casey Martin
Joan Njenga
Ryan Aloof
Dante Cofano
Aidan Gaetano
Kyle Parker

Last Updated: [3/22/2025 12:12:18 PM]

Table of Contents

[Update the Table of Contents]

Document Revision History

Revision Number	**Revision Date**	**Description**	**Rationale**
0.0	3/10/2025	Initial Draft	Created based on project overview
1.0	3/10/2025	Final Draft	Document is in an up-to-date state as of Milestone 2

List of Figures

List of Tables

1. Introduction

The Bravens project is a bullet hell shooting game developed using the MonoGame framework. This game is designed to provide players with a fast paced and challenging experience, characterized by intense enemy encounters and different bullet patterns which may be challenging a aplayer based on the game's difficulty level.

To achieve this, the game features a component-based architecture that promotes modularity and reusability, allowing for flexible game object behaviors. This also allows the developers working iteratively on it to apply modifications and enhancements easily without disrupting the overall system. The project includes various assets, game logic, and manager classes to handle different functionalities, such as enemy waves and player controls.

1.1 Architectural Design Goals

This section discusses the architectural design of the bravens project which focuses on several key qualities that are essential for delivering a high quality experience, including:

- **Performance**: The game should run smoothly with multiple enemies and projectiles on screen. Our biggest concern with regards to optimization is constantly instantiating and destroying game objects. While we could take steps to optimize our game via object pooling, our project is a 2D game, so optimizations are not a concern at this time.
- **Modularity**: The component-based architecture allows for easy addition and modification of game behaviors without affecting other parts of the system. Tactics include encapsulation and composition which enable the developers working on this game to create and manage game components independently.
- **Maintainability:** The design promotes clean code practices and separation of concerns, making it easier for developers to maintain and update the game over time. This has been crucial for addressing bugs, implementing new features especially in this milestone and ensuring an overall smooth player experience**.**

2. Software Architecture

This section documents the architectural model of the Bravens' implementation of the game. The architecture is designed to support the core gameplay mechanics and facilitate interactions between various components. To illustrate the system functionality, we have identified three key use cases that represent critical interactions within the game as shown by the figure below:

Figure 2-1: **Use Case 1:** AI Spawns enemies - This use case describes how the game's AI system manages the spawning of enemy units, determining their types and behaviors based on predefined logic.

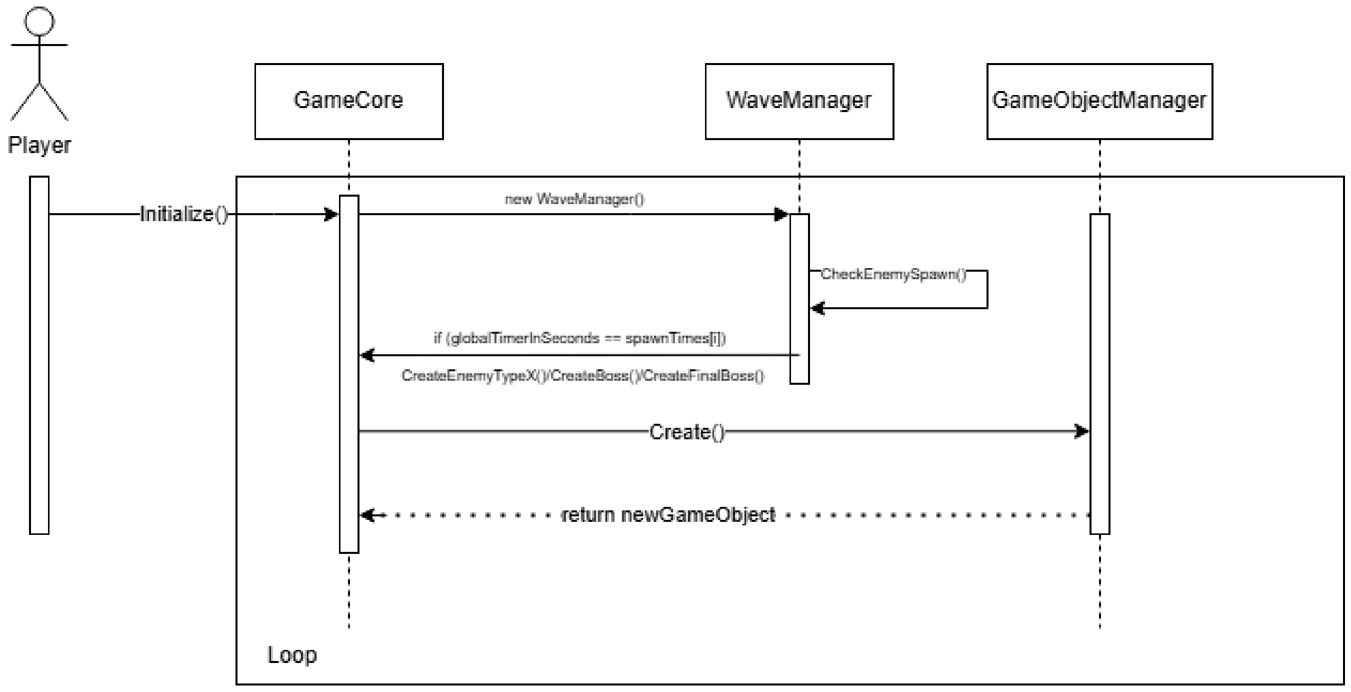


Figure 2-2: **Use Case 2:** User controls player to move - This use case shows how players interact with the game through input controls, allowing them to maneuver their character and avoid enemy projectiles.

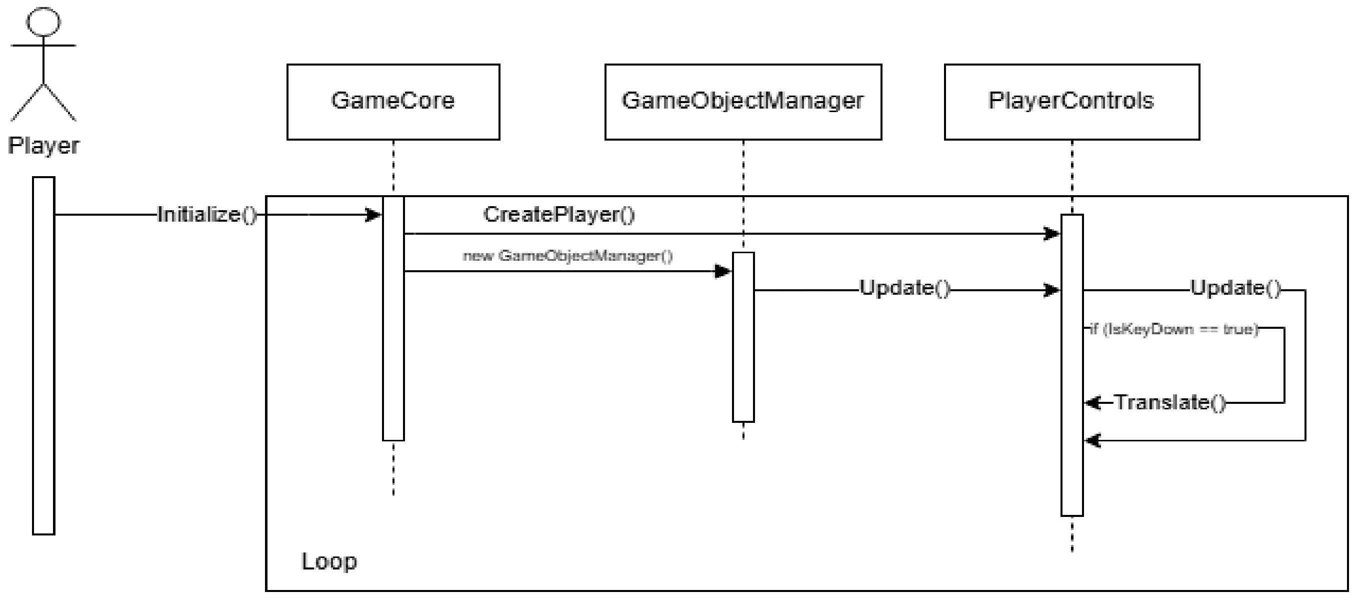
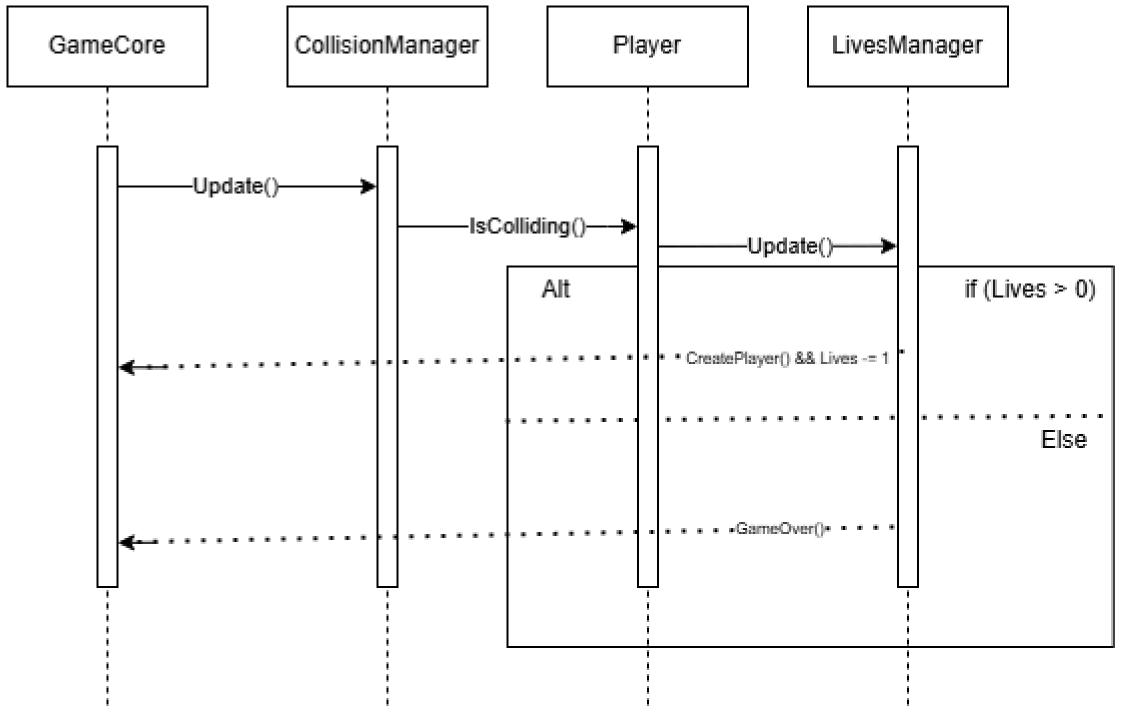


Figure 2-3: **Use Case 3:** Player is hit by enemy bullets - This use case shows the consequences of player interactions with enemy projectiles, including death conditions which leads to game over.



2.1 Overview

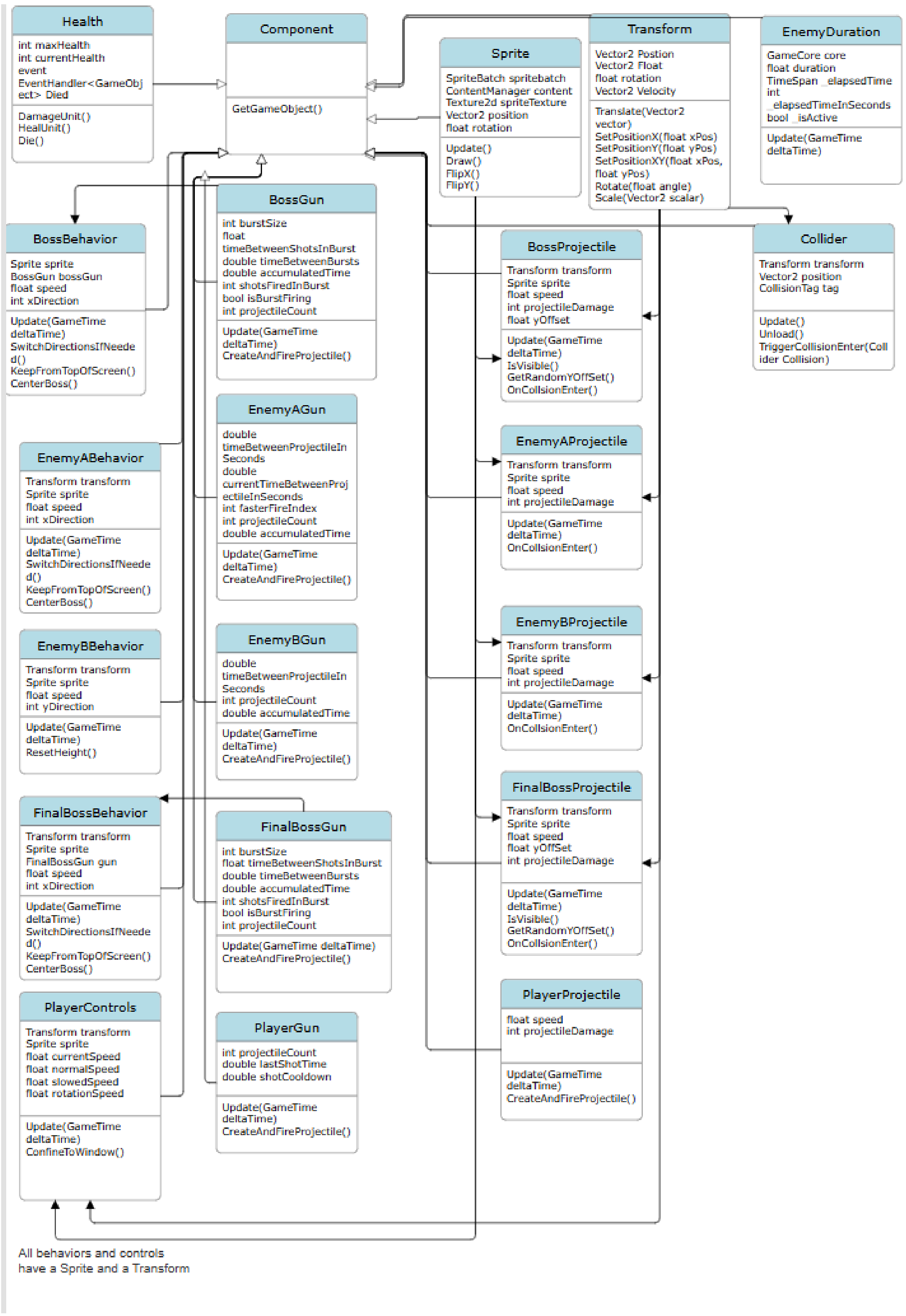
The architecture follows a component-based design pattern, allowing for flexible and reusable components that can be attached to game objects. This pattern while it allows for modularity, it also allows for flexible attachments of behaviors to game objects, enabling developers to easily modify or extend functionalities without impacting the entire system. The main subsystems include:

- **GameCore**: Manages the overall game loop and initializes game objects.
- **GameObjectManager**: Responsible for creating, updating, and rendering game objects.
- **CollisionManager:** Manages Colliders, determines when to activate collisions between them.
- **WaveManager**: Manages the spawning of enemy waves and bosses.
- **LivesManager:** Responsible for the systems relating to the Player being hit and the results.
- **Components:** Classes that can be contained within GameObjects to give them different properties and behaviors.

All of this has been further illustrated by the detailed diagrams in section 2.2.

Components subsystems include a collection of classes that can be attached to game objects to provide various properties and behaviors, such as movement, shooting, and health. Figure 2.9 below helps illustrate the architectural pattern adopted, which further shows the relationships and dependencies among the subsystems.

Figure 2-4: **Components Diagram**



2.2 Subsystem Decomposition

In this section, we will break down the system into its individual subsystems, providing a brief description of each subsystem and their responsibilities in a detailed manner.

Figure 2-5: **GameCore Diagram :** This subsystem is the central hub that manages the overall game loop, including initialization, updating game states, and coordinates interactions between subsystems. It serves as an entry point for the game.

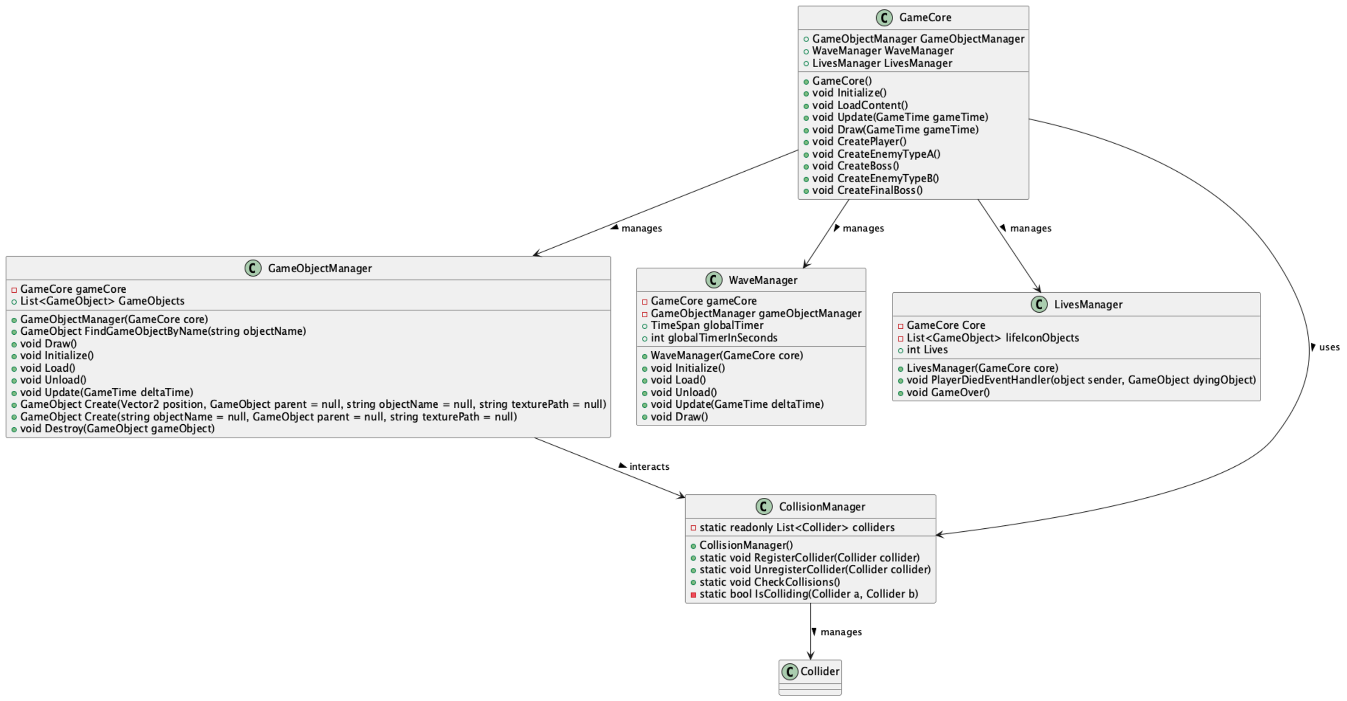

GameCore
GameObjectManager GameObjectManager
WaveManager WaveManager
LivesManager LivesManager
GameCore()
void Initialize()
void LoadContent()
void Update(GameTime gameTime)
void Draw(GameTime gameTime)
void CreatePlayer()
void CreateEnemyTypeA()
void CreateBoss()
void CreateEnemyTypeB()
void CreateFinalBoss()
manages
manages
manages
uses
GameObjectManager
GameCore gameCore
List<GameObject> GameObjects
GameObjectManager(GameCore core)
GameObject FindGameObjectByName(string objectName)
void Draw()
void Initialize()
void Load()
void Unload()
void Update(GameTime deltaTime)
GameObject Create(Vector2 position, GameObject parent = null, string objectName = null, string texturePath = null)
GameObject Create(string objectName = null, GameObject parent = null, string texturePath = null)
void Destroy(GameObject gameObject)
WaveManager
GameCore gameCore
GameObjectManager gameObjectManager
TimeSpan globalTimer
int globalTimerInSeconds
WaveManager(GameCore core)
void Initialize()
void Load()
void Unload()
void Update(GameTime deltaTime)
void Draw()
LivesManager
GameCore Core
List<GameObject> lifeIconObjects
int Lives
LivesManager(GameCore core)
void PlayerDiedEventHandler(object sender, GameObject dyingObject)
void GameOver()
interacts
CollisionManager
static readonly List<Collider> colliders
CollisionManager()
static void RegisterCollider(Collider collider)
static void UnregisterCollider(Collider collider)
static void CheckCollisions()
static bool IsColliding(Collider a, Collider b)
manages
Collider

Figure 2-6 **CollisionManager Diagram :** This subsystem deals with collision detection and response between game objects, ensuring that all interactions are handled correctly. It determines when collision occurs between game objects and ensures the detection is effective and responsive, allowing for real time interactions.

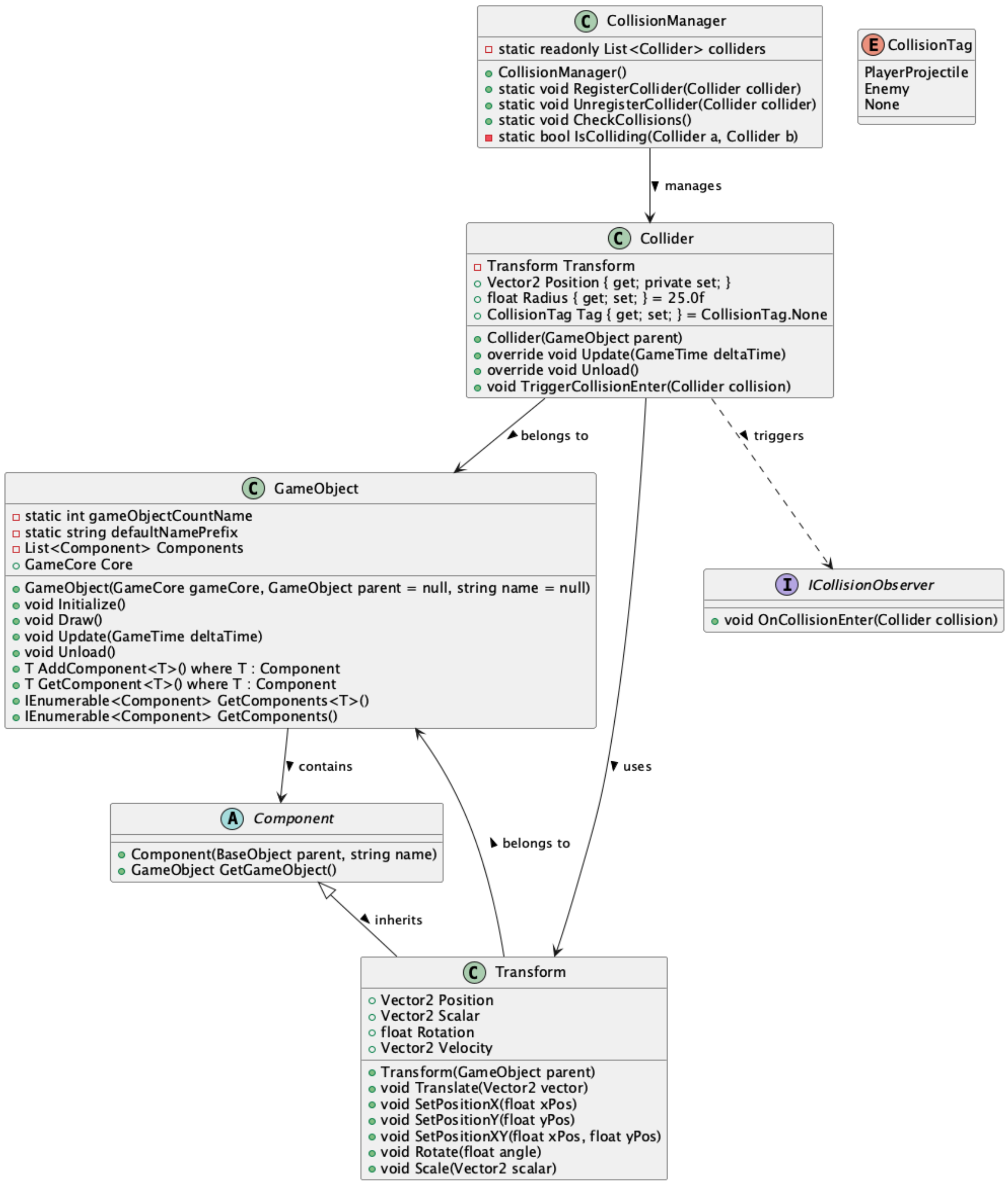

C CollisionManager
static readonly List<Collider> colliders
CollisionManager()
static void RegisterCollider(Collider collider)
static void UnregisterCollider(Collider collider)
static void CheckCollisions()
static bool IsColliding(Collider a, Collider b)
E CollisionTag
PlayerProjectile
Enemy
None
manages
C Collider
Transform Transform
Vector2 Position { get; private set; }
float Radius { get; set; } = 25.0f
CollisionTag Tag { get; set; } = CollisionTag.None
Collider(GameObject parent)
override void Update(GameTime deltaTime)
override void Unload()
void TriggerCollisionEnter(Collider collision)
belongs to
triggers
C GameObject
static int gameObjectCountName
static string defaultNamePrefix
List<Component> Components
GameCore Core
GameObject(GameCore gameCore, GameObject parent = null, string name = null)
void Initialize()
void Draw()
void Update(GameTime deltaTime)
void Unload()
T AddComponent<T>() where T : Component
T GetComponent<T>() where T : Component
IEnumerable<Component> GetComponents<T>()
IEnumerable<Component> GetComponents()
I ICollisionObserver
void OnCollisionEnter(Collider collision)
contains
uses
A Component
Component(BaseObject parent, string name)
GameObject GetGameObject()
belongs to
inherits
C Transform
Vector2 Position
Vector2 Scalar
float Rotation
Vector2 Velocity
Transform(GameObject parent)
void Translate(Vector2 vector)
void SetPositionX(float xPos)
void SetPositionY(float yPos)
void SetPositionXY(float xPos, float yPos)
void Rotate(float angle)
void Scale(Vector2 scalar)

Figure 2-7: **GameObjectManager Diagram :** This subsystem is responsible for creating, updating, and rendering all game objects, including players, enemies, and projectiles. It maintains a collection of game objects and handles their lifecycle, ensuring that they are properly managed throughout the game.

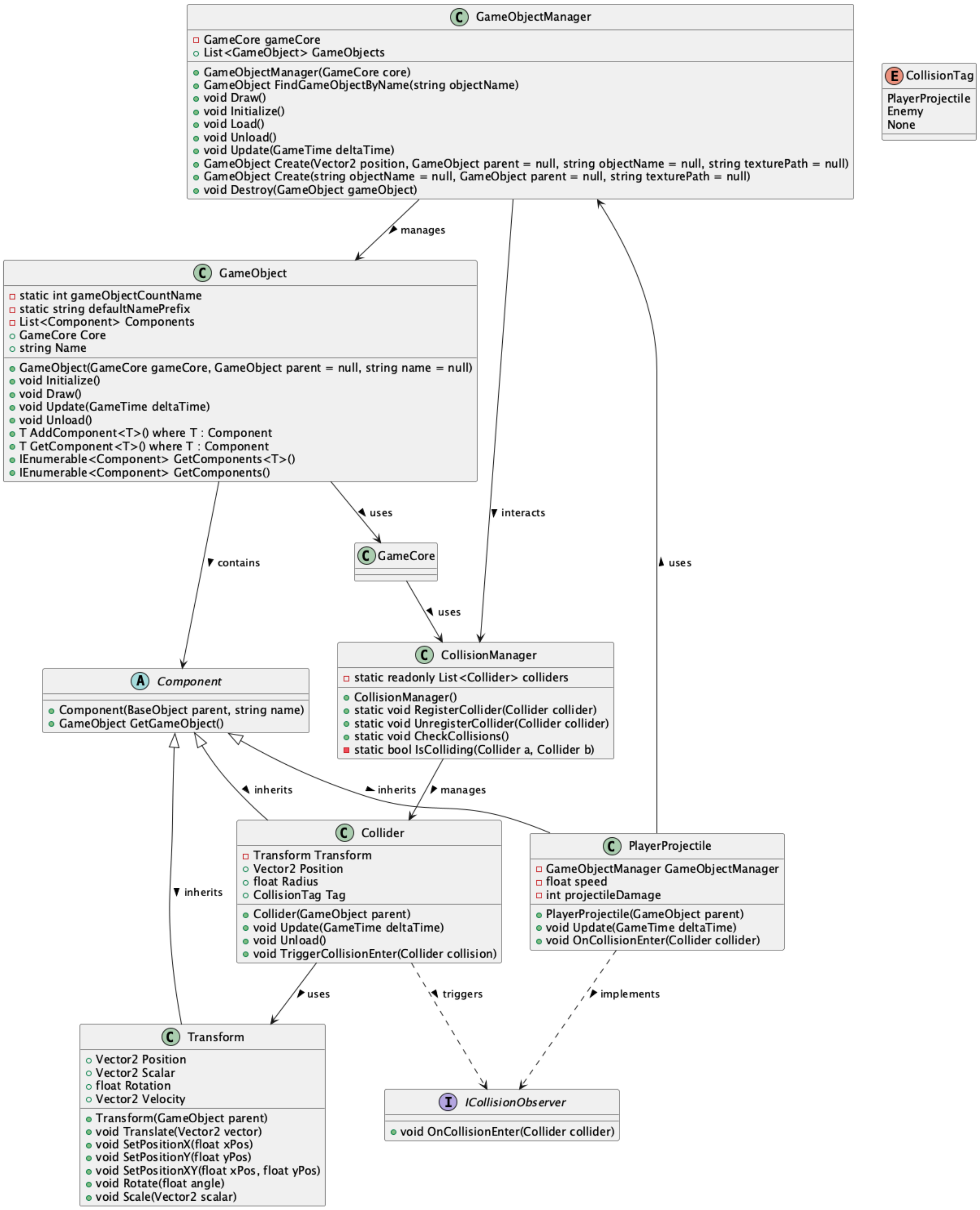

GameObjectManager
GameCore gameCore
List<GameObject> GameObjects
GameObjectManager(GameCore core)
GameObject FindGameObjectByName(string objectName)
void Draw()
void Initialize()
void Load()
void Unload()
void Update(GameTime deltaTime)
GameObject Create(Vector2 position, GameObject parent = null, string objectName = null, string texturePath = null)
GameObject Create(string objectName = null, GameObject parent = null, string texturePath = null)
void Destroy(GameObject gameObject)
CollisionTag
PlayerProjectile
Enemy
None
manages
GameObject
static int gameObjectCountName
static string defaultNamePrefix
List<Component> Components
GameCore Core
string Name
GameObject(GameCore gameCore, GameObject parent = null, string name = null)
void Initialize()
void Draw()
void Update(GameTime deltaTime)
void Unload()
T AddComponent<T>() where T : Component
T GetComponent<T>() where T : Component
IEnumerable<Component> GetComponents<T>()
IEnumerable<Component> GetComponents()
uses
interacts
uses
contains
GameCore
uses
CollisionManager
static readonly List<Collider> colliders
CollisionManager()
static void RegisterCollider(Collider collider)
static void UnregisterCollider(Collider collider)
static void CheckCollisions()
static bool IsColliding(Collider a, Collider b)
Component
Component(BaseObject parent, string name)
GameObject GetGameObject()
inherits
inherits
manages
Collider
Transform Transform
Vector2 Position
float Radius
CollisionTag Tag
Collider(GameObject parent)
void Update(GameTime deltaTime)
void Unload()
void TriggerCollisionEnter(Collider collision)
PlayerProjectile
GameObjectManager GameObjectManager
float speed
int projectileDamage
PlayerProjectile(GameObject parent)
void Update(GameTime deltaTime)
void OnCollisionEnter(Collider collider)
inherits
uses
triggers
implements
Transform
Vector2 Position
Vector2 Scalar
float Rotation
Vector2 Velocity
Transform(GameObject parent)
void Translate(Vector2 vector)
void SetPositionX(float xPos)
void SetPositionY(float yPos)
void SetPositionXY(float xPos, float yPos)
void Rotate(float angle)
void Scale(Vector2 scalar)
ICollisionObserver
void OnCollisionEnter(Collider collider)

Figure 2-8: **WaveManager Diagram :** This subsystem manages the spawning of enemy waves and bosses. It's in charge of the flow of gameplay and difficulty progression over time by implementing predefined logic. It ensures a balanced gameplay experience for the player.

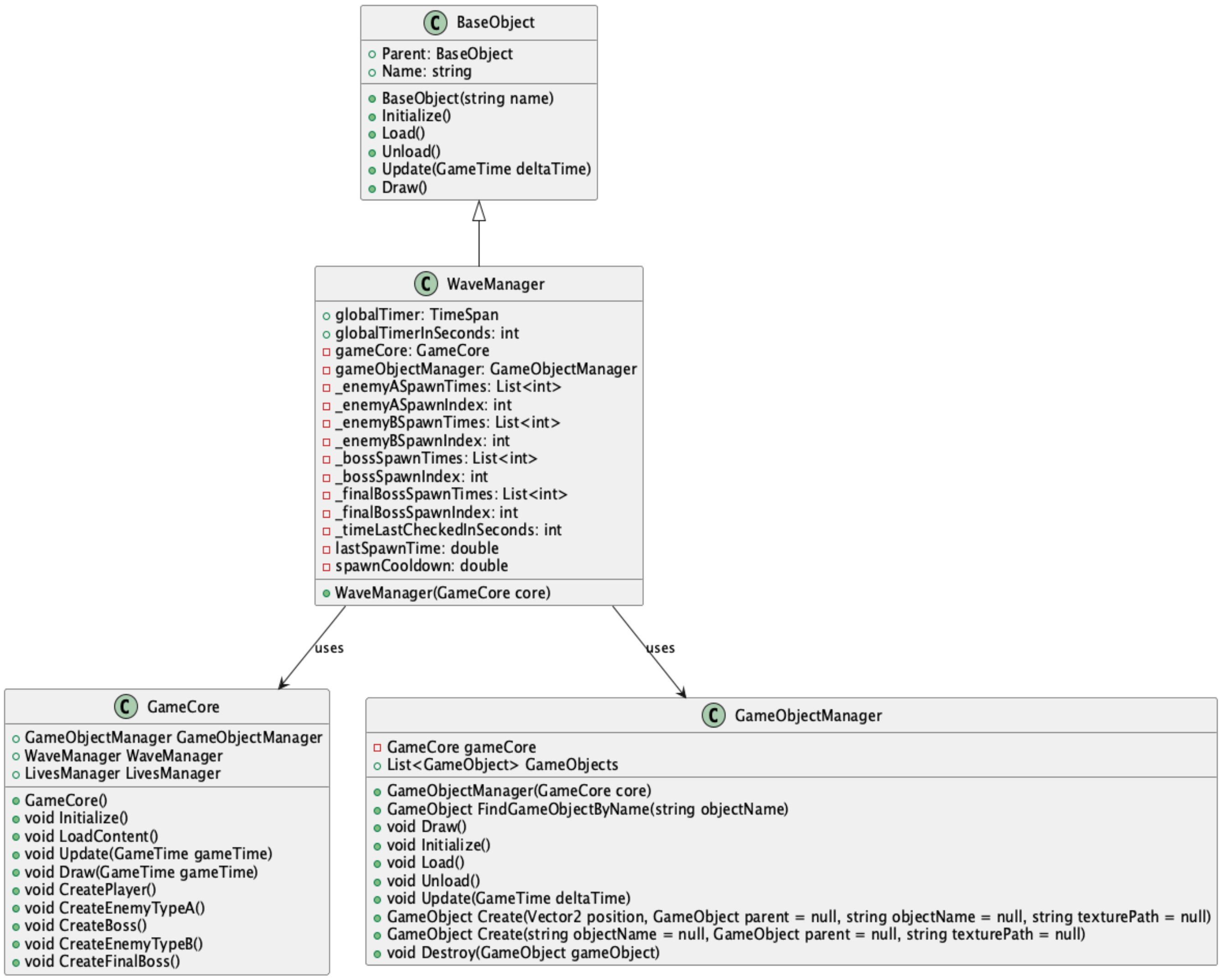
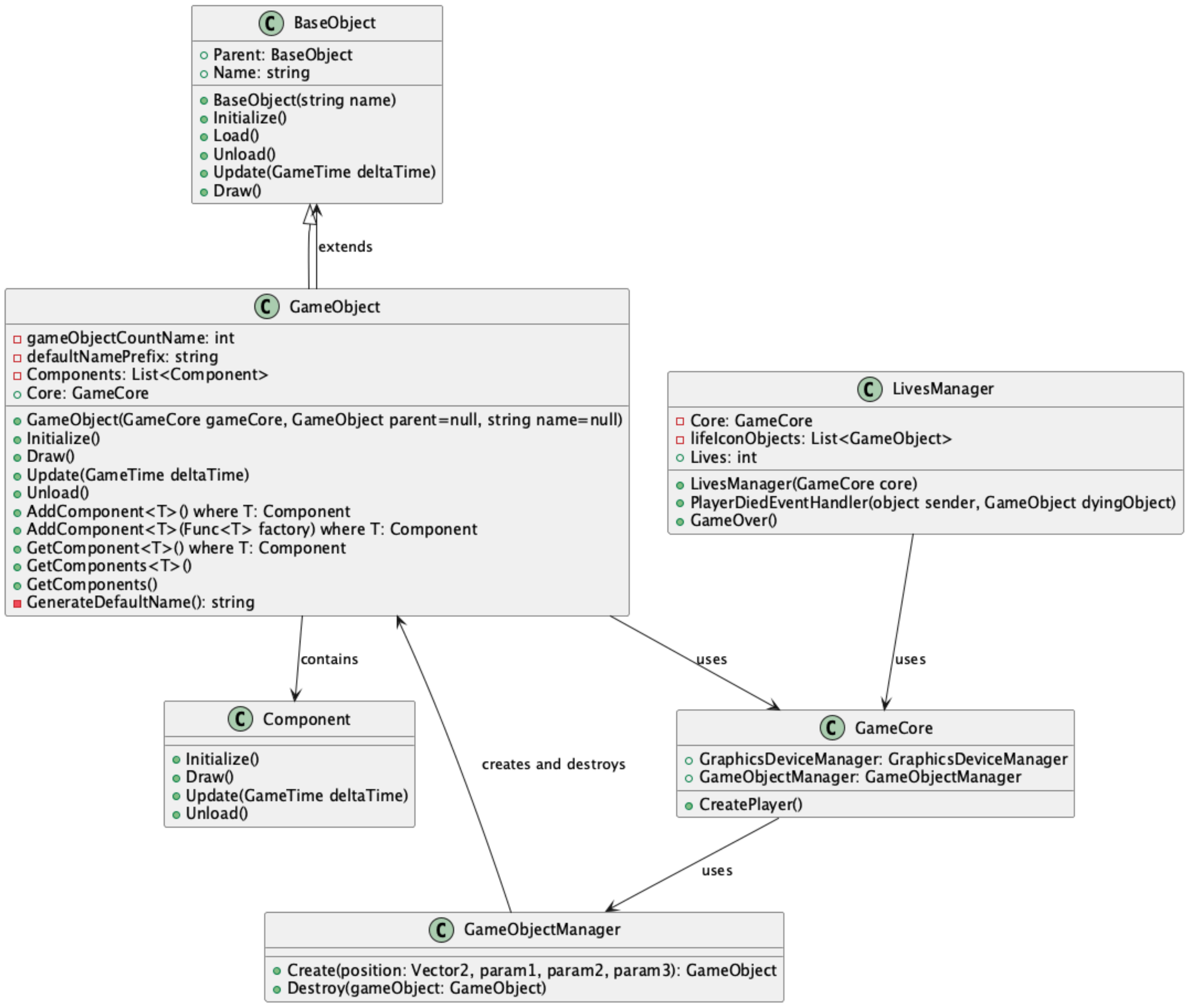

BaseObject
Parent: BaseObject
Name: string
BaseObject(string name)
Initialize()
Load()
Unload()
Update(GameTime deltaTime)
Draw()
WaveManager
globalTimer: TimeSpan
globalTimerInSeconds: int
gameCore: GameCore
gameObjectManager: GameObjectManager
_enemyASpawnTimes: List<int>
_enemyASpawnIndex: int
_enemyBSpawnTimes: List<int>
_enemyBSpawnIndex: int
_bossSpawnTimes: List<int>
_bossSpawnIndex: int
_finalBossSpawnTimes: List<int>
_finalBossSpawnIndex: int
_timeLastCheckedInSeconds: int
lastSpawnTime: double
spawnCooldown: double
WaveManager(GameCore core)
uses
uses
GameCore
GameObjectManager GameObjectManager
WaveManager WaveManager
LivesManager LivesManager
GameCore()
void Initialize()
void LoadContent()
void Update(GameTime gameTime)
void Draw(GameTime gameTime)
void CreatePlayer()
void CreateEnemyTypeA()
void CreateBoss()
void CreateEnemyTypeB()
void CreateFinalBoss()
GameObjectManager
GameCore gameCore
List<GameObject> GameObjects
GameObjectManager(GameCore core)
GameObject FindGameObjectByName(string objectName)
void Draw()
void Initialize()
void Load()
void Unload()
void Update(GameTime deltaTime)
GameObject Create(Vector2 position, GameObject parent = null, string objectName = null, string texturePath = null)
GameObject Create(string objectName = null, GameObject parent = null, string texturePath = null)
void Destroy(GameObject gameObject)

Figure 2-9: **LivesManager Diagram :** This subsystem manages player lives, keeping track of when a player is hit and determines the consequences of losing lives including death of players.

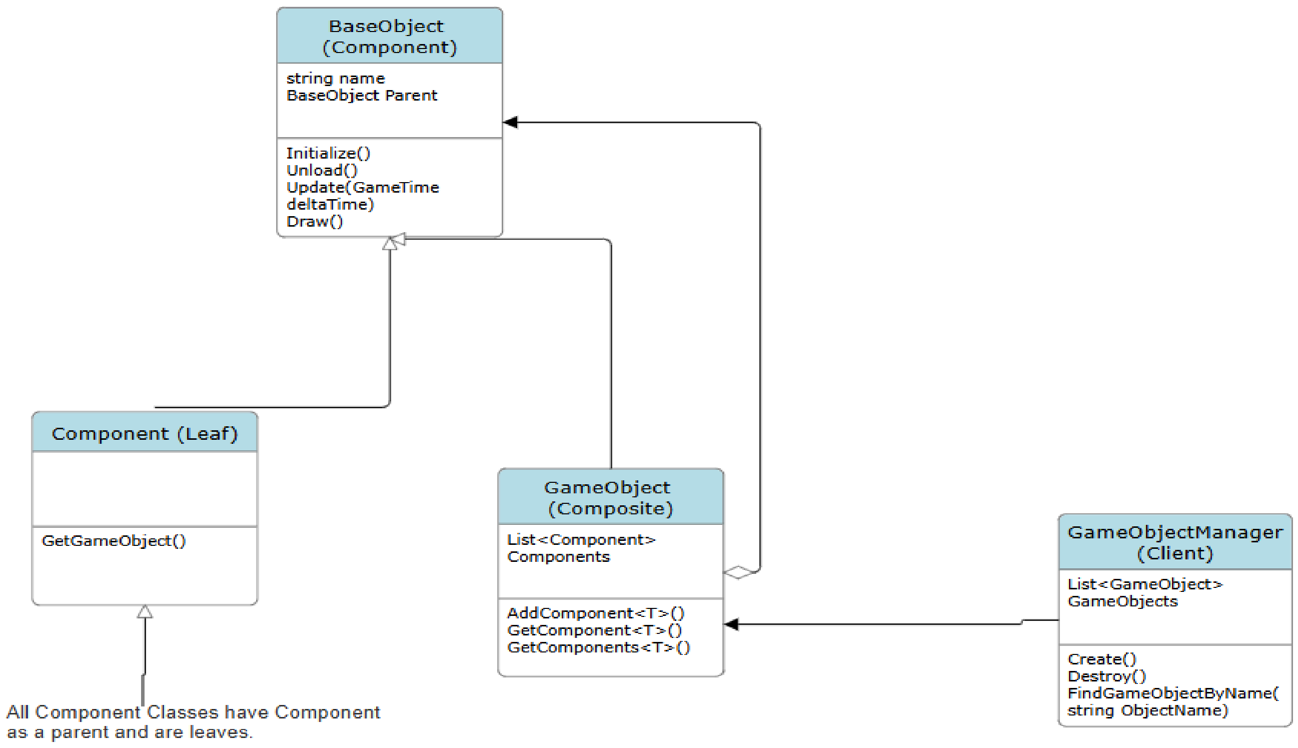

BaseObject
Parent: BaseObject
Name: string
BaseObject(string name)
Initialize()
Load()
Unload()
Update(GameTime deltaTime)
Draw()
extends
GameObject
gameObjectCountName: int
defaultNamePrefix: string
Components: List<Component>
Core: GameCore
GameObject(GameCore gameCore, GameObject parent=null, string name=null)
Initialize()
Draw()
Update(GameTime deltaTime)
Unload()
AddComponent<T>() where T: Component
AddComponent<T>(Func<T> factory) where T: Component
GetComponent<T>() where T: Component
GetComponents<T>()
GetComponents()
GenerateDefaultName(): string
LivesManager
Core: GameCore
lifeIconObjects: List<GameObject>
Lives: int
LivesManager(GameCore core)
PlayerDiedEventHandler(object sender, GameObject dyingObject)
GameOver()
contains
uses
uses
Component
Initialize()
Draw()
Update(GameTime deltaTime)
Unload()
creates and destroys
GameCore
GraphicsDeviceManager: GraphicsDeviceManager
GameObjectManager: GameObjectManager
CreatePlayer()
uses
GameObjectManager
Create(position: Vector2, param1, param2, param3): GameObject
Destroy(gameObject: GameObject)

2.2.1 Design Patterns

So far in our implementation we have utilized several design patterns to enhance the architecture of the game. This includes:

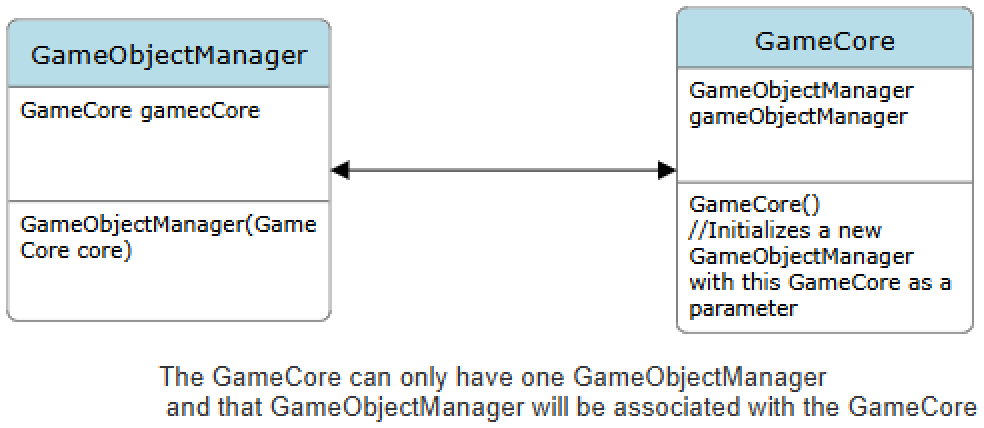
1. **Component Pattern**: Used to allow flexible attachment of behaviors to game objects. It also allows for dynamic composition of functionalities.

Figure 2-10: **Component Pattern Diagram**


BaseObject (Component)
string name
BaseObject Parent
Initialize()
Unload()
Update(GameTime deltaTime)
Draw()
Component (Leaf)
GetGameObject()
All Component Classes have Component as a parent and are leaves.
GameObject (Composite)
List<Component> Components
AddComponent<T>()
GetComponent<T>()
GetComponents<T>()
GameObjectManager (Client)
List<GameObject> GameObjects
Create()
Destroy()
FindGameObjectByName(string ObjectName)

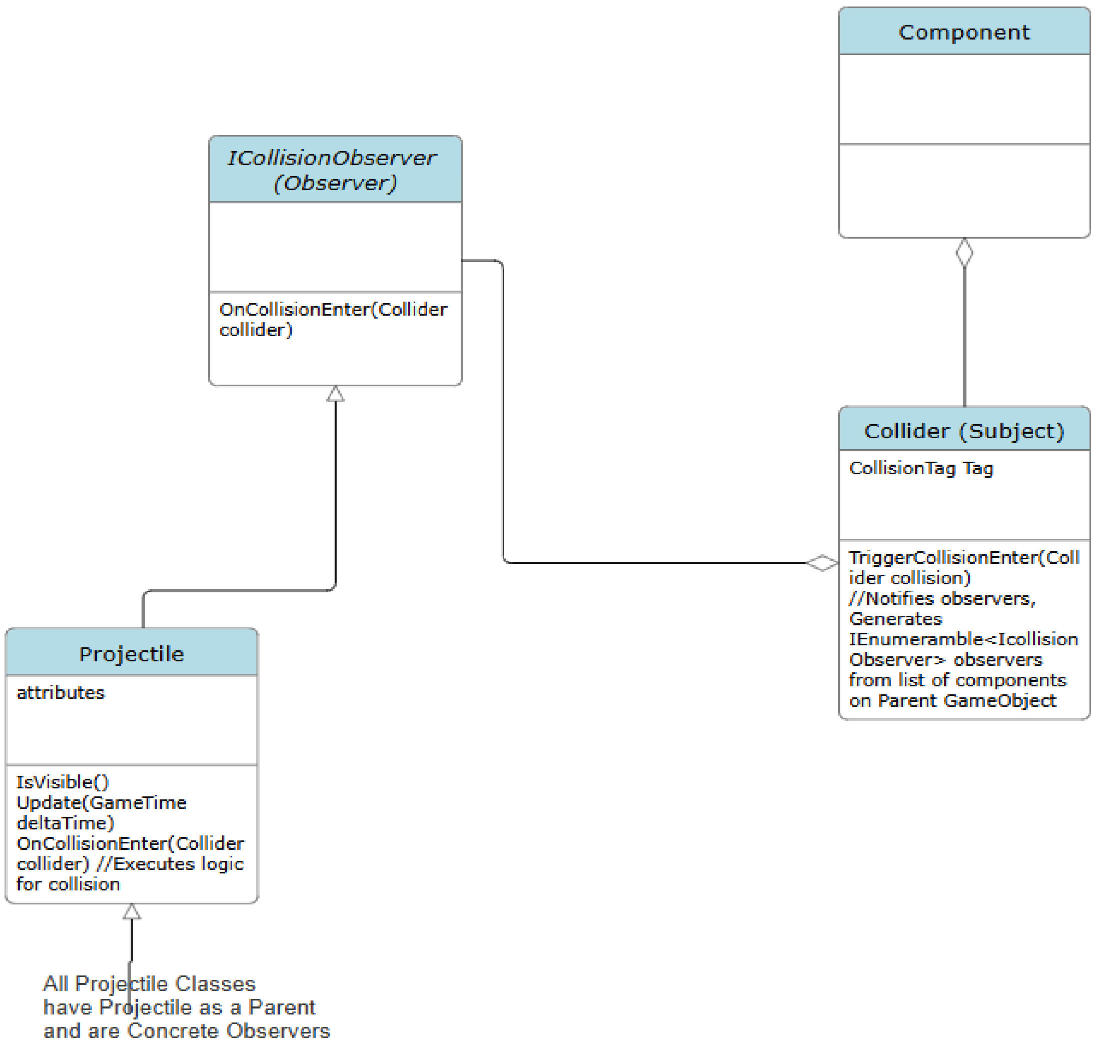
2. **Singleton Pattern**: While this does not exactly match the standard Singleton pattern in that GameObjectManager does not necessarily ensure that only one of it is instantiated ever, the instance variables in both GameCore and GameObjectManager ensure that only one instance of each ever interacts with each other.

Figure 2-11: **Singleton Pattern Diagram**


GameObjectManager
GameCore gamecCore
GameObjectManager(Game Core core)
GameCore
GameObjectManager gameObjectManager
GameCore()
//Initializes a new GameObjectManager with this GameCore as a parameter
The GameCore can only have one GameObjectManager and that GameObjectManager will be associated with the GameCore

3. **Observer Pattern**: Used for event handling, where game objects can subscribe to events like collisions and state changes, promoting loose coupling between components.

Figure 2-11: **Observer Pattern Diagram**


Component
ICollisionObserver (Observer)
OnCollisionEnter(Collider collider)
Collider (Subject)
CollisionTag Tag
TriggerCollisionEnter(Collider collision) //Notifies observers, Generates IEnumeramble<Icollision Observer> observers from list of components on Parent GameObject
Projectile
attributes
IsVisible()
Update(GameTime deltaTime)
OnCollisionEnter(Collider collider) //Executes logic for collision
All Projectile Classes have Projectile as a Parent and are Concrete Observers

3. Subsystem Services

This section describes the services provided by each subsystem, detailing their responsibilities and interactions with other subsystems

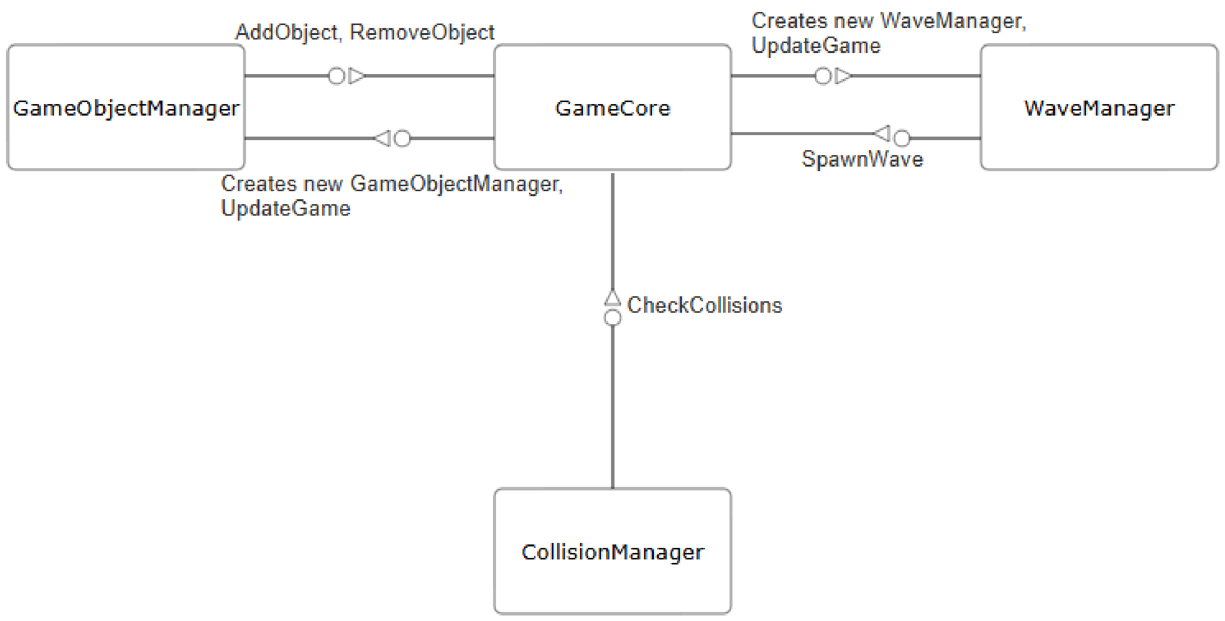
Table 3-1**: Subsystem Services**

Providing Subsystem	**Required By Subsystem**	**Service Name**	**Description**
GameCore	GameCore	InitializeGame	Initializes the game and starts the main loop.
GameCore	GameObjectManager, WaveManager, LivesManager	UpdateGame	Updates the game state each frame, processing input and managing game logic.
GameObjectManager	GameCore, WaveManager, LivesManager	Create	Adds a new game object to the scene, initializing its properties.
GameObjectManager	WaveManager, LivesManager	Destroy	Removes a game object from the scene, cleaning up the relevant resources to the object as needed.
WaveManager	GameCore, WaveManager, LivesManager	CheckEnemySpawn	Spawns a new wave of enemies based on predefined logic.
CollisionManager	GameCore, WaveManager, LivesManager	CheckCollisions	Allows the activation of all collider related activities in the scene on command.

LivesManager	GameCore, GameObjectManager, WaveManager, LivesManager	GameOver	Triggered when a player has been hit and is dead after the playerisdead event handler is triggered.

This structured approach to defining subsystem services ensures clarity in the interactions and dependencies among various components of the game architecture. Each service plays a crucial role in maintaining the overall functionality and flow of the game, facilitating communication between subsystems while keeping to the architectural design principles established in earlier sections.

Figure 3-1: **Subsystem Interactions**



GameCore provides the core operations of the game and maintains the state as the game progresses. It also calls on GameObjectManager and WaveManager to handle the moment-to-moment actions of the game within these categories and keeps them on the same time schedule. Collisions are handled by CollisionManager, which is a class with static functions, at the request of GameCore.