Project Alterac Valley : Technical Design Document (Gameplay)

Landon Grant
Jean-François Carnovale
Jaymie Xu
Brandon MacLean
Nathan Ellenberger
Ethan Butler

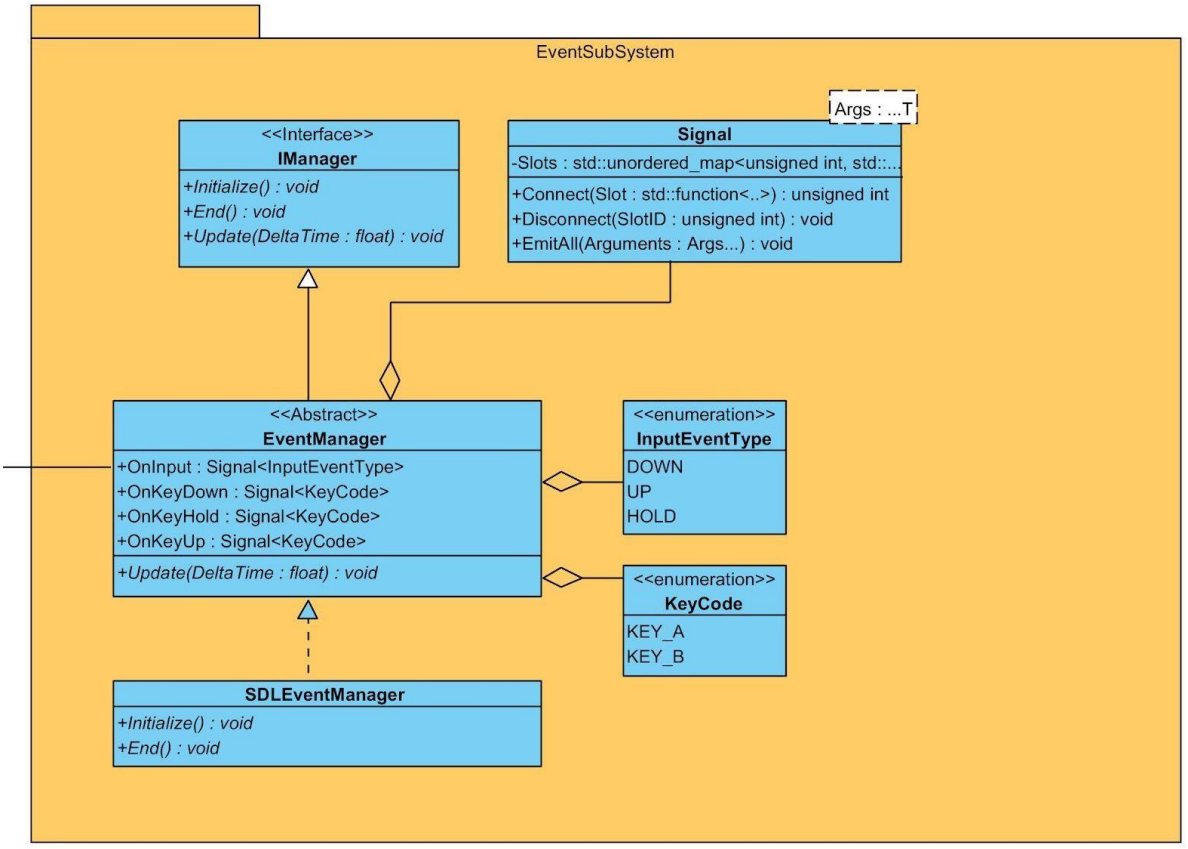
Contents

Table of Contents

General Architecture

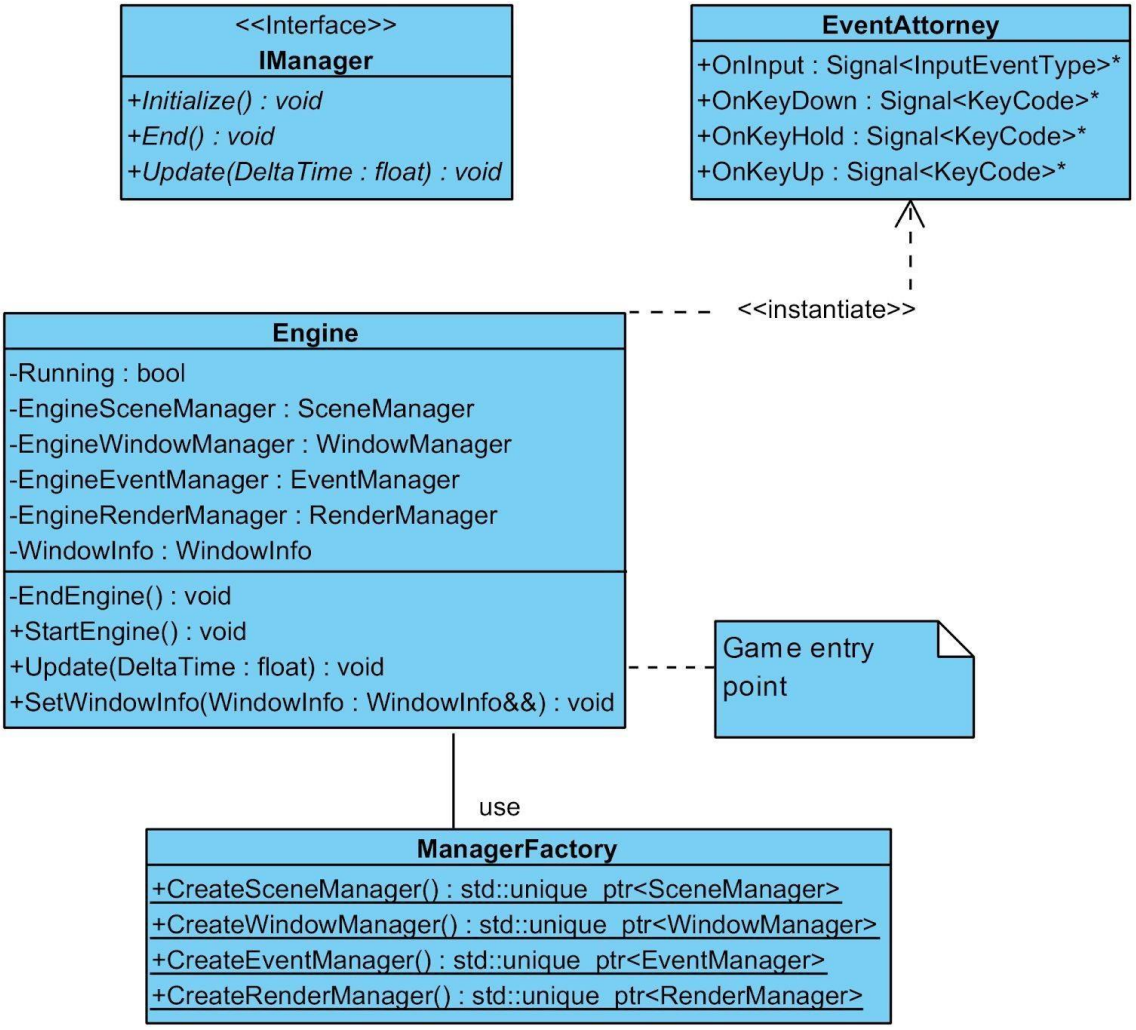
Game Entry Logic

Project Alterac Valley will be coded in C++. At this stage of development, most teams have agreed upon using OpenGL and SDL as two of our main graphic and event handling API. Dependencies such as OpenGL and or SDL may change later (For argument sake, we decided to use DirectX and GLFW instead in future), therefore we decided to heavily rely on the dependency inversion principle for most of our engine subsystems.



(Full UML exported as zoomable .pdf files)

As an example, The EventManager class is an abstract class with some common functionalities. In ordered to handle events a programmer must use the EventManager as a base class and write their own event manager; in this case we wrote SDLEventManager since SDL is the event API dependency here. During the actual creation of the engine there will be another class called ManagerFactory which will construct the appropriate concrete EvenManager base on precompile macro settings.



(Full UML exported as zoomable .pdf files)

System of Entities and Components

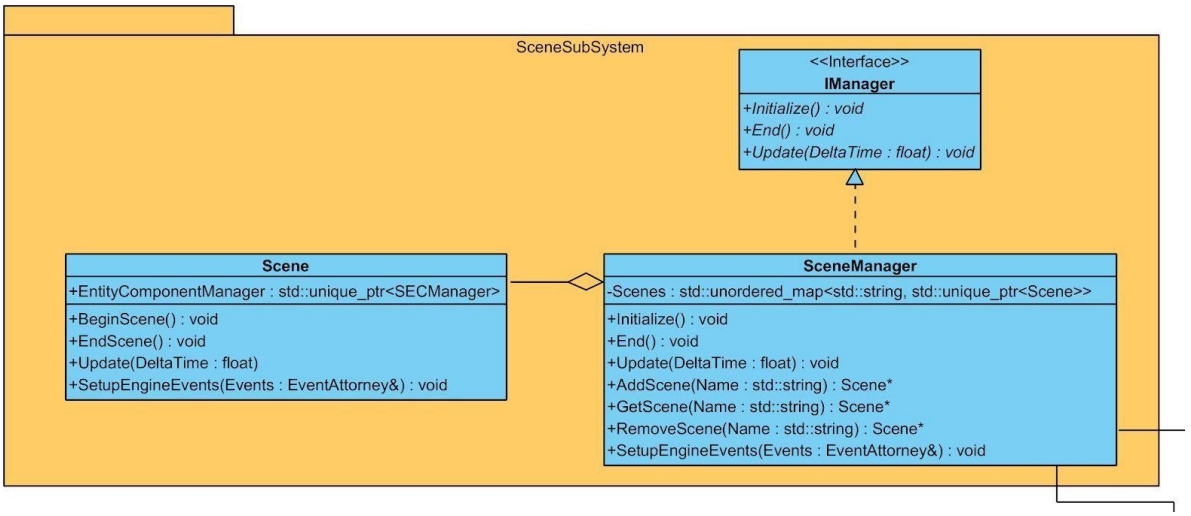
We decided to make Project Alterac Valley's inner working system a heavily modular one. This means that almost everything, aside from some of the major managers and engine code, should be either a GameObject (aka an entity) or a Component (Some game level managers are an exception to this rule). To illustrate this, a playable character is nothing more than just a GameObject with MeshRendererComponent,
AnimationPlayerComponent, UnitComponent, UnitStatComponent, and finally a CollisionComponent. On the other hand, an AI player maybe the same as a playable character, with one exception: It also has an AIControllerComponent. Without this system of entity and components such behaviour can only be achieved with complex strategy pattern or with heavy usage of inheritance and repeated code.

Each IComponentBase and IGameObjectBase object will use a GUID to identify their type, for example a RenderComponent's GUID is 1 and a CollisionComponent GUID is 2. To generate this ID, we used CRTP (Curiously Recurring Template Pattern) to allow derived class to pass their type by template into super class. Once the is in the super class, the system would then call a static GUID class to generate an ID based on an object's type (See UML for detail). We would employ several template meta programming tricks as well as local static variable to achieve this effect.

In order to make sure our components are store continuously in memory. We would be creating all component by ComponentManager which will store then in a HashMap of Vectors (Note that Vector in this context is referring to std::vector) the key being the component GUID. GameObjects will have pointers to all their components thus creating the illusion that GameObjects owns Component (GameObject can easily access their own component) while in truth ComponentManager manages it all. This also allows search by component type easily implemented (Using the HashMap get Component type GUID to fetch the correct Vector of components).

Scene Management

Project Alterac Valley's Scene class will be an inheritable abstract class. Making scenes inheritable allows direct modification inside them. Programmers may derive their own scenes for different areas of the game. For example: a menu scene, a game scene, etc. Programmers my also add their own managers to dictate how game objects will interact with each other in a scene.



(Full UML exported as zoomable .pdf files)

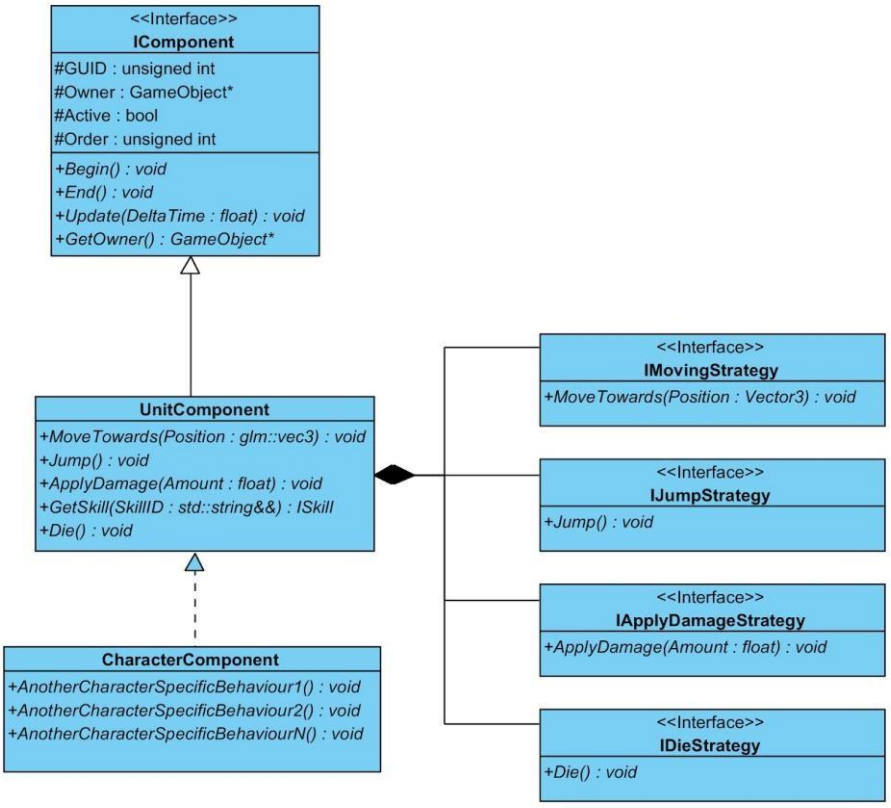
Unit Class and Player Logic

Unit Logic Component

All the characters (Players and NPCs) controlling logic in the game will use a class (or one of its specialized subclass) called UnitComponent (Inherit from IComponentBase interface) The UnitComponent base class will have common behaviour functions such as Moving, Dying, AutoAttack and Respawning. UnitComponent will also have some basic members such as a mesh, animations etc. Since every player class and NPC will have some similarities and some differences, we decided to use the strategy pattern to achieve this effect.

For example, a Knight class can be made of a UnitComponent class with MoveStrategy1, DieStrategy2, AutoAttackStrategy2, and RespawnStrategy4 while a Mage class can be made of a UnitComponent class with MoveStrategy1, DieStrategy3, AutoAttackStrategy2, RespawnStrategy1. The Knight and Mage both uses the same MoveStrategy andAutoAttackStrategy by using the strategy pattern we essentially avoid the need of repeating unnecessary code. Furthermore, because most Class/NPC behaviours are wrapped inside Strategy classes we can use a factory class to build the desired class at runtime.

This flexibility enables us the ability to easily define class archetype in a JSON file (see example below). Should there ever be a need to change these classes' behaviours in the future, all we need to do is modify the JSON file and change a single line of our game code.



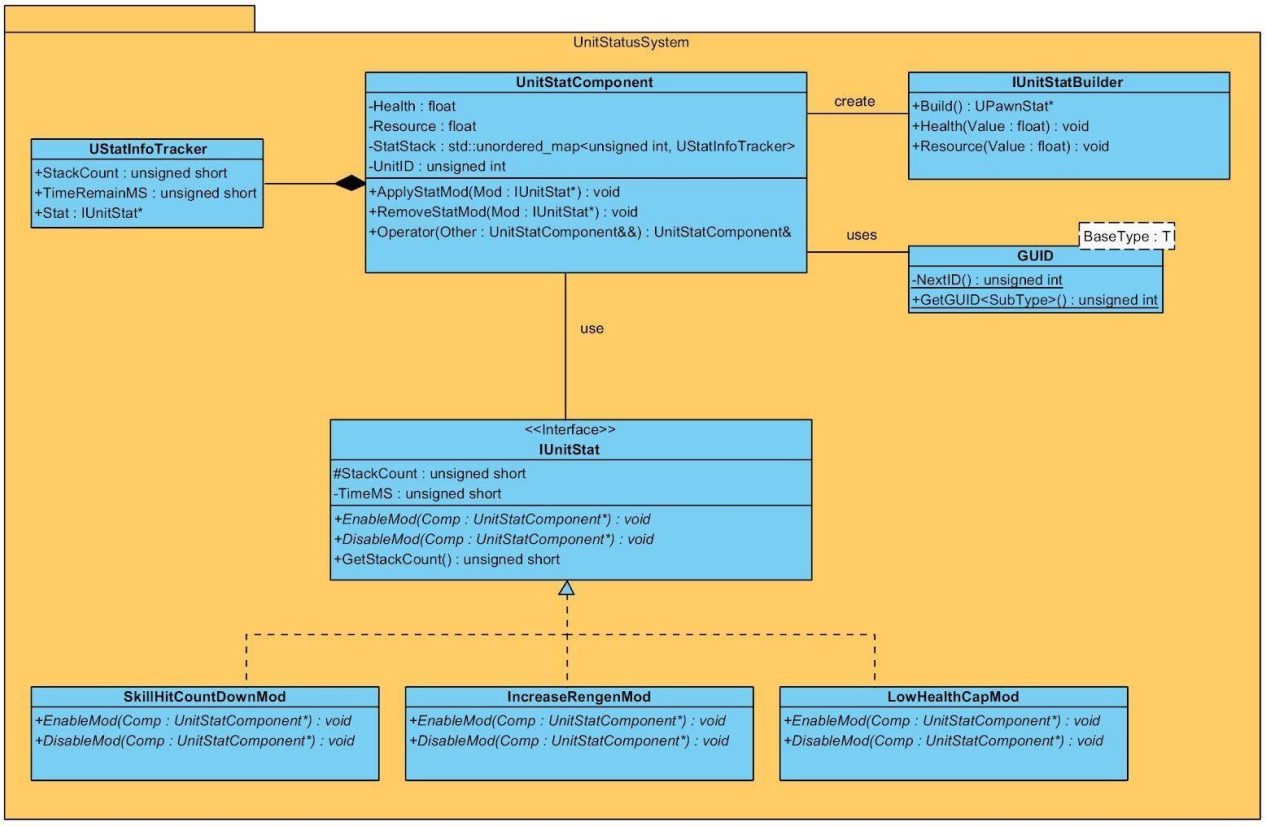
We also recognized that there will be some cases where defining specific traits of a certain archetype may be difficult when using the strategy pattern above. As such, all the base functions in the character

class like Move, AutoAttack, Die, and Respawn can be overwritten. This allows us to derive a subclass from the UnitComponent class, for example:
UnitComponent (Derive)-> SpecialUnitComponent. And since UnitComponent is a child class from the base class UnitComponent, it may also use the same strategy classes while also having the ability to override or add functionalities.

Lastly, skills are handled differently from strategies. Each UnitComponent object will contain a list (std::vector object) of Skill based object. A Skill object contains some basic information about a skill/ability such as if the damage range is area of effect or single target. Skill class will also have a mandatory pure virtual function to define how a skill should be invoked (does it shoots a projectile or does it spawn a bunch of fireballs that rain down from sky etc).

Unit Status System

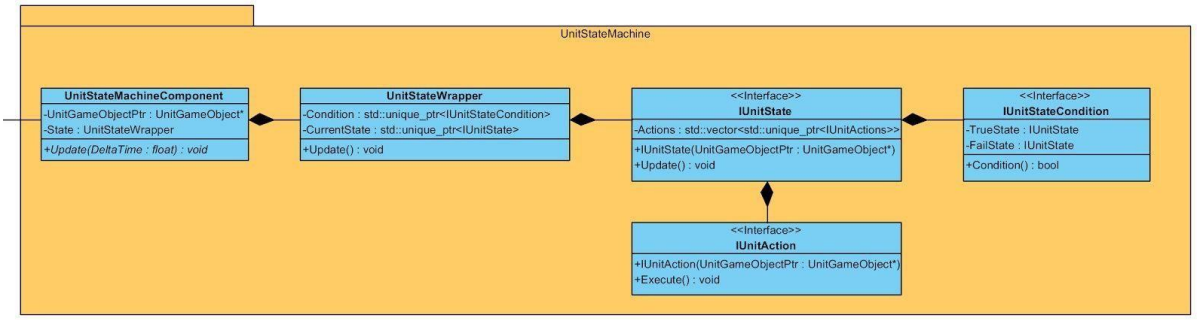
Every single Unit be it player or AI can have status on them at any given point. A player may have a fast regen stat and a quick cast stat, it may be tempting to create 2 sets of variables for each status: current_health, max_health, current_cast_mod, base_cast_mod, etc. This approach would work for small projects; however, we are dealing with an MMO and there are all sorts of buffs, even skill from another player is a “debuff” which by the end of that buff would case its attached player to lose some health. Therefore, it is crucial that we make the buff system as modular as possible. Instead of storing two variables for each stat a player may have we created a component called UnitStatComponent and a base interface class called IUnitStat, every buff/debuff including health regen will be derived from IUnitStat and this interface has 2 primary functions EnableMod and DisableMod both takes in a parameter of UnitStatComponent and have their own rules of modifying it. Whenever there is a new buff UnitStatComponent calls ApplyStatMod and whenever that buff is over UnitStatComponent can call RemoveStatMod. That way UnitStatComponent is decoupled from all the possible buff that could ever be done to a character. Furthermore, this also allows buff stacking as you can call ApplyStatMod on the same buff multiple times.



(Full UML exported as zoomable .pdf files)

Unit Behaviour / State Machine

On the other hand, a unit must behave a certain way. By that we mean a player should be able walk/run when it is currently not stunned, similarly an AI should not mindlessly wonder randomly on map then eventually gets kill by player. The easiest way to solve this problem is to use a state machine. Each character player/ai would have a specific dynamically set states. For example, player start off in moveable state which can transition into a stunned state or a casting state; all state may transition to a death state. This way, with server validation system, player cannot move while getting stunned unless the server tells the player to transition back from stunned state to moving state. In contrast the AI may use the state machine in a different manner, instead of telling the AI what they can do, the AI state machine would dictate what they could do. For instance, an AI begins with patrol a certain area and will only transition to chase state (sees player) if it encounters a player.

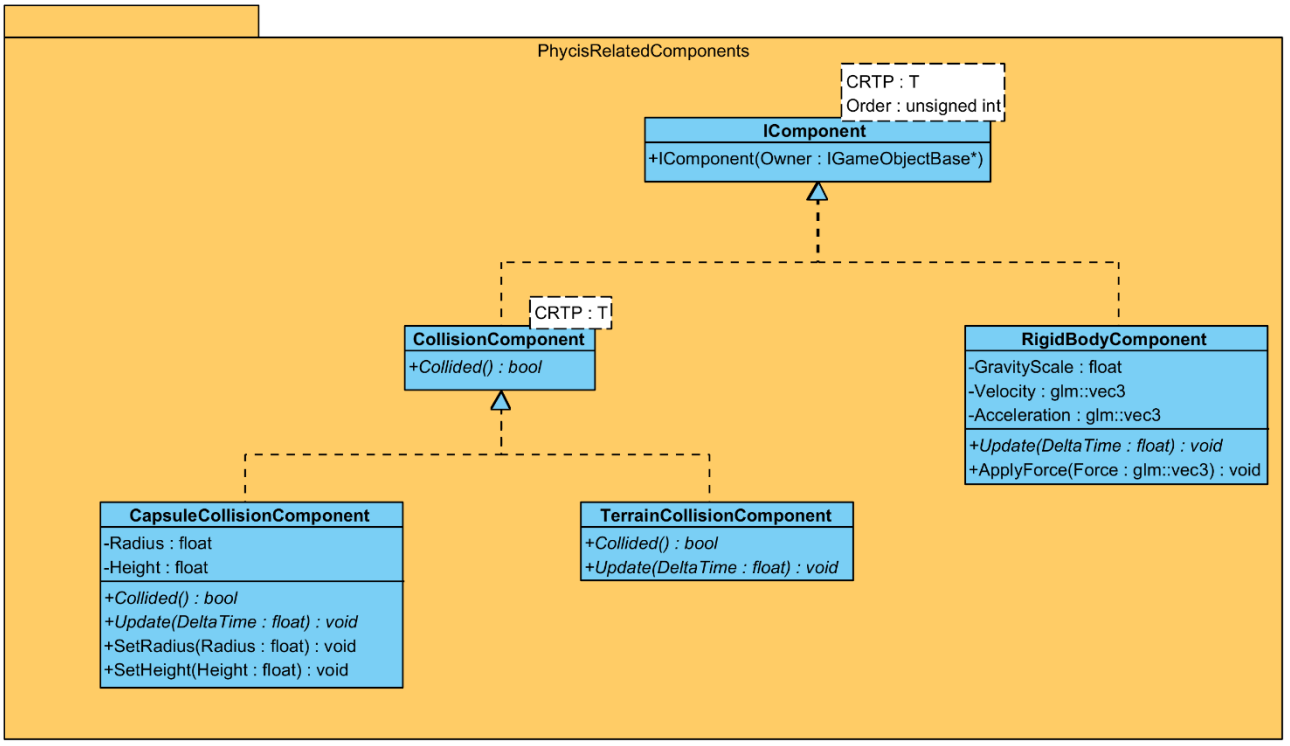


(Full UML exported as zoomable .pdf files)

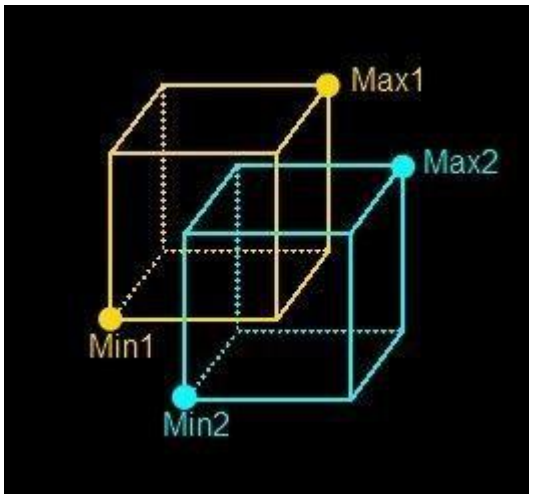
Collision and Physics Components

Component classes are all derived from the super interface IComponentBase. The number of components is expected to be increase as the development proceeds. At this moment the primary Components used by Gameplay team, aside from all the unit components, are the collision and physics components.

First, all collision components will inherit from an abstract call called CollisionComponent which would provide an overridable function called Collided which will check if this component has detected a collision with another collider component of any type and return a boolean value. So far, we have CapsuleCollisionComponent which takes in a height and a radius, this will most likely use its bottom sphere to check collision with another object. The other collider we have was the terrain collider which would detect if any object has meet the terrain.



There are many methods used in games to detect collision. Examples of such are GJK SAT AABB, since MMO is not a physics heavy game we decided to go for the least costing algorithm AABB collision detection for our terrain. AABB stands for Axis Aligned Bounding Box, it utilizes 2 bounding boxes generated from 2 object to detect if they ever need to be collided. Character are most likely to be collided with terrain, however they utilize a capsule collider (This is used to get a somewhat smooth motion while going up a ramp) AABB cannot directly act upon a capsule and a bounding box/volume (We need to have 3 axes overlap). Therefore, we need to first take the bottom most sphere of the capsule approximates a bounding volume if AABB passes we uses the capsule collider to further adjust the character's position, otherwise the character and the terrain are not collided.



Lastly the physics component in our case would have one job: pulling the player down due to gravity. This is determined by using basic physics equations (for acceleration and velocity) to determine the y position overtime and should properly reflect projectile motion. The amount of GavityScale will be determined later once testing begins. These calculations will be called every fixed update. Physics component's relation to collision detection comes into play when touching the ground. So that game objects affected by gravity do not fall through the ground a state change could enable and disable the gravity physics. This would be done using the TerrainCollisionComponent Collided() method.

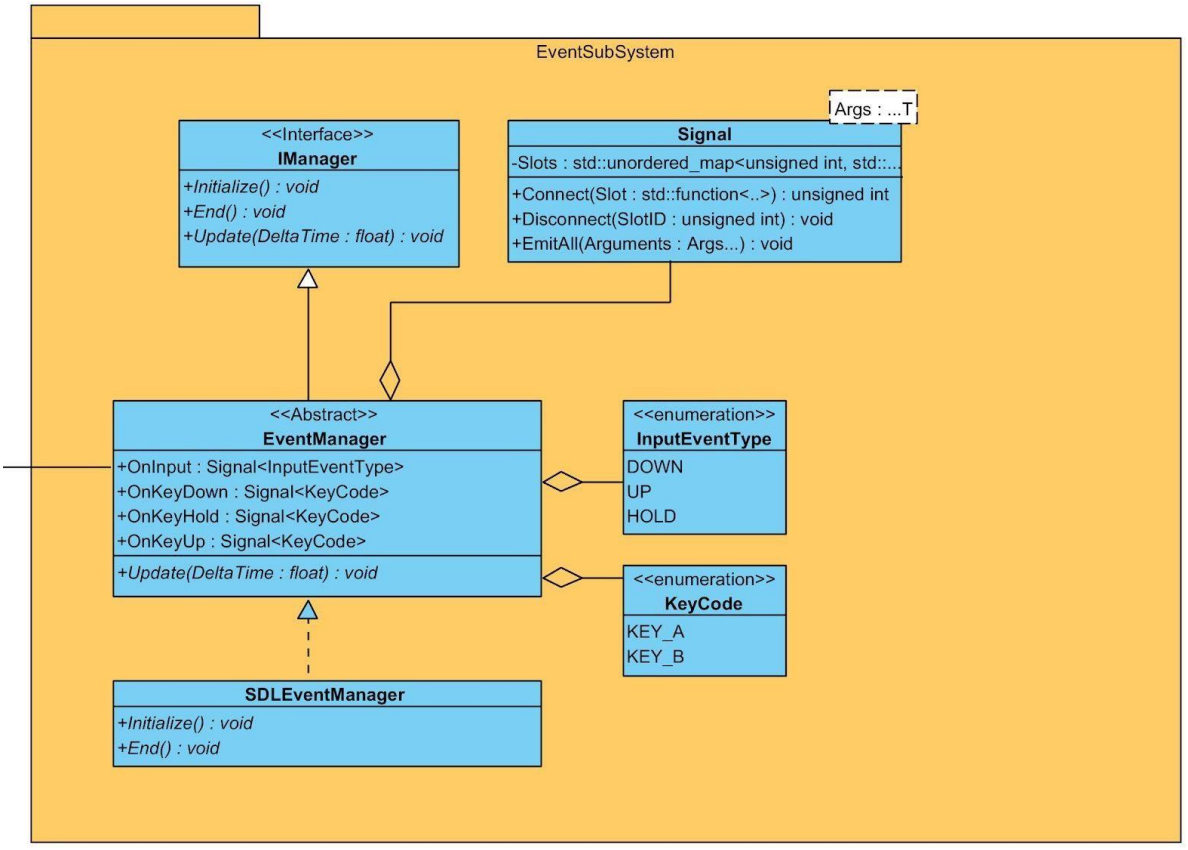
Event Handle and Data Manager

Input/Event Handle

Events will be handled by EventManager and as mentioned previously, it is an engine level manager. At this moment we have decided to use SDL as our event handling API, to make sure that SDL do not couple with our own code we have safely wrapped it inside an EvenManager subclass called SDLEventManager. This way all the engine needs to know is a pointer to EvenManager. The actual event is forwarded by a system called Signals and Slots (This was used in QT: a GUI lib for C++). Signals and Slots are very much like C#'s delegate and an observer pattern that which does not required the user to set observer class as a super class. A signal is essentially a function pointer (Reason behind us not using std::function is that member functions have a different signature in C++ compares to a lambda or global function), slots are all the function registered to it for now we have 4 signals OnInput, OnKeyDown, OnKeyHold, OnKeyUp, of course more can be added later such as mouse input. In order to avoid singleton and allow event manager to leak into every part of our engine, we decided to use dependency injection. First we made an EventAttorny class that hides some of the EventManager functions such as Update(User of the engine should not have the ability to force called Update function that was handled by the Engine class, yes people have tried to do this before to my engine) EventAttorny also has pointer to all the Signal object the EventManager class has so essentially the EventAttorny class is like a proxy class - a simplified version of a complex object.
To handle event in a Game level one must called the function SetupEngineEvents(EventAttorney& Events) on GameObject or Component, the

EventAttorney here is the injected dependency and you would now need to bind a function to one of the signals. For example
Events.OnKeyDown.Connect(&Object::HandleKeyDown(KeyCode));

Object::HandleKeyDown(KeyCode e) { if (e == KEY_A) { … } }

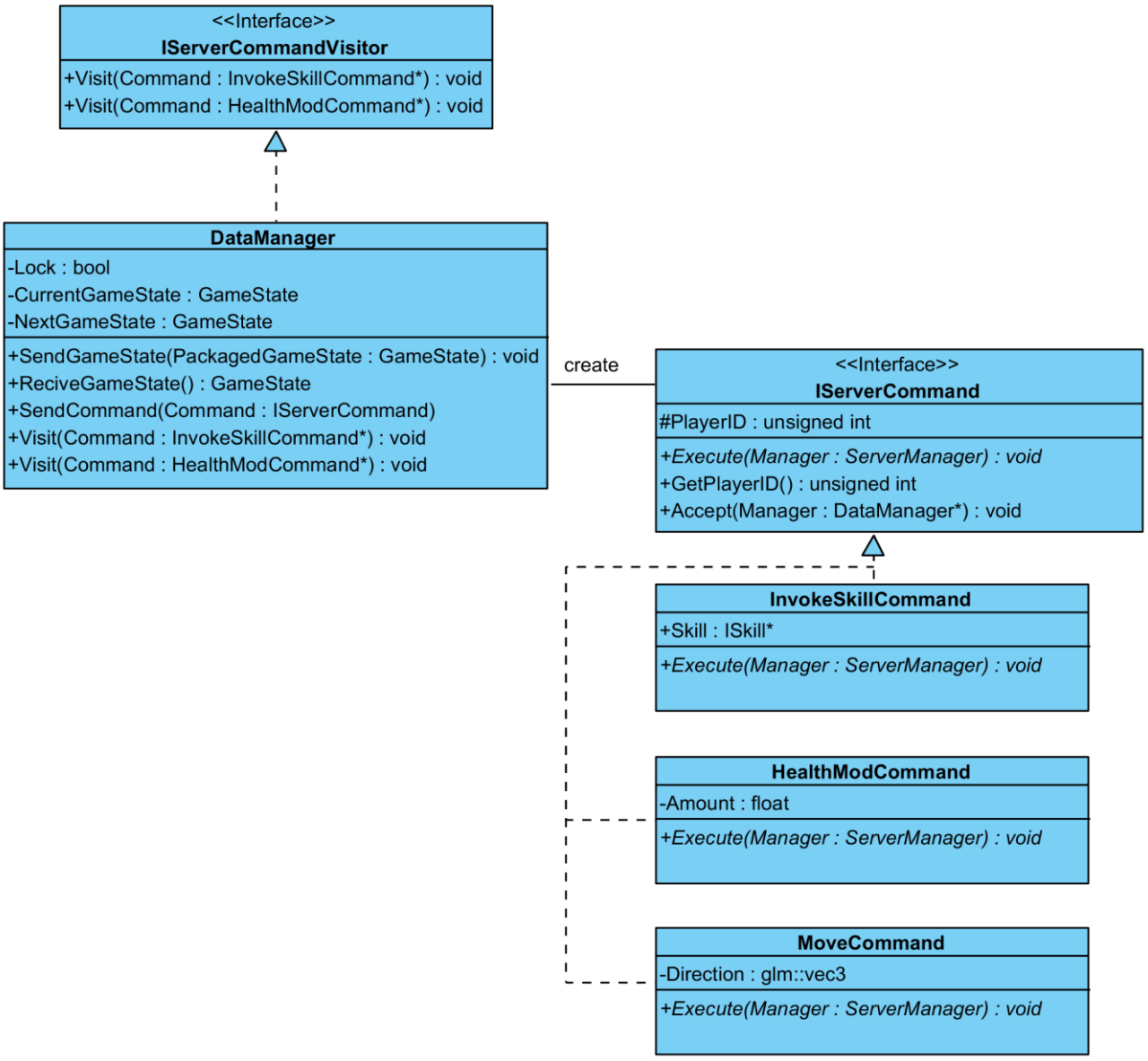


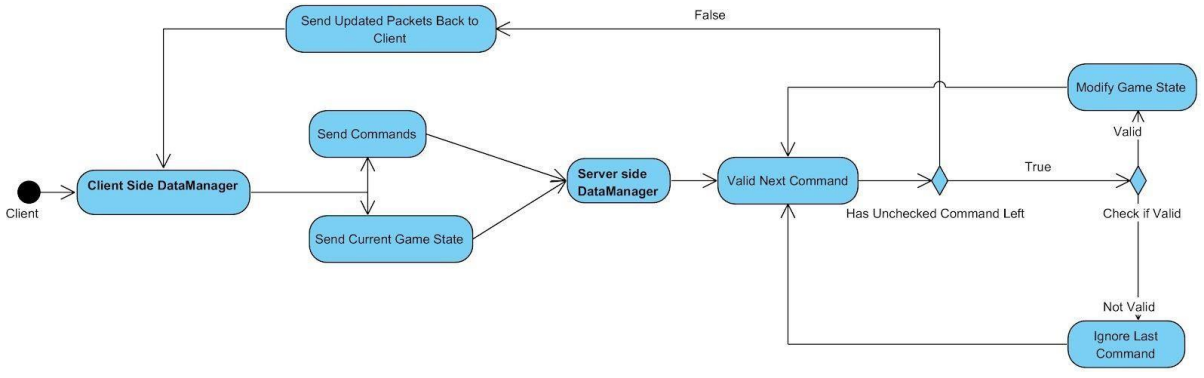
(Full UML exported as zoomable .pdf files)

Data Manager and Server Communication

All the communication between the client and the server will be handled by a class called DataManager. DataManager would send out one instance of the current game state of the current player (Health, Position, State Machine State) and a series of IServerCommand objects. A player may not directly ask to change their current state, if a player wishes to regain health, the client side must send out an IServerCommand based object like HealthModCommand and request the server to change this state for them. On the other hand, the server will implement a visitor pattern to determine which command the player just send and check whether or not this command is currently valid (See below for rule validation). If not, valid server does nothing, if the command is valid the server would call Command.Execute(this); to modify the state. After the server finishing executing all commands for a player it returns the newly modified GameState back to player.

One important thing to note here is that this communication does not happen every frame. Therefore, on the player's end, if a server didn't respond they may still able to move around before the server kicks back in and rubber bands their position and stats back.

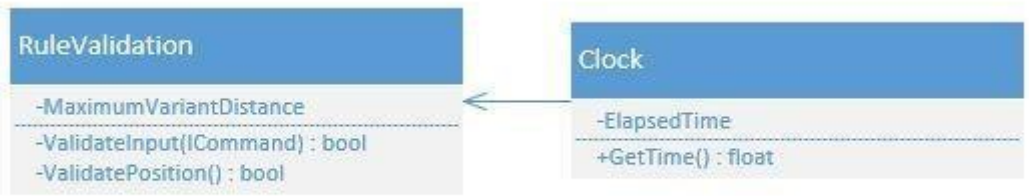




(Full UML exported as zoomable .pdf files)

Rule Validation

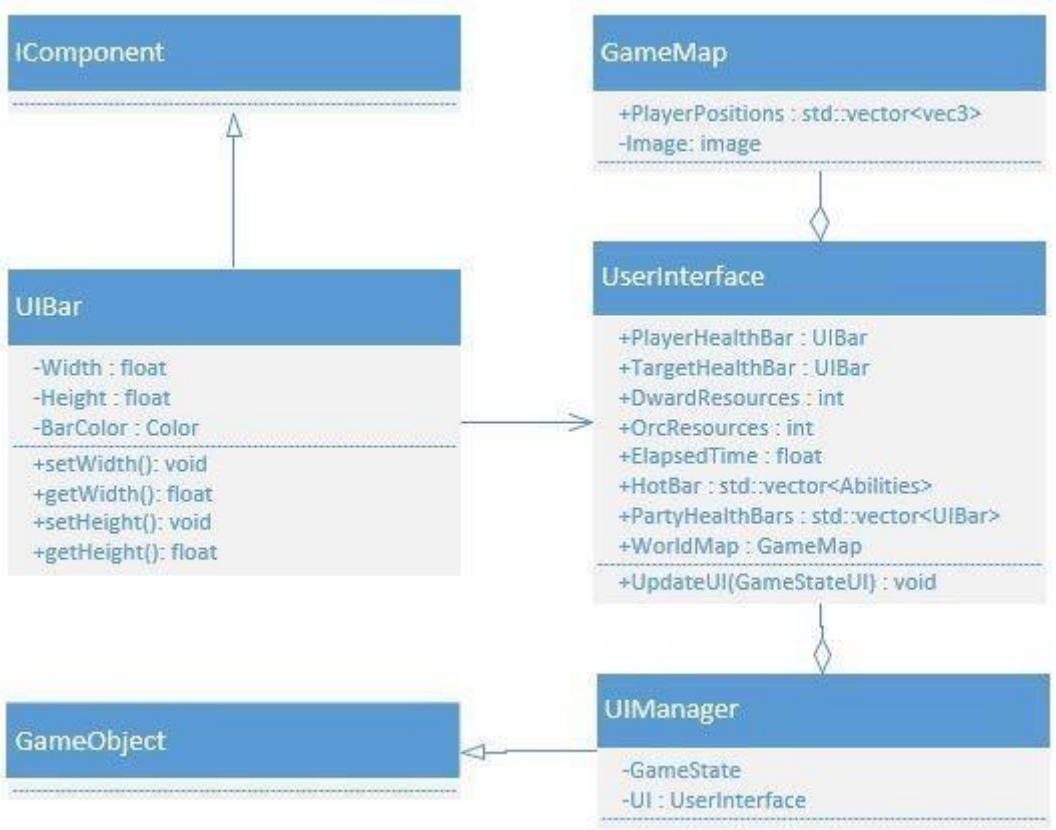
This static class is used to validate player position and inputs made by the player. For example, if the player is stunned, rather than handling inputs normally, it should prevent the associated action from taking place unless appropriate and instead display a message and play a sound effect notifying the player that they cannot make that action due to being stunned.



The position validation will prevent the client from getting out of sync with the server as well as prevent cheating. The class has a MaximumVariantDistance threshold which, if exceeded, will cause the player character on the client to rubberband back to its position on the server. It works by taking the time elapsed since the last update and checking how much the unit has moved since then. Depending on that units' abilities, it may be able to travel a large amount of distance in a short time frame. This is where the variant distance comes into play, allowing a bit of leeway before deciding to determine the movement as invalid.

UIManager

The UIManager gets the information of the GameState and breaks up the information to be appropriate to Update the UI, stores it into a GameStateUI and sends it to the UserInterface by calling the public UpdateUI method on the UserInterface. The UserInterface handles this information and updates each of its members as needed. For example, the width and value of the HealthBar will be adjusted to match that of the GameState.



Scene Information Splitting

As the game will have various Game objects of different types that will all need to be persistent and accounted for at all time, that inherently means that there will be a lot of information in the current game scene at every iteration. With such a large amount of information, not all of it would necessarily be relevant to every type of entity in the scene, having each entity in the scene receive information from every other entity and having to do whatever type of comparison operations with those entities is unrealistic, and can cause severe performance issue. However, there are ways to split up that information in a way where only the most relevant information of the various game objects relative to each object's current location will be checked. The best way to achieve this goal is to store references to the game objects based on their position in the world space inside of a binary space partitioning tree (BSP tree). There are 3 main BSP tree variations that we looked at as a solution to this particular problem, we have opted to use two different ones.

There are k-d trees, which are k dimensional trees. These trees function in a way where it is divided by dimensional axis and cycles through the dimensions per partition. The location of the partition is also not uniform as in order to make a well-balanced k-d tree you would divide the partition at the median of all the data points that are used for that section of the tree. This makes k-d trees well balance and quick as the information stored in the tree will be of similar distance to the root node. However, as you need the median of the information stored in the tree, you would need to give it the information upfront and therefore variable information and addition of new nodes will result in an imbalanced tree

and reduce its performance. The advantage to a k-d tree comes out mostly in the case of things like collision detection with persistent objects as their locations are pre-defined. Another advantage is that the way information is partitioned in a k-d tree is by a greater than, less than relationship as any node that is less than the median will be on one side of the partition where things that are greater will be on the other, as such in cases like collision or render culling these trees are advantageous, as they can easily discard anything that is at a greater distance and as such of no concern.

The other type is the quadtree, quad trees are typically used in two-dimensional space and are the two-dimensional equivalent of the octree. A quadtree has either a node that stores a set amount of information and no children or contains a reference to exactly four sub children. Each quadrant contains either its own set amount of information or its own reference to four of its own children. The advantages are similar to the octree as well however applied in two-dimensional space, this also means that they aren't as large as the octrees if you do not require three-dimensions of information. They can be used to easily determine whether or not something is outside of a relevant distance to a point or area that you are searching for information in and can also be used for collision and render culling.

For our solution we have decided to utilize both of these trees for specialized information. The as k-d trees are better optimized and have better uses as trees for persistence data that is loaded upfront, we will be using them in order to store our obstacle information. The k-d tree will have all trees, walls, rocks, and other such persistent world objects in order to have an optimized comparison algorithm to handle colliding with those sources. In our solution we will be utilizing the quad trees to handle any information sharing and comparisons between the units in the scene. Traditionally quad trees would be used for two dimensional situations, however since this is a game where there isn't much three-dimensional movement, the information in the third dimension is irrelevant, as such we can use the simpler and lighter quadtrees versus the octrees. It will b

The information for the quad tree will be stored on the server as it will contain all of the units that are in the game scene. At each update the tree will be recreated as with any movement the tree will need to be rebuilt in order to update all the positions. As for the passing of the information, players will query the server for any relevant information to them and their game state, the server will receive the players location, use it to generate an area that will be used to query the quad tree for a list of every unit that falls within that range, and then send all that information to the client. The client will then compare the list of units and update their game state accordingly so as to only include the units that are within that range and their updates at that pass, in order to save performance and cut back on the amount of information that is being passed between each object at each game state update. The information of the k-d tree will be loaded locally on the game client as it is persistent to the map and will never be updating, that way the comparison for collision into the obstacles will be handled client side, saving query time to the server reducing comparison operations as the search radius can be much smaller and improving the accuracy of collision in the game for the player as it does not need any validation from the server for the collision operation. Thus we will have one large tree that will contain all the information on the units in the game, that only the server will have access too, and will query search radiuses around each unit in order to return to each unit only the relevant information to them and their game state, as well as a second tree for all the locational information that will be stored client side and will have its comparisons be handled there.

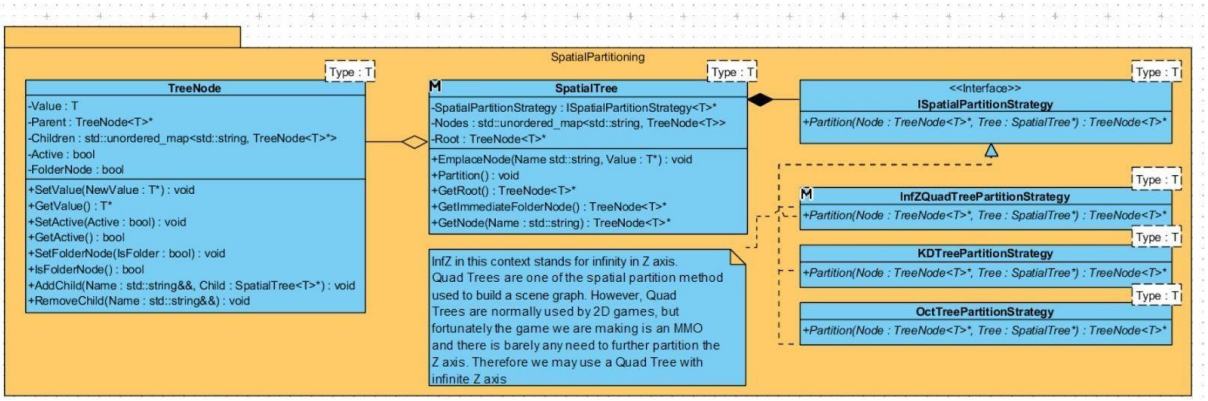
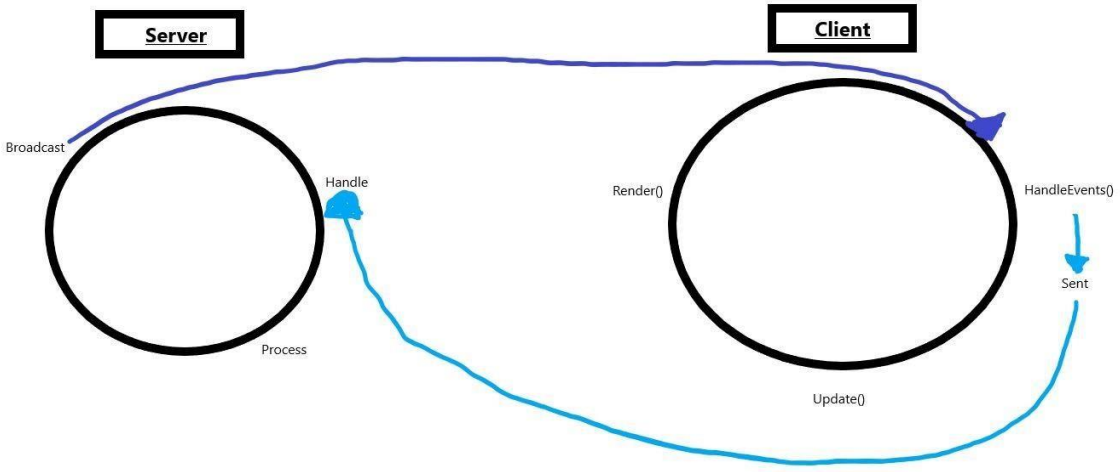
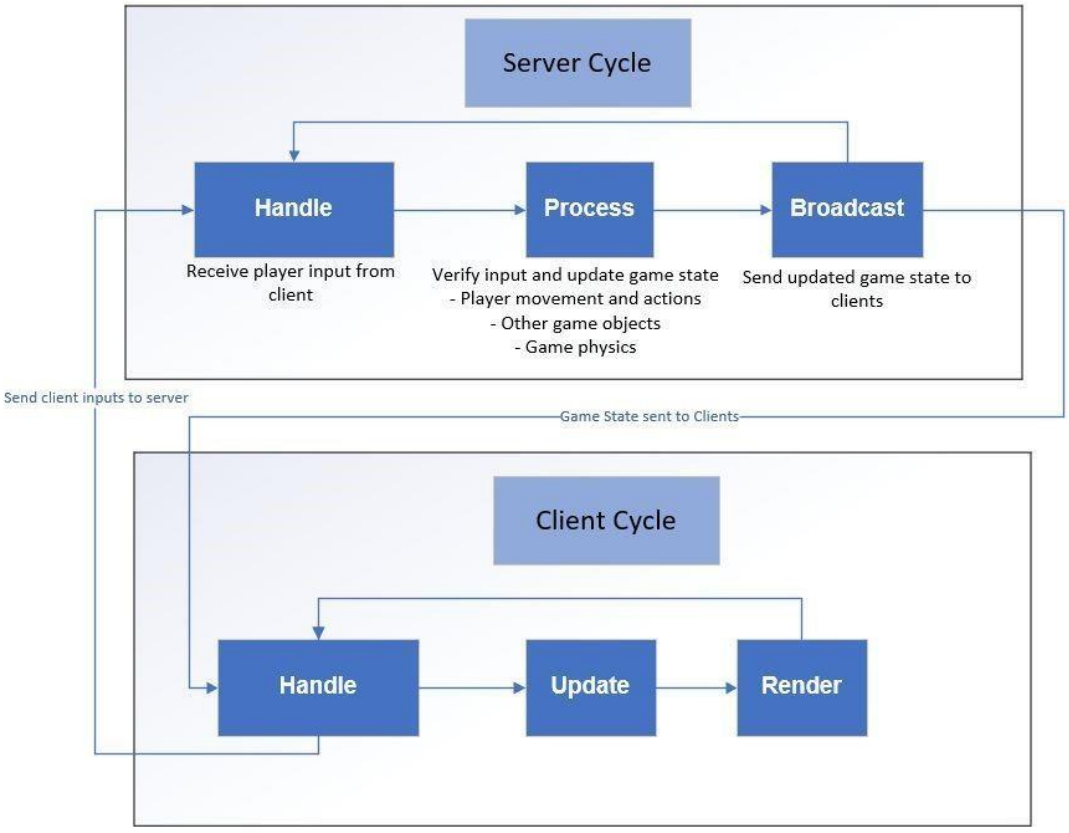


Diagram & Game Loop explained





Server

The server is going to be set up and everyone will be able to connect to it.
Broadcast/Handle: The server will first receive input/actions the player is doing which will be sent to the server via the main game loop. The server will take this input and then process it and check whether the move that was performed is valid and then let the player perform the action. The server will combine all certain packets of input received and validate it so that the next update call everyone can be in sync and the server has no problems.

Main Game Loop

Update(): The update function will constantly be changing the players position, status, etc. It will constantly be updating the game state to keep the player aware of everybody's actions being performed. If the player is out of sync with the server at some point in the update call, it will sync the player with the rest of the game.

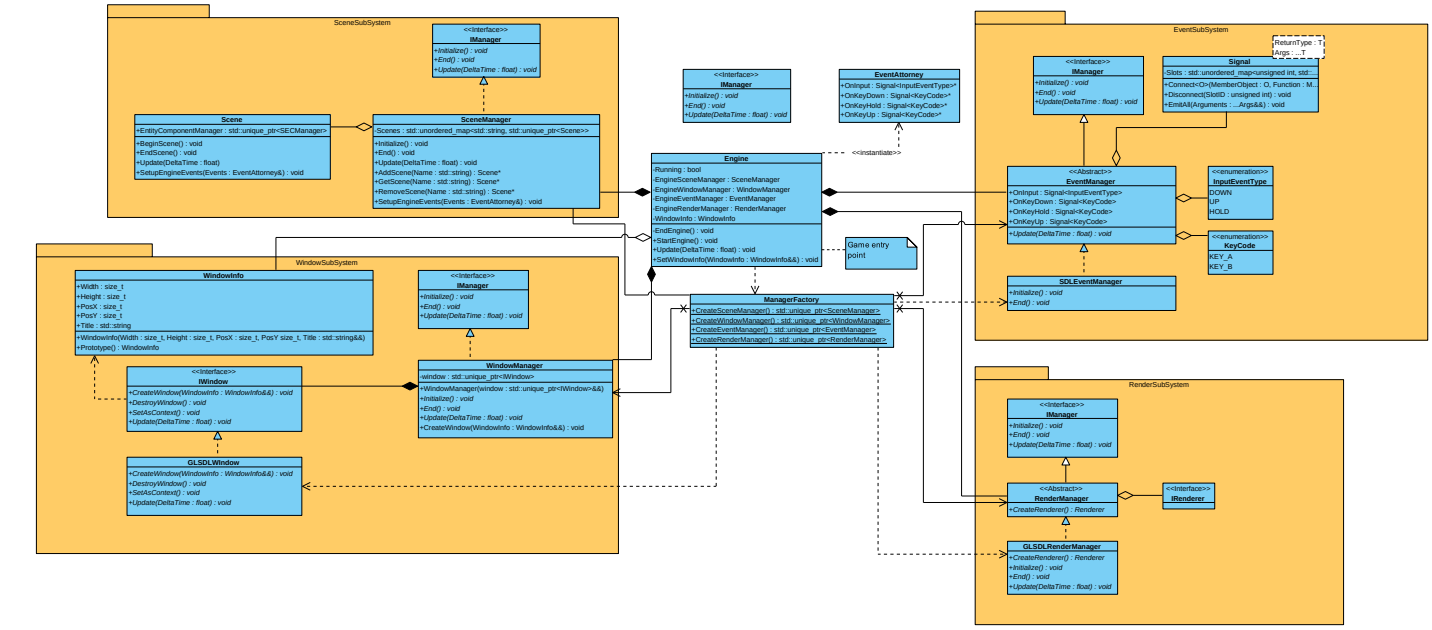
Every time the game updates, the four main managers have their update functions called. SceneManager's Update will cause the components of game objects within the scene to update, while EventManager's Update will update the event that the EventAttorney class passes to components when checking for events. WindowManager will update the game window in case any changes are made to it, and RenderManager will make any necessary changes to renderables in the game.

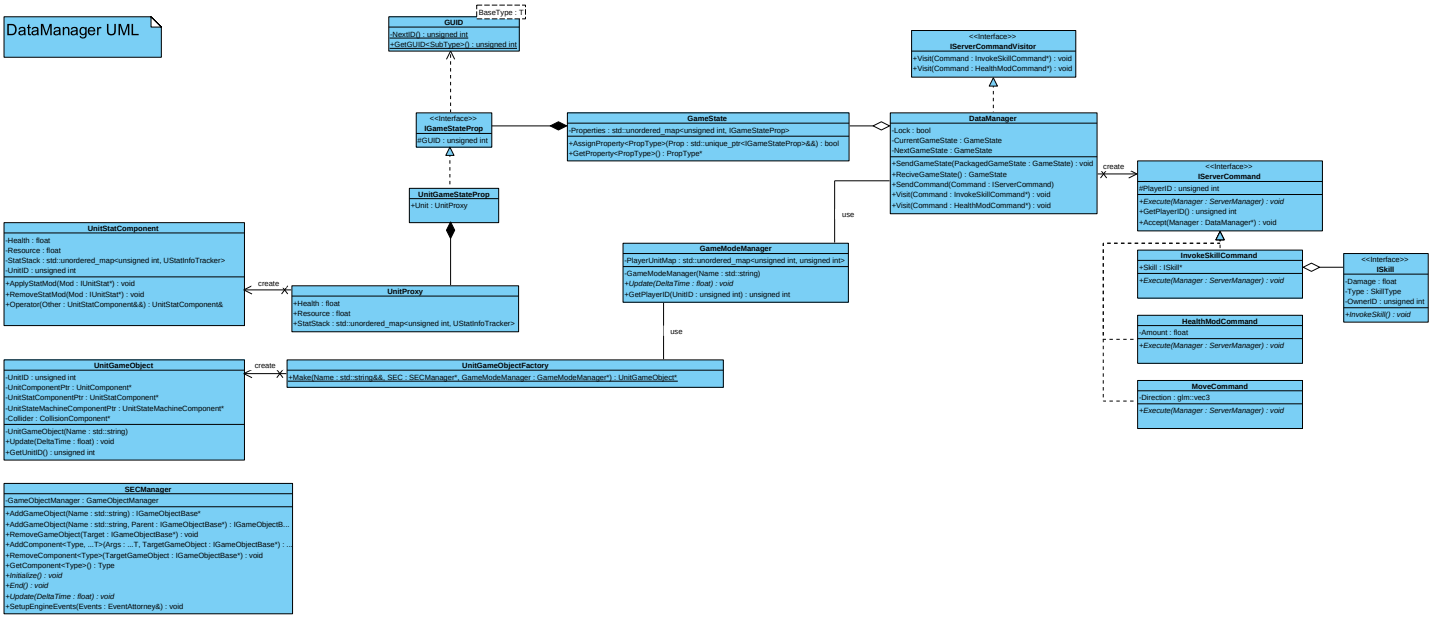
Handle Events(): This method will be constantly checking every loop if there was a player input and then it will carry out the appropriate action associated with the key press if all goes correctly. These events will be sent off to the server to be validated. Our handle events function is similar to the way that unreal engine handle's inputs. There is going to be a function within the component class called SetupEngineEvents which will be using dependency injection to bring in an EventAttorney class that contains the details of the event that occurred. E.G. EventAttorney.onKeyboardinput.register(your function here).

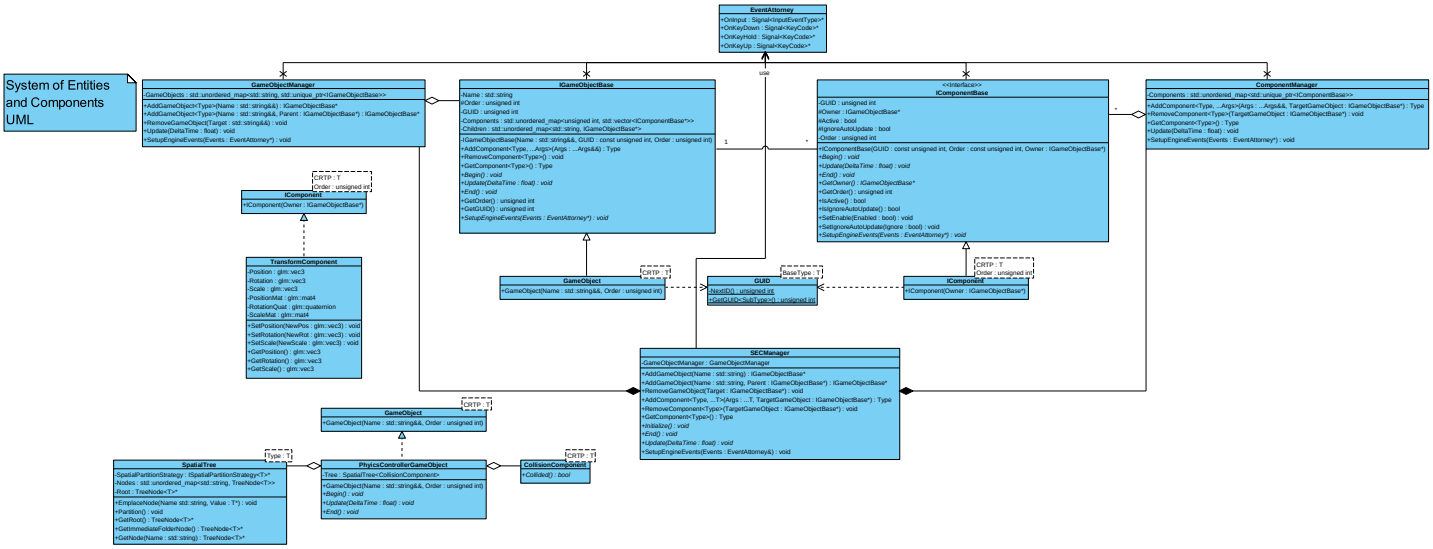
Render(): Our render function will be incorporated inside the main game loop and will be handling all the rendering of our game scene.

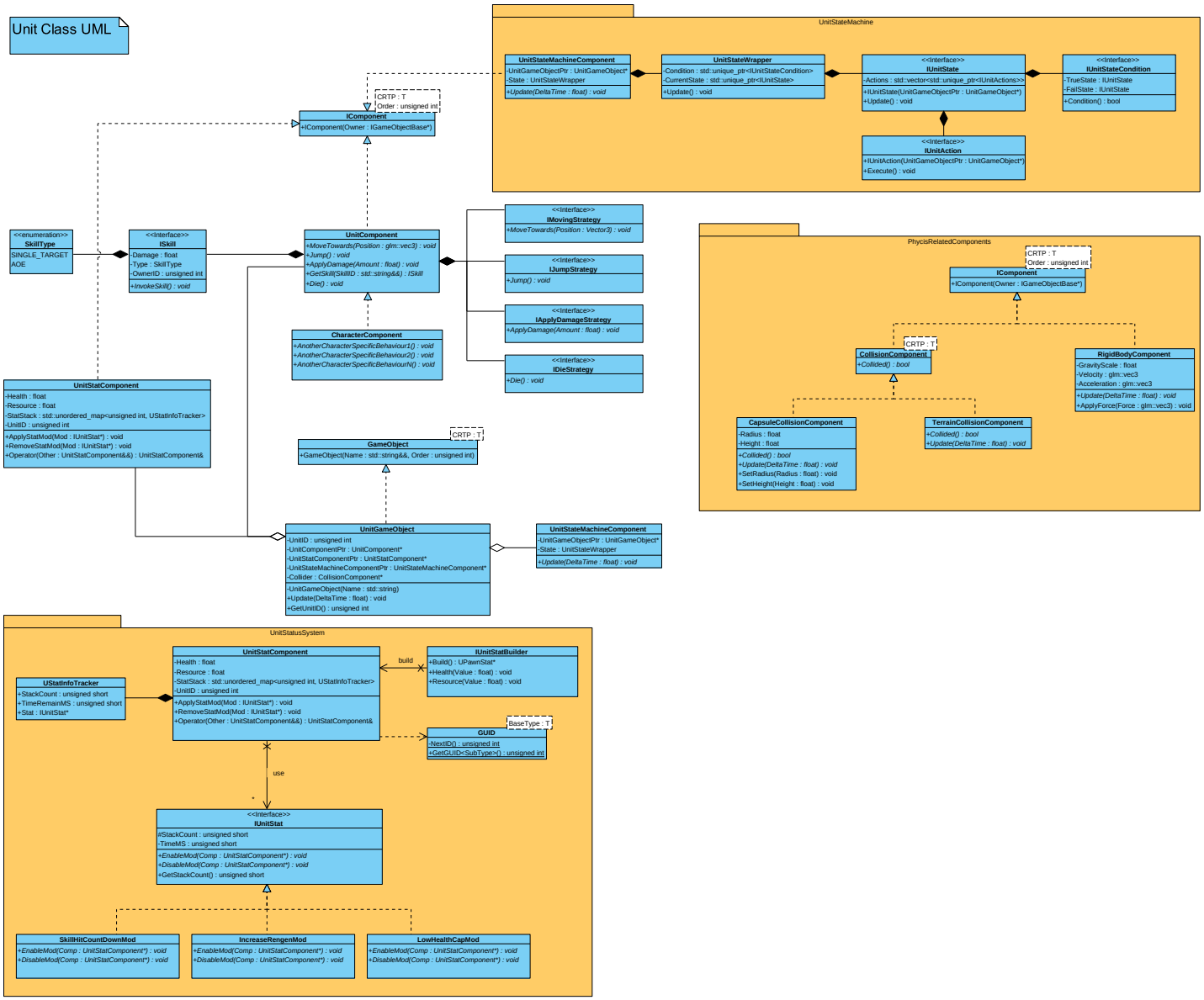
UML

Engine API UML




DataManager UML
BaseType : T
GUID
-NextID() : unsigned int
+GetGUID<SubType>() : unsigned int
<<Interface>>
IServerCommandVisitor
+Visit(Command : InvokeSkillCommand*) : void
+Visit(Command : HealthModCommand*) : void
<<Interface>>
IGameStateProp
#GUID : unsigned int
GameState
-Properties : std::unordered_map<unsigned int, IGameStateProp>
+AssignProperty<PropType>(Prop : std::unique_ptr<IGameStateProp>&&) : bool
+GetProperty<PropType>() : PropType*
DataManager
-Lock : bool
-CurrentGameState : GameState
-NextGameState : GameState
+SendGameState(PackagedGameState : GameState) : void
+ReciveGameState() : GameState
+SendCommand(Command : IServerCommand)
+Visit(Command : InvokeSkillCommand*) : void
+Visit(Command : HealthModCommand*) : void
create
<<Interface>>
IServerCommand
#PlayerID : unsigned int
+Execute(Manager : ServerManager) : void
+GetPlayerID() : unsigned int
+Accept(Manager : DataManager*) : void
UnitGameStateProp
+Unit : UnitProxy
use
UnitStatComponent
+Health : float
-Resource : float
-StatStack : std::unordered_map<unsigned int, UStatInfoTracker>
-UnitID : unsigned int
+ApplyStatMod(Mod : IUnitStat*) : void
+RemoveStatMod(Mod : IUnitStat*) : void
+Operator(Other : UnitStatComponent&&) : UnitStatComponent&
create
UnitProxy
+Health : float
+Resource : float
+StatStack : std::unordered_map<unsigned int, UStatInfoTracker>
GameModeManager
-PlayerUnitMap : std::unordered_map<unsigned int, unsigned int>
-GameModeManager(Name : std::string)
+Update(DeltaTime : float) : void
+GetPlayerID(UnitID : unsigned int) : unsigned int
use
InvokeSkillCommand
+Skill : ISkill*
+Execute(Manager : ServerManager) : void
<<Interface>>
ISkill
-Damage : float
-Type : SkillType
-OwnerID : unsigned int
+InvokeSkill() : void
HealthModCommand
-Amount : float
+Execute(Manager : ServerManager) : void
MoveCommand
-Direction : glm::vec3
+Execute(Manager : ServerManager) : void
UnitGameObject
-UnitID : unsigned int
-UnitComponentPtr : UnitComponent*
-UnitStatComponentPtr : UnitStatComponent*
-UnitStateMachineComponentPtr : UnitStateMachineComponent*
-Collider : CollisionComponent*
-UnitGameObject(Name : std::string)
+Update(DeltaTime : float) : void
+GetUnitID() : unsigned int
create
UnitGameObjectFactory
+Make(Name : std::string&&, SEC : SECManager*, GameModeManager : GameModeManager*) : UnitGameObject*
SECManager
-GameObjectManager : GameObjectManager
+AddGameObject(Name : std::string) : IGameObjectBase*
+AddGameObject(Name : std::string, Parent : IGameObjectBase*) : IGameObjectB...
+RemoveGameObject(Target : IGameObjectBase*) : void
+AddComponent<Type, ...T>(Args : ...T, TargetGameObject : IGameObjectBase*) : ...
+RemoveComponent<Type>(TargetGameObject : IGameObjectBase*) : void
+GetComponent<Type>() : Type
+Initialize() : void
+End() : void
+Update(DeltaTime : float) : void
+SetupEngineEvents(Events : EventAttorney&) : void


System of Entities and Components UML
EventAttorney
+OnInput : Signal<InputEventType>*
+OnKeyDown : Signal<KeyCode>*
+OnKeyHold : Signal<KeyCode>*
+OnKeyUp : Signal<KeyCode>*
use
GameObjectManager
-GameObjects : std::unordered_map<std::string, std::unique_ptr<IGameObjectBase>>
+AddGameObject<Type>(Name : std::string&&) : IGameObjectBase*
+AddGameObject<Type>(Name : std::string&&, Parent : IGameObjectBase*) : IGameObjectBase*
+RemoveGameObject(Target : std::string&&) : void
-Update(DeltaTime : float) : void
+SetupEngineEvents(Events : EventAttorney*) : void
IGameObjectBase
-Name : std::string
#Order : unsigned int
-GUID : unsigned int
-Components : std::unordered_map<unsigned int, std::vector<IComponentBase*>>
-Children : std::unordered_map<std::string, IGameObjectBase*>
+IGameObjectBase(Name : std::string&&, GUID : const unsigned int, Order : unsigned int)
+AddComponent<Type, ...Args>(Args : ...Args&&) : Type
+RemoveComponent<Type>() : void
+GetComponent<Type>() : Type
+Begin() : void
+Update(DeltaTime : float) : void
+End() : void
+GetOrder() : unsigned int
+GetGUID() : unsigned int
+SetupEngineEvents(Events : EventAttorney*) : void
1
*
<<Interface>>
IComponentBase
-GUID : unsigned int
#Owner : IGameObjectBase*
#Active : bool
#IgnoreAutoUpdate : bool
-Order : unsigned int
+IComponentBase(GUID : const unsigned int, Order : const unsigned int, Owner : IGameObjectBase*)
+Begin() : void
+Update(DeltaTime : float) : void
+End() : void
+GetOwner() : IGameObjectBase*
+GetOrder() : unsigned int
+IsActive() : bool
+IsIgnoreAutoUpdate() : bool
+SetEnable(Enabled : bool) : void
+SetIgnoreAutoUpdate(ignore : bool) : void
+SetupEngineEvents(Events : EventAttorney*) : void
*
ComponentManager
-Components : std::unordered_map<std::unique_ptr<IComponentBase>>
+AddComponent<Type, ...Args>(Args : ...Args&&, TargetGameObject : IGameObject*) : Type
+RemoveComponent<Type>(TargetGameObject : IGameObject*) : void
+GetComponent<Type>() : Type
+Update(DeltaTime : float) : void
+SetupEngineEvents(Events : EventAttorney*) : void
CRTP : T
Order : unsigned int
IComponent
+IComponent(Owner : IGameObject*)
TransformComponent
-Position : glm::vec3
-Rotation : glm::vec3
-Scale : glm::vec3
-PositionMat : glm::mat4
-RotationQuat : glm::quaternion
-ScaleMat : glm::mat4
+SetPosition(NewPos : glm::vec3) : void
+SetRotation(NewRot : glm::vec3) : void
+SetScale(NewScale : glm::vec3) : void
+GetPosition() : glm::vec3
+GetRotation() : glm::vec3
+GetScale() : glm::vec3
CRTP : T
GameObject
+GameObject(Name : std::string&&, Order : unsigned int)
GUID
-NextID() : unsigned int
+GetGUID<SubType>() : unsigned int
BaseType : T
SECManager
-GameObjectManager : GameObjectManager
+AddGameObject(Name : std::string) : IGameObjectBase*
+AddGameObject(Name : std::string, Parent : IGameObjectBase*) : IGameObjectBase*
+RemoveGameObject(Target : IGameObject*) : void
+AddComponent<Type, ...T>(Args : ...T, TargetGameObject : IGameObject*) : Type
+RemoveComponent<Type>(TargetGameObject : IGameObject*) : void
+GetComponent<Type>() : Type
+Initialize() : void
+End() : void
+Update(DeltaTime : float) : void
+SetupEngineEvents(Events : EventAttorney&) : void
CRTP : T
GameObject
+GameObject(Name : std::string&&, Order : unsigned int)
Type : T1
SpatialTree
-SpatialPartitionStrategy : ISpatialPartitionStrategy<T>*
-Nodes : std::unordered_map<std::string, TreeNode<T>>
-Root : TreeNode<T>*
+EmplaceNode(Name std::string, Value : T*) : void
+PartBon() : void
+GetRoot() : TreeNode<T>*
+GetImmediateFolderNode() : TreeNode<T>*
+GetNode(Name : std::string) : TreeNode<T>*
PhysicsControllerGameObject
-Tree : SpatialTree<CollisionComponent>
+GameObject(Name : std::string&&, Order : unsigned int)
+Begin() : void
+Update(DeltaTime : float) : void
+End() : void
CRTP : T1
CollisionComponent
+Collided() : bool


Unit Class UML
UnitStateMachine
UnitStateMachineComponent
-UnitGameObjectPtr : UnitGameObject*
-State : UnitStateWrapper
+Update(DeltaTime : float) : void
UnitStateWrapper
-Condition : std::unique_ptr<IUnitStateCondition>
-CurrentState : std::unique_ptr<IUnitState>
+Update() : void
<<Interface>>
IUnitState
-Actions : std::vector<std::unique_ptr<IUnitActions>>
+IUnitState(UnitGameObjectPtr : UnitGameObject*)
+Update() : void
<<Interface>>
IUnitStateCondition
-TrueState : IUnitState
-FailState : IUnitState
+Condition() : bool
<<Interface>>
IUnitAction
+IUnitAction(UnitGameObjectPtr : UnitGameObject*)
+Execute() : void
CRTP : T
Order : unsigned int
IComponent
+IComponent(Owner : IGameObjectBase*)
<<enumeration>>
SkillType
SINGLE_TARGET
AOE
<<Interface>>
ISkill
-Damage : float
-Type : SkillType
-OwnerID : unsigned int
+InvokeSkill() : void
UnitComponent
+MoveTowards(Position : glm::vec3) : void
+Jump() : void
+ApplyDamage(Amount : float) : void
+GetSkill(SkillID : std::string&&) : ISkill
+Die() : void
<<Interface>>
IMovingStrategy
+MoveTowards(Position : Vector3) : void
<<Interface>>
IJumpStrategy
+Jump() : void
<<Interface>>
IApplyDamageStrategy
+ApplyDamage(Amount : float) : void
<<Interface>>
IDieStrategy
+Die() : void
CharacterComponent
+AnotherCharacterSpecificBehaviour1() : void
+AnotherCharacterSpecificBehaviour2() : void
+AnotherCharacterSpecificBehaviourN() : void
UnitStatComponent
-Health : float
-Resource : float
-StatStack : std::unordered_map<unsigned int, UStatInfoTracker>
-UnitID : unsigned int
+ApplyStatMod(Mod : IUnitStat*) : void
+RemoveStatMod(Mod : IUnitStat*) : void
+Operator(Other : UnitStatComponent&&) : UnitStatComponent&
CRTP : T
GameObject
+GameObject(Name : std::string&&, Order : unsigned int)
UnitGameObject
-UnitID : unsigned int
-UnitComponentPtr : UnitComponent*
-UnitStatComponentPtr : UnitStatComponent*
-UnitStateMachineComponentPtr : UnitStateMachineComponent*
-Collider : CollisionComponent*
-UnitGameObject(Name : std::string)
+Update(DeltaTime : float) : void
+GetUnitID() : unsigned int
UnitStateMachineComponent
-UnitGameObjectPtr : UnitGameObject*
-State : UnitStateWrapper
+Update(DeltaTime : float) : void
PhycisRelatedComponents
CRTP : T
Order : unsigned int
IComponent
+IComponent(Owner : IGameObjectBase*)
CRTP : T
CollisionComponent
+Collided() : bool
RigidBodyComponent
-GravityScale : float
-Velocity : glm::vec3
-Acceleration : glm::vec3
+Update(DeltaTime : float) : void
+ApplyForce(Force : glm::vec3) : void
CapsuleCollisionComponent
-Radius : float
-Height : float
+Collided() : bool
+Update(DeltaTime : float) : void
+SetRadius(Radius : float) : void
+SetHeight(Height : float) : void
TerrainCollisionComponent
+Collided() : bool
+Update(DeltaTime : float) : void
UnitStatusSystem
UStatInfoTracker
+StackCount : unsigned short
+TimeRemainMS : unsigned short
+Stat : IUnitStat*
UnitStatComponent
-Health : float
-Resource : float
-StatStack : std::unordered_map<unsigned int, UStatInfoTracker>
-UnitID : unsigned int
+ApplyStatMod(Mod : IUnitStat*) : void
+RemoveStatMod(Mod : IUnitStat*) : void
+Operator(Other : UnitStatComponent&&) : UnitStatComponent&
build
IUnitStatBuilder
+Build() : UPawnStat*
+Health(Value : float) : void
+Resource(Value : float) : void
BaseType : T
GUID
-NextID() : unsigned int
+GetGUID<SubType>() : unsigned int
use
*
<<Interface>>
IUnitStat
#StackCount : unsigned short
-TimeMS : unsigned short
+EnableMod(Comp : UnitStatComponent*) : void
+DisableMod(Comp : UnitStatComponent*) : void
+GetStackCount() : unsigned short
SkillHitCountDownMod
+EnableMod(Comp : UnitStatComponent*) : void
+DisableMod(Comp : UnitStatComponent*) : void
IncreaseRengenMod
+EnableMod(Comp : UnitStatComponent*) : void
+DisableMod(Comp : UnitStatComponent*) : void
LowHealthCapMod
+EnableMod(Comp : UnitStatComponent*) : void
+DisableMod(Comp : UnitStatComponent*) : void

Utilities UML