### **Hello. I am a game developer working on a project in C# using the MonoGame framework. I am about to undertake a significant refactoring of my game's architecture to implement a modern, scalable Entity-Component-System (ECS).**

### **I have a detailed, multi-step roadmap for this process. I will be providing you with one step at a time. Your task is to act as an expert C# and game architecture programmer. For each step I provide, you will perform the following:**

### **1 Analyze the Request: Carefully read the Objective and Detailed Instructions for the step.**

### **2 Process the Context: I will provide you with all the necessary existing C# files as context. You must use these files to understand the current state of the project.**

### **3 Generate New Files: Create the complete, fully-functional C# code for any new files listed under "Files to Create." The code must be clean, well-commented, and adhere to standard C# conventions.**

### **4 Refactor Existing Files: Modify the provided context files exactly as described in the instructions. This may involve adding, changing, or removing methods, properties, and fields.**

### **5 Identify Deletions: If the instructions mention deleting a file, you will not provide code for it but will explicitly state, "The file [FileName.cs] should now be deleted from the project."**

### **6 Provide a Complete Response: Your final output for each step should be a single, coherent response containing all the new and modified code, clearly demarcated by file name. Do not ask me for clarification unless there is a direct and unresolvable contradiction in the instructions. Assume the instructions are the source of truth.**

### **Our goal is to collaboratively refactor my game. I will verify the output of each step before proceeding to the next. Let's begin.**

### **We are on step []**

### **Here is your step:**

### **Step 1: Establish the ECS Core Primitives**

**Objective:**Create the foundational, non-game-specific classes for our Entity-Component-System (ECS) architecture. This involves creating an entity manager, a base component interface, and a component storage system.

**Detailed Instructions:**

1. **Create IComponent.cs:**
   * This will be a simple marker interface. It will have no methods or properties. Its purpose is to enforce a type constraint on our components.
   * public interface IComponent { }
2. **Create EntityManager.cs:**
   * This class will be responsible for creating and destroying unique entity IDs. To be efficient, it should recycle destroyed IDs.
   * It will be a singleton, accessible via Instance.
   * **Fields:**
     + private Queue<int> \_availableIds = new Queue<int>();
     + private int \_nextId = 0;
   * **Methods:**
     + public int CreateEntity(): If the queue is not empty, dequeue and return an ID. Otherwise, return \_nextId++.
     + public void DestroyEntity(int entityId): Enqueue the entityId back into \_availableIds.
3. **Create ComponentStore.cs:**
   * This class will store all components for all entities, organized by component type.
   * It will be a singleton, accessible via Instance.
   * **Field:**
     + private Dictionary<Type, Dictionary<int, IComponent>> \_components = new Dictionary<Type, Dictionary<int, IComponent>>();
     + This is a dictionary where the key is the Type of the component (e.g., typeof(PositionComponent)) and the value is another dictionary that maps an entity ID (int) to its instance of that component (IComponent).
   * **Methods:**
     + public void AddComponent<T>(int entityId, T component) where T : IComponent: Adds a component for a given entity. It must first check if the outer dictionary contains the type T. If not, it creates a new inner dictionary for it.
     + public T GetComponent<T>(int entityId) where T : IComponent: Retrieves a component of type T for a given entity. Returns default(T) if not found.
     + public bool HasComponent<T>(int entityId) where T : IComponent: Checks if an entity has a specific component type.
     + public void RemoveComponent<T>(int entityId) where T : IComponent: Removes a component of type T from an entity.
     + public void EntityDestroyed(int entityId): A helper method that iterates through all component types and removes any components associated with the given entityId. This should be called when an entity is destroyed.
4. **Integrate into Core.cs:**
   * In the Core class, add static properties to access the singletons:
     + public static EntityManager EntityManager => \_entityManager;
     + public static ComponentStore ComponentStore => \_componentStore;
   * Instantiate them: private static readonly EntityManager \_entityManager = new(); and private static readonly ComponentStore \_componentStore = new();

**Files to Create:**

* IComponent.cs
* EntityManager.cs
* ComponentStore.cs

**Context Files (Provide these to the AI):**

* Core.cs

### **Step 2: Refactor the Player into an Entity**

**Objective:**Deconstruct the monolithic Player class and represent the player as a standard entity with a collection of components. This is a critical step to prove the viability of the ECS architecture.

**Detailed Instructions:**

1. **Create New Component Files:**
   * **PositionComponent.cs**:
     + Will implement IComponent.
     + Will contain public Vector2 WorldPosition { get; set; }.
   * **LocalPositionComponent.cs**:
     + Will implement IComponent.
     + Will contain public Vector2 LocalPosition { get; set; }.
   * **StatsComponent.cs**:
     + Will implement IComponent.
     + This component will hold all the data currently in PlayerStats.cs. Copy all properties from PlayerStats (e.g., MaxEnergyPoints, CurrentEnergyPoints, WalkSpeed, etc.) and all methods (ExertEnergy, Rest, CanExertEnergy).
   * **ActionQueueComponent.cs**:
     + Will implement IComponent.
     + Will contain public List<PendingAction> ActionQueue { get; } = new List<PendingAction>();.
   * **PlayerTagComponent.cs**:
     + Will implement IComponent.
     + This is a "marker" component. It will be an empty class, used only to identify which entity is the player.
2. **Refactor GameState.cs:**
   * **Remove Player Field:** Delete private Player \_player;.
   * **Add Player Entity ID:** Add public int PlayerEntityId { get; private set; }.
   * **Constructor Changes:**
     + In the GameState constructor, after initializing the managers, create the player entity:
       - PlayerEntityId = Core.EntityManager.CreateEntity();
     + Remove the line \_worldEntities.Add(\_player);.
     + Add all the new components for the player entity using Core.ComponentStore.AddComponent():
       - Add a PositionComponent with the initial position.
       - Add a LocalPositionComponent with the initial local position (32, 32).
       - Add a StatsComponent, initializing it with the same values as the old PlayerStats constructor.
       - Add an ActionQueueComponent.
       - Add the PlayerTagComponent.
   * **Update Property Accessors:** Go through every property and method in GameState that used \_player and change it to query the ComponentStore.
     + PlayerWorldPos becomes Core.ComponentStore.GetComponent<PositionComponent>(PlayerEntityId).WorldPosition.
     + PlayerLocalPos becomes Core.ComponentStore.GetComponent<LocalPositionComponent>(PlayerEntityId).LocalPosition.
     + PendingActions becomes Core.ComponentStore.GetComponent<ActionQueueComponent>(PlayerEntityId).ActionQueue.
     + PlayerStats becomes Core.ComponentStore.GetComponent<StatsComponent>(PlayerEntityId).
   * **Update Methods:** Systematically refactor methods like SimulateActionQueueEnergy, ApplyActionEffects, QueueMovementInternal, etc., to get player data from the ComponentStore instead of the \_player object. For example, \_player.SetLocalPosition(newPos) becomes Core.ComponentStore.GetComponent<LocalPositionComponent>(PlayerEntityId).LocalPosition = newPos;.
3. **Delete Obsolete Files:**
   * The Player.cs file is now redundant and should be deleted.
   * The PlayerStats.cs file is now redundant (its logic is in StatsComponent.cs) and should be deleted.
   * The Entity.cs file is no longer the pattern we are following. Delete it.

**Files to Create:**

* PositionComponent.cs
* LocalPositionComponent.cs
* StatsComponent.cs
* ActionQueueComponent.cs
* PlayerTagComponent.cs

**Context Files (Provide these to the AI):**

* GameState.cs
* Player.cs
* PlayerStats.cs
* Entity.cs
* Core.cs
* IComponent.cs
* EntityManager.cs
* ComponentStore.cs
* PendingAction.cs (from GameState.cs)

### **Step 3: Create the First Systems for Input and Action Execution**

**Objective:**Move game logic out of the monolithic GameState class and into dedicated, reusable System classes that operate on components.

**Detailed Instructions:**

1. **Create ISystem.cs:**
   * Create an interface that all systems will implement.
   * It should contain one method: void Update(GameTime gameTime);
2. **Create PlayerInputSystem.cs:**
   * This system will handle player-specific command processing.
   * Implement ISystem.
   * The Update method will not be used for this system. Instead, it will have public methods that are called by the CommandProcessor.
   * Move the logic from the following GameState methods into this new class:
     + QueueNewPath
     + AppendPath
     + CancelPendingActions
     + QueueRest
     + QueueMovementInternal (and its public wrappers QueueRunMovement and QueueWalkMovement)
   * These methods will no longer be part of GameState. They will operate on the PlayerEntityId from GameState and access its components via the ComponentStore. You will need to pass Core.CurrentGameState or its PlayerEntityId to this system's methods.
3. **Create ActionExecutionSystem.cs:**
   * This system will be responsible for executing actions from any entity's queue and applying their effects.
   * Implement ISystem.
   * Move the logic from the following GameState methods into this class's Update method:
     + UpdateMovement
     + ApplyActionEffects
     + CalculateSecondsForAction
   * The Update method in this system will be the main driver of action execution. It will check GameState's \_isExecutingPath flag. If true, it will get the ActionQueueComponent for the PlayerEntityId and process the current action.
   * **Crucially, this system should be designed to eventually work on *any* entity, not just the player.** For now, it's acceptable to hard-code it to the player's entity ID for simplicity.
4. **Refactor Core.cs:**
   * Add a list of systems: private static readonly List<ISystem> \_updateSystems = new();
   * Instantiate your new systems and add them to the list.
   * In the Core.Update method (within the \_sceneManager.Update call for the TerminalMapScene), add a loop: foreach (var system in \_updateSystems) { system.Update(gameTime); }.
   * The CommandProcessor will now need a reference to an instance of PlayerInputSystem to call its methods.
5. **Refactor GameState.cs:**
   * Remove all the methods that were just moved to the new systems. The GameState class should now be significantly smaller, primarily holding state flags (\_isExecutingPath, CurrentMapView) and the PlayerEntityId.

**Files to Create:**

* ISystem.cs
* PlayerInputSystem.cs
* ActionExecutionSystem.cs

**Context Files (Provide these to the AI):**

* Core.cs
* GameState.cs
* CommandProcessor.cs (to see how commands are handled)
* ComponentStore.cs
* PositionComponent.cs
* LocalPositionComponent.cs
* StatsComponent.cs
* ActionQueueComponent.cs
* PlayerTagComponent.cs

**Step 4: Spatial Partitioning - The Chunk System**

**Objective:**Implement the core data structures for spatial partitioning. This involves creating a Chunk class to hold entities for a single world coordinate and a ChunkManager to organize the world. This step lays the groundwork for loading and processing only the entities relevant to the player's current world location.

**Detailed Instructions:**

1. **Create Chunk.cs:**
   * This class represents the data associated with a single **1x1 grid cell** on the world map.
   * **Properties:**
     + public Point ChunkCoords { get; }: The coordinate of this chunk, which directly corresponds to a world map position (e.g., (0,0), (0,1)).
     + public HashSet<int> EntityIds { get; } = new HashSet<int>();: A collection of entity IDs currently located within this chunk (i.e., at this world coordinate). A HashSet is used for fast additions and removals.
   * **Constructor:**
     + public Chunk(Point chunkCoords): Initializes the chunk with its coordinates.
2. **Create ChunkManager.cs:**
   * This class manages the entire collection of chunks. It will be a singleton.
   * **Fields:**
     + private Dictionary<Point, Chunk> \_chunks = new Dictionary<Point, Chunk>();: Stores all loaded chunks, keyed by their Point coordinates.
   * **Methods:**
     + public Point WorldToChunkCoords(Vector2 worldPosition): A static helper method that converts a Vector2 world position into a Point chunk coordinate. The logic is simply a cast: new Point((int)worldPosition.X, (int)worldPosition.Y).
     + public void RegisterEntity(int entityId, Vector2 worldPosition):
       - Calculates the chunk coordinates for the given worldPosition.
       - Checks if a Chunk already exists at those coordinates in the \_chunks dictionary. If not, it creates a new Chunk and adds it.
       - Adds the entityId to the EntityIds HashSet of the correct chunk.
       - Updates the entity's PositionComponent to store its current chunk coordinates.
     + public void UnregisterEntity(int entityId, Vector2 worldPosition):
       - Calculates the chunk coordinates.
       - Finds the corresponding chunk and removes the entityId from its EntityIds set.
     + public void UpdateEntityChunk(int entityId, Vector2 oldPosition, Vector2 newPosition):
       - Calculates the chunk coordinates for both oldPosition and newPosition.
       - If the coordinates are the same, do nothing.
       - If they are different, call UnregisterEntity with the oldPosition and RegisterEntity with the newPosition.
     + public List<int> GetEntitiesInChunk(Point chunkCoords): Returns a list of entity IDs from the specified chunk. Returns an empty list if the chunk doesn't exist.
     + public List<int> GetEntitiesInTetherRange(Point centralChunkCoords):
       - This method's purpose is to get all entities relevant to the player. In this design, that means **only the entities in the player's current chunk**.
       - The implementation should simply call GetEntitiesInChunk(centralChunkCoords) and return the result. The 9-chunk logic is not needed.
3. **Modify PositionComponent.cs:**
   * Add a new property: public Point CurrentChunk { get; set; }. This will be set by the ChunkManager and allows for efficient lookup of an entity's current chunk without recalculating.
4. **Integrate into Core.cs:**
   * Add a static property for the ChunkManager singleton: public static ChunkManager ChunkManager => \_chunkManager;
   * Instantiate it: private static readonly ChunkManager \_chunkManager = new();
5. **Integrate into GameState.cs:**
   * In the constructor, after creating the player entity and its PositionComponent, register it with the ChunkManager:
     + var playerPos = Core.ComponentStore.GetComponent<PositionComponent>(PlayerEntityId).WorldPosition;
     + Core.ChunkManager.RegisterEntity(PlayerEntityId, playerPos);

### **Step 5: Activating Entities via the Chunk System**

**Objective:**Make the game systems aware of the spatial partitioning. The ActionExecutionSystem will now update an entity's chunk registration upon movement. We will also introduce the concept of "active entities" (those within the player's tether range) to limit processing.

**Detailed Instructions:**

1. **Refactor ActionExecutionSystem.cs:**
   * Modify the logic that handles WalkMove and RunMove actions.
   * After calculating the nextPosition and before updating the PositionComponent, store the oldPosition.
   * After updating the PositionComponent with the newPosition, call the ChunkManager:
     + Core.ChunkManager.UpdateEntityChunk(entityId, oldPosition, newPosition);
   * This ensures that every time an entity moves, its chunk registration is correctly updated.
2. **Modify GameState.cs:**
   * Add a new property to hold the list of entities that should be processed this frame:
     + public List<int> ActiveEntities { get; private set; } = new List<int>();
   * Create a new method UpdateActiveEntities():
     + This method will be called once per game tick.
     + It gets the player's current PositionComponent.
     + It converts the player's world position to chunk coordinates using ChunkManager.WorldToChunkCoords.
     + It calls ChunkManager.GetEntitiesInTetherRange with the player's chunk coordinates.
     + It sets the ActiveEntities list to the result of this call.
     + **High-Importance Entities:** This is where you implement the "always processed" feature from your roadmap.
       - Create a new marker component: HighImportanceComponent.cs which implements IComponent.
       - In UpdateActiveEntities, after getting the tethered entities, query the ComponentStore for all entities with a HighImportanceComponent.
       - Add these entities to the ActiveEntities list (use Union or a loop with a check to avoid duplicates).
3. **Refactor Core.cs:**
   * In the main Update loop (inside the TerminalMapScene logic), before the systems are updated, call the new method on GameState:
     + CurrentGameState.UpdateActiveEntities();
   * Now, all systems updated after this point will have access to a fresh list of active entities for the current frame.
4. **Update Systems to Use Active Entities:**
   * For now, this is a conceptual change. In later steps, systems like AISystem and RenderSystem will be modified to iterate over GameState.ActiveEntities instead of all entities in the world. The ActionExecutionSystem is a special case because it only needs to act on the player, which is always active.

**Files to Create:**

* HighImportanceComponent.cs

**Context Files (Provide these to the AI):**

* ActionExecutionSystem.cs
* GameState.cs
* Core.cs
* ChunkManager.cs
* PositionComponent.cs
* ComponentStore.cs
* IComponent.cs

### **Step 6: Tiered Update Frequencies & System Management**

**Objective:**Implement the "Tiered Update Frequencies" from the roadmap. This will allow different systems to run at different intervals (e.g., AI every 0.5 seconds, rendering every frame), saving significant CPU resources. We will create a SystemManager to orchestrate this.

**Detailed Instructions:**

1. **Create SystemManager.cs:**
   * This class will replace the simple List<ISystem> in Core.cs.
   * **Fields:**
     + private List<SystemEntry> \_systems = new List<SystemEntry>();
   * **Methods:**
     + public void RegisterSystem(ISystem system, float updateIntervalSeconds = 0f):
       - This method adds a system to the manager.
       - An updateIntervalSeconds of 0 means the system should run every frame.
       - It creates a SystemEntry object and adds it to the \_systems list.
     + public void Update(GameTime gameTime):
       - This is the main update loop for all systems.
       - It iterates through the \_systems list.
       - For each SystemEntry, it adds gameTime.ElapsedGameTime.TotalSeconds to the entry's \_accumulator.
       - If the \_accumulator is greater than or equal to the UpdateInterval, it calls the system's Update method and then subtracts the UpdateInterval from the \_accumulator. This handles both per-frame and periodic updates correctly.
2. **Create SystemEntry.cs (as a private nested class or a separate file):**
   * This is a small data-holding class used by SystemManager.
   * **Properties:**
     + public ISystem System { get; }
     + public float UpdateInterval { get; }
     + public float Accumulator { get; set; }
   * **Constructor:**
     + Takes the system and interval as parameters.
3. **Refactor Core.cs:**
   * Remove the \_updateSystems list and the foreach loop that updates them.
   * Add a static property for the SystemManager: public static SystemManager SystemManager => \_systemManager;
   * Instantiate it: private static readonly SystemManager \_systemManager = new();
   * In the Initialize or LoadContent method, register your existing systems with the new manager:
     + \_systemManager.RegisterSystem(new ActionExecutionSystem(), 0f); // Runs every frame
     + (The PlayerInputSystem doesn't have an Update loop, so it doesn't need to be registered here).
   * In the Core.Update method (for the TerminalMapScene), call the manager's update:
     + \_systemManager.Update(gameTime);

**Files to Create:**

* SystemManager.cs
* SystemEntry.cs (can be inside SystemManager.cs)

**Context Files (Provide these to the AI):**

* Core.cs
* ISystem.cs
* ActionExecutionSystem.cs
* PlayerInputSystem.cs

### **Step 7: Generalize Action Data Structures**

**Objective:**Refactor the PendingAction class into a more flexible, object-oriented system using an IAction interface. Crucially, we will **not** treat actions as components yet. This step safely isolates the change to the data structure itself, building a stable foundation for the next step.

**Detailed Instructions:**

1. **Create IAction.cs:**
   * This will be the base interface for all actions.
   * **It will NOT implement IComponent in this step.**
   * Properties:
     + public int ActorId { get; }
     + public bool IsComplete { get; set; }
2. **Create Concrete Action Classes:**
   * **MoveAction.cs:**
     + Implements IAction.
     + Properties: ActorId, IsComplete, public Vector2 Destination { get; }, public bool IsRunning { get; }.
     + Should have a constructor to set these values.
   * **RestAction.cs:**
     + Implements IAction.
     + Properties: ActorId, IsComplete, public RestType RestType { get; }.
     + Should have a constructor to set these values.
3. **Refactor ActionQueueComponent.cs:**
   * Change the ActionQueue property from List<PendingAction> to public Queue<IAction> ActionQueue { get; } = new Queue<IAction>();.
4. **Refactor PlayerInputSystem.cs:**
   * Update all methods that created PendingAction objects. They should now create instances of MoveAction or RestAction and enqueue them into the player's ActionQueueComponent.
5. **Refactor ActionExecutionSystem.cs:**
   * The system's Update loop will still be hard-coded to the player for now.
   * It should check if the player's ActionQueue has items. If so, it should Peek() at the current action without removing it.
   * Use if (action is MoveAction moveAction) and if (action is RestAction restAction) to handle the logic for each action type.
   * When an action's logic is finished, set its IsComplete flag to true.
   * At the end of the processing logic, if the current action at the front of the queue IsComplete, Dequeue() it from the queue.
6. **Delete Obsolete Code:**
   * The PendingAction class and ActionType enum are now obsolete and should be removed.

**Files to Create:**

* IAction.cs
* MoveAction.cs
* RestAction.cs

**Context Files (Provide these to the AI):**

* ActionQueueComponent.cs
* PlayerInputSystem.cs
* ActionExecutionSystem.cs
* GameState.cs (to remove PendingAction and ActionType)
* IComponent.cs

### **Step 8: Implement Action Component Mechanics**

**Objective:**Refactor the system to mechanically handle actions as components. This step focuses *only* on the process of attaching a dequeued action to an entity and then cleaning it up once it's marked complete. We will **not** implement the actual processing logic yet, allowing us to safely test the new architectural scaffolding.

**Detailed Instructions:**

1. **Modify IAction.cs:**
   * Update the IAction interface to implement IComponent.
   * The new signature should be: public interface IAction : IComponent
2. **Refactor ActionExecutionSystem.cs:**
   * This is a complete rewrite of the system's Update method.
   * The main loop should now iterate over all GameState.ActiveEntities.
   * The new logic must follow a strict three-phase cycle:
   * **Phase 1 - INITIATION:**
     + Iterate through all ActiveEntities. For each entityId:
     + Check if the entity has an ActionQueueComponent with ActionQueue.Count > 0.
     + **AND** check that the entity does **NOT** currently have any IAction component attached.
     + If both conditions are true, Dequeue the next action and use Core.ComponentStore.AddComponent(entityId, nextAction) to attach it.
   * **Phase 2 - DUMMY PROCESSING (for testing):**
     + Query the ComponentStore for all entities that have an IAction component.
     + For each found action, immediately set its IsComplete property to true. This is only to test the cleanup mechanism.
   * **Phase 3 - CLEANUP:**
     + Query the ComponentStore for all entities that have an IAction component where IsComplete is true.
     + For each of these, remove the action component from the entity.

**Files to Create:**

* None. This step involves only refactoring existing files.

**Context Files (Provide these to the AI):**

* IAction.cs
* ActionExecutionSystem.cs
* ActionQueueComponent.cs
* ComponentStore.cs
* GameState.cs

### **Step 9: Implement Action Processing Logic**

**Objective:**Build upon the verified mechanics from the previous step by adding the actual processing logic. The system will now execute the behavior defined by the attached action component over time, bringing movement back to life within the new, robust architecture.

**Detailed Instructions:**

1. **Refactor ActionExecutionSystem.cs:**
   * Modify the system created in the previous step. The Initiation (Phase 1) and Cleanup (Phase 3) logic are correct and should not be changed.
   * You will **replace the "Dummy Processing" phase** with the real logic.
   * **Phase 2 - REAL PROCESSING:**
     + **Process MoveActions:** Query the ComponentStore for all entities that have a MoveAction component. For each entity and its moveAction:
       - Execute the movement logic (calculate time, move the entity, update position).
       - When the movement to the destination is finished, set moveAction.IsComplete = true.
     + **Process RestActions:** Query the ComponentStore for all entities that have a RestAction component. For each entity and its restAction:
       - Execute the rest logic (e.g., restore energy).
       - When the rest is finished, set restAction.IsComplete = true.

**Files to Create:**

* None. This step involves only refactoring an existing file.

**Context Files (Provide these to the AI):**

* ActionExecutionSystem.cs
* ComponentStore.cs
* PositionComponent.cs
* StatsComponent.cs
* MoveAction.cs
* RestAction.cs

### **Step 10: Implementing Basic AI**

**Objective:**Bring the first non-player entity to life. We will create a simple "Wanderer" NPC that moves randomly, proving that our new, generalized action systems can handle multiple entities with minimal effort.

**Detailed Instructions:**

1. **Create AI-Related Components:**
   * **AIComponent.cs:** Implements IComponent. Will hold the AI's state. Properties: public AIState CurrentState { get; set; }, public float StateTimer { get; set; }.
   * **AIState.cs:** An enum with initial states: Idle, Wandering.
   * **NPCTagComponent.cs:** An empty marker component, similar to PlayerTagComponent, to identify NPCs.
2. **Create AISystem.cs:**
   * Implements ISystem. This system is the "brain" for our NPCs.
   * In its Update method, it iterates through all ActiveEntities from GameState.
   * For each entity, it checks if it has an AIComponent, an NPCTagComponent, and does **not** have an active IAction component (meaning it's not busy).
   * **FSM Logic:**
     + If CurrentState is Idle, it decrements the StateTimer. When the timer hits zero, it transitions to Wandering, sets a new random timer, and enqueues a new MoveAction to a random adjacent passable tile.
     + If CurrentState is Wandering, it does nothing (it's waiting for the ActionExecutionSystem to complete the move). Once the MoveAction is removed, on the next AISystem tick, the AI will see the entity is not busy and transition it back to Idle.
3. **Spawn an NPC in GameState.cs:**
   * In the constructor, after creating the player, create a new NPC entity.
   * Give it the necessary components: PositionComponent, ActionQueueComponent, AIComponent, and NPCTagComponent.
   * Register it with the ChunkManager.
4. **Register the AISystem in Core.cs:**
   * In the SystemManager registration block, add the new system. It should run at a lower frequency.
   * \_systemManager.RegisterSystem(new AISystem(), 0.5f); // AI thinks twice a second

**Files to Create:**

* AIComponent.cs
* AIState.cs
* NPCTagComponent.cs
* AISystem.cs

**Context Files (Provide these to the AI):**

* GameState.cs
* Core.cs
* SystemManager.cs
* ComponentStore.cs
* ActionQueueComponent.cs
* MoveAction.cs
* IAction.cs

### **Step 11: Create and Load Archetype Data Structures**

**Objective:**Create the necessary classes for managing archetypes and the JSON data files themselves. This step focuses only on *loading* the data from disk and deserializing it. We will not use it to spawn entities yet, allowing us to verify that file I/O is working correctly in isolation.

**Detailed Instructions:**

1. **Create Archetype.cs:**o This class will be a direct mapping of our JSON structure.  
   o Properties: public string Id { get; set; }, public string Name { get; set; }, public List<Dictionary<string, object>> Components { get; set; }.
2. **Create ArchetypeManager.cs:**o A singleton responsible for loading and storing archetypes.  
   o Implement the standard singleton pattern (public static readonly Instance, private constructor).  
   o It will contain a private readonly Dictionary<string, Archetype> \_archetypes.  
   o Methods: public void LoadArchetypes(string directoryPath) which reads all .json files, deserializes them into Archetype objects, and stores them in the dictionary. public Archetype GetArchetype(string id) which retrieves a loaded archetype by its ID.
3. **Create JSON Archetype Files:**o Create a new folder: Content/Archetypes.  
   o **player.json**: Define the player archetype. It must include PositionComponent, StatsComponent, ActionQueueComponent, PlayerTagComponent, and HighImportanceComponent. The StatsComponent should have a Properties object with values for MaxEnergyPoints, CurrentEnergyPoints, etc.  
   o **wanderer\_npc.json**: Define the NPC archetype. It must include PositionComponent, ActionQueueComponent, AIComponent, and NPCTagComponent. The AIComponent should have a Properties object with values for CurrentState and StateTimer.  
   o **Note:** The Type string in the JSON must be the fully-qualified C# type name (e.g., "MyGame.Components.PositionComponent, MyGame"). The assembly name after the comma is often required.
4. **Refactor Core.cs to Load Archetypes:**o In the Core.LoadContent method, add the call: ArchetypeManager.Instance.LoadArchetypes("Content/Archetypes");. This ensures they are loaded before they are needed.

**Files to Create:**

* Archetype.cs
* ArchetypeManager.cs
* Content/Archetypes/player.json
* Content/Archetypes/wanderer\_npc.json

**Context Files (Provide these to the AI):**

* Core.cs
* GameState.cs

### **Step 12: Create the Entity Spawner**

**Objective:**Create a static Spawner class that can create an entity and its components from a loaded archetype. This step isolates the complex reflection logic into a single, dedicated class, making it easier to debug.

**Detailed Instructions:**

1. **Create Spawner.cs:**o This will be a static helper class.  
   o Method: public static int Spawn(string archetypeId, Vector2 position).  
   o This method gets the archetype from ArchetypeManager, creates an entity ID from EntityManager, and then iterates through the components defined in the archetype.  
   o For each component, it uses reflection (Type.GetType, Activator.CreateInstance) to create an instance.  
   o It then checks for a "Properties" dictionary in the JSON data. If found, it iterates through each property, uses reflection (PropertyInfo.SetValue) to populate the component instance with the values from the JSON. This will require a helper method to convert JsonElement objects to the correct C# types (int, float, enum, etc.).  
   o After creating and populating a component, it is added to the ComponentStore.  
   o After all components are added, it specifically sets the WorldPosition on the entity's PositionComponent using the position parameter from the Spawn method.  
   o Finally, it registers the newly created entity with the ChunkManager and returns the new entityId.

**Files to Create:**

* Spawner.cs

**Context Files (Provide these to the AI):**

* Core.cs
* ArchetypeManager.cs
* EntityManager.cs
* ComponentStore.cs
* ChunkManager.cs
* IComponent.cs
* All existing Component class files (PositionComponent.cs, StatsComponent.cs, etc.)

### **Step 13: Integrate Spawner and Fix Initialization Order**

**Objective:**Replace the hard-coded entity creation with the new data-driven Spawner. This step fixes the critical initialization order problem, ensuring that archetypes are loaded from disk *before* any code attempts to spawn an entity from them.

**Detailed Instructions:**

1. **Refactor GameState.cs:**o Remove all entity creation and component-adding logic from the GameState constructor. The constructor should become very simple.  
   o Create a new method: public void InitializeWorld().  
   o Move the entity creation logic into this new method, but replace it with calls to the new spawner:  
   § PlayerEntityId = Spawner.Spawn("player", new Vector2(32, 32));  
   § Spawner.Spawn("wanderer\_npc", new Vector2(40, 40));
2. **Refactor Core.cs:**o Locate the LoadContent method.  
   o At the very end of this method, after the call to ArchetypeManager.Instance.LoadArchetypes(...), add the call to initialize the game world:  
   § CurrentGameState.InitializeWorld();  
   o This change guarantees that Spawner.Spawn is only ever called after the ArchetypeManager has loaded all the necessary data.

**Files to Create:**

* None. This step involves only refactoring existing files.

**Context Files (Provide these to the AI):**

* Core.cs
* GameState.cs
* Spawner.cs

### **Step 14 (Revised): The Combat Data Layer & Action Costs**

**Objective:**Establish the foundational data structures for combat. This includes creating components for health and combat prowess, defining an AttackAction, and, crucially, introducing the concept of a TimeCost to all actions. This TimeCost is the cornerstone of our pseudo-turn-based system, representing how much "in-combat time" an action consumes. We will also create a centralized, static CombatLog to decouple combat messages from the UI, which we will build in a later step.

**Detailed Instructions:**

1. **Create New Combat Components:**
   * **HealthComponent.cs:** Implements IComponent. It should have properties for CurrentHealth and MaxHealth, both integers. It should also have a method TakeDamage that accepts an integer amount and reduces CurrentHealth without letting it go below zero.
   * **CombatantComponent.cs:** Implements IComponent. It should have integer properties for AttackPower and float properties for AttackRange and AggroRange.
2. **Create New Action Class:**
   * **AttackAction.cs:** Implements IAction. It should have properties for ActorId, IsComplete, and TargetId. It also needs a constructor to set these values upon creation.
3. **Refactor Action Interfaces and Classes to include Time Cost:**
   * **Modify IAction.cs:** Add a new integer property named TimeCost. This represents the number of "action ticks" or "time units" the action takes to perform.
   * **Modify MoveAction.cs:** Implement the TimeCost property. The value should be conditional; for example, the time cost could be 50 if running and 100 if walking.
   * **Modify RestAction.cs:** Implement the TimeCost property with a hard-coded value, such as 100.
   * **Modify AttackAction.cs:** Implement the TimeCost property with a hard-coded value, such as 100.
4. **Create a Centralized Combat Log:**
   * **Create CombatLog.cs:** This will be a public static class. It will manage a simple list of combat messages. Add a static event that is triggered when a message is logged. Add a static Log method that takes a string message and invokes the event. This decouples the combat logic from the UI renderer.
5. **Create the Core Combat Logic System:**
   * **Create CombatSystem.cs:** Implements ISystem. Its Update method will query the ComponentStore for any entities with an active AttackAction component. For each AttackAction, it will find the target entity, get the target's HealthComponent and the attacker's CombatantComponent, and then call the TakeDamage method on the target. It should then log the result using the static CombatLog.Log method. For example, it could log a red-colored string indicating who attacked whom and for how much damage. Finally, it should set the attackAction.IsComplete flag to true.
6. **Update Archetypes:**
   * **Modify Content/Archetypes/player.json:** Add HealthComponent and CombatantComponent with default values.
   * **Rename Content/Archetypes/wanderer\_npc.json to goblin\_warrior.json**.
   * **Modify Content/Archetypes/goblin\_warrior.json:** Add HealthComponent and CombatantComponent. Make it weaker than the player.
7. **Register New System in Core.cs:**
   * In the SystemManager registration block, add the new CombatSystem and set it to run every frame.

**Files to Create:**

* HealthComponent.cs
* CombatantComponent.cs
* AttackAction.cs
* CombatLog.cs
* CombatSystem.cs

**Files to Refactor:**

* IAction.cs
* MoveAction.cs
* RestAction.cs
* Core.cs
* Content/Archetypes/player.json
* Content/Archetypes/wanderer\_npc.json (rename to goblin\_warrior.json and modify)

**Context Files (Provide these to the AI):**

* IAction.cs, MoveAction.cs, RestAction.cs (to show the existing action structure)
* IComponent.cs
* Core.cs
* SystemManager.cs
* ComponentStore.cs
* ActionExecutionSystem.cs
* The JSON archetype files.

### **Step 15: Combat State Management & Initiation**

**Objective:**Make the game formally aware of a "combat state." This involves adding flags and lists to GameState to track who is in combat. We will then modify the AISystem to be the trigger that initiates combat, which in turn will cause the SceneManager to transition to the CombatScene.

**Detailed Instructions:**

1. **Refactor GameState.cs to Manage Combat State:**
   * Add new properties: a boolean IsInCombat (defaulting to false) and a list of integers named Combatants.
   * Add a new method InitiateCombat that takes an initiator ID and a target ID. If combat is already active, it should just add both entities to the fight. Otherwise, it should set IsInCombat to true, clear any pending actions for both entities, add them to the Combatants list, and log that combat has begun.
   * Add a new method AddEntityToCombat that takes an entity ID. It should check if the entity is already a combatant before adding them to the Combatants list and logging that they have joined the fight.
   * Add a new method EndCombat that sets IsInCombat to false, clears the Combatants list, and logs that combat has ended.
2. **Refactor AISystem.cs to Trigger Combat:**
   * In the Update method, within the Chasing state logic, change the behavior. Instead of just enqueuing actions, it should now check the distance to the player against its AttackRange from its CombatantComponent. If the player is within attack range, it should call the new GameState.InitiateCombat method. If the player is only within the wider AggroRange, it should continue to enqueue a MoveAction as before.
3. **Refactor Core.cs to Handle Scene Transition:**
   * In the Core.Update method, before the scene manager's update call, add a new block for scene state management. This logic should check if the game state's IsInCombat flag is true while the current scene is not the CombatScene. If so, it should change the scene to CombatScene. Conversely, if the IsInCombat flag is false but the current scene is the CombatScene, it should change the scene back to the TerminalMapScene. After this state check, the scene manager's update method should be called as normal.
4. **Refactor CombatSystem.cs to End Combat:**
   * In the Update method, after processing all attacks, add logic to check if combat should end. A simple condition would be to check if the player is dead or if there are no hostile NPCs (identifiable by their NPCTagComponent) left in the GameState.Combatants list. If the condition is met, call GameState.EndCombat.

**Files to Create:**

* None.

**Files to Refactor:**

* GameState.cs
* AISystem.cs
* Core.cs
* CombatSystem.cs

**Context Files (Provide these to the AI):**

* GameState.cs
* AISystem.cs
* Core.cs
* CombatSystem.cs
* SceneManager.cs
* CombatScene.cs
* TerminalMapScene.cs
* CombatLog.cs
* NPCTagComponent.cs

### **Step 16: The Pseudo-Turn-Based Engine**

**Objective:**Implement the core mechanics of the time-based combat system. This involves creating a component to track an entity's available "action time" and a new system, CombatTurnSystem, to manage the flow of combat turns. Actions will now consume this time, and the ActionExecutionSystem will be updated to enforce this.

**Detailed Instructions:**

1. **Create ActionTimeComponent.cs:**
   * Implements IComponent.
   * It should have a single integer property: ActionTime. This will store the number of "time units" an entity has available to spend on actions.
2. **Refactor GameState.cs:**
   * Add a new public constant integer named COMBAT\_TURN\_TIME with a value like 500. This represents the amount of ActionTime granted to each combatant at the start of a "turn".
3. **Create CombatTurnSystem.cs:**
   * Implements ISystem.
   * The Update method should only run if GameState.IsInCombat is true.
   * The update logic should get the list of combatants and ensure each one has an ActionTimeComponent. It should then check if any combatant has an action queued. If no one has an action queued, it signifies the end of a turn. In this case, it should grant ActionTime to all combatants. The amount of time granted should be the COMBAT\_TURN\_TIME constant plus a bonus derived from the entity's Agility stat (e.g., Agility multiplied by 10). It should then log that a new turn has started.
4. **Refactor ActionExecutionSystem.cs:**
   * The Phase 1 - INITIATION logic needs to be updated for combat. When iterating through active entities, if the game is in combat, it must perform an additional check. Before dequeuing an action, it must get the entity's ActionTimeComponent and the TimeCost of the pending action. Only dequeue and attach the action if the entity's available ActionTime is greater than or equal to the action's TimeCost.
   * The Phase 3 - CLEANUP logic also needs an update. When an action component is removed, it should get the IAction interface from it to access its TimeCost. It should then get the entity's ActionTimeComponent and subtract the completed action's TimeCost from the entity's total ActionTime.
5. **Refactor AISystem.cs:**
   * In the Chasing state, before enqueuing an AttackAction, the AI must now also check if it has enough ActionTime to perform the attack. It should get its ActionTimeComponent and compare its value against the TimeCost of a potential AttackAction. Only enqueue the action if it has sufficient time.
6. **Register New System in Core.cs:**
   * Register the CombatTurnSystem with the SystemManager. It should be registered to run before the AISystem and ActionExecutionSystem.

**Files to Create:**

* ActionTimeComponent.cs
* CombatTurnSystem.cs

**Files to Refactor:**

* GameState.cs
* ActionExecutionSystem.cs
* AISystem.cs
* Core.cs

**Context Files (Provide these to the AI):**

* GameState.cs
* ActionExecutionSystem.cs
* AISystem.cs
* Core.cs
* SystemManager.cs
* StatsComponent.cs
* IAction.cs

### **Step 17: Implementing the Combat Scene & UI**

**Objective:**Flesh out the CombatScene to make it functional. This involves creating a dedicated CombatTerminalRenderer to display combat logs, a PlayerCombatInputSystem to handle player commands during a fight, and updating the CombatScene to use these new pieces. We will also make the game world's clock continue to tick during combat.

**Detailed Instructions:**

1. **Create CombatTerminalRenderer.cs:**
   * This class will be very similar to TerminalRenderer but will be for display only. It will not handle user input.
   * It needs a public AddLog method that takes a message, parses it for colors, wraps it, and adds it to its internal history.
   * It needs a Draw method to render the log to the screen in a designated area.
   * In its constructor, it should subscribe to the static CombatLog.OnMessageLogged event.
2. **Create PlayerCombatInputSystem.cs:**
   * Implements ISystem.
   * This system will handle player input only when GameState.IsInCombat is true.
   * Its Update method will be empty; it will have public methods called by CombatScene.
   * Create a method HandleAttackCommand that takes a target ID. This method will get the player's entity ID, check if the player has enough ActionTime for an attack, and if so, enqueue a new AttackAction targeting the given ID.
3. **Refactor CombatScene.cs:**
   * Add private fields for the CombatTerminalRenderer and PlayerCombatInputSystem and instantiate them upon entering the scene.
   * In the Update method, call Core.CurrentWorldClockManager.Update to ensure time passes in the background. Check for player input, such as clicking on an enemy. If an enemy is clicked, call the HandleAttackCommand method from the PlayerCombatInputSystem.
   * In the Draw method, draw the scene background, the combatants (with health bars), and then call the Draw method of the CombatTerminalRenderer.
4. **Register New System in Core.cs:**
   * Instantiate PlayerCombatInputSystem in Core.cs and make it accessible via a static property. It does not need to be registered with the SystemManager's update loop, as it is driven by the CombatScene.

**Files to Create:**

* CombatTerminalRenderer.cs
* PlayerCombatInputSystem.cs

**Files to Refactor:**

* CombatScene.cs
* Core.cs

**Context Files (Provide these to the AI):**

* CombatScene.cs
* Core.cs
* TerminalRenderer.cs (as a template for CombatTerminalRenderer)
* CombatLog.cs
* GameState.cs
* WorldClockManager.cs
* AttackAction.cs

### **Step 18: Advanced Combat: Dynamic Entry & World Simulation**

**Objective:**Implement the final, advanced features of the combat system. This includes creating a system that allows nearby non-combatant NPCs to join an ongoing fight, and ensuring that non-combatant entities continue their basic behaviors (like wandering) in the background while the player is in the CombatScene.

**Detailed Instructions:**

1. **Create CombatProximitySystem.cs:**
   * Implements ISystem.
   * Its Update method should only run if GameState.IsInCombat is true.
   * The update logic should get the list of current combatants. Then, it should find all NPCs that are not currently in combat. For each of these potential joiners, it should check their distance to every current combatant. If the distance is less than the potential joiner's AggroRange, it should call GameState.AddEntityToCombat to make them join the fight.
2. **Refactor Core.cs for Background Simulation:**
   * First, modify SystemManager.cs. Add a boolean property IsCombatSystem to the SystemEntry class. Modify the RegisterSystem method to accept an optional isCombatSystem boolean parameter.
   * Next, in Core.cs, when registering systems, mark the combat-specific ones (CombatSystem, CombatTurnSystem, CombatProximitySystem) by passing true for the new isCombatSystem parameter.
   * Finally, modify the SystemManager.Update method. Inside the main loop that iterates through systems, add a conditional check. If the game is in combat, all systems should update. However, if the game is not in combat, the update logic should only be executed for systems where the IsCombatSystem flag is false.
3. **Final Polish on AISystem.cs:**
   * In the AISystem.Update method, at the very beginning of the loop for each entity, add a check: if the entity is in the GameState.Combatants list, continue to the next entity. This prevents the AISystem from making non-combat decisions (like wandering) for an entity that is already engaged in a fight.

**Files to Create:**

* CombatProximitySystem.cs

**Files to Refactor:**

* SystemManager.cs
* Core.cs
* AISystem.cs

**Context Files (Provide these to the AI):**

* SystemManager.cs
* Core.cs
* AISystem.cs
* GameState.cs
* PositionComponent.cs
* CombatantComponent.cs
* NPCTagComponent.cs