### **Hello. I am a game developer working on a project in C# using the MonoGame framework. I am undertaking 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. If you are missing context, do not hallucinate a files contents, ask for it before proceeding.**

### **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.**

### **7 At the very end provide a detail, but brief explanation of exactly what changes, what was removed, and why**

### **Ask me for clarification if something is unclear or there are multiple possible appraoches ESPECIALLY if a direct and unresolvable contradiction is in the instructions.**

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

### **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:**

### **Tier 1: Critical Architectural Refinements**

*These are the most impactful changes that address fundamental architectural issues. Tackling these first will make all subsequent refactoring easier and more effective.*

**1. Dismantle the Core and Global God Objects with Dependency Injection (DI)**

* **Problem:** The Core.cs class is a classic "God Object" singleton. It creates and holds static references to every single manager and system in the game. This creates extremely tight coupling. Every system knows about every other system, making the code difficult to test, maintain, and reason about. For example, ActionExecutionSystem directly calls Core.CurrentTerminalRenderer, which it has no business knowing about directly. Global.cs has a similar, smaller-scale issue by holding direct references to CurrentGraphics and CurrentSpriteBatch.
* **Solution:** Implement a Dependency Injection (DI) pattern.
  + The main Core class (your Game class) should be responsible for *instantiating* the managers and systems.
  + When creating a system, pass the specific dependencies it needs into its constructor. For example, new ActionExecutionSystem(componentStore, worldClockManager, entityManager) instead of the system reaching out to Core.ComponentStore.
  + This immediately breaks the tight coupling. ActionExecutionSystem now only knows about the components and the clock, as it should. It no longer has access to the TerminalRenderer or SceneManager.
  + To communicate between decoupled systems, introduce a simple event bus or use C# Action / event patterns. (See Tier 2, Item 5).
* **Impact:** **(Highest)**. This is the single most important refactor. It will fundamentally improve the entire architecture, enabling unit testing, modularity, and parallel development.

**2. Eliminate Runtime Reflection in the Spawner**

* **Problem:** The Spawner uses Type.GetType(string) and reflection (Activator.CreateInstance, GetProperty, SetValue) to create components from JSON definitions at runtime. While this is very flexible for rapid prototyping, it is a significant performance bottleneck. Reflection is orders of magnitude slower than direct object instantiation. Spawning many entities (e.g., bullets, particles, enemies in a wave) would severely impact performance.
* **Solution:** Create a two-stage loading process.
  1. **Loading/Baking:** During game startup, ArchetypeManager should still load and parse the JSON files. However, instead of storing the raw JSON, it should use that data to create "template" or "factory" objects for each archetype. These factories would know how to create the entity and its components using direct new Component() calls, not reflection.
  2. **Spawning:** The Spawner will then ask the ArchetypeManager for an archetype's factory/template and simply Clone() or Create() a new entity from it. This is extremely fast.
* **Impact:** **(High)**. Drastically improves entity creation performance, which is critical for any game where objects are created and destroyed during gameplay.

### **Tier 2: Significant Performance & Design Improvements**

*These items address major performance bottlenecks and design patterns that can be significantly improved.*

**3. Optimize Rendering with RenderTarget2D**

* **Problem:** MapRenderer and TerminalRenderer appear to be redrawing a large number of static elements every single frame.
  + MapRenderer generates a list of GridElements for the entire visible map grid and draws them one by one, every frame. The underlying terrain doesn't change unless the player moves to a new world tile.
  + TerminalRenderer redraws all visible lines of text, including complex color parsing and wrapping, every frame. Most of the terminal history is static.
* **Solution:** Use RenderTarget2D as a cache.
  + **For MapRenderer:** Create a RenderTarget2D for the terrain. When the player enters a new world tile (or the map view changes), draw the static terrain (water, hills, etc.) to this render target *once*. Then, in your main Draw loop, just draw this single texture. Dynamic elements like the player, path, and hover selector can be drawn on top of it.
  + **For TerminalRenderer:** Use a RenderTarget2D for the terminal history. When a new line is added, draw only that new line onto the bottom of the render target (you may need to scroll the existing content up). Then, in the main Draw loop, draw the relevant portion of this render target to the screen.
* **Impact:** **(High)**. This will massively reduce the number of draw calls per frame, significantly boosting rendering performance and lowering CPU usage.

**4. Improve ECS Query Performance**

* **Problem:** The ComponentStore uses a Dictionary<Type, Dictionary<int, IComponent>>. When a system needs to operate on entities with multiple components (e.g., PositionComponent and RenderableComponent), it has to perform multiple lookups or get all entities with one component and then check if they have the other. This can be inefficient for systems that run every frame.
* **Solution:** Implement a "View" or "Query" system.
  + When a system is initialized, it can register a query with the ECS for a specific combination of components (e.g., ecs.CreateQuery(typeof(PositionComponent), typeof(RenderableComponent))).
  + The ECS would then maintain a cached List<int> of all entity IDs that match this query. The list is updated only when components are added or removed from entities.
  + The system can then iterate over this pre-filtered, cached list, which is much faster than repeated dictionary lookups.
* **Impact:** **(Medium-High)**. Speeds up the core logic loop of your game, especially as the number of entities and systems grows.

**5. Formalize an Event-Driven Architecture**

* **Problem:** Systems communicate by directly calling methods on each other via the Core god object (e.g., Core.CurrentTerminalRenderer.AddOutputToHistory). This is a direct result of the tight coupling mentioned in Tier 1.
* **Solution:** Create a simple, centralized event bus or use C# events more formally.
  + Instead of calling a method, a system should raise an event. For example, ActionExecutionSystem would raise an ActionCompleted(action) event. CombatResolutionSystem would raise a DamageDealt(attacker, target, damage) event.
  + Other systems, like TerminalRenderer, CombatLogPanel, or a future SoundManager, would subscribe to these events and react accordingly. The TerminalRenderer would listen for ActionCompleted and format a message. The SoundManager would listen for DamageDealt and play a hit sound.
* **Impact:** **(High)**. This is the other half of solving the God Object problem. It fully decouples systems, making them independent and reusable modules.

### **Tier 3: Code Quality & Maintainability**

*These changes will make the code cleaner, easier to understand, and less prone to bugs.*

**6. Refactor GameState**

* **Problem:** GameState.cs is becoming a secondary God Object, managing player state, world state, combat state, and UI state. This violates the Single Responsibility Principle. The combat-related properties (IsInCombat, Combatants, InitiativeOrder, etc.) do not belong in the main GameState.
* **Solution:** Break GameState into more focused classes.
  + Create a CombatState class to hold all combat-specific data. The main GameState could hold an instance of CombatState which is only non-null when in combat.
  + Player-specific data like PlayerEntityId and PendingActions should ideally be accessed through a dedicated PlayerManager or directly via the player's components, rather than being top-level properties on GameState.
* **Impact:** **(Medium)**. Improves code organization and makes it clearer where to find and modify specific parts of the game's state.

**7. Refine Input Handling Logic**

* **Problem:** The input handling is spread across InputHandler, MapInputHandler, and PlayerInputSystem. The logic in PlayerInputSystem for modifying the action queue by converting a Queue<T> to a List<T> and back is highly inefficient.
* **Solution:**
  + Clarify the responsibilities: InputHandler for raw device state, MapInputHandler for translating clicks to world actions, and PlayerInputSystem for processing those actions into the queue.
  + Change the player's action queue from Queue<IAction> to List<IAction>. A List provides the flexibility needed for appending, removing, and inserting actions, which you are already trying to do. A Queue is the wrong data structure for a path that needs to be modified.
* **Impact:** **(Medium)**. Simplifies input logic and fixes a significant inefficiency in action queue management.

**8. Consolidate and Abstract UI Components**

* **Problem:** You have many UI panel classes (CombatLogPanel, ActionMenuPanel, TargetInfoPanel, etc.) and UI element classes (Button, Slider, Dialog). The panels have repeated code for drawing borders and backgrounds. The Button class has a Function string property which is used in a large switch statement in MapInputHandler - this is brittle.
* **Solution:**
  + Create a base UIPanel class that handles common functionality like drawing a frame, border, and title. The specific panels can then inherit from this.
  + Instead of a Function string, have the MapInputHandler (or a new UIManager) directly subscribe to the OnClick event of each button with a specific lambda or method. This is type-safe and less error-prone. \_buttonMap.TryGetValue("go", out Button goButton) is a good start, but wiring the event is better.
* **Impact:** **(Medium)**. Reduces code duplication in UI classes and makes the UI interaction logic more robust.

### **Tier 4: Future-Proofing & Best Practices**

*These are good practices that will set you up for future success.*

**9. Use the MonoGame Content Pipeline for Archetypes**

* **Problem:** JSON files are being loaded directly from the Content directory using File.ReadAllText. This won't work on all platforms (like consoles or mobile) and is less efficient than using pre-processed binary assets.
* **Solution:** Create a custom Content Pipeline processor for your .json archetype files. The processor would read the JSON during the build process and write it out as a compiled .xnb binary asset. Your ArchetypeManager would then load it using Content.Load<Archetype>. This integrates seamlessly with the "baking" process from Tier 1.
* **Impact:** **(Low now, High later)**. Ensures cross-platform compatibility and improves loading times.

**10. Introduce Unit Testing**

* **Problem:** There are no tests. It's impossible to verify that complex logic like the Pathfinder, WorldClockManager time calculations, or StatsComponent secondary stat calculations are correct without running the entire game.
* **Solution:** Once DI is implemented (Tier 1), systems and logic classes will be testable. Create a separate test project and start writing unit tests for your pure logic classes.
* **Impact:** **(Medium)**. Dramatically increases code quality and confidence, preventing regressions as you add new features.

### **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 : The Combat Data Layer & Turn Foundation**

**Objective:**Establish the foundational data structures for your new turn-based combat model. This involves creating components for health, combat stats (including the new "Action Points" resource), and a list of available attacks. This step lays the groundwork for the "time budget" system.

**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 an integer property for AttackPower and float properties for AttackRange and AggroRange.
   * **CombatStatsComponent.cs:** Implements IComponent. This will hold the dynamic stats for a single combat turn. It should have one integer property: ActionPoints. This is the "currency" an entity spends to perform attacks during a turn.
   * **AvailableAttacksComponent.cs:** Implements IComponent. This component will hold a list of attacks an entity can perform. For now, create a simple inner class or struct named CombatAttack with properties for Name (string), DamageMultiplier (float), and ActionPointCost (int). The component itself should have a List<CombatAttack> property.
2. **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.
3. **Update Archetypes:**
   * **Modify Content/Archetypes/player.json:** Add HealthComponent, CombatantComponent, CombatStatsComponent, and AvailableAttacksComponent. In the AvailableAttacksComponent, give the player a basic "Punch" attack with a low action point cost.
   * **Rename Content/Archetypes/wanderer\_npc.json to bandit.json**.
   * **Modify Content/Archetypes/bandit.json:** Add the same new combat components. Make its stats weaker than the player's and give it a basic "Punch" attack.

**Files to Create:**

* HealthComponent.cs
* CombatantComponent.cs
* CombatStatsComponent.cs
* AvailableAttacksComponent.cs
* CombatLog.cs

**Files to Refactor:**

* Content/Archetypes/player.json
* Content/Archetypes/wanderer\_npc.json (rename to bandit.json and modify)

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

* IComponent.cs
* StatsComponent.cs
* The JSON archetype files.

### **Step 15: Combat State, Turn Management, & Scene Transitions**

**Objective:**Define the "combat state" and create the main CombatTurnSystem that will drive the entire flow of combat. This system will be responsible for managing the 10-second turn timer, granting Action Points, and triggering the resolution of all chosen attacks.

**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 public constant integer COMBAT\_TURN\_DURATION\_SECONDS and set it to 10.
   * Add methods InitiateCombat, AddEntityToCombat, and EndCombat as described in the previous plan. Their role is to manage the IsInCombat flag and the Combatants list.
2. **Refactor AISystem.cs to Trigger Combat:**
   * In the Update method, within the Chasing state logic, change the behavior. If the player is within the AI's AttackRange, it should call GameState.InitiateCombat. This is the trigger that starts a fight.
3. **Refactor Core.cs to Handle Scene Transition:**
   * In the Core.Update method, before the scene manager's update call, add the scene state management logic. This will check the IsInCombat flag and switch to the CombatScene when combat begins, and back to the TerminalMapScene when it ends.
4. **Create the Combat Turn Clock System:**
   * **Create CombatTurnSystem.cs:** Implements ISystem. This is the master clock for combat.
   * It should have a private float \_turnTimer.
   * Its Update method should only run if GameState.IsInCombat is true.
   * Inside Update, it should increment \_turnTimer by gameTime.ElapsedGameTime.TotalSeconds.
   * When \_turnTimer exceeds GameState.COMBAT\_TURN\_DURATION\_SECONDS:
     1. It should log that the turn is resolving.
     2. **(Crucially, it will call another system to do the attack logic in the next step. For now, just add a comment placeholder here).**
     3. It should call Core.CurrentWorldClockManager.PassTime(seconds: GameState.COMBAT\_TURN\_DURATION\_SECONDS).
     4. It should iterate through all entities in GameState.Combatants. For each one, it should get their CombatStatsComponent and StatsComponent and grant them new ActionPoints for the next turn (e.g., 100 + (stats.Agility \* 10)).
     5. Finally, it should reset \_turnTimer by subtracting the turn duration.
5. **Register New System in Core.cs:**
   * Register the new CombatTurnSystem with the SystemManager.

**Files to Create:**

* CombatTurnSystem.cs

**Files to Refactor:**

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

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

* GameState.cs
* AISystem.cs
* Core.cs
* SceneManager.cs
* CombatScene.cs
* TerminalMapScene.cs
* CombatLog.cs
* StatsComponent.cs
* CombatStatsComponent.cs
* WorldClockManager.cs

### **Step 16: Choosing Attacks & Combat Resolution**

**Objective:**Implement the systems for choosing and resolving attacks. This involves creating a component to hold an entity's chosen attack for the turn, a system to process all chosen attacks when the turn ends, and updating the AI to make tactical decisions.

**Detailed Instructions:**

1. **Create a Component for Chosen Actions:**
   * **Create ChosenAttackComponent.cs:** Implements IComponent. This is a temporary component that holds the attack an entity has decided to make this turn. It should have properties for the TargetId (int) and the AttackName (string, matching a name from the AvailableAttacksComponent).
2. **Create the Combat Resolution System:**
   * **Create CombatResolutionSystem.cs:** Implements ISystem. This system does the actual fighting.
   * Its Update method should do nothing on its own. Instead, create a public method ResolveTurn().
   * Inside ResolveTurn, it should query the ComponentStore for all entities that have a ChosenAttackComponent.
   * For each entity, it should:
     1. Get the attacker's CombatantComponent and the target's HealthComponent.
     2. Find the specific attack details from the attacker's AvailableAttacksComponent using the AttackName from the ChosenAttackComponent.
     3. Calculate the final damage (e.g., attacker.AttackPower \* attack.DamageMultiplier).
     4. Call TakeDamage on the target's health component.
     5. Log the result to the CombatLog.
     6. After processing the attack, it should remove the ChosenAttackComponent from the attacker.
3. **Refactor CombatTurnSystem.cs:**
   * In the Update method, where the placeholder comment was, now call Core.CombatResolutionSystem.ResolveTurn(). You will need to add a static instance of CombatResolutionSystem to Core.cs to make it accessible.
4. **Refactor AISystem.cs to Make Combat Decisions:**
   * In the Update method, add a new logic block that runs if an entity is in the GameState.Combatants list.
   * Inside this block, the AI should decide what to do. The logic should be:
     1. If the AI already has a ChosenAttackComponent, do nothing (it has already decided).
     2. Get its CombatStatsComponent to see its available ActionPoints.
     3. Get its AvailableAttacksComponent to see its attack options.
     4. Iterate through its available attacks and pick one it can afford (i.e., ActionPoints >= ActionPointCost).
     5. If it picks an attack, add a ChosenAttackComponent to itself, targeting the player. Then, subtract the ActionPointCost from its ActionPoints in its CombatStatsComponent.
5. **Register New Systems in Core.cs:**
   * Add a static instance of CombatResolutionSystem. It does not need to be registered with the SystemManager's update loop.
   * Register the CombatSystem (from the original plan, which you can create now or rename CombatResolutionSystem to CombatSystem to keep it simple). Its job is simply to check for combat end conditions (player dead or no enemies left).

**Files to Create:**

* ChosenAttackComponent.cs
* CombatResolutionSystem.cs (or CombatSystem.cs)

**Files to Refactor:**

* CombatTurnSystem.cs
* AISystem.cs
* Core.cs

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

* CombatTurnSystem.cs
* AISystem.cs
* Core.cs
* GameState.cs
* ComponentStore.cs
* CombatStatsComponent.cs
* AvailableAttacksComponent.cs
* HealthComponent.cs

### **Step 17: Unify Scenes and Remove Combat Transition**

**Objective:**Dismantle the separate CombatScene and remove the logic that transitions the player away from the main map view. This step prepares the TerminalMapScene to become the sole view for both exploration and combat.

**Detailed Instructions:**

1. **Refactor Core.cs:**
   * In the Update method, locate and remove the entire if/else if block that checks CurrentGameState.IsInCombat to switch between CombatScene and TerminalMapScene. The game should no longer change scenes when combat starts or ends.
2. **Refactor AISystem.cs:**
   * In the DecideNextAction method, the call to gameState.InitiateCombat(...) is correct, but it will no longer trigger a scene change. This is the desired behavior. No code change is needed here, but it's important to understand the effect of the Core.cs change.
3. **Delete Obsolete Files:**
   * The CombatScene.cs file is now redundant and should be deleted from the project.

**Files to Create:**

* None.

**Files to Refactor:**

* Core.cs

**Files to Delete:**

* CombatScene.cs

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

* Core.cs
* AISystem.cs
* GameState.cs
* CombatScene.cs

### **Step 18: Establish Combat UI State & Data**

**Objective:**Create the foundational data structures and state management for the new in-place combat UI. This involves defining the different UI modes and adding properties to GameState to track the player's interactions during a fight.

**Detailed Instructions:**

1. **Create CombatUIState.cs:**
   * This enum will manage which menu or interaction mode is currently active for the player.
   * The code should be: public enum CombatUIState { Default, SelectAttack, SelectSkill, SelectMove, SelectTarget, Busy }
     + Default: The main action menu is shown (Attack, Skills, Move, etc.).
     + SelectAttack: The player is choosing from a list of available attacks.
     + SelectSkill: The player is choosing from a list of available skills.
     + SelectMove: The player is choosing a tile on the local map to move to.
     + SelectTarget: The player is choosing an enemy to target with an action.
     + Busy: Player input is disabled (e.g., during an enemy's turn or an animation).
2. **Refactor GameState.cs:**
   * Add properties to track the UI's current state and the player's selections.
   * **Add Properties:**
     + public CombatUIState UIState { get; set; } = CombatUIState.Default;
     + public int? SelectedTargetId { get; set; } = null; (A nullable int to store the ID of the enemy the player is interacting with).
   * **Modify Combat Initiation:**
     + In the InitiateCombat method, after setting IsInCombat = true, add a line to lock the map view: CurrentMapView = MapView.Local;
   * **Modify Combat End:**
     + In the EndCombat method, add a line to reset the UI state: UIState = CombatUIState.Default;

**Files to Create:**

* CombatUIState.cs

**Files to Refactor:**

* GameState.cs

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

* GameState.cs

### **Step 19: Create the Turn Order Display**

**Objective:**Create the UI panel that shows the turn order (initiative list). This panel will display the names of all combatants, handle duplicate names, and highlight whose turn it is.

**Detailed Instructions:**

1. **Create TurnOrderPanel.cs:**
   * This class will be responsible for drawing the initiative list.
   * It should have a Draw(SpriteBatch spriteBatch) method.
   * **Draw Method Logic:**
     + Check if GameState.IsInCombat is true. If not, do nothing.
     + Get the InitiativeOrder list from GameState.
     + Create a temporary dictionary to store the display names (Dictionary<int, string>).
     + Create another dictionary to count occurrences of archetype names (Dictionary<string, int>).
     + Iterate through the InitiativeOrder list to generate unique names:
       - For each entityId, get its RenderableComponent and find its archetype Name from the ArchetypeManager.
       - If the archetype name is already in the count dictionary, increment the count and append the number to the name (e.g., "Bandit 2"). Otherwise, add it with a count of 1 (e.g., "Bandit 1").
       - Store the generated unique name in the display name dictionary.
     + Iterate through the InitiativeOrder list again to draw the names:
       - Use the generated unique names from your dictionary.
       - Use Color.Yellow for the player's name.
       - Use Color.White or Color.LightGray for all other combatants' names.
       - Check if the current entityId in the loop matches GameState.CurrentTurnEntityId. If it does, draw a highlighted box (e.g., a red rectangle) around that entity's name.
2. **Refactor TerminalMapScene.cs:**
   * Add a private field: private TurnOrderPanel \_turnOrderPanel;
   * Instantiate it in the constructor: \_turnOrderPanel = new TurnOrderPanel();
   * In the Draw method, inside the SpriteBatch.Begin() call, invoke \_turnOrderPanel.Draw(spriteBatch);.

**Files to Create:**

* TurnOrderPanel.cs

**Files to Refactor:**

* TerminalMapScene.cs

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

* TerminalMapScene.cs
* GameState.cs
* ArchetypeManager.cs
* ComponentStore.cs
* RenderableComponent.cs

### **Step 20: Create Combat UI Panels (Player Status, Target Status, Action Menu)**

**Objective:**Create the three core UI panels that will display information and provide interaction points during combat. This step focuses only on the visual layout and state-driven rendering.

**Detailed Instructions:**

1. **Create PlayerStatusPanel.cs:**
   * This class will draw the player's combat information.
   * It needs a Draw(SpriteBatch spriteBatch) method that only runs if GameState.IsInCombat is true.
   * Inside Draw, get the player's HealthComponent and CombatStatsComponent to display their current/max health, action points, etc., in a designated area (e.g., bottom-left).
2. **Create TargetInfoPanel.cs:**
   * This class will display information about the currently selected enemy.
   * It needs a Draw(SpriteBatch spriteBatch) method.
   * Inside Draw, check if GameState.SelectedTargetId has a value.
   * If it does, get the target's HealthComponent and archetype Name. Display the name and a health bar in a designated area (e.g., below the truncated terminal).
3. **Create ActionMenuPanel.cs:**
   * This is the most complex panel. It will handle drawing the interactive menus.
   * It needs a Draw(SpriteBatch spriteBatch) method that only runs if GameState.IsInCombat is true.
   * The Draw method must check GameState.UIState:
     + If UIState is Default, draw the main action buttons: "Attack", "Skills", "Move", "Item", "End Turn".
     + If UIState is SelectAttack, get the player's AvailableAttacksComponent and draw a list of their attacks. Visually distinguish affordable attacks from unaffordable ones (e.g., by color).
     + If UIState is SelectSkill, draw a placeholder list: "Block", "Power Strike".
     + If UIState is SelectTarget, draw the text "Select a target..." over the menu area.
     + If UIState is SelectMove, draw the text "Select a destination..." over the menu area.
   * It should also have a placeholder method for the next step: public string HandleInput(Point mousePosition) { return null; }. This will later be used to detect which button is clicked.

**Files to Create:**

* PlayerStatusPanel.cs
* TargetInfoPanel.cs
* ActionMenuPanel.cs

**Files to Refactor:**

* None.

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

* GameState.cs
* HealthComponent.cs
* CombatStatsComponent.cs
* AvailableAttacksComponent.cs
* ComponentStore.cs
* CombatUIState.cs

### **Step 21: Integrate UI Panels and Refactor Terminal**

**Objective:**Integrate the newly created UI panels into the main game scene and modify the existing TerminalRenderer to accommodate the new combat UI layout.

**Detailed Instructions:**

1. **Refactor TerminalRenderer.cs:**
   * Modify the DrawTerminal method.
   * Add a check at the beginning: bool isInCombat = Core.CurrentGameState.IsInCombat;.
   * Change the terminalHeight calculation. If isInCombat is true, set the height to roughly half of Global.DEFAULT\_TERMINAL\_HEIGHT. If false, use the full height.
   * Hide the terminal input line entirely when isInCombat is true. This includes the > prompt, the input text, and the carat. The status text ("Actions Queued:", etc.) should also be hidden.
2. **Refactor TerminalMapScene.cs:**
   * Add private fields for the new panels:
     + private PlayerStatusPanel \_playerStatusPanel;
     + private TargetInfoPanel \_targetInfoPanel;
     + private ActionMenuPanel \_actionMenuPanel;
   * Instantiate them in the constructor.
   * In the Draw method, inside the SpriteBatch.Begin() call, invoke the Draw method for each of the new panels.

**Files to Create:**

* None.

**Files to Refactor:**

* TerminalRenderer.cs
* TerminalMapScene.cs

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

* TerminalRenderer.cs
* TerminalMapScene.cs
* GameState.cs
* PlayerStatusPanel.cs
* TargetInfoPanel.cs
* ActionMenuPanel.cs

### **Step 22: Implement In-Combat Map Visuals**

**Objective:**Modify the MapRenderer to display crucial combat information directly on the local map grid, such as identifying combatants, the selected target, and the entity whose turn it is.

**Detailed Instructions:**

1. **Refactor MapRenderer.cs:**
   * Locate the DrawLocalMap method (or the local map drawing logic within DrawMap).
   * After drawing the base grid and entities, add a new drawing section that only runs if GameState.IsInCombat is true.
   * **Inside this new section:**
     + Iterate through all entityId in GameState.Combatants.
     + For each combatant, get its LocalPositionComponent.
     + Use MapCoordsToScreen to find its position on the screen.
     + **Draw Bounding Box:**
       - If the combatant's ID is the GameState.SelectedTargetId, draw a pink bounding box around its grid cell.
       - Otherwise, draw a teal bounding box around its grid cell.
     + **Draw Turn Indicator:**
       - Check if the combatant's ID is the GameState.CurrentTurnEntityId.
       - If it is, draw an 8x8 teal pixel (or a small sprite) just above its grid cell to indicate it's their turn.

**Files to Create:**

* None.

**Files to Refactor:**

* MapRenderer.cs

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

* MapRenderer.cs
* GameState.cs
* LocalPositionComponent.cs
* ComponentStore.cs

### **Step 23: Implement Combat Interactivity**

**Objective:**Bring the combat UI to life by creating an input system that allows the player to select targets on the map, interact with the action menu, and execute attacks.

**Detailed Instructions:**

1. **Create PlayerCombatInputSystem.cs:**
   * This class will be responsible for handling all player input *during combat*. It will be called directly from TerminalMapScene.
   * Create a public method: public void ProcessInput().
   * **Inside ProcessInput:**
     + Only run the logic if GameState.IsInCombat is true and GameState.UIState is not Busy.
     + Get the current mouse state and check for a left-click.
     + **Targeting Logic:**
       - Call a new method on MapRenderer, GetEntityIdAt(Point mousePosition), which you will implement next. This method should check if the mouse is over any combatant's grid cell and return that entity's ID.
       - If an ID is returned, set GameState.SelectedTargetId to that value.
     + **Menu Interaction Logic:**
       - If the mouse was clicked, call \_actionMenuPanel.HandleInput(mousePosition).
       - The string returned from HandleInput will be the name of the button/attack clicked.
       - If "Attack" was clicked, change GameState.UIState to SelectAttack.
       - If an attack name was clicked (e.g., "Punch"), change GameState.UIState to SelectTarget.
     + **Executing an Action:**
       - Add a check: if GameState.UIState is SelectTarget, the mouse was clicked, and GameState.SelectedTargetId is not null, then execute the action.
       - To execute: find the chosen attack, check for sufficient ActionPoints, and if affordable, add a ChosenAttackComponent to the player entity targeting the SelectedTargetId.
       - Finally, set GameState.UIState back to Default or Busy.
2. **Refactor MapRenderer.cs:**
   * Implement the public int? GetEntityIdAt(Point mousePosition) method. It should iterate through GameState.Combatants, convert their LocalPosition to screen coordinates, and check if the mouse position is within the bounds of their grid cell.
3. **Refactor ActionMenuPanel.cs:**
   * Fully implement the HandleInput method. It should define Rectangles for each button it draws. When called, it checks if the mousePosition is within any of these rectangles and returns the corresponding button's text (e.g., "Attack", "Punch").
4. **Refactor TerminalMapScene.cs:**
   * Instantiate the new PlayerCombatInputSystem.
   * In the Update method, call \_playerCombatInputSystem.ProcessInput().

**Files to Create:**

* PlayerCombatInputSystem.cs

**Files to Refactor:**

* MapRenderer.cs
* ActionMenuPanel.cs
* TerminalMapScene.cs

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

* TerminalMapScene.cs
* MapRenderer.cs
* ActionMenuPanel.cs
* GameState.cs
* ComponentStore.cs
* CombatStatsComponent.cs
* ChosenAttackComponent.cs
* AvailableAttacksComponent.cs

### **Step 24: Fix Player Turn and Path Interruption**

**Objective:**Resolve two critical logic issues. First, ensure the player can take actions on their turn. Second, make sure any ongoing player movement is immediately cancelled when combat begins.

**Detailed Instructions:**

1. **Refactor PlayerCombatInputSystem.cs:**
   * Locate the ProcessInput method.
   * Modify the initial condition that checks if the system should run. It should now check if it is the **player's turn** in combat.
   * The new condition should be: if (Core.CurrentGameState.IsInCombat && Core.CurrentGameState.CurrentTurnEntityId == Core.CurrentGameState.PlayerEntityId && Core.CurrentGameState.UIState != CombatUIState.Busy)
2. **Refactor GameState.cs:**
   * Locate the InitiateCombat method.
   * This method is called when an AI's aggro range is triggered. We need to interrupt any actions the player was performing.
   * At the very beginning of the InitiateCombat method, before any other logic, add a call to cancel the player's current action queue: CancelExecutingActions(true);

**Files to Create:**

* None.

**Files to Refactor:**

* PlayerCombatInputSystem.cs
* GameState.cs

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

* PlayerCombatInputSystem.cs
* GameState.cs

### **Step 25: Implement Target Information Panel & Hover Tooltip**

**Objective:**Create a dedicated UI panel to display information about the selected target and enhance the map hover tooltip to show the same data for any visible combatant.

**Detailed Instructions:**

1. **Refactor TargetInfoPanel.cs:**
   * This panel will now display more detailed information.
   * Modify the Draw method.
   * If GameState.SelectedTargetId has a value, get the following components for the target entity: HealthComponent, LocalPositionComponent, and RenderableComponent.
   * Get the player's LocalPositionComponent.
   * Calculate the distance between the player and the target using Vector2.Distance().
   * Use the ArchetypeManager to get the target's display Name from its archetype.
   * Draw the target's Name, Health Bar (Current/Max), and Distance (e.g., "Distance: 5.2m").
2. **Refactor MapRenderer.cs:**
   * Locate the UpdateHover method (or the main Update method where hover logic resides).
   * Enhance the tooltip logic for the local map view.
   * When \_hoveredGridPos has a value, use a new method GetEntityIdAt(\_hoveredGridPos.Value) to see if an entity is on that tile.
   * If an entity is found and the game is in combat (GameState.IsInCombat), build a multi-line tooltip string containing the entity's Name, Health, and Distance from the player, mirroring the logic from TargetInfoPanel.
   * Request the tooltip using Core.CurrentTooltipManager.RequestTooltip().
3. **Refactor TerminalMapScene.cs:**
   * Ensure the TargetInfoPanel is being instantiated and its Draw method is being called in the main Draw loop. This should already be in place from previous steps, but verify it.

**Files to Create:**

* None.

**Files to Refactor:**

* TargetInfoPanel.cs
* MapRenderer.cs

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

* TargetInfoPanel.cs
* MapRenderer.cs
* GameState.cs
* HealthComponent.cs
* LocalPositionComponent.cs
* RenderableComponent.cs
* ComponentStore.cs
* ArchetypeManager.cs
* TooltipManager.cs

### **Step 26: Centralize Entity Naming and Enhance Combat Logging**

**Objective:**Create a centralized helper method to get a unique, user-friendly display name for any entity. Use this method to improve all combat logs, ensuring entities are referred to as "Player" or "Bandit 2" instead of "Entity 1". Also, add logging for key combat events like death.

**Detailed Instructions:**

1. **Create a Naming Utility:**
   * It is recommended to add a new static helper class, e.g., EntityNamer.cs, or add a static method to an existing utility class.
   * Create a public static method: public static string GetName(int entityId).
   * Inside this method, get the entity's archetype Name. If the entity is the player, return "Player". Otherwise, return the archetype name.
   * Create another public static method: public static Dictionary<int, string> GetUniqueNames(IEnumerable<int> entityIds).
     + This method will take a list of entity IDs (like the InitiativeOrder).
     + It will perform the name-counting logic (e.g., "Bandit", "Bandit 2") and return a dictionary mapping each entityId to its unique name for that context.
2. **Refactor TurnOrderPanel.cs:**
   * Modify the Draw method to use the new EntityNamer.GetUniqueNames(gameState.InitiativeOrder) method to get the display names, simplifying its internal logic.
3. **Refactor CombatResolutionSystem.cs:**
   * In the ResolveTurn method, replace the manual name-fetching logic with calls to EntityNamer.GetName() for both the attacker and the target.
   * After calling targetHealthComp.TakeDamage(damage), add a check: if (targetHealthComp.CurrentHealth <= 0).
   * If the target is dead, log it to the CombatLog (e.g., CombatLog.Log($"[red]{targetName} has been defeated![/]");).
4. **Refactor AISystem.cs:**
   * In the ProcessCombatTurn method, use EntityNamer.GetName() when logging the AI's chosen action or lack thereof.
5. **Refactor CombatTurnSystem.cs:**
   * In the EndCurrentTurn method, use EntityNamer.GetName() to log whose turn it is now.
6. **Refactor GameState.cs:**
   * In the InitiateCombat method, use EntityNamer.GetUniqueNames() to generate the names for the initial initiative order log message.

**Files to Create:**

* EntityNamer.cs (Recommended)

**Files to Refactor:**

* TurnOrderPanel.cs
* CombatResolutionSystem.cs
* AISystem.cs
* CombatTurnSystem.cs
* GameState.cs

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

* TurnOrderPanel.cs
* CombatResolutionSystem.cs
* AISystem.cs
* CombatTurnSystem.cs
* GameState.cs
* ArchetypeManager.cs
* ComponentStore.cs
* RenderableComponent.cs
* HealthComponent.cs

### **Step 27: Implement AI Combat Decision-Making**

**Objective:**Overhaul the AISystem to give NPCs intelligent combat behavior. The AI will now evaluate its position relative to the player, decide whether to move or attack, and execute its chosen action.

**Detailed Instructions:**

1. **Create a Move Action for AI:**
   * The MoveAction class already exists. We need a system that can process it for any entity, not just the player.
   * **Refactor ActionExecutionSystem.cs:**
     + The Update method currently only processes the player's actions. This system needs to be generalized.
     + The Update loop should now iterate over all GameState.ActiveEntities.
     + For each entity, it should perform the three-phase cycle (Initiation, Processing, Cleanup) for MoveAction and RestAction components, just as it does for the player.
     + **Crucially**, the ApplyMoveActionEffects method needs to be updated. It currently assumes the move is for the player and resets local position on world moves. It must now handle moves for any entity. For now, since combat is locked to the local map, you only need to ensure the local position update works for any entityId.
2. **Refactor AISystem.cs:**
   * This is a complete rewrite of the ProcessCombatTurn method.
   * **New ProcessCombatTurn Logic:**
     + Get the AI's components: LocalPositionComponent, CombatStatsComponent, AvailableAttacksComponent, CombatantComponent.
     + Get the player's LocalPositionComponent.
     + Calculate the distance to the player.
     + **Decision Tree:**
       - **1. Attack or Move?** Check if the distance to the player is within the AI's AttackRange from its CombatantComponent.
       - **2. If in range (Attack Logic):**
         * Find the best attack it can afford from its AvailableAttacksComponent (e.g., the one with the highest damage that costs less than or equal to its current ActionPoints).
         * If an affordable attack is found, add a ChosenAttackComponent to the AI entity, targeting the player. Subtract the ActionPointCost. Log the choice.
         * If no affordable attack is found, the AI does nothing this turn. Log this.
       - **3. If not in range (Move Logic):**
         * Use the Pathfinder to find a path from the AI's current LocalPosition to a tile adjacent to the player's LocalPosition.
         * If a path is found, get the first step (the next tile to move to).
         * Check if the AI has enough ActionPoints to move (for now, assume a move costs a fixed amount, e.g., 5 AP).
         * If it can afford to move, enqueue a MoveAction to the next tile in its ActionQueueComponent. Subtract the move cost from its ActionPoints. Log the move.
         * If no path is found or it can't afford to move, the AI does nothing. Log this.
     + After making a decision (or deciding to do nothing), call Core.CombatTurnSystem.EndCurrentTurn() to pass control.
3. **Refactor CombatTurnSystem.cs:**
   * The EndCurrentTurn method should now have a small delay or be triggered by the AI system after a short timer to create a "realistic pace". For now, a simple approach is fine: the call to EndCurrentTurn at the end of the AI's logic is sufficient. The "pace" can be refined later. The key is that the AI makes one decision per turn.

**Files to Create:**

* None.

**Files to Refactor:**

* AISystem.cs
* ActionExecutionSystem.cs
* CombatTurnSystem.cs

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

* AISystem.cs
* ActionExecutionSystem.cs
* CombatTurnSystem.cs
* GameState.cs
* ComponentStore.cs
* LocalPositionComponent.cs
* CombatantComponent.cs
* CombatStatsComponent.cs
* AvailableAttacksComponent.cs
* ActionQueueComponent.cs
* MoveAction.cs
* Pathfinder.cs
* EntityNamer.cs (if created in the previous step)

**Step 28: Create the Combat Action System**

**Objective:**Create the new, dedicated system for processing *only* tactical, in-combat actions. This system will be the foundation for unifying all combat actions and will initially take over the AI's movement logic.

**Detailed Instructions:**

1. **Create CombatActionSystem.cs:**
   * This new class must implement the ISystem interface.
   * Its Update method will be the core of its logic.
   * Inside Update, iterate through all entities in GameState.Combatants.
   * For each combatant, check if they have a MoveAction in their ActionQueueComponent.
   * If a MoveAction exists, dequeue it and call a *new, private* method within this class called ApplyMoveActionEffects(int entityId, MoveAction action). This method's logic should be copied directly from the existing ApplyMoveActionEffects in ActionExecutionSystem.cs, but simplified to *only* handle local map position changes.
2. **Refactor ActionExecutionSystem.cs:**
   * Locate the Update method.
   * Remove the entire if (gameState.IsInCombat) block. This system will no longer be aware of combat. Its sole purpose is now processing the player's out-of-combat action queue.
3. **Refactor Core.cs:**
   * Add a new static, readonly instance of the CombatActionSystem.
   * In the Update method, add a new conditional block. If CurrentGameState.IsInCombat is true, call the Update method of your new \_combatActionSystem instance. The existing block that updates the \_systemManager should be in an else clause, so it only runs when *not* in combat.

**Files to Create:**

* CombatActionSystem.cs

**Files to Refactor:**

* ActionExecutionSystem.cs
* Core.cs

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

* ActionExecutionSystem.cs
* Core.cs
* GameState.cs
* ComponentStore.cs
* ActionQueueComponent.cs
* MoveAction.cs
* LocalPositionComponent.cs
* ISystem.cs

**Step 29: Decouple AI Decision from Execution**

**Objective:**Modify the AI to only *decide* on an action during its turn, adding the action as a component. The AI will no longer execute the action itself. This is a critical step in decoupling decision-making from action-processing.

**Detailed Instructions:**

1. **Refactor AISystem.cs:**
   * Locate the ProcessCombatTurn method.
   * If the AI decides to attack, it should add a ChosenAttackComponent to itself and then immediately call Core.CombatTurnSystem.EndCurrentTurn(). **Remove the call to Core.CombatResolutionSystem.ResolveAction()**.
   * If the AI decides to move, it should add a MoveAction to its ActionQueueComponent and then immediately call Core.CombatTurnSystem.EndCurrentTurn(). **The action will not be processed here.**
   * The AI's turn is now simply the act of adding a component that represents its intent.

**Files to Create:**

* None

**Files to Refactor:**

* AISystem.cs

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

* AISystem.cs
* CombatTurnSystem.cs
* CombatResolutionSystem.cs
* ActionQueueComponent.cs
* ChosenAttackComponent.cs
* MoveAction.cs

**Step 30: Decouple Player Decision from Execution**

**Objective:**Modify the player's input system to follow the same "decide, don't execute" pattern as the AI. When the player clicks to attack, it will now simply register their intent.

**Detailed Instructions:**

1. **Refactor PlayerCombatInputSystem.cs:**
   * Locate the ExecutePlayerAttack method.
   * When the player successfully chooses a valid target for an attack, it should add a ChosenAttackComponent to the player entity.
   * After adding the component, it should immediately call Core.CombatTurnSystem.EndCurrentTurn().
   * **Remove the call to Core.CombatResolutionSystem.ResolveAction()**. The input system is no longer responsible for resolving the action.

**Files to Create:**

* None

**Files to Refactor:**

* PlayerCombatInputSystem.cs

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

* PlayerCombatInputSystem.cs
* CombatTurnSystem.cs
* CombatResolutionSystem.cs
* ChosenAttackComponent.cs

**Step 31: Create the Unified Combat Processing System**

**Objective:**Create the final, central system that processes the chosen actions of the entity whose turn it is. This system will replace the direct calls to the resolution and execution systems, unifying the action pipeline.

**Detailed Instructions:**

1. **Create CombatProcessingSystem.cs:**
   * This new class must implement the ISystem interface.
   * Its Update method will be called by Core only when IsInCombat is true.
   * **Logic for Update method:**
     + Get the CurrentTurnEntityId from GameState.
     + Check if this entity has a ChosenAttackComponent. If so, call Core.CombatResolutionSystem.ResolveAction() with the entity's ID and the component, then **remove the ChosenAttackComponent**.
     + Check if this entity has a MoveAction in its ActionQueueComponent. If so, dequeue the action and process it using the logic from CombatActionSystem (it's best to move the ApplyMoveActionEffects logic into this new system to centralize it).
     + **This system does not end the turn.** The turn is ended by the input/AI systems after they have added their action components.
2. **Refactor CombatActionSystem.cs:**
   * The logic from this system's Update method should be moved into the new CombatProcessingSystem.cs.
   * After moving the logic, the CombatActionSystem.cs file can be deleted.
3. **Refactor Core.cs:**
   * Remove the instance of CombatActionSystem.
   * Add a new static, readonly instance of CombatProcessingSystem.
   * In the Update method, replace the call to \_combatActionSystem.Update() with a call to \_combatProcessingSystem.Update().

**Files to Create:**

* CombatProcessingSystem.cs

**Files to Delete:**

* CombatActionSystem.cs

**Files to Refactor:**

* Core.cs

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

* Core.cs
* CombatActionSystem.cs
* CombatResolutionSystem.cs
* GameState.cs
* ComponentStore.cs
* ChosenAttackComponent.cs
* ActionQueueComponent.cs
* MoveAction.cs
* ISystem.cs

**Step 32: Finalizing the Turn Flow**

**Objective:**Ensure the turn ends correctly *after* an action has been processed. The current flow has the turn ending when an action is *chosen*, not when it's *resolved*. This step corrects the sequence.

**Detailed Instructions:**

1. **Refactor CombatProcessingSystem.cs:**
   * At the end of the Update method, after checking for and processing either a ChosenAttackComponent or a MoveAction, add a final check. If an action was processed for the CurrentTurnEntityId, call Core.CombatTurnSystem.EndCurrentTurn(). This ensures the turn only advances after the action is complete.
2. **Refactor PlayerCombatInputSystem.cs:**
   * In ExecutePlayerAttack, **remove the call to Core.CombatTurnSystem.EndCurrentTurn()**. The player's input system should now only be responsible for adding the ChosenAttackComponent.
3. **Refactor AISystem.cs:**
   * In ProcessCombatTurn, **remove the call to Core.CombatTurnSystem.EndCurrentTurn()**. The AI system is now only responsible for adding the ChosenAttackComponent or MoveAction.

**Files to Create:**

* None

**Files to Refactor:**

* CombatProcessingSystem.cs
* PlayerCombatInputSystem.cs
* AISystem.cs

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

* CombatProcessingSystem.cs
* PlayerCombatInputSystem.cs
* AISystem.cs
* CombatTurnSystem.cs