Below is the updated project requirements document, now including details on how the action execution follows the Command Pattern and how available commands are determined based on the unit's current state.

**Siberian Warfare: Educational Game Project Requirements**

**Overview**

The goal of this project is to build a simple strategy game that serves as a hands-on example of several key programming concepts. The game—named *Siberian Warfare*—allows players to control various game objects (such as infantry units and war factories) and execute actions (such as movement). The design of the game illustrates important design patterns, object-oriented programming (OOP) principles, and concepts like immutability, equality, comparability, and the Command Pattern.

**Core Objectives**

* **Demonstrate Design Patterns:**
  + Use the **Component Pattern** to compose game objects.
  + Apply the **State Pattern** to manage behaviors (e.g., unit movement states).
  + Utilize the **Command Pattern** to encapsulate game actions, ensuring that each action is an independent command object.
* **OOP Principles:**
  + Organize code into objects with clear responsibilities.
  + Emphasize encapsulation, modularity, and separation of concerns.
* **Immutability:**
  + Implement immutable components (for example, via an interface like IImmutableName) to ensure that an object’s identity, once set, cannot be altered.
* **Equality & Comparability:**
  + Define proper equality and, optionally, comparability for game objects to support robust comparisons, sorting, and integrity checks.

**Game Architecture & Components**

**1. Base Game Object (SWGameObject)**

* **Purpose:**
  + Acts as the fundamental entity in the game. All game objects (units, buildings, etc.) either inherit from or are composed of this class.
* **Responsibilities:**
  + Managing a collection of components.
  + Dynamically adding, retrieving, or removing components.

**2. Component System**

* **TransformComponent:**
  + Manages spatial properties such as position, rotation, and scale.
* **UnitTypeComponent:**
  + Stores details about the unit (name, category such as "Infantry" or "Building", and a brief description).
  + Implements immutability via an interface like IImmutableName, ensuring that the unit's identity remains unchanged.
* **MovementStateMachine:**
  + **State-Driven Behavior:**
    - Manages the movement state of a game object (e.g., MOVABLE versus STATIONARY).
    - Uses the **State Pattern** to provide different behaviors based on the unit's current state.
  + **Command Pattern Integration:**
    - Provides a list of available commands (actions) that a unit can perform in its current state via a method like GetAvailableActions().
    - Each command (for instance, MoveAction) encapsulates the details required to perform the action.
    - The available commands are determined dynamically; only those that are valid in the current state of the unit are returned.

**3. Command Pattern Details**

* **Encapsulation of Actions:**
  + Each game action is implemented as a command object. For example, MoveAction encapsulates both the logic to move a unit and the parameters required for the movement.
* **Execution:**
  + When a command is selected (e.g., a move command from the list of available actions), it is executed by passing in the necessary arguments (like the target position).
* **Decoupling Invoker from Receiver:**
  + By using the Command Pattern, the execution of an action is decoupled from the object that triggers it (the player), which allows for flexible action management such as undo/redo operations in a more advanced implementation.

**4. Player Class**

* **Purpose:**
  + Represents a game participant who controls one or more game objects.
* **Key Functions:**
  + **AddUnit / RemoveUnit:**
    - Methods for adding or removing game objects under the player’s control.
  + **Scenario Execution:**
    - Demonstrates a typical gameplay scenario:
      * Searches for a specific type of unit (e.g., an "Infantry" unit) by using its immutable name.
      * Retrieves the list of available commands from the unit’s movement state machine, which are state-dependent.
      * Selects and executes a command (such as a move command with provided movement arguments) following the Command Pattern.

**5. Game State Management (SiberianWarfareGameState)**

* **Purpose:**
  + Oversees the initialization and management of game objects as well as the overall game flow.
* **Responsibilities:**
  + Registering new game objects.
  + Initializing the game scene by creating units (e.g., infantry) and buildings (e.g., war factories), setting up their components (transform, unit type, movement state), and registering them.
  + Instantiating the player, assigning units, and triggering gameplay scenarios that illustrate the interaction between components, states, and commands.

**Design Patterns & Key Concepts**

* **Component Pattern:**
  + Facilitates the assembly of game objects from a set of interchangeable components, promoting flexibility and reusability.
* **State Pattern (MovementStateMachine):**
  + Enables a unit to exhibit different behaviors depending on its current state (e.g., movable vs. stationary), determining the available set of commands.
* **Command Pattern:**
  + Each action a unit can perform (like moving) is encapsulated as a command object. This pattern decouples the action request from the action execution, making it easier to manage actions, log them, or extend functionality (e.g., adding undo features).
* **Immutability:**
  + Using interfaces such as IImmutableName ensures that once a game object’s identifying attributes are set, they cannot be changed.
* **Equality and Comparability:**
  + Properly implemented equality and comparison methods ensure that game objects can be reliably compared and sorted, which is essential for managing collections and ensuring game integrity.

**Implementation Expectations**

Students should:

* **Design and Implement Classes:**
  + Create the necessary classes (SWGameObject, Player, and various components) with clear responsibilities and well-defined interaction protocols.
* **Utilize OOP Techniques:**
  + Emphasize encapsulation, abstraction, inheritance (where appropriate), and polymorphism in the design.
* **Apply Design Patterns:**
  + Incorporate the Component, State, and Command Patterns in the game’s architecture.
* **Ensure Robustness:**
  + Implement immutability and provide appropriate equality and comparability methods to maintain object integrity.
* **Develop a Demo Scenario:**
  + Implement a sample gameplay scenario (as shown in the provided high-level code) where a player retrieves an infantry unit, accesses its state-dependent available commands, and executes a move command using the Command Pattern.

This project is designed to be both educational and practical, providing students with real-world examples of advanced programming techniques and design patterns. It challenges them to think about how objects interact in a game environment while adhering to solid software design principles, including how to manage actions through a robust Command Pattern implementation.

Feel free to modify or extend these requirements to suit specific educational goals or to include additional challenges for your students.