REQ5: Design Rationale (Stateful Creatures: Chimera Elemental Transformations)

A. Representing state and behavior encapsulation

Design A1: State flags with conditional branching. Store the current state as an enum (DEFAULT, FIRE, ICE, POISON) on Chimera; playTurn() contains large switch/if-else blocks that branch on the state flag to determine weapon, behavior, and transitions.

Pros	Cons
Minimal extra classes; state is a simple field.	Violates SRP: Chimera grows bloated with all state-specific logic mixed together.
Easy to see all states in one place.	Hard to extend: adding a new state requires editing the monolithic playTurn() and scattered conditionals.
	DRY violation: state transition checks, weapon selection, and behavior rules duplicated across methods.
	Low cohesion: combat logic, movement patterns, and transition rules entangled.

Design A2: State Pattern with ChimeraState interface. Introduce a ChimeraState interface defining getBehaviorAction(), getStateWeapon(), attemptStateTransition(), createAttackAction(), and onEnterState(). Concrete states (DefaultChimeraState, FireState, IceState, PoisonState) implement the interface and encapsulate all state-specific rules (weapon, behavior, transitions, special effects). Chimera maintains a reference to its current state and delegates behavior calls.

Pros	Cons
High cohesion: each state class owns its behavior, weapon, and transition logic.	Adds several classes (one per state).
OCP: new states can be added without modifying Chimera or existing states.	Requires indirection through the state interface.
SRP: Chimera focuses on state management; states focus on behavior.	

Clear ownership: state transitions, special effects, and timers live in their respective state classes.	
LSP-friendly: all states are interchangeable through the same interface.	

We choose **Design A2** because it provides modularity, extensibility, and clear separation of concerns. Each state class is self-contained, making it trivial to add new elemental forms or modify existing ones without touching the Chimera context or other states, adhering to the Open-Closed Principle.

B. State transition logic and predetermined paths

Design B1: Random state selection. On each turn, roll a die and transition to any state randomly (e.g., 25% chance for each of four states).

Pros	Cons
Maximum variety; unpredictable behavior.	Violates the requirement: transitions must be predetermined, not random.
Simple probability check.	No strategic progression; incoherent elemental narrative (fire → fire repeatedly).
	Hard to balance difficulty or test reliably.

Design B2: Predetermined state graph with probabilistic timing. Define a fixed transition graph: Default → Fire → Ice → Poison → Fire (with branches: Ice → Default, Poison → Default). Each state tracks internal counters (turnsInState, turnsAlone, enemiesAttacked) and attempts transitions only along allowed edges. Transition probability is tuned per state (e.g., Fire → Ice after 3 turns with 60% chance; Ice → Poison after 3 turns with 40% chance; Ice → Default after 4 turns alone with 30% chance).

Pros	Cons
Satisfies requirement: next state is always predetermined (no unreachable transitions).	Requires careful counter management in each state.
Probabilistic timing adds natural variation and tunable difficulty.	More complex than pure deterministic intervals.

Creates a coherent elemental cycle with meaningful progression.	
Easy to reason about and test (fixed edges, variable timing).	

We choose **Design B2** because it strictly follows the requirement's mandate for predetermined transitions while allowing probability to govern when transitions occur. This design balances predictability (strategic counterplay) with variability (replayability), and each state's transition conditions are encapsulated within the state itself, promoting SRP.

C. Special state effects and terrain interaction

Design C1: Effects hardcoded in Chimera. Chimera's playTurn() checks the current state flag and manually applies effects (e.g., if state == FIRE, burn surrounding tiles; if state == ICE, buff max HP).

Pros	Cons
All effects visible in one method.	Tight coupling: Chimera must know implementation details of every state's effects.
	DRY violation: effect application logic scattered across Chimera's methods.
	Closed to extension: adding a new state's effect requires modifying Chimera.
	Low cohesion: terrain manipulation, status effect application, and buff management mixed with core actor logic.

Design C2: State-specific inner attack actions with effect hooks. Each state provides a createAttackAction() factory that returns a customized inner action class (e.g., FireState::FireAttackAction, IceState::IceAttackAction, PoisonState::PoisonAttackAction). These inner actions extend the standard AttackAction and override execute() to apply state-specific effects: FireAttackAction burns two surrounding tiles and applies BurnEffect; IceAttackAction applies FrostBiteEffect; PoisonAttackAction applies PoisonEffect. States also implement onEnterState() hooks to apply buffs (e.g., IceState grants +5 max HP to chimera and tamer).

Pros	Cons
SRP: state classes own their attack effects; Chimera remains agnostic.	Requires inner classes or separate action classes per state.
High cohesion: terrain burning, status effect application, and buff logic live with the state that uses them.	Slightly more code per state.
OCP: new states with new effects don't modify existing actions or Chimera.	
Reuses engine's AttackAction while adding state-specific behavior (LSP).	
Clear extension point for future effects (e.g., lightning chains, wind knockback).	

We choose **Design C2** because it localizes special effects within state classes, adhering to SRP and OCP. The FireState's terrain-burning behavior (creating Fire ground objects) satisfies the requirement by modifying the environment, while status effects (burn, frostbite, poison) provide combat depth without polluting the Chimera class with conditional effect logic.

D. Weapon assignment and state-specific combat

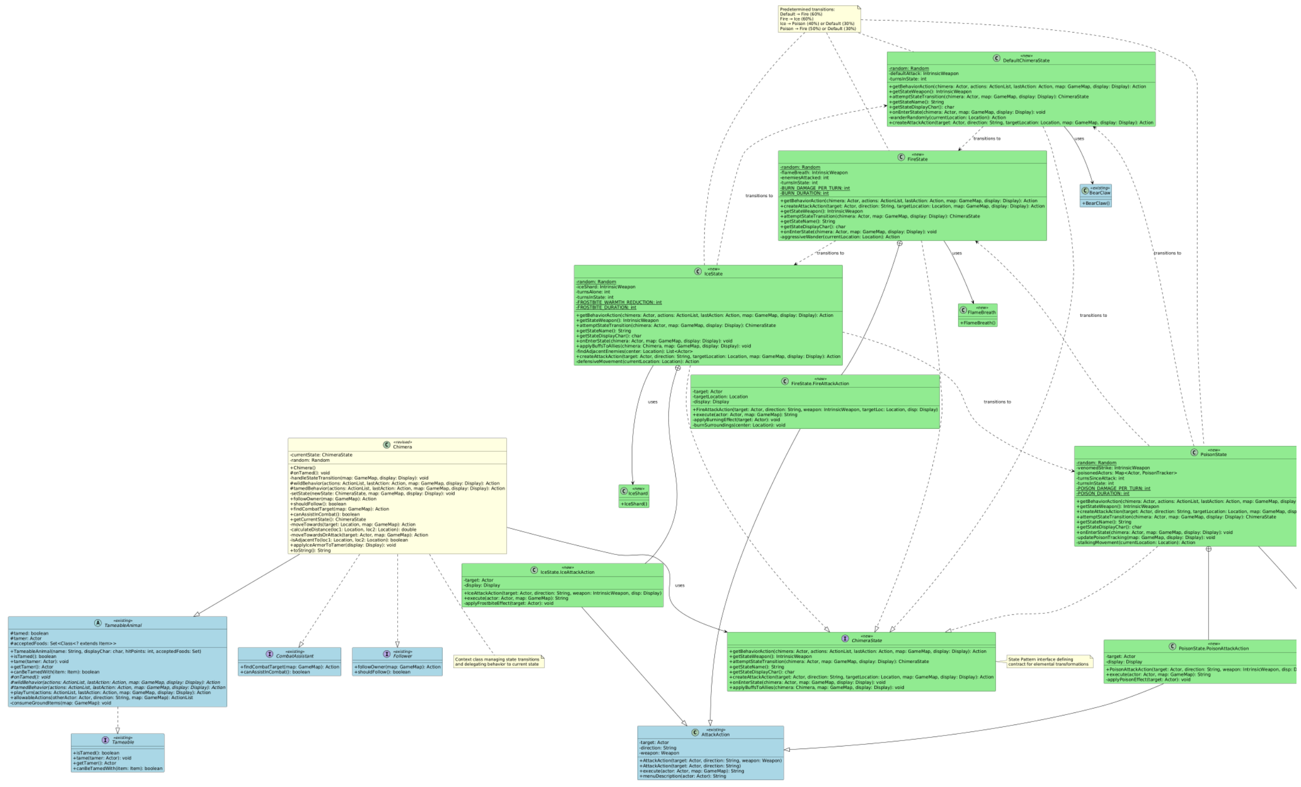
Design D1: Chimera's playTurn() selects weapon based on state flag. Chimera's getIntrinsicWeapon() switches on the state enum and returns the appropriate weapon instance; damage/hit chance are fields on Chimera.

Pros	Cons
Weapons are "close" to the actor.	Conflates actor state with weapon stats (low cohesion).
Fewer classes.	Hard to extend: adding a new weapon requires editing Chimera's switch statement.
	Weapon parameters (damage, hit chance, verb) scattered in conditionals.
	Violates SRP: Chimera must know every weapon's details.

Design D2: State provides IntrinsicWeapon subclass; Chimera delegates. Each state owns an IntrinsicWeapon subclass (BearClaw for Default, FlameBreath for Fire, IceShard for Ice, VenomedStrike for Poison) that encodes damage, verb, and hit chance. ChimeraState interface defines getStateWeapon(); Chimera calls setIntrinsicWeapon(currentState.getStateWeapon()) during state transitions. AttackAction uses the weapon abstraction without knowing which state produced it.

Pros	Cons
SRP: weapon stats live with the weapon; state logic lives with the state.	A few small weapon classes to hold constants.
OCP: new states add new weapon classes without modifying Chimera or AttackAction.	
High cohesion: each weapon is a self-contained data object.	
Plays naturally with engine's weapon abstraction; reusable for any actor.	
Clear extension point for future weapons (e.g., ToxicFang, ThunderStrike).	

We choose **Design D2** because it separates weapon identity from actor logic, enabling clean delegation and easy extension. When a state transition occurs, Chimera simply asks the new state for its weapon and updates its intrinsic weapon reference, keeping weapon selection logic decentralized and state-specific.



The diagram above shows the final design of requirement 5, which utilizes each Design 2 of parts A, B, C, and D stated above to implement the Chimera as a stateful creature that transforms between four elemental states (Default, Fire, Ice, Poison) with predetermined transitions, state-specific weapons, and unique combat effects. In this design, Chimera acts as the context class in the State Pattern, delegating behavior to its current ChimeraState implementation. Each concrete state (DefaultChimeraState, FireState, IceState, PoisonState) extends the ChimeraState interface and encapsulates its own behavior logic, transition rules, intrinsic weapon, and special effects (terrain burning, status application, buff granting). State transitions follow a predetermined graph (Default to Fire to Ice to Poison/Default, Fire to Ice, Ice to Poison/Default, Poison to Fire/Default), with probabilistic timing governed by internal counters (turnsInState, turnsAlone, enemiesAttacked), satisfying the requirement's mandate for non-random, predetermined paths. The FireState's terrain-burning behavior (spawning Fire ground objects on adjacent tiles) fulfills the "new behaviours similar to deers spawned from meadows" clause by modifying the environment dynamically. The design rationale and UML diagrams are in perfect alignment with each other and with the code implementation across all functional expectations, as per the relevant requirement(s). All relevant requirements have been addressed or attempted and they are covered in the implementation, UML diagrams, and design rationale.

Since state-specific logic (behavior, weapons, transitions, effects) is encapsulated entirely within state classes, we avoid duplicating transition checks, weapon selection, or effect application code (DRY). Using the ChimeraState interface as the contract ensures that Chimera depends on an abstraction rather than concrete state implementations (Dependency Inversion Principle), and adding new states or modifying existing ones does not require changes to the Chimera class (Open-Closed Principle). Each state class has a single responsibility (managing its own behavior, weapon, and transition conditions) while Chimera's responsibility is limited to maintaining the current state reference and delegating appropriately (Single Responsibility Principle). The state interfaces remain focused (getBehaviorAction, getStateWeapon, attemptStateTransition) without forcing non-producing states to implement irrelevant methods (Interface Segregation Principle), and all states are interchangeable through the same abstraction, enabling polymorphic treatment (Liskov Substitution Principle).