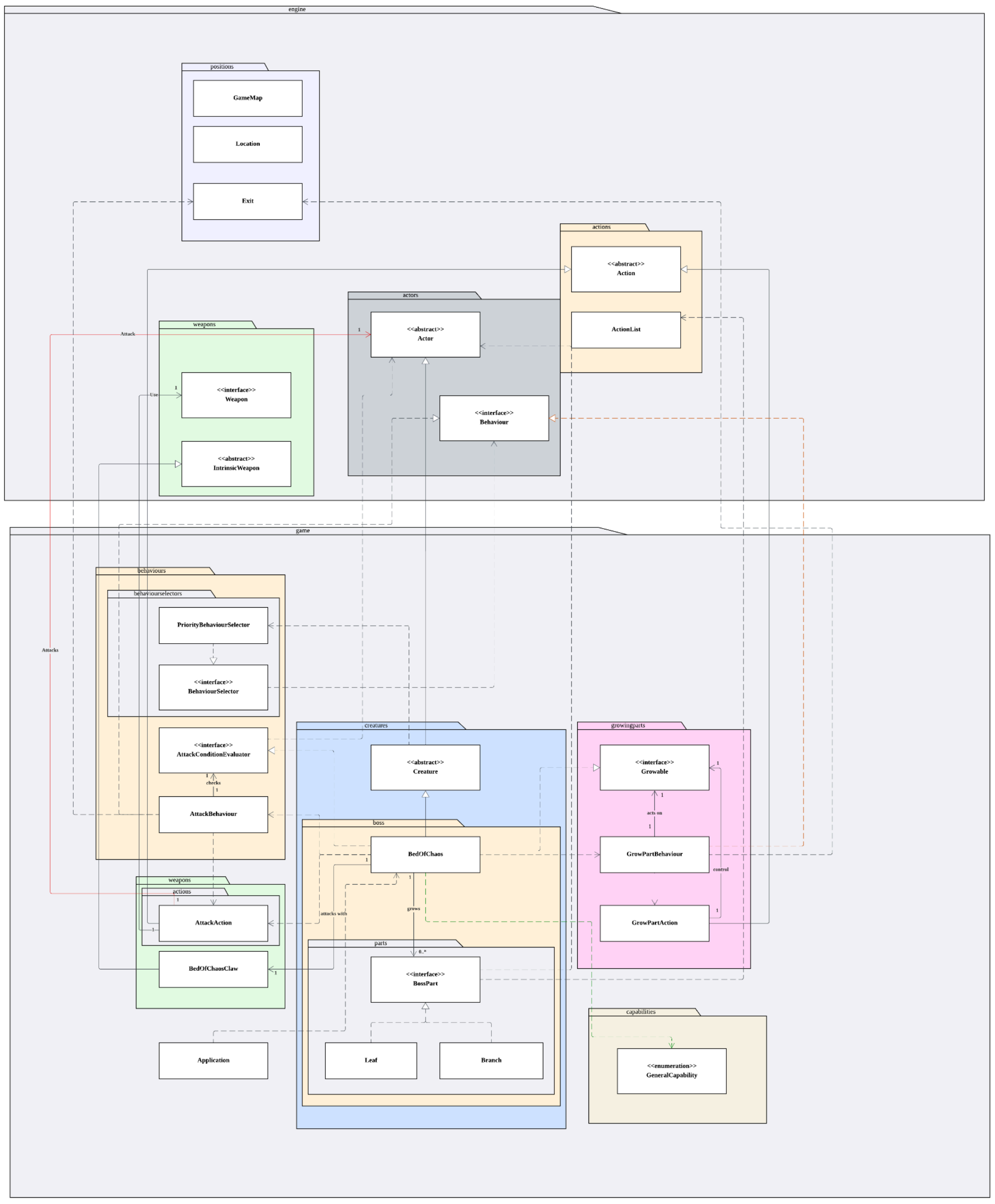
Design Rationale: Bed of Chaos Boss Implementation



1. Introduction

This document outlines the design for the "Bed of Chaos" boss, focusing on its composite structure, dynamic growth, variable damage calculation and attack mechanism. The primary goal is to create a modular, extensible, and maintainable system using object-oriented principles.

2. Design Goals

- Implement the "Bed of Chaos" boss, which remains in a fixed location.
- Implement a stationary boss that can dynamically grow Branch and Leaf parts.
- Allow Branch parts to grow subsequent parts.
- The boss's total attack power is the sum of its base damage and the damage from all its parts.

3. Design Analysis & Decisions

1. Boss Part Structure & Effect Calculation (Damage/Heal)

Option A: Boss Part Structure Composite Pattern (Chosen)

- **Description:** The boss will manage a single List<BossPart>, where BossPart is an interface implemented by Branch and Leaf. To calculate total damage, the boss will polymorphically call a getDamageContribution() method on each part in the list.
- **Maintainability:** High. Logic for each part is self-contained. Adding a new part type does not require changes to the BedOfChaos calculation.

Pros	Cons
Adheres to **SRP** and **OCP**. New part types can be added without modifying BedOfChaos.	More number of small classes
Low connascence of meaning where the effect calculation logic relies on an interface	

Option B: Separate Lists for Each Boss Part Type

- **Description:** The BedOfChaos class would maintain separate, concrete lists for each part type, such as private List<Branch> branches, and private List<Leaf> leaves. To calculate total damage, it would need multiple loops: one to iterate through the branches list (multiplying count by 3) and another for the leaves list (multiplying count by 1).
- **Maintainability:** Poor. Adding a new part type requires modifying the BedOfChaos class to add a new list and a new loop to process it.

Pros	Cons
More type-safe at compile time	Adding a new part requires modifying BedOfChaos to add List<Thorn> thorns (OCP).
Logic for a specific part type is handled in a dedicated loop within the boss class.	**Violates DRY** as part types increase, similar looping logic will be duplicated within BedOfChaos.

Decision & Justification

- **Chosen Design:** Option A.
- **Rationale:** This design is highly extensible. New part types (e.g., ThornPart) can be added without modifying the BedOfChaos class, adhering to the Open/Closed Principle. An alternative approach using separate lists for each part type (List<Branch>, List<Leaf>) was rejected because it would require modifying the core boss code for every new part, making it rigid and unmaintainable.

2. Dynamic Growth Mechanism

Option A: Shared List with Chain-Reaction Growth (Chosen)

- **Description:** The BedOfChaos will iterate through its single list of parts and call a grow() method on each. Branch parts can then add new parts directly to this same shared list, creating a simple and effective cascading growth effect within a single cycle.
- **Maintainability:** Moderate. The logic is concise, but the side effect (a method modifying a list passed as a parameter) requires careful handling.

Pros	Cons
Adheres to the **LSP**. The growth loop in BedOfChaos works with any object that implements BossPart.	Modifying a collection while iterating over it can be prone to bugs..
Low connascence of algorithm where the growth process relies on a uniform grow() method.	

Option B: Parent and Child Lists Hierarchical Structure

- **Description:** This would be a more traditional implementation of the Composite Pattern. The BedOfChaos class would have a List<BossPart>, and each Branch object would also have its own internal List<BossPart> to manage its direct children. Growth would be strictly recursive.
- **Maintainability:** High for relationship-dependent logic, but lower for overall state calculations.

Pros	Cons
Parent-child relationships are explicitly defined in the data structure.	High connascence of algorithms where the recursive traversal depends on the tree structure.
	Increased memory overhead and object management complexity.

Decision & Justification

- **Chosen Design:** Option A.
- **Rationale:** This method efficiently simulates the desired cascading growth with minimal complexity. It is better suited to the requirement of calculating a collective effect than a traditional hierarchical Composite Pattern (with explicit parent/child lists), which would introduce unnecessary overhead and require complex recursive traversals for state calculation.

3.3 Boss Attack Mechanism.

Option A: Intrinsic Weapon with Bonus Damage Calculation (Chosen)

- **Description**: The attack logic will be delegated to a separate AttackBehaviour component (Strategy Pattern). This component will evaluate conditions and execute attacks. The boss's total damage, calculated from all its parts, will be used to update an intrinsic BedOfChaosClaw weapon.
- **Maintainability**: High. Damage calculation is obtained from one method, and attack logic is delegated to AttackBehaviour, making it easy to extend/

Pros	Cons
BedOfChaosClaw handles weapon mechanics and AttackBehaviour decides when to attack. (SRP)	The BedOfChaosClaw's damage depends on the growables' state, requiring synchronization between growth and attack phases (High connascence of state)
AttackBehaviour delegates decision logic to AttackConditionEvaluator, reducing coupling to specific conditions (Low connascence of execution)	Increased complexity with additional classes (AttackBehaviour, AttackConditionEvaluator), though manageable with clear separation.

Option B: Hardcoded Attack Logic in BedOfChaos

- **Description**: The BedOfChaos directly checks for nearby players in playTurn, calculates total damage by iterating over BossPart lists, and uses a hardcoded IntrinsicWeapon with fixed damage. No AttackBehaviour or BedOfChaosClaw is used; attack logic is embedded in the boss.
- **Maintainability**: Poor. Adding new attack conditions or damage modifiers requires modifying BedOfChaos.

Pros	Cons
Simple initial implementation with no additional classes.	BedOfChaos handles growth, attack decision, and damage calculation (SRP)

Low connascence of control, all attack logic is centralized	High connascence of algorithm where attack logic is tightly coupled to the boss’s internal structure.

Decision & Justification

- **Chosen Design**: Option A.
- **Rationale**: This approach decouples attack logic from the boss's core responsibilities, adhering to the Single Responsibility Principle. It allows attack conditions to be modified or extended easily without altering the BedOfChaos class itself. Hardcoding attack logic directly within the boss class was rejected as it would create a monolithic, inflexible system that is difficult to maintain or extend.

4. Conclusion

By implementing the Composite and Strategy patterns, this design ensures the "Bed of Chaos" boss is modular, maintainable, and extensible. The polymorphic BossPart structure allows for the seamless addition of new components, while delegating growth and attack logic effectively separates concerns. The result is a robust system prepared for future enhancements.