State Pattern

1. Brief Introduction to State Pattern
    1.1 What is State Pattern
2. Previous Implementation and Problems
    2.1 Original Implementation Approach
    2.2 Problems with Original Approach
    2.3 Specific Pain Points
3. Overall Refactoring Design
    3.1 Architecture Overview
    3.2 Design Decisions
    3.3 UML Class Diagram
        Before Refactoring (Tag-Based Approach)
        After Refactoring (State Pattern)
    3.4 State Transition Diagram (BerserkBoss Example)
4. Detailed Implementation
    4.1 Core File Changes
    4.2 TransitionCondition System
        Design Philosophy
        Concrete Condition Example: HealthCondition
    4.3 MonsterState Implementation
        Core Design: Lambda-Based Actions
        Key Methods Explained
        Transition Priority System
    4.4 MonsterStateMachine Implementation
        Design: Centralized State Management
        Key Methods Explained
        Complete Example: Building a State Machine
5. Advantages of Refactored Design
    5.1 Separation of Concerns
    5.2 Code Reusability
    5.3 Support for Complex Conditions
    5.4 Easy Testing
    5.5 Extensibility: Adding New Conditions

State Pattern

1. Brief Introduction to State Pattern

1.1 What is State Pattern

The **State Pattern** is a behavioral design pattern that allows an object to alter its behavior when its internal state changes. The pattern appears to change the class of the object by encapsulating state-specific behavior into separate state classes.

Key Components:

- **Context**: The object whose behavior changes (Monster in our case)
- **State Interface**: Defines common interface for all states
- **Concrete States**: Implement specific behaviors for each state
- **Transition Logic**: Rules that determine when and how states change

2. Previous Implementation and Problems

2.1 Original Implementation Approach

Before refactoring, the monster AI system used a **tag-based state machine** with extensive if-else statements to manage monster behavior.

Example: Mandibular_worm Monster (Original Implementation)

```
class Mandibular_worm : public Monster {
public:
    Mandibular_worm() : Monster(NORMAL, 20, "Mandibular_worm", 2) {}
    void takeEffect() {
        std::shared_ptr<Creature> thisMonster =
CombatSystem::getInstance()->getMonsterPointer(this);
        // State 0: Attack
        if (tag == 0) {
            // transition needs a lot of nested if-else
            if (randomValue < 0.6) {
                tag = 1;  // 60% transition to buff state
            } else {
                tag = 2;  // 40% transition to attack+defend state
            }
        }
        // other actions...
};
```

2.2 Problems with Original Approach

1. Mixed Responsibilities

- Behavior logic and state transition logic are intertwined in a single method
- Difficult to understand which part handles "what to do" vs "when to transition"

2. Code Complexity Grows Exponentially

- For 3 states: 53 lines of code with nested if-else
- Adding more states or conditions quickly becomes unmaintainable
- Code duplication across similar behaviors
- Cannot easily implement:
    - Health-based transitions (e.g., enter berserk mode at 50% HP)
    - Round-based transitions (e.g., ultimate attack every 4 rounds)
    - Player state-based transitions (e.g., execute when player HP < 30)
    - Combined conditions (e.g., HP < 50% AND round >= 3)

3. Testing Challenges

- Cannot test individual states in isolation
- Must test entire takeEffect() method as a monolithic unit
- Difficult to reproduce specific state transitions

2.3 Specific Pain Points

Example: Adding a Half-Health Berserk Feature

With the tag-based approach, implementing "enter berserk mode when HP ≤ 50%" would require:

```
void takeEffect() {
    // PROBLEM: Need to check health condition in EVERY state
    if (getHealth() <= getMaxHealth() * 0.5 && !berserkTriggered) {
        tag = 3; // Berserk state
        berserkTriggered = true;
        // Add berserk buff...
        return;
    }
    // ... more duplication
}
```

This leads to:

- Code duplication (health check repeated in multiple places)
- New member variable `berserkTriggered` to track one-time event
- Increased cognitive load (hard to understand flow)

---

3. Overall Refactoring Design

3.1 Architecture Overview

The refactored system uses the **State Pattern** combined with a **Condition System** to separate concerns and improve extensibility.

Core Components:

```
Monster (Context)
    ├── MonsterStateMachine (Manages states and transitions)
    │       ├── map<string, MonsterState> (All states)
    │       └── MonsterState* currentState (Current state)
    │
    ├── MonsterState (Individual state)
    │       ├── function<void(Creature*)> action (Behavior - Lambda)
    │       ├── vector<StateTransition> transitions (Transition rules)
    │       └── IntentionInfo (UI display information)
    │
    └── TransitionCondition (Condition system)
            ├── HealthCondition (HP percentage)
```

```
                ├── RoundCondition (Round number)
                ├── ProbabilityCondition (Random probability)
                ├── PlayerStateCondition (Player HP/Block/Energy)
                └── CompositeCondition (AND/OR logic)
```

3.2 Design Decisions

1. State Behavior via Lambda Expressions

- Instead of creating separate strategy classes for each action, we use `std::function` with lambda expressions
- Simpler and more concise than full Strategy Pattern
- Avoids over-engineering for straightforward behaviors

2. Priority-Based Transition

- Each transition rule has a priority (int value)
- Higher priority conditions are checked first
- Enables critical transitions (e.g., HP-based) to override random ones

3. Backward Compatibility

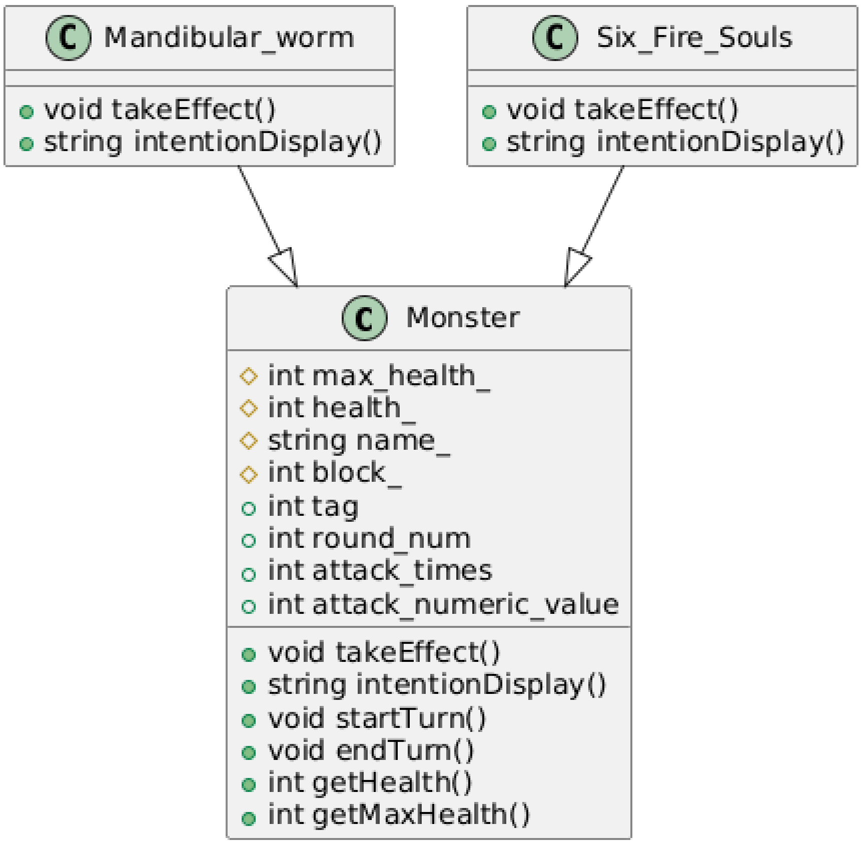
- Existing monsters continue using tag-based approach
- `stateMachine_` is an optional member in Monster base class
- No breaking changes to current codebase

4. Separation of Concerns

- **TransitionCondition**: "When to transition" (reusable across monsters)
- **MonsterState**: "What to do" + "Which states to transition to"
- **MonsterStateMachine**: "Manage all states and execute current state"

3.3 UML Class Diagram

Before Refactoring (Tag-Based Approach)

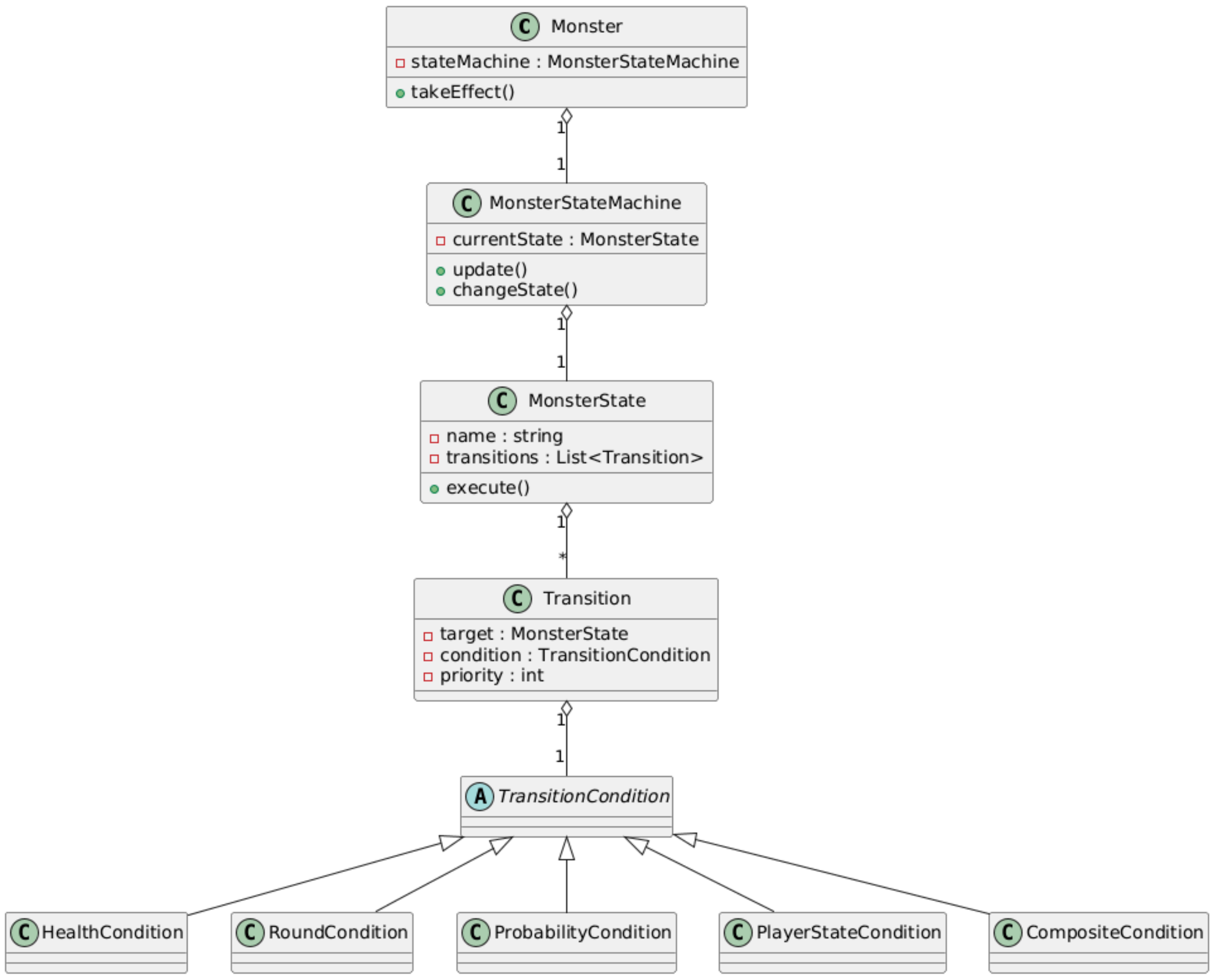


Key Issues in Original Design:

- **Tight Coupling**: State behavior and transition logic are inseparable
- **No Reusability**: Each monster reimplements similar logic
- **Hard to Test**: Cannot test states independently
- **Poor Scalability**: Adding states requires modifying large methods

---

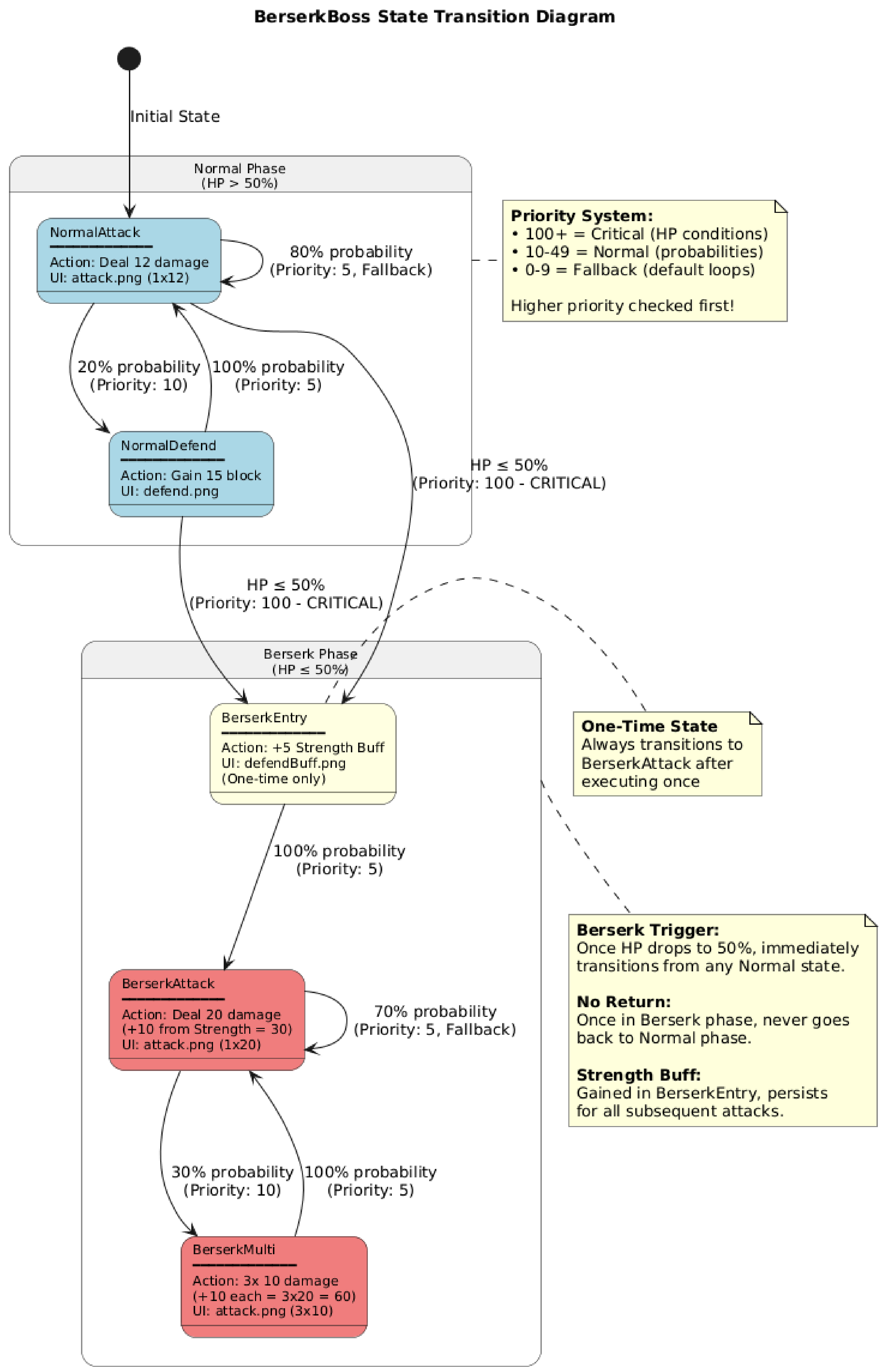
After Refactoring (State Pattern)



Key Improvements in New Design:

- **Separation of Concerns**: Conditions, States, and StateMachine are independent
- **Reusability**: Conditions can be shared across monsters
- **Extensibility**: New conditions/states without modifying existing code
- **Testability**: Each component can be tested in isolation

3.4 State Transition Diagram (BerserkBoss Example)


BerserkBoss State Transition Diagram
Initial State
Normal Phase (HP > 50%)
NormalAttack
Action: Deal 12 damage
UI: attack.png (1x12)
80% probability (Priority: 5, Fallback)
20% probability (Priority: 10)
100% probability (Priority: 5)
NormalDefend
Action: Gain 15 block
UI: defend.png
HP ≤ 50% (Priority: 100 - CRITICAL)
HP ≤ 50% (Priority: 100 - CRITICAL)
Priority System:
• 100+ = Critical (HP conditions)
• 10-49 = Normal (probabilities)
• 0-9 = Fallback (default loops)
Higher priority checked first!
Berserk Phase (HP ≤ 50%)
BerserkEntry
Action: +5 Strength Buff
UI: defendBuff.png
(One-time only)
One-Time State
Always transitions to BerserkAttack after executing once
100% probability (Priority: 5)
BerserkAttack
Action: Deal 20 damage (+10 from Strength = 30)
UI: attack.png (1x20)
70% probability (Priority: 5, Fallback)
30% probability (Priority: 10)
100% probability (Priority: 5)
BerserkMulti
Action: 3x 10 damage (+10 each = 3x20 = 60)
UI: attack.png (3x10)
Berserk Trigger:
Once HP drops to 50%, immediately transitions from any Normal state.
No Return:
Once in Berserk phase, never goes back to Normal phase.
Strength Buff:
Gained in BerserkEntry, persists for all subsequent attacks.

Transition Priority Explanation:

1. **Priority 100** (Critical): Health-based berserk trigger - checked FIRST in every state
2. **Priority 10** (Normal): Random probability transitions
3. **Priority 5** (Fallback): Default loops (always return to same/next state)

---

4. Detailed Implementation

4.1 Core File Changes

Involved Files :

```
Classes/Monster/
├── TransitionCondition.h
├── TransitionCondition.cpp
├── MonsterState.h
├── MonsterStateMachine.h
└── BerserkBoss.cpp

```

4.2 TransitionCondition System

The `TransitionCondition` system is the foundation of flexible state transitions. It defines **when** a state should transition to another state, completely separated from the **what** (the action the state performs).

Design Philosophy

- **Open/Closed Principle**: New condition types can be added without modifying existing code
- **Reusability**: Same condition instance can be used across multiple monsters and states
- **Testability**: Each condition can be tested independently
- **Composability**: Complex conditions can be built by combining simple ones

Base Interface:

```cpp
// Refactored with State Pattern
class TransitionCondition {
public:
    virtual ~TransitionCondition() = default;

    // Check if transition condition is met
    virtual bool check(std::shared_ptr<Creature> monster) const = 0;

    // Get text description (for debugging)
    virtual std::string describe() const = 0;
};
```

Interface Methods:

- `check()`: Returns `true` if the condition is satisfied. Takes a monster pointer to access its current state (HP, buffs, round count, etc.)
- `describe()`: Returns human-readable description for debugging and logging (e.g., "Health <= 50%")

Concrete Condition Example: HealthCondition

This condition triggers when a monster's health reaches a specific threshold, commonly used for:

- Entering berserk/enrage mode at low HP
- Phase transitions in boss fights
- Defensive states when damaged

```cpp
class HealthCondition : public TransitionCondition {
private:
    float percentage_;  // 0.0~1.0 (e.g., 0.5 = 50%)
    bool isLessThan_;   // true: <=, false: >
public:
    HealthCondition(float percentage, bool isLessThan);
    bool check(std::shared_ptr<Creature> monster) const override {
        if (!monster) return false;
        float currentPercentage = static_cast<float>(monster-
>getHealth()) /
```

```cpp
                                  static_cast<float>(monster->getMaxHealth());
        return isLessThan_ ? (currentPercentage <= percentage_)
                           : (currentPercentage > percentage_);
    }
};
```

4.3 MonsterState Implementation

`MonsterState` represents a single state in the monster's behavior. Each state encapsulates:

1. **What to do**: The action/behavior when in this state (using lambda functions)
2. **When to leave**: A list of possible transitions to other states
3. **UI Information**: Display data for player intention preview (attack icon, damage value, etc.)

Core Design: Lambda-Based Actions

Instead of creating separate classes for each action (which would require the full Strategy Pattern), we use `std::function` with lambda expressions. This provides:

- **Simplicity**: Inline behavior definition without extra files
- **Flexibility**: Each state can have completely different logic
- **Closure Capture**: Can capture local variables from the monster's constructor

```cpp
class MonsterState {
private:
    std::string stateName_;
    std::function<void(std::shared_ptr<Creature>)> action_;  // Lambda!
    std::vector<StateTransition> transitions_;
public:
    MonsterState(const std::string& name,
                 std::function<void(std::shared_ptr<Creature>)> action)
        : stateName_(name), action_(action),
          intentionIcon_(""), attackTimes_(0), attackValue_(0) {}
    // Execute state behavior ...
    // Check transitions (sorted by priority descending)
    std::string checkTransitions(std::shared_ptr<Creature> monster) {
        // ...
        }
        return "";  // No transition
    }
    //...
}
```

Key Methods Explained

1. **`execute()`** - Performs the state's action

- Called by `MonsterStateMachine` during the monster's turn
- Executes the lambda function stored in `action_`
- The lambda receives the monster pointer to access combat APIs (attack, defend, add buffs, etc.)

2. **`checkTransitions()`** - Determines next state

- Called **after** `execute()` to determine if state should change
- **Priority-based evaluation**: Higher priority conditions checked first
- Returns next state name, or empty string if no transition occurs

Transition Priority System

The priority system ensures critical transitions (like HP-based phase changes) take precedence over random ones:

```cpp
// Priority 100: Critical health-based transition - checked FIRST
attackState->addTransition("Berserk", healthCondition, 100);

// Priority 10: Normal random transition - checked SECOND
attackState->addTransition("Defend", randomCondition, 10);

// Priority 5: Fallback - checked LAST
attackState->addTransition("Attack", alwaysTrueCondition, 5);
```

Evaluation Order: When `checkTransitions()` runs, it sorts transitions by priority (descending) and returns the **first** condition that evaluates to `true`.

4.4 MonsterStateMachine Implementation

`MonsterStateMachine` is the **Context** in the State Pattern. It manages:

1. **State Registry**: All possible states the monster can be in
2. **Current State Tracking**: Which state is currently active
3. **State Execution**: Running the current state's action
4. **State Transitions**: Switching between states based on conditions

Design: Centralized State Management

The state machine acts as a container and controller, separating state management from the Monster class itself. This allows:

- **Single Responsibility**: Monster class focuses on attributes (HP, name, etc.), StateMachine handles behavior
- **Runtime Flexibility**: States can be added/modified dynamically
- **Reusability**: Same state machine logic works for all monsters

```cpp
// Refactored with State Pattern
class MonsterStateMachine {
private:
    std::map<std::string, std::shared_ptr<MonsterState>> states_;
    std::shared_ptr<MonsterState> currentState_;

public:
    MonsterStateMachine() : currentState_(nullptr) {}

    void addState(std::shared_ptr<MonsterState> state) {
        if (state) {
            states_[state->getName()] = state;
        }
    }

    void setInitialState(const std::string& stateName) {
        auto it = states_.find(stateName);
        if (it != states_.end()) {
            currentState_ = it->second;
        } else {
            CCLOG("Warning: Initial state '%s' not found", 
stateName.c_str());
        }
    }

    void execute(std::shared_ptr<Creature> monster) {
        if (!currentState_) {
            return;
        }
        currentState_->execute(monster);
    }

    void updateState(std::shared_ptr<Creature> monster) {
        if (!currentState_) return;

        std::string nextStateName = currentState_-
>checkTransitions(monster);

        if (!nextStateName.empty()) {
            auto it = states_.find(nextStateName);
            if (it != states_.end()) {
                currentState_ = it->second;
            }
        }
    }
    // ...
};
```

Key Methods Explained

1. `addState()` - Register a new state

- Stores state in the `states_` map using state name as key
- Called during monster initialization to build the state graph

2. `setInitialState()` - Set starting state

- Called once after all states are added

- Sets `currentState_` pointer to the specified state

3. `execute()` - Run current state's behavior

- Called during monster's turn in combat (from `Monster::takeEffect()`)
- Delegates to `currentState_->execute()`

4. `updateState()` - Handle state transitions

- Called **after** `execute()` each turn
- Asks current state to check its transitions
- If a valid next state is found, switches `currentState_` pointer

Complete Example: Building a State Machine

```cpp
// In BerserkBoss constructor
BerserkBoss::BerserkBoss() : Monster(ELITE, 80, "BerserkBoss", 3) {
    stateMachine_ = std::make_unique<MonsterStateMachine>();

    // Create states
    auto normalAttack = std::make_shared<MonsterState>("NormalAttack",
        [](auto m) { /* attack logic */ });
    auto defend = std::make_shared<MonsterState>("Defend",
        [](auto m) { /* defend logic */ });
    auto berserk = std::make_shared<MonsterState>("Berserk",
        [](auto m) { /* powerful attack */ });

    // Add transitions
    normalAttack->addTransition("Berserk",
        std::make_shared<HealthCondition>(0.5f, true), 100);  // Critical
    normalAttack->addTransition("Defend",
        std::make_shared<ProbabilityCondition>(0.3f), 10);    // Random

    // Register states
    stateMachine_->addState(normalAttack);
    stateMachine_->addState(defend);
    stateMachine_->addState(berserk);

    // Set starting state
    stateMachine_->setInitialState("NormalAttack");
}
```

---

5. Advantages of Refactored Design

5.1 Separation of Concerns

In the original implementation, *what* the monster does and *when* it transitions to another state are tightly coupled in the same block of code. This quickly becomes hard to maintain as more conditions and branches are added.

```cpp
// Before: Mixed logic
if (tag == 0) {
    attack();  // What to do
    if (random < 0.6) tag = 1;  // When to transition
    else tag = 2;
}
```

After refactoring, the action of the state and the transition logic are cleanly separated. The state only encapsulates the behavior (*WHAT* to do), while the transitions independently encode the conditions and priorities (*WHEN* to move to another state). This adheres to the Single Responsibility Principle and makes each part easier to change in isolation.

```cpp
// After: Clear separation
auto state = std::make_shared<MonsterState>("Attack",
    [](auto m) { attack(); });  // WHAT to do

state->addTransition("State1", condition1, 10);  // WHEN to transition
state->addTransition("State2", condition2, 5);
```

5.2 Code Reusability

In the refactored design, conditions are first-class objects that can be reused across multiple states and even different monsters. This avoids duplicating intricate logical expressions in many places and makes behavior rules more consistent.

```cpp
// Condition instances can be reused across different monsters
auto halfHealthCondition = std::make_shared<HealthCondition>(0.5f, true);

// Used in Monster A
stateA->addTransition("Berserk", halfHealthCondition, 100);

// Used in Monster B
stateB->addTransition("Enrage", halfHealthCondition, 100);

// Used in Monster C
stateC->addTransition("Phase2", halfHealthCondition, 100);
```

By encapsulating logic inside reusable `TransitionCondition` classes (such as `HealthCondition`), any change to the condition is automatically reflected in all monsters that use it. This increases consistency, reduces code duplication, and lowers the risk of subtle bugs caused by slightly different copies of the same logic.

5.3 Support for Complex Conditions

With the original approach, combining multiple factors (health, round number, player status, etc.) would require deeply nested `if` statements, extra flags, or ad-hoc logic scattered across the codebase.

```cpp
// Before: Impossible or very difficult
if (health <= 50% && round >= 3 && player_health < 30) {
    // This would require deeply nested conditions and multiple flags
}
```

The refactored design introduces a `CompositeCondition` that can combine multiple simple conditions using logical operators like AND. This makes it straightforward to express complex behavior in a declarative and extensible way.

```cpp
// After: Straightforward
auto composite = std::make_shared<CompositeCondition>(CompositeCondition::AND);
composite->addCondition(std::make_shared<HealthCondition>(0.5f, true));
composite->addCondition(std::make_shared<RoundCondition>(3, GREATER_EQUAL));
composite->addCondition(std::make_shared<PlayerStateCondition>(HEALTH, 30, true));

state->addTransition("SpecialState", composite, 100);
```

This composition-based approach makes the system more flexible: complex rules are built by combining small, reusable building blocks instead of writing one-off, hard-coded logic.

5.4 Easy Testing

By modeling behavior as states, transitions, and conditions, each piece becomes independently testable. Instead of having to simulate entire combat scenarios, we can write focused unit tests for each condition, state, and the overall state machine.

```cpp
// Test condition independently
TEST(HealthCondition, ChecksCorrectly) {
    auto monster = createMockMonster(50, 100);  // 50/100 HP
    auto condition = HealthCondition(0.5f, true);

    EXPECT_TRUE(condition.check(monster));  // 50% <= 50%
}

// Test state independently
TEST(MonsterState, ExecutesAction) {
    bool actionCalled = false;
    auto state = MonsterState("Test",
        [&](auto m) { actionCalled = true; });

    state.execute(mockMonster);
    EXPECT_TRUE(actionCalled);
}

// Test state machine independently
TEST(MonsterStateMachine, TransitionsCorrectly) {
    // ... isolated test
}
```

This improves reliability and makes regression testing easier. When changing behavior logic, we can quickly verify that conditions, actions, and transitions still behave as expected without needing to run a full game.

5.5 Extensibility: Adding New Conditions

Adding new gameplay rules no longer requires modifying existing `if` chains or touching multiple parts of the code. Instead, we introduce a new `TransitionCondition` subclass and plug it into the state machine where needed.

```cpp
// Step 1: Create new condition class
class BuffCountCondition : public TransitionCondition {
private:
    std::string buffName_;
    int minLayers_;
public:
    BuffCountCondition(const std::string& buff, int min)
        : buffName_(buff), minLayers_(min) {}
    bool check(std::shared_ptr<Creature> monster) const override {
        for (const auto& buff : monster->buffs_) {
            if (buff->name_ == buffName_ &&
                buff->effect_layers >= minLayers_) {
                return true;
            }
        }
        return false;
    }
};

// Step 2: Use it immediately
state->addTransition("PoweredUp",
    std::make_shared<BuffCountCondition>("StrengthBuff", 5), 50);
```

This design follows the Open/Closed Principle: the behavior system is open for extension (we add new condition classes) but closed for modification (we rarely need to change existing core logic). As a result, new mechanics and AI behaviors can be added quickly and safely, which is especially valuable in a growing game codebase.