CS 247

Software Engineering Principles

Chapter 16

The State Pattern

Victoria Sakhnini

Table of Contents

Introduction

Every sufficiently complex object has phases. A network socket is not always connected. A media player is not always playing. A vending machine is not always waiting for a coin. In each case, the object is the same throughout its life, but its behaviour changes dramatically depending on which phase or state it is currently in.

The naive way to model this is with an integer or enum field and a cascade of if/else or switch branches at the start of every method. This approach works for two or three states. It collapses under its own weight as soon as requirements grow: every new state requires touching every existing method, and every new method requires a case for every existing state. The code that started as a neat conditional becomes an unmaintainable tangle.

The State Pattern solves this problem by turning each state into its own class. Each state class encapsulates everything the object does while in that state. Adding a new state means adding one new class; nothing else changes. This chapter develops the pattern from first principles using the Gumball Machine example from Head First Design Patterns, then generalizes it to a full C++ implementation.

The State Pattern Defined

The Gang of Four definition is:

> *Allow an object to alter its behaviour when its internal state changes. The object will appear to change its class.*

Two sentences, two ideas.

"Allow an object to alter its behaviour when its internal state changes." The Context object, the thing clients call methods on, delegates every state-dependent call to a separate State object. When the state changes, the Context swaps which State object it delegates to. The next call gets completely different behaviour from the same method signature.

"The object will appear to change its class." From the caller's perspective, a GumballMachine that has a quarter behaves like an entirely different type than a GumballMachine that does not. In reality it is the same object; only the internal State pointer has been swapped. The illusion of type change arises because behaviour is fully determined by the delegate.

> **Tip:**
>
> **State vs. Strategy.** The class diagrams for State and Strategy are identical: a Context holds a pointer to an abstract base class, and concrete subclasses vary the behaviour. The difference is intent. In the Strategy Pattern, the client usually chooses and sets the strategy; in the State Pattern, the states themselves drive transitions. Strategy is typically configured once; State changes continuously as the system evolves.

The Problem it Solves: Gumball Machine

Mighty Gumball, Inc. needs a software controller for their gumball machine. The machine has four observable states and four user actions.

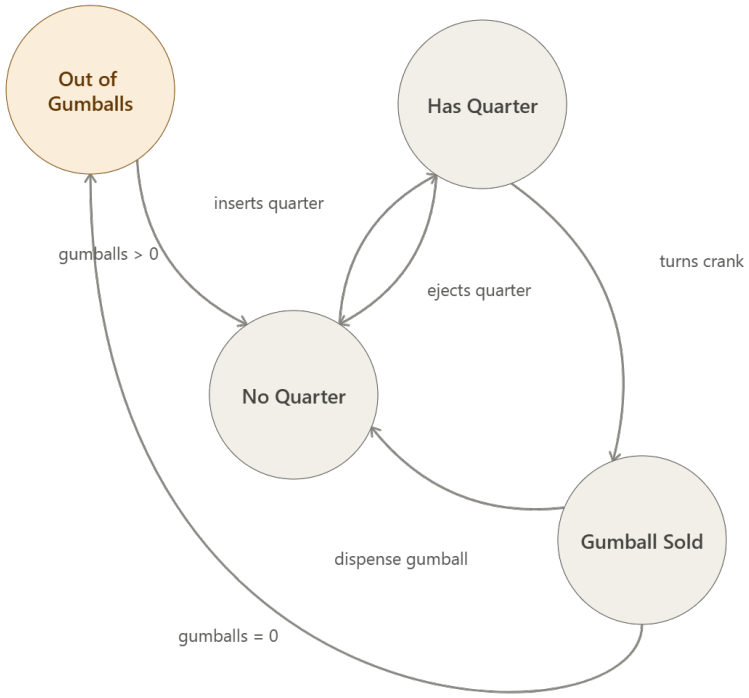

Figure 1. The Mighty Gumball, Inc. state-transition diagram. Circles are states; arrows are the actions that cause transitions. The machine's behaviour at any moment is fully determined by which circle it is currently in.

The naive approach and why it breaks

A first implementation puts all logic in a single class, gated on an integer field:

```
class GumballMachine {
    static constexpr int SOLD_OUT   = 0;
    static constexpr int NO_QUARTER = 1;
    static constexpr int HAS_QUARTER= 2;
    static constexpr int SOLD       = 3;

    int state_ = SOLD_OUT;
    int count_ = 0;

public:
```

```
    void insertQuarter() {
        if state_ == NO_QUARTER){
             std::cout << "You inserted a quarter\n";
             state_ = HAS_QUARTER;
        }
        else if (state_ == HAS_QUARTER){
           std::cout << "You can't insert another quarter\n";
        }
        else if (state_ == SOLD_OUT){
          std::cout << "You can't insert a quarter, the machine is sold
out\n";
        }
        else if (state_ == SOLD){
         std::cout << "Please wait, we're already giving you a gumball\n";
        }
    }
    // ... and the same structure for ejectQuarter, turnCrank, dispense
};
```

This works for four states. Then Mighty Gumball calls back: they want a WinnerState where 10% of cranks dispense two gumballs for free. Now every method needs a new branch. The code that started manageable has become a maintenance hazard:

- Every new state requires editing every existing method.
- Every new method requires a case for every existing state.
- State-specific logic is scattered across the entire class instead of being in one place.
- Testing any single state in isolation is impossible without running through the entire class.

> ⚠ **Warning:**
>
> **Open/Closed violation.** This design is open for modification (you have to modify it for every change) and closed for extension (you cannot add a state without reopening every method). The State Pattern upholds the Open/Closed Principle: adding a new state introduces one new class and modifies at most the states that transition into it.

The State Pattern solution

The solution is to localize each state's behaviour into its own class. The slide notes describe the three-step approach:

1. Define a State interface that contains a method for every action in the machine.
2. Implement a State class for every state of the machine. Each class is responsible for the machine's behaviour in its corresponding state.
3. Remove all the conditional code from the Context and delegate every action call to the current State object.

The behaviour mapping below shows what each state class needs to do for each action:

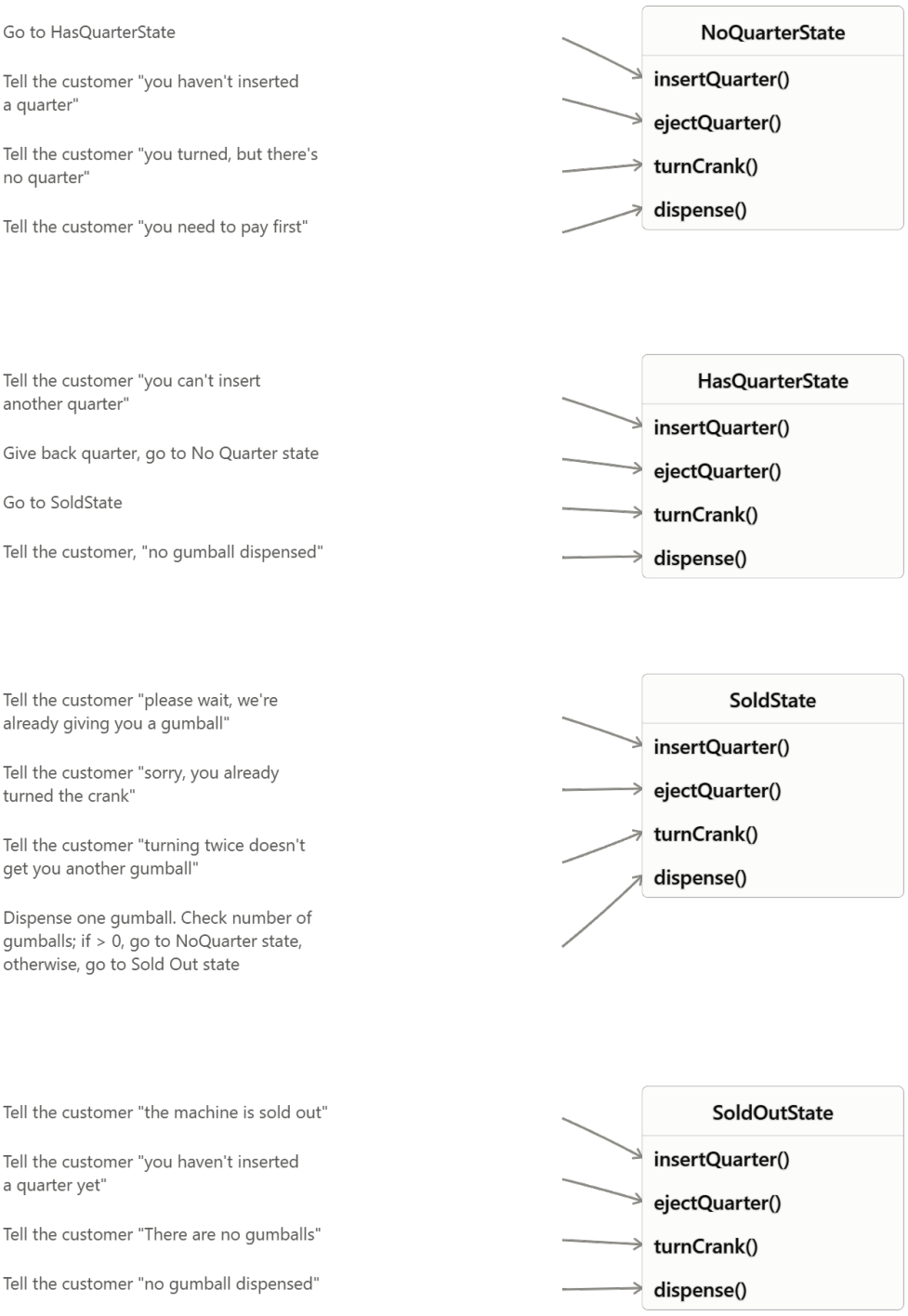

Figure 2. Behaviour mapping: for each of the four states (columns on the right), what should each action do? This table is the specification for the four concrete state classes. Each cell becomes the body of one override method.

Design: Three Participants

Every State Pattern implementation has three roles: the State interface, one or more Concrete State classes, and the Context.

The State interface

The State interface declares a pure virtual method for every action the Context exposes. In the gumball machine there are five: insertQuarter, ejectQuarter, turnCrank, dispense, and getStateName (for diagnostics). The protected default constructor prevents direct instantiation while still allowing derived classes to construct normally.

A critical structural point: State.h must refer to GumballMachine (so states can call machine_.setState(...) during transitions), but GumballMachine.h must refer to State (so the Context can hold State* pointers). This circular dependency is broken with a forward declaration:

```
// Forward declaration breaks the circular dependency
class GumballMachine;

// State interface
class State {
public:
    virtual ~State() = default;
    virtual void insertQuarter()  = 0;
    virtual void ejectQuarter()   = 0;
    virtual void turnCrank()      = 0;
    virtual void dispense()       = 0;
    virtual std::string_view getStateName() const = 0;
protected:
    State() = default;
};
```

> **Tip:**
>
> **Why forward-declare instead of #include?** A forward declaration tells the compiler that GumballMachine is a class name, which is enough information to declare a reference (GumballMachine&) or pointer (GumballMachine*) to it. The full definition is only needed in the .cpp file where methods are actually implemented. This breaks the cycle without exposing either header to the other's internals.

Concrete State classes

Each concrete state inherits from State and holds a GumballMachine& reference, a reference rather than a pointer because references cannot be null and because the state object is meaningless without the machine it belongs to. The explicit keyword on the constructors prevents accidental implicit conversions.

```
// Concrete States
class NoQuarterState final : public State {
public:
    explicit NoQuarterState(GumballMachine& machine): machine_(machine) {}
    void insertQuarter()  override;
    void ejectQuarter()   override;
    void turnCrank()      override;
    void dispense()  override;
    std::string_view getStateName() const override { return "NoQuarter"; }
private:
    GumballMachine& machine_;
};

class HasQuarterState final: public State {
public:
    explicit HasQuarterState(GumballMachine& machine): machine_(machine) {}
    void insertQuarter()  override;
    void ejectQuarter()   override;
    void turnCrank()      override;
    void dispense()       override;
    std::string_view getStateName() const override { return "HasQuarter"; }
private:
    GumballMachine& machine_;
};

class SoldState final: public State {
public:
    explicit SoldState(GumballMachine& machine): machine_(machine) {}
    void insertQuarter()  override;
    void ejectQuarter()   override;
    void turnCrank()      override;
    void dispense()       override;
    std::string_view getStateName() const override { return "Sold"; }
private:
    GumballMachine& machine_;
};

class SoldOutState final: public State {
public:
    explicit SoldOutState(GumballMachine& machine): machine_(machine) {}
    void insertQuarter()  override;
    void ejectQuarter()   override;
    void turnCrank()      override;
    void dispense()       override;
    std::string_view getStateName() const override { return "SoldOut"; }
private:
    GumballMachine& machine_;
};
```

The Context class (GumballMachine)

The Context owns all the ConcreteState objects via std::unique_ptr members. It exposes one raw (non-owning) getter for each state so that state methods can call machine_.setState(machine_.getNoQuarterState()) to trigger transitions. The state_ raw pointer is a non-owning observer into one of the four unique_ptrs; it is set in the constructor and updated by setState.

```
// Context class

class GumballMachine {
public:
    explicit GumballMachine(int numberGumballs)
        : count_(numberGumballs)
        , soldOutState_  (std::make_unique<SoldOutState>  (*this))
        , noQuarterState_(std::make_unique<NoQuarterState>(*this))
        , hasQuarterState_(std::make_unique<HasQuarterState>(*this))
        , soldState_     (std::make_unique<SoldState>     (*this))
    {
        state_ = (numberGumballs > 0)
                    ? noQuarterState_.get()
                    : soldOutState_.get();
    }

    // Delegation, every public action forwards to the current state
    void insertQuarter() { state_->insertQuarter(); }
    void ejectQuarter()  { state_->ejectQuarter();  }
    void turnCrank()     { state_->turnCrank(); state_->dispense(); }

    // Transition helper called by state methods
    void setState(State* state) { state_ = state; }

    void releaseBall() {
        std::cout << "A gumball comes rolling out the slot...\n";
        if (count_ > 0) --count_;
    }

    // Non-owning state accessors used by ConcreteStates during transitions
    State* getSoldOutState()const noexcept { return soldOutState_.get(); }
    State* getNoQuarterState()const noexcept {
                return noQuarterState_.get();}
    State* getHasQuarterState() const noexcept {
                 return hasQuarterState_.get();}
    State* getSoldState() const noexcept { return soldState_.get(); }
    int    getCount() const noexcept { return count_; }

    friend std::ostream& operator<<(std::ostream& os, const GumballMachine& gm)
    {
        os << "\nMighty Gumball, Inc.\n"
           << "C++-enabled Standing Gumball Model #2017\n"
           << "Inventory: " << gm.count_ << " gumball"
           << (gm.count_ != 1 ? "s" : "") << "\n"
           << "Machine is " << gm.state_->getStateName() << " state\n";
        return os;
```

```
    }

private:
    std::unique_ptr<State> soldOutState_;
    std::unique_ptr<State> noQuarterState_;
    std::unique_ptr<State> hasQuarterState_;
    std::unique_ptr<State> soldState_;
    State* state_{nullptr};
    int    count_{0};
};
```

> **Tip:**
>
> **Why own via unique_ptr but refer via raw pointer?** The unique_ptr members in GumballMachine own the state objects and guarantee they are destroyed when the machine is destroyed. The state_ raw pointer is a non-owning observer; it simply records which owned state is currently active. Mixing ownership (unique_ptr) with observation (raw pointer) is idiomatic C++17 for this pattern.

The Complete C++ Implementation

With the headers in place, each state's method bodies implement the behaviour for that state. The structure is consistent: legal actions perform their work and call machine_.setState(...) to transition; illegal actions print an error message.

State implementations

```
// State implementations
void NoQuarterState::insertQuarter() {
    std::cout << "You inserted a quarter\n";
    machine_.setState(machine_.getHasQuarterState());
}
void NoQuarterState::ejectQuarter() {
    std::cout << "You haven't inserted a quarter\n";
}
void NoQuarterState::turnCrank() {
    std::cout << "You turned, but there's no quarter\n";
}
void NoQuarterState::dispense() {
    std::cout << "You need to pay first\n";
}
```

```
void HasQuarterState::insertQuarter() {
    std::cout << "You can't insert another quarter\n";
}
void HasQuarterState::ejectQuarter() {
    std::cout << "Quarter returned\n";
    machine_.setState(machine_.getNoQuarterState());
}
void HasQuarterState::turnCrank() {
    std::cout << "You turned...\n";
    machine_.setState(machine_.getSoldState());
}
void HasQuarterState::dispense() {
    std::cout << "No gumball dispensed\n";
}
```

```
void SoldState::insertQuarter() {
    std::cout << "Please wait, we're already giving you a gumball\n";
}
void SoldState::ejectQuarter() {
    std::cout << "Sorry, you already turned the crank\n";
}
void SoldState::turnCrank() {
    std::cout << "Turning twice doesn't get you another gumball!\n";
}
void SoldState::dispense() {
    machine_.releaseBall();
    if (machine_.getCount() > 0) {
        machine_.setState(machine_.getNoQuarterState());
    } else {
        std::cout << "Oops, out of gumballs!\n";
        machine_.setState(machine_.getSoldOutState());
    }
}
```

```
void SoldOutState::insertQuarter() {
    std::cout << "You can't insert a quarter, the machine is sold out\n";
}
void SoldOutState::ejectQuarter() {
    std::cout << "You can't eject, you haven't inserted a quarter yet\n";
}
void SoldOutState::turnCrank() {
    std::cout << "You turned, but there are no gumballs\n";
}
void SoldOutState::dispense() {
    std::cout << "No gumball dispensed\n";
}
```

Notice the design that emerges. Every state implements every method, even the nonsensical ones. The correct response to a nonsensical action is an informative error message, not a crash or silence. This is

an intentional design choice: it means the machine always produces predictable, debuggable output, even when used incorrectly.

> **? Question:**
>
> **Why must illegal actions print a message rather than do nothing?** Silent failure is the enemy of debugging. If a client calls insertQuarter() on a machine that is sold out and nothing happens, the client does not know whether the machine is broken, already has a quarter, or is in an unknown state. An explicit message points directly to the problem. Designing for observability is part of designing for correctness.

main(): exercising the machine

```
int main() {
    GumballMachine gumballMachine(5);
    std::cout << gumballMachine;

    gumballMachine.insertQuarter();
    gumballMachine.turnCrank();
    std::cout << gumballMachine;

    gumballMachine.insertQuarter();
    gumballMachine.turnCrank();
    gumballMachine.turnCrank();         // second crank, should reject
    gumballMachine.insertQuarter();
    gumballMachine.ejectQuarter();
    gumballMachine.turnCrank();
    std::cout << gumballMachine;

    gumballMachine.insertQuarter();
    gumballMachine.turnCrank();
    std::cout << gumballMachine;

    gumballMachine.insertQuarter();
    gumballMachine.turnCrank();
    std::cout << gumballMachine;

    gumballMachine.insertQuarter();
    gumballMachine.turnCrank();
    std::cout << gumballMachine;

    return 0;
}
```

The UML Class Diagram

The complete class diagram for the gumball machine implementation shows the three-participant structure of the State Pattern.

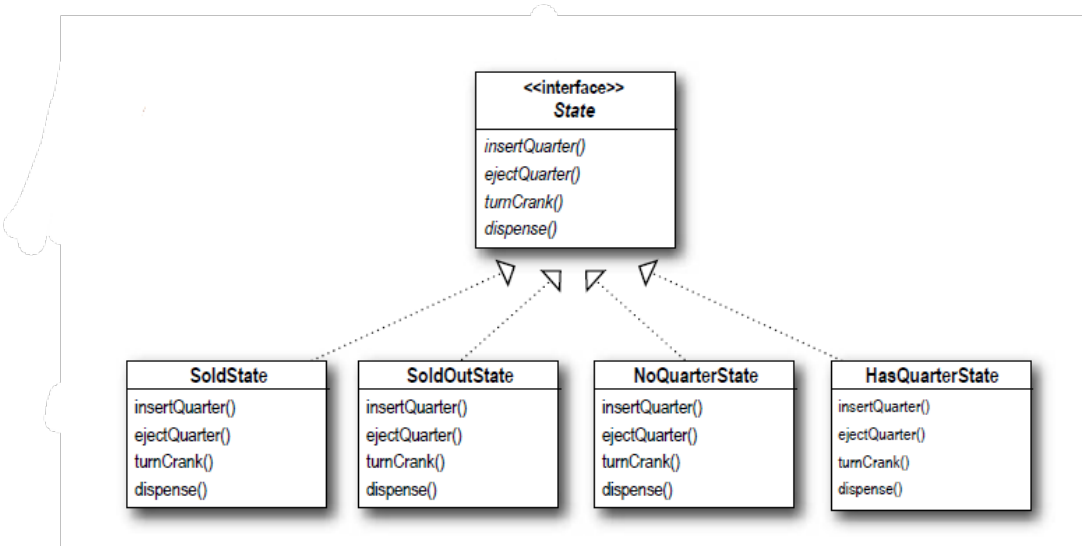

Figure 3. UML class diagram for the gumball machine.

The structure generalizes directly. In any State Pattern application:

- The Context holds a pointer to the current State and delegates method calls to it.
- The State interface declares one pure virtual method per action.
- Each ConcreteState implements all methods for its particular state and triggers transitions by calling context.setState(...).

Transitions are always initiated from inside ConcreteState methods, never from the Context itself. This is what keeps the Context free of conditional logic.

State vs. Strategy

State and Strategy share the same class diagram: a Context is composed of an abstract base, and concrete subclasses vary the behaviour. They have the same class diagram but differ in intent.

Dimension	State Pattern	Strategy Pattern
Who controls transitions?	The ConcreteState objects themselves call context.setState(...)	Usually the client, or the Context on construction
How often does it change?	Continuously, as the system moves through its lifecycle	Rarely; typically chosen once and kept
Do states know each other?	Yes, each state knows which states to transition to	No, strategies are independent of each other
Primary intent	Model an object lifecycle with distinct behavioural phases	Select and swap algorithms or behavioural variants
Class count effect	Increases class count significantly (one class per state)	Also increases class count (one class per algorithm)

A practical test: if the behaviour-switching object decides for itself when to switch (self-transitioning), it is State. If external code decides when and what to switch, it is Strategy.

Tasks: Extending the Design

You will work on two extension tasks that test understanding of the pattern. Both can be completed without modifying any existing state class, which demonstrates that the Open/Closed Principle is working.

Task 1: Add a WinnerState

Mighty Gumball wants 10% of cranks to dispense two gumballs instead of one. The WinnerState is reached from HasQuarterState when the crank is turned and a random check succeeds.

Update the UML diagram and update the code to add a WinnerState. Requirements:

- WinnerState is a new ConcreteState that releases two gumballs.
- HasQuarterState::turnCrank() should transition to WinnerState 10% of the time (std::rand() % 10 == 0) and to SoldState otherwise.
- WinnerState::dispense() should call machine_.releaseBall() twice, then transition based on count.
- No existing state class other than HasQuarterState should need modification.

Task 2: Add a refill() method

When the machine is in SoldOutState, an operator needs to be able to refill it. This is a Context-level operation, not a State-level one, because the refill comes from outside the normal user interaction cycle.

Add a refill(int count) method to GumballMachine. The implementation is straightforward:

```
void GumballMachine::refill(int count) {
    count_ = count;
    state_ = noQuarterState_.get();
}
```

Why is this on GumballMachine and not on State? Because refilling is an operator action that bypasses the state machine, it resets state to a known starting point regardless of what state the machine is currently in. Putting it on a specific State class (e.g., SoldOutState::refill()) would mean the method is only callable in that one state, which is not the right abstraction.

Real-World Uses

The State Pattern appears in any system where an object's behaviour changes significantly depending on its current phase or status.

Infrastructure and networking

- **TCP connections.** A TCP socket moves through states: LISTEN, SYN_SENT, SYN_RECEIVED, ESTABLISHED, FIN_WAIT_1, FIN_WAIT_2, CLOSE_WAIT, CLOSING, LAST_ACK, TIME_WAIT, CLOSED. The socket's behaviour for send(), recv(), and close() is entirely different in each state. RFC 793 is, in essence, a State Pattern specification.
- **HTTP/2 streams.** Each HTTP/2 stream moves through idle → open → half-closed (local) → half-closed (remote) → closed. The allowed frame types and their effects depend entirely on the current stream state.
- **Thread lifecycle.** A thread moves through NEW → RUNNABLE → RUNNING → WAITING/BLOCKED → TERMINATED. The operations legal in each state (start(), join(), interrupt()) produce different behaviour or throw exceptions depending on the current state.

E-commerce and business workflows

- **E-commerce checkout.** An order moves through CART → SHIPPING_INFO → PAYMENT → CONFIRMED → PROCESSING → SHIPPED → DELIVERED (or CANCELLED at various points). The operations available to the customer change completely at each stage.
- **Document approval workflows.** A document moves through DRAFT → IN_REVIEW → APPROVED → PUBLISHED → ARCHIVED. Who can edit it, what actions are available, and what notifications fire all depend on the current state.

Physical systems and games

- **Vending machines.** Idle → coin inserted → item selected → dispensing → sold out. The lecture example is a direct model of this physical system.
- **Traffic lights.** GREEN → YELLOW → RED → GREEN, with pedestrian crossing states, fault states, and maintenance modes. Each state controls which lights are on and for how long.
- **Game characters.** IDLE → WALKING → RUNNING → JUMPING → ATTACKING → TAKING_DAMAGE → DEAD. Each state controls which animation plays, which physics apply, and which inputs are accepted.

Summary

Recognition: what to spot in code

- *A Context class* that holds a pointer or reference to an abstract State type and delegates every state-sensitive method call to it.
- *A State abstract interface* with one pure virtual method per action that the Context supports.
- *ConcreteState classes* (one per state) that implement all methods and call context.setState(...) to trigger transitions.
- *Self-triggered transitions:* the ConcreteState decides when to move to the next state, not the Context and not external code.

Comprehension: how the pieces fit

- The State Pattern eliminates cascading conditionals by distributing state-specific logic across dedicated classes, one class per state.
- The Context owns the ConcreteState objects (typically via unique_ptr) and holds a raw non-owning pointer to the current state. Transitioning is O(1): just swap the pointer.
- State and Strategy have the same class diagram but different intent. State involves self-triggered transitions and a changing internal context; Strategy is typically set externally and changes rarely.
- Using the State Pattern increases the number of classes in the design. This is intentional: more, smaller, focused classes are easier to test and modify than one large class with conditional logic.

Application: what to actually do

- **Start with the state-transition diagram.** Draw the circles (states) and arrows (triggering actions) before writing any code. Each circle becomes a ConcreteState class; each arrow becomes a conditional branch inside a method of the source state.
- **Use std::unique_ptr in the Context.** The Context should own the ConcreteState objects. Use raw non-owning State* only for the current-state pointer and the transition accessors.
- **Hold GumballMachine& in states.** A reference cannot be null and communicates that the state is permanently bound to its machine. Use a pointer only if the binding needs to be reseatable.
- **Forward-declare the Context in the State header.** This breaks the circular dependency without exposing either class's internals to the other.
- **Implement every method in every state.** Illegal actions should produce informative error messages, not silent no-ops. Observability matters.

Practice Problems

1.

A GumballMachine is created with 2 gumballs. Trace the following sequence of calls, stating after each call: (a) which ConcreteState method executed, (b) what was printed, and (c) what state the machine is now in.

```
gumballMachine.insertQuarter();
gumballMachine.turnCrank();       // (dispenses 1 gumball, 1 remaining)
gumballMachine.insertQuarter();
gumballMachine.turnCrank();       // (dispenses 1 gumball, 0 remaining)
gumballMachine.insertQuarter();   // machine should now be sold out
```

After the last call, what state is the machine in and why?

2.

Before applying the State Pattern, the naive GumballMachine violated the Open/Closed Principle.

- Explain in one paragraph exactly how the naive implementation violates OCP. Be specific about which code must be modified when a new state is added.
- Explain how the State Pattern restores OCP compliance. Which files are modified when WinnerState is added? Which are not?
- Is there any modification at all when a new state is added? (Hint: think about the states that transition into the new state.)

3.

Add a WinnerState to the complete implementation. Requirements:

- WinnerState::dispense() calls machine_.releaseBall() twice. After dispensing, it transitions to NoQuarterState if count > 0, or SoldOutState otherwise.
- HasQuarterState::turnCrank() transitions to WinnerState with 10% probability (use std::rand() % 10 == 0) and to SoldState otherwise.
- No state class other than HasQuarterState should require modification.
- Write a main() that demonstrates WinnerState being triggered (you may need to run it several times or temporarily change the probability to 100% for testing).

4.

For each scenario below, decide whether it is better modelled as a State Pattern or a Strategy Pattern, and justify your choice in one sentence.

- A text editor supports three editing modes: normal, insert, and visual selection. The behaviour of every keystroke depends on which mode is active. Modes switch via specific keystrokes.
- A sorting library uses bubble sort for arrays with fewer than 10 elements and quicksort for larger ones. The algorithm is chosen automatically on each call.
- An ATM moves through Idle → Card Inserted → PIN Entered → Transaction → Dispensing → Receipt → Idle. Each transition is triggered by a specific user action.
- A payment processor supports Stripe, PayPal, and bank transfer. The client selects which one to use at checkout.

5.

Model a traffic light using the State Pattern. The light cycles through GREEN (30 seconds) → YELLOW (5 seconds) → RED (25 seconds) → GREEN. There is also a FAULT state that can be entered from any state when a hardware error is detected; in FAULT the light flashes yellow.

- Draw the state-transition diagram.
- Write C++ class declarations (headers only) for the Context (TrafficLight) and all ConcreteState classes.
- Implement TrafficLight::tick(), which advances the timer by one second and triggers a transition when the timer expires.
- Implement TrafficLight::reportFault() and TrafficLight::clearFault(), which transition to and from FAULT state.

6.

The following implementation has three design problems. Identify each one and explain how to fix it.

```
class IdleState : public State {
public:
    void coinInserted() override {
        machine_.setState(new CoinInsertedState(machine_)); // (1)
    }
    void selectItem() override {
        // do nothing, not valid in this state   // (2)
    }
    void dispense() override { }
};

class VendingMachine {
    State* current_;
    void processInput(Input in) {
        if (in == Input::Coin)   current_->coinInserted();
        if (in == Input::Select) current_->selectItem();
        if (in == Input::Dispense) {
            current_->dispense();
            current_ = new IdleState(*this);       // (3)
        }
    }
};
```

7.

A content management system has a document that moves through: DRAFT → IN_REVIEW → APPROVED → PUBLISHED → ARCHIVED. Valid transitions:

- DRAFT: submit() → IN_REVIEW; (delete is also legal but out of scope)
- IN_REVIEW: approve() → APPROVED; reject() → DRAFT
- APPROVED: publish() → PUBLISHED; reject() → DRAFT
- PUBLISHED: archive() → ARCHIVED
- ARCHIVED: no further transitions

Write the full implementation: the Document class (Context), the DocumentState interface, and all five ConcreteState classes with their complete method bodies. Illegal actions should print a clear message explaining why the action is not permitted in the current state.

8.

The refill(int count) method from Task 2 resets the machine to NoQuarterState unconditionally.

- Is this correct if the machine is in HasQuarterState when refill() is called? What should happen to the quarter that was inserted? Revise the method or the design to handle this correctly.
- Suppose the machine runs in a multi-threaded environment where one thread processes insertQuarter() calls and another calls refill(). Identify the data race. What is the minimal change needed to make the machine thread-safe? (You do not need to implement the full solution, describe it in prose and show the relevant declarations.)

9.

Extend the GumballMachine so that external observers can be notified whenever the machine changes state. Requirements:

- Add an Observer interface with a single method: void onStateChanged(std::string_view newState).
- GumballMachine should maintain a list of registered observers and notify all of them inside setState().
- Write a ConsoleDashboard class that implements Observer and prints a formatted status line whenever the machine changes state.
- Write a SalesLogger class that implements Observer and counts the number of times the machine enters SoldState (i.e., counts gumballs dispensed).

This combination, State Pattern for internal lifecycle management plus Observer Pattern for external notification, is extremely common in production systems.

10.

Choose one of the following systems and design a full State Pattern implementation for it. Deliver: state-transition diagram, Context class declaration, State interface, all ConcreteState class declarations, and a main() that exercises every state and every transition.

- An ATM: Idle → Card Inserted → PIN Entry → Menu → Transaction → Dispensing → Receipt → Idle (plus a Card Retained fault state).
- A game character: Idle → Walking → Running → Jumping → Attacking → Taking Damage → Dead → Respawning → Idle.
- A smart home door lock: Locked → Unlocking → Unlocked → Locking → Locked (plus Tamper Alert, which can be entered from Locked or Unlocked).