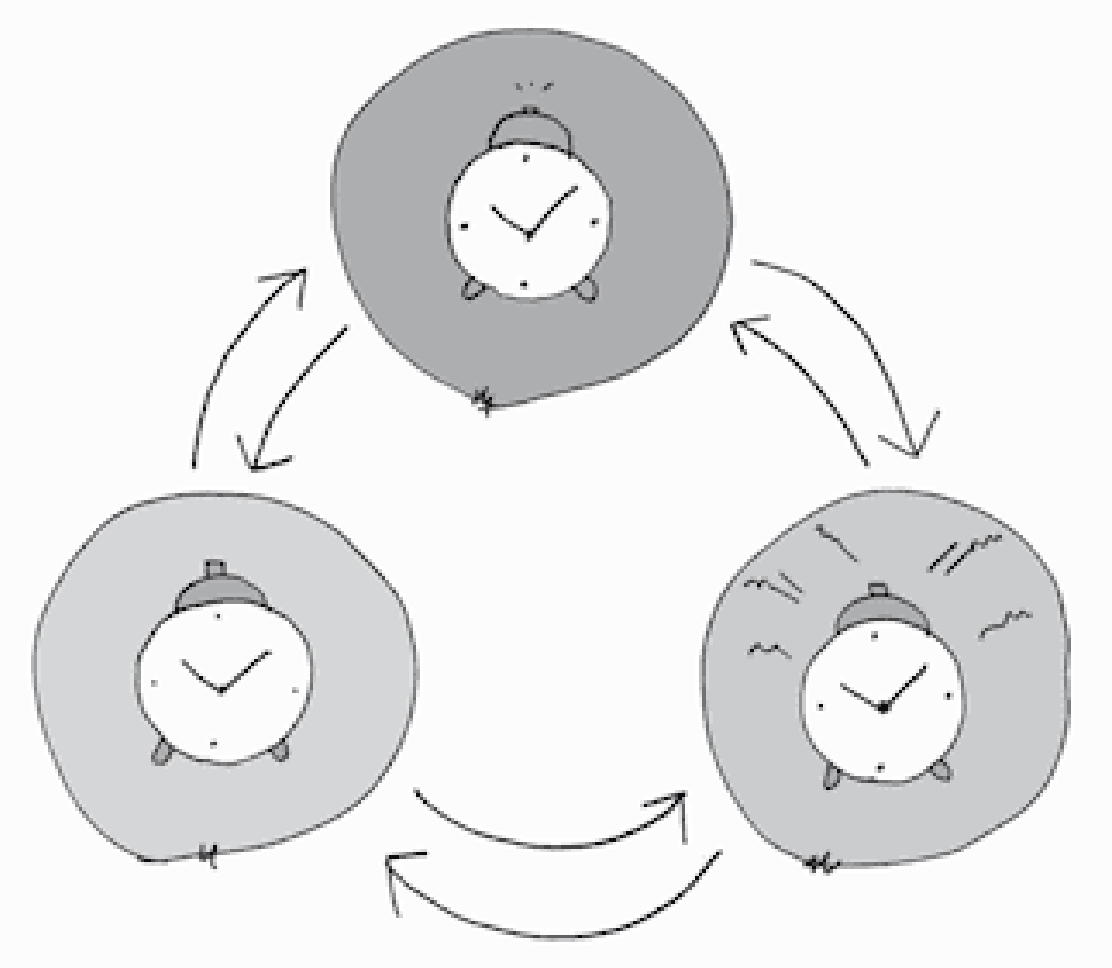
State Pattern

Allow Objects to Alter Behavior When Internal State Changes



State Pattern

Think of a **security system** that behaves completely differently based on the time of day:

- **Security system**: Day mode (normal use) vs Night mode (emergency alerts)
- **Traffic lights**: Red state (stop) vs Green state (go) vs Yellow state (caution)

The Problem

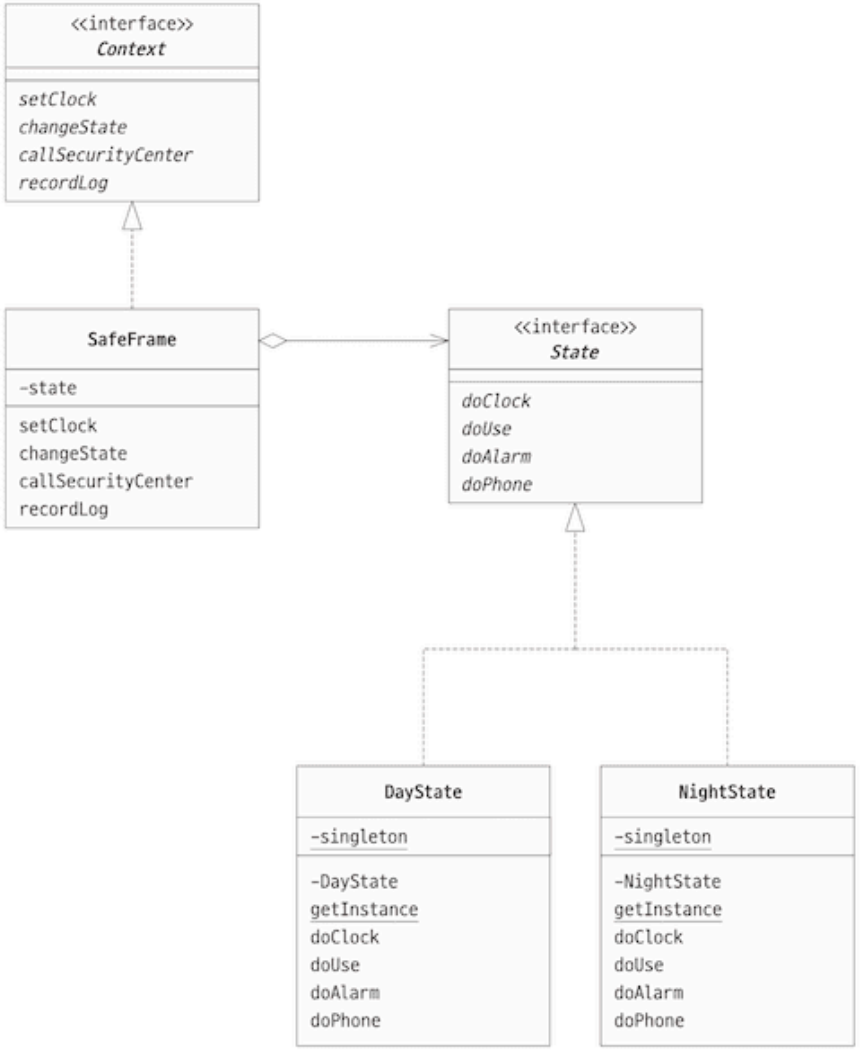
- We have a **security system** that must behave differently during **day** and **night**.
- **Same operations** (use safe, emergency alarm, phone calls) require **different responses** based on the current state.
- Traditional approach: large if-else statements scattered throughout code.

The challenge: how to make state-dependent behavior clean and maintainable?

The *State* as the Solution

- We **encapsulate** state-specific behavior in separate **state objects**.
- **Context** delegates behavior to the current **state object**.
- **State transitions** are handled by the states themselves.

The Solution (Design)



Step 1: Understand the Players

In this design, we have three key components:

- *State*: Defines an interface for encapsulating state-specific behavior
    - **ConcreteState** (DayState, NightState): Implements behavior for specific states

Context maintains the current state and delegates operations to other components.

- *Context* (SecuritySystem)

Step 2: State transitions are managed

- **States** decide when to transition to other states.
- **Context** provides an interface but delegates to the current state.

Step 3: State Pattern Key Points

Context: Maintains current state reference and delegates requests to it.

State Interface: Defines common operations (`do_use()` , `do_alarm()` , `do_phone()` , `do_clock()`).

Concrete States:

- Implement state-specific behavior
- Decide when to transition to other states
- Example: `DayState` (normal operations) vs `NightState` (security mode)

Code

Simple Traffic Controller

We have two states Red/Green and they switch to the other state with the change() method in the controller.

States

```
class TrafficLightState:  # State
    def handle(self, light):
        pass

class RedState(TrafficLightState):  # ConcreteState
    def handle(self, light):
        print("Red light - Stop!")
        light.set_state(GreenState())
```

With the `change()` method, states are updated to Green or Red.

```
# State manager
class TrafficLight:  # Context
    def __init__(self):
        self._state = RedState()

    def set_state(self, state):
        self._state = state

    def change(self):
        self._state.handle(self)
```

Complex Security System

Our security system behaves differently based on time:

Day State (9:00 - 17:00)

- ✅ Safe usage: Normal operation (logged)
- 🚨 Emergency alarm: Calls the security center
- 📞 Phone calls: Forwarded to security center

Night State (17:00 - 9:00)

- 🚨 Safe usage: **SECURITY ALERT** (unauthorized access!)
- 🚨 Emergency alarm: Calls the security center
- 🎙️ Phone calls: **Recorded** for security review

```python
from security_system import SecuritySystem

def main():
    # Create security system context
    system = SecuritySystem("Corporate Security System")

    # Test different times to show state transitions
    scenarios = [
        (10, "Morning - Business Hours"),
        (18, "Evening - After Hours"),
        (9, "Business Hours Resume")
    ]

    for hour, description in scenarios:
        print(f"=== {description} ({hour:02d}:00) ===")
        system.set_time(hour)  # May trigger state transition

        # Same operations, different behaviors!
        system.use_safe()
        system.make_phone_call()
        system.trigger_alarm()
```

Context Class (SecuritySystem)

SecuritySystem invokes the state methods.

```
class SecuritySystem:
    def __init__(self, name="Security System"):
        self.name = name
        self.state = DayState.get_instance()  # Initial state
        self.current_time = "00:00"
        self.log_entries = []

    def use_safe(self):
        print(f"User action: USE SAFE at {self.current_time}")
        self.state.do_use(self)  # Delegate to current state

    def make_phone_call(self):
        print(f"User action: PHONE CALL at {self.current_time}")
        self.state.do_phone(self)  # Delegate to current state

    def set_time(self, hour):
        self.current_time = f"{hour:02d}:00"
        self.state.do_clock(self, hour)  # May trigger transition
```

Key Points: Context Delegation

1. **Delegates everything**: All behavior goes to the current state object
2. **Simple interface**: Client just calls methods normally
3. **State management**: Provides methods for state transitions
4. **Logging support**: States can log actions through context

State Classes - Abstract Interface

```
from abc import ABC, abstractmethod

class State(ABC):
    @abstractmethod
    def do_use(self, context):
        """Handle safe usage - behavior varies by state"""
        pass

    @abstractmethod
    def do_alarm(self, context):
        """Handle emergency alarm - behavior varies by state"""
        pass

    @abstractmethod
    def do_phone(self, context):
        """Handle phone call - behavior varies by state"""
        pass

    @abstractmethod
    def do_clock(self, context, hour):
        """Handle time change - may trigger transitions"""
        pass
```

DayState Implementation

```
class DayState(State):
    def do_use(self, context):
        # Normal daytime behavior
        context.record_log("✅ Safe used (normal daytime operation)")

    def do_phone(self, context):
        # Forward calls to security
        context.call_security_center("📞 Normal call forwarded")

    def do_clock(self, context, hour):
        # Check for transition to night
        if hour < 9 or hour >= 17:
            print("⏰ Switching to NIGHT mode")
            context.change_state(NightState.get_instance())
```

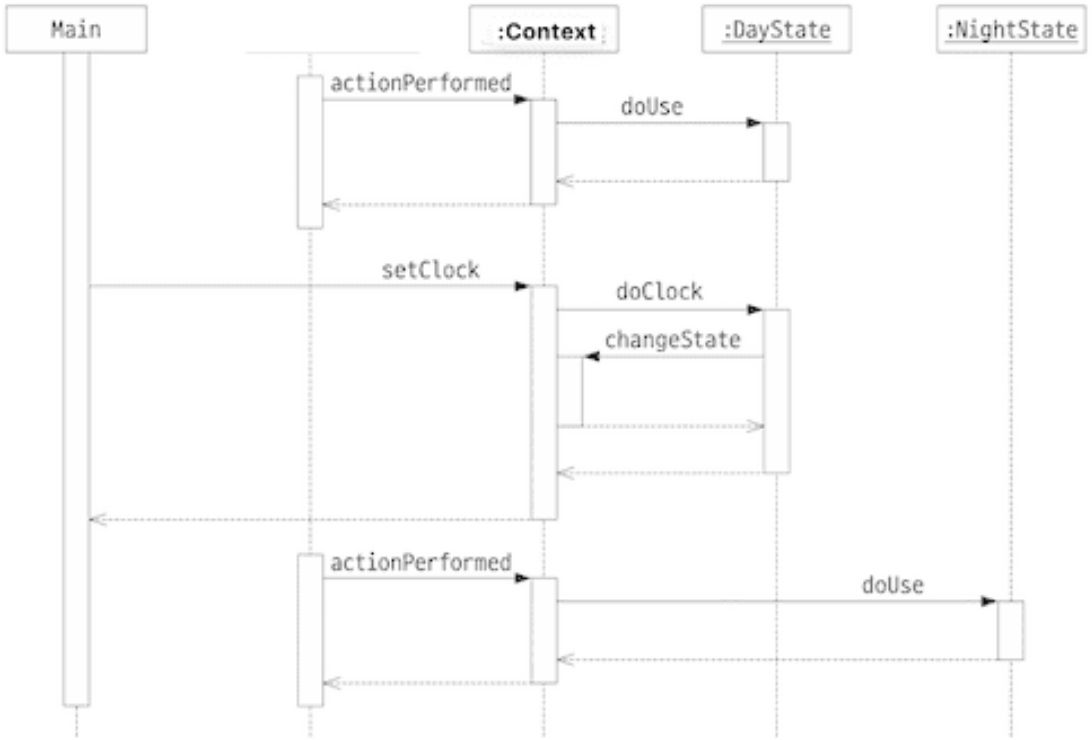
NightState Implementation

```
class NightState(State):
    def do_use(self, context):
        # DIFFERENT behavior - security alert!
        context.call_security_center("SECURITY ALERT: Unauthorized access!")

    def do_phone(self, context):
        # DIFFERENT behavior - record instead of forward
        context.record_log("Phone call recorded for review")

    def do_clock(self, context, hour):
        # Check for transition to day
        if 9 <= hour < 17:
            print("⏰ Switching to DAY mode")
            context.change_state(DayState.get_instance())
```

Sequence of Operations



1. **Client** calls method on **Context**
2. **Context** delegates to current **State** object
3. **State** executes state-specific behavior
4. **State** may trigger a transition to a different state
5. **Context** updates current state reference

Behavior Comparison

Same Method Call - Different Results:

```
# Day State (10:00)
system.use_safe()
# Output: ✅ Safe used (normal daytime operation)

# Night State (22:00)
system.use_safe()
# Output: 🚨 SECURITY ALERT: Unauthorized access!

# Same method, completely different behavior!
```

Discussion

Misunderstanding of the State

In State pattern, the Context class contains large conditional statements to handle different states.

Wrong!

State pattern eliminates conditionals by delegating behavior to state objects.

When to Use State

- When object behavior depends significantly on its state
- When operations have large conditional statements based on object state
- When state transitions are complex and need explicit management
- When you want to avoid duplicate state-checking code

Traditional vs State Pattern

Traditional Conditional Approach:

```
def handle_use(self):
    if self.current_time >= 9 and self.current_time < 17:
        self.log("Normal use")
    else:
        self.alert("Security alert!")
```

State Pattern Approach:

```
def handle_use(self):
    self.state.do_use(self)  # Behavior determined by state
```

Related Patterns

- **Strategy**: Similar structure but different intent
- **Singleton**: State objects are often implemented as singletons
- **Observer**: States might notify observers of changes

State vs Strategy

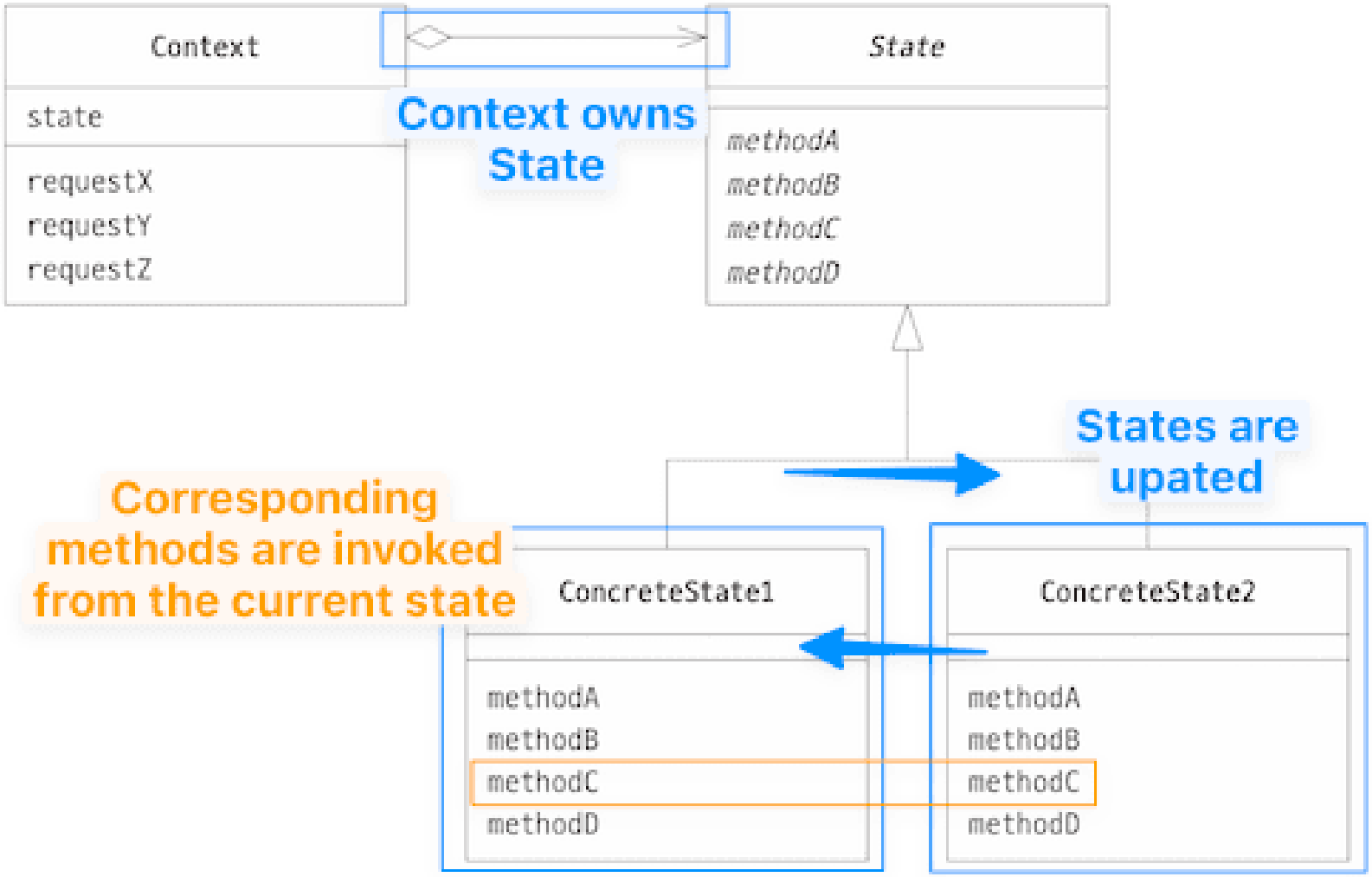
State Pattern:

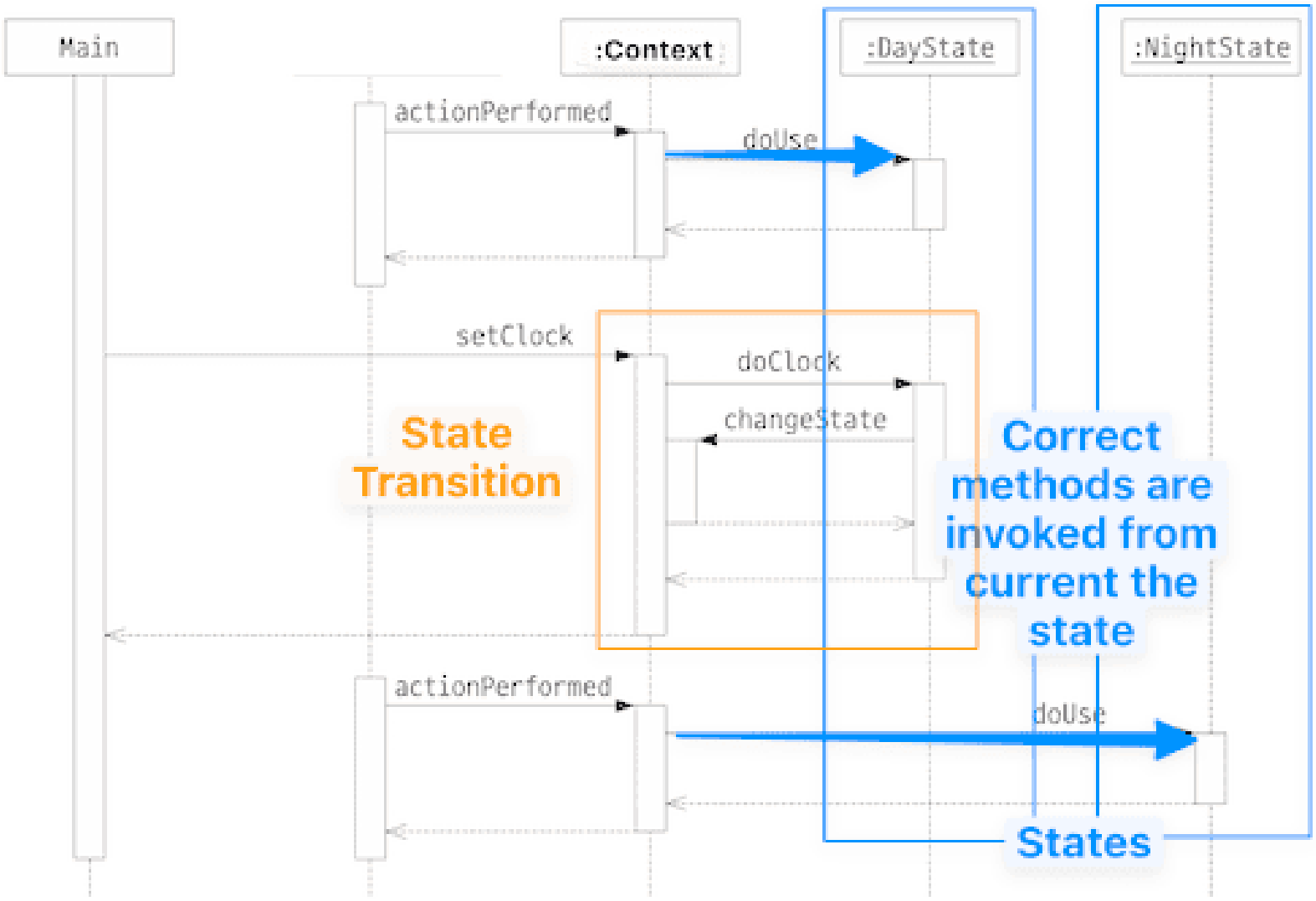
- **Context** aware of state changes, states control transitions
- **States** reference each other for transitions
- **Behavior changes** based on internal state evolution

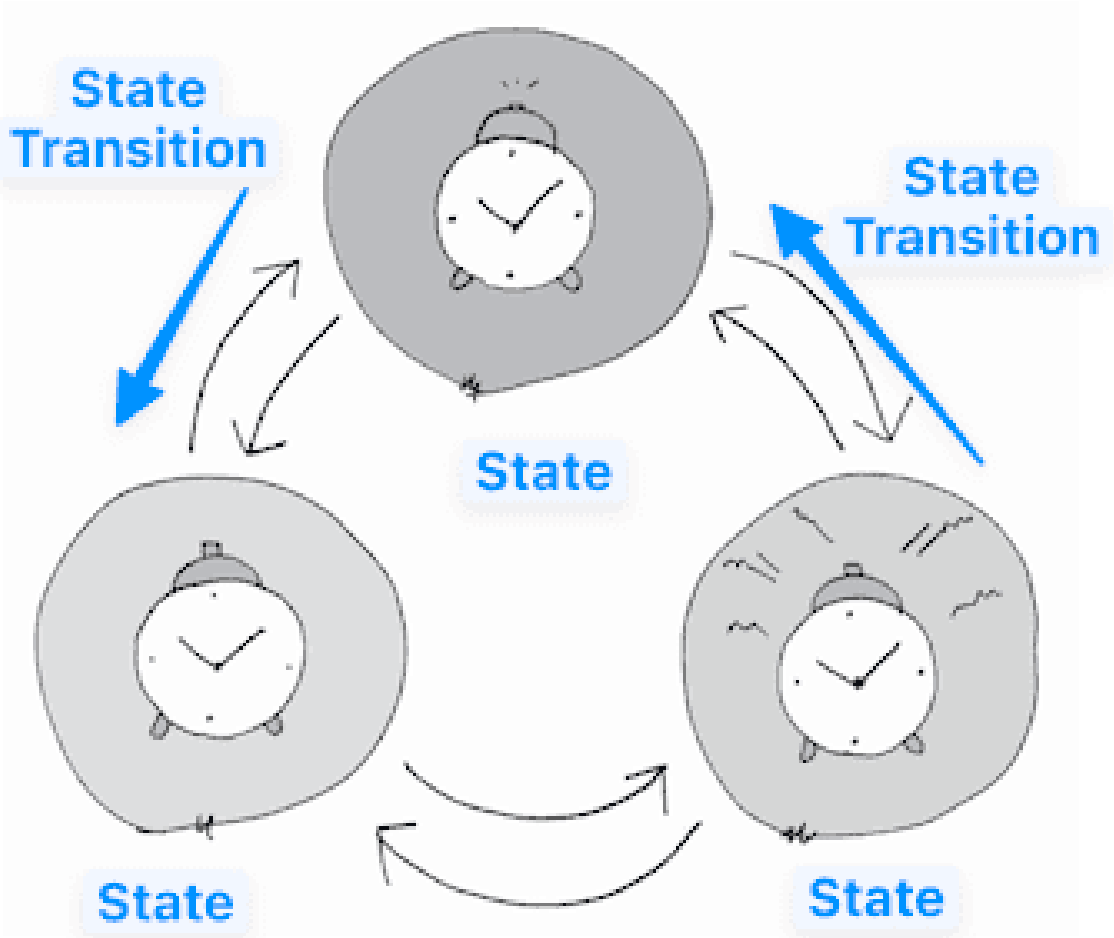
Strategy Pattern:

- **Context** chooses strategy, strategies are independent
- **Strategies** don't know about each other
- **Behavior changes** based on external configuration

UML


Context
state
requestX
requestY
requestZ
Context owns State
State
methodA
methodB
methodC
methodD
States are upated
Corresponding methods are invoked from the current state
ConcreteState1
methodA
methodB
methodC
methodD
ConcreteState2
methodA
methodB
methodC
methodD


Main
:Context
:DayState
:NightState
actionPerformed
doUse
setClock
doClock
changeState
State Transition
Correct methods are invoked from current the state
actionPerformed
doUse
States


State
Transition
State
Transition
State
State
State