Object-Oriented Programming
State Pattern

CSIE Department, NTUT
Woei-Kae Chen
State: Intent

- Allow an object to alter its behavior when its internal state changes.
  - The object will appear to change its class.
- Also known as: Object for States.
An object can be in one of several different states.
Example: Petri Net simulator

State: Motivation (2)

- SimulatorGUI
  - LBDown()
  - LBUp()
  - MouseMove()

- SimulatorState
  - LBDown()
  - LBUp()
  - MouseMove()

- SimulatorAddLink
  - LBDown()
  - LBUp()
  - MouseMove()

- SimulatorMove
  - LBDown()
  - LBUp()
  - MouseMove()

- SimulatorAutoFire
  - LBDown()
  - LBUp()
  - MouseMove()
State: Applicability

- Use the State pattern in either of the following cases
  - An object’s behavior depends on its state, and it must change its behavior at run-time depending on its state.
  - Operations have large, multipart conditional statements that depend on the object’s state.
State: Structure (1)

State as object
Similar to XXX?
State: Structure (2)

**State ↔ Strategy**

- **Context**
  - +ContextInterface()

- **Strategy**
  - +Algorithminterface()

- **ConcreteStrategyA**
  - +Algorithminterface()

- **ConcreteStrategyB**
  - +Algorithminterface()

- **ConcreteStrategyC**
  - +Algorithminterface()
State: Participants

- **Context (TCPConnection)**
  - defines the interface to client.
  - maintains an instance of a ConcreteState subclass.

- **State (TCPState)**
  - defines an interface for encapsulating the behavior associated with a particular state of the Context.

- **ConcreteState subclasses (TCPListen, etc.)**
  - each subclass implement a behavior associated with a state of the Context.
State: Collaborations

- Context delegates state-specific requests to the current ConcreteState object.
- A context may pass itself as an argument to the State object handling the request.
- Context is the primary interface for clients. Clients can configure a context with State objects.

Either Context or the ConcreteState subclasses can decide which state succeeds another and under what circumstances.
State: Consequences

- It localizes state-specific behavior and partitions behavior for different states.
  - avoids switch statements (if there are many states)
  - increase the number of classes

- It makes state transitions explicit.
  - separate objects for different states makes state transitions more explicit
  - states transitions are atomic (one variable; not several)
    → protect context from inconsistent internal states.

- State objects can be shared.
  - if State objects have no instance variables, then contexts can share a State object → Flyweight pattern
State: Implementation (1)

- **Who defines the state transitions?**
  - **Context**
    - if state transitions can be implemented entirely in the Context
  - **ConcreteState**
    - allow State subclasses to specify their successor state and make the transition by themselves
    - add an interface to the Context that lets State objects set the context’s current state
    - disadvantage: State subclasses have knowledge of other State subclasses ➔ dependency

- **A table-based alternative**
  - the table-driven approach focuses on defining state transitions
  - the State pattern models state-specific behavior
State: Implementation (2)

- Creating and destroying State objects.
  - Trade-off
    - create State objects ahead of time and never destroying them
    - create State objects only when they are needed and destroy them thereafter

- Using dynamic inheritance
  - changing the object’s class at run-time
    - not possible in most object-oriented languages
    - possible with Self and other delegation-based languages
State: Related patterns

- Flyweight pattern explains when and how State objects can be shared
- State objects are often Singletons
  - when ConcreteState perform state transitions
- Patterns using similar ideas (inheritance and polymorphism)
  - Command: command as object
  - Strategy: algorithm as object
  - Iterator: pointer as object
  - State: state as object
  - Composite: composite as object (with uniform interface)
  - Decorator: decorator as object (with uniform interface)
  - Proxy: proxy as object (with uniform interface)