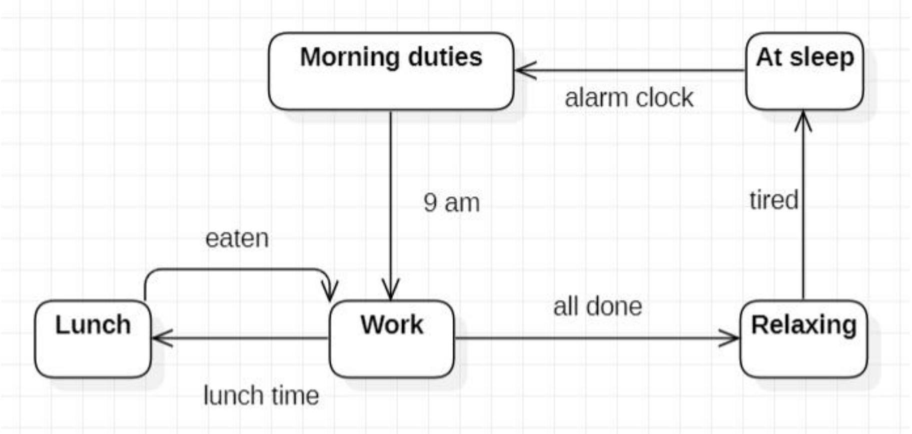
State

An object-behavioral pattern

Learning goals

1. Learn the idea, structure, and Java implementation of the State design pattern.
2. Learn to apply the State DP in your own programming.

Background: Finite State Machines



- The State DP is for situations where the object's state can be represented as a **Finite State Machine** (FSM).
- A finite state machine consists of a set of states, events, and transitions.
- **States**: The state machine comprises different states that represent the system's state or phase.
  - In this example, there are five states.
  - One of the states is the **initial state** where the system is at the time of startup (not specified in the diagram)
- Transitions: Transitions represent state changes that are triggered by **events**.
  - For instance, a transition from the **Relaxing** state into the **At sleep** state is triggered by the **tired** event

Examples of FSM in software

- Player character states in video games:
  - normal Mario, Super Mario, Fire Mario, etc
- Network connection states:
  - connection established, connection opened, connection closed, connection error, etc.
- UI component states:
  - active, inactive, selected,deselected, etc.
- Automatic control systems:
  - startup, running, shutdown, fault, etc.
- Robotics control states:
  - forward motion, turning, charging, executing tasks, etc.

Idea of State DP

- When there are states involved, a trivial solution would be to code the state changes and the state-specific behavior using many if/else of switch/case statements.
- Complex, messy code!

```
public void changeLight() {
   if (currentColor.equals("red")) {
      currentColor = "red_yellow";
   } else if (currentColor.equals("red_yellow")) {
      currentColor = "green";
   } else if (currentColor.equals("green")) {
      currentColor = "yellow";
   } else if (currentColor.equals("yellow")) {
      currentColor = "red";
   } else {
      System.out.println("Invalid traffic light color!");
   }
}
```

✗

```
public void displayLight() {
   if (currentColor.equals("red")) {
      System.out.println("The traffic light is red.");
   } else if (currentColor.equals("red_yellow")) {
      System.out.println("The traffic light is red-yellow.");
   } else if (currentColor.equals("green")) {
      System.out.println("The traffic light is green.");
   } else if (currentColor.equals("yellow")) {
      System.out.println("The traffic light is yellow.");
   } else {
      System.out.println("Invalid traffic light color!");
   }
}
```

Idea of State

- In the State DP, each State is represented as a subclass of the State abstract class.
- The State superclass specifies the operations whose implementation may change from one subclass to another.
- The objects state is expresses by a reference to one of the State subclasses. This reference may change at runtime, causing the object's behavior to change.

Example

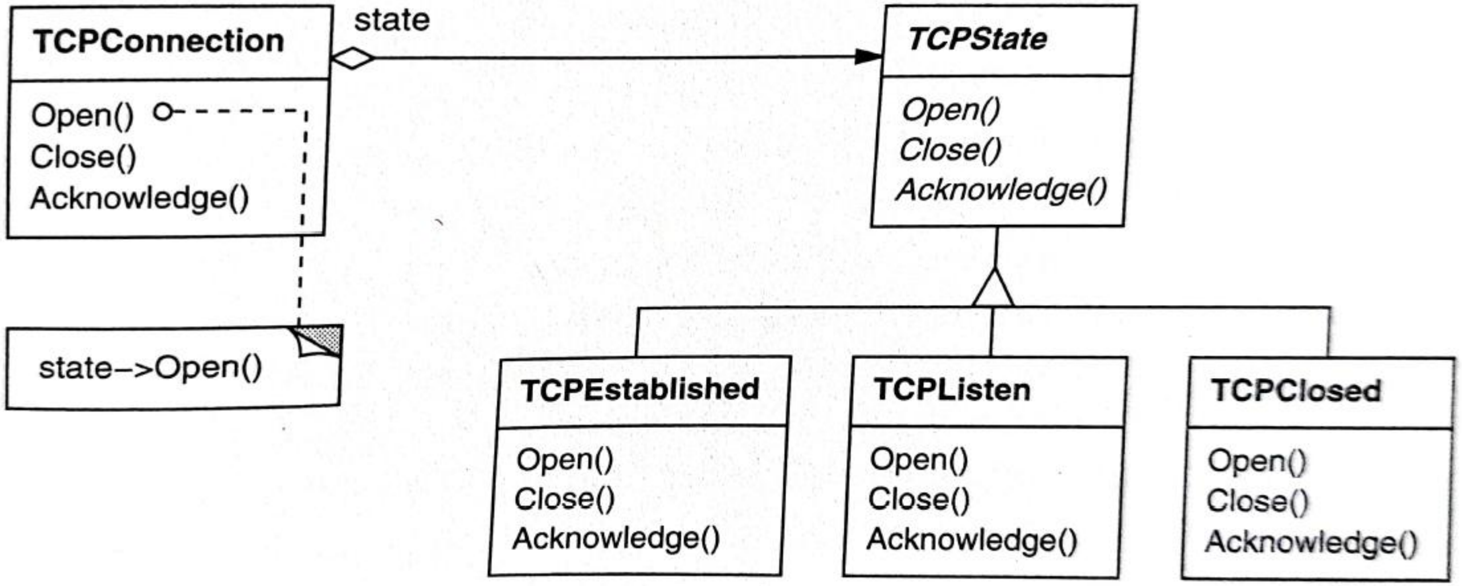


Image: Gamma et al., Design Patterns. Elements of Reusable Object-Oriented Software. Addison Wesley Longman (1995), p. 305

Example

- In the example, the TCP connection can be in three possible states.
- The behavior of **open()**, **close()** and **acknowledge()** depends on the current state.
- The TCPConnection might have, in addition, not state-specific behavior, which is coded in the TCPConnection class itself.

General structure

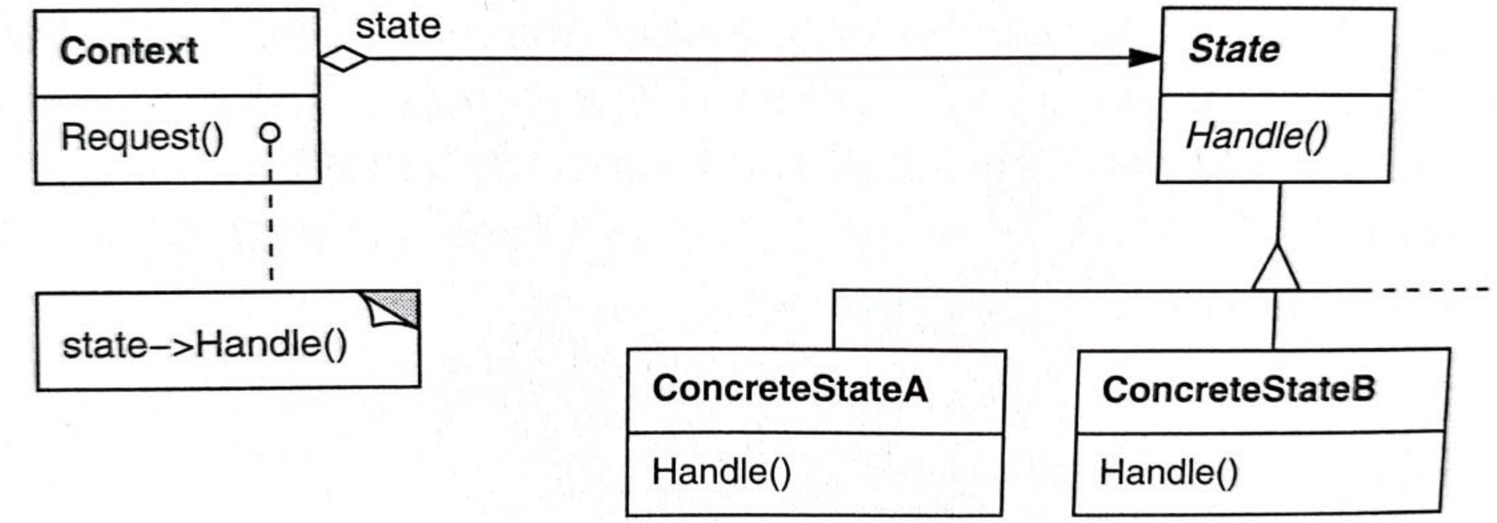


Image: Gamma et al., Design Patterns. Elements of Reusable Object-Oriented Software. Addison Wesley Longman (1995), p. 306

Roles

- **Context**: Maintains the reference to the current state. Is used by the client.
- **State:** Declares the state-specific methods.
- **Concrete State**: implements one of the states and its state-specific behaviour.

Practical issues

- The State DP encapsulates the state-specific behaviour. It becomes easy to add new states (good expandability) or remove existing ones.
- The client should only deal with Context, not directly with states.
- The DP makes states and transitions explicit. The states are represented as classes, not just variable values.
- The state objects can be created *ad hoc* or as at once as the execution starts.
  - This may sometimes be relevant from the resources’ point of view.
- In most cases, the State subclasses are made responsible of state changes.