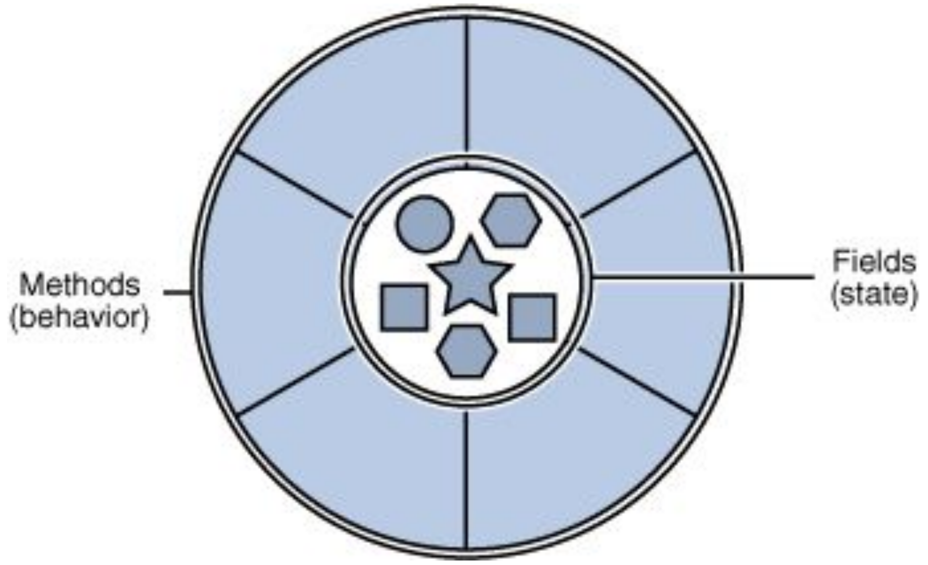
State Pattern



Intent

- Allows an object to alter its behavior when its internal state changes. The object will appear to change its class.

Applicability

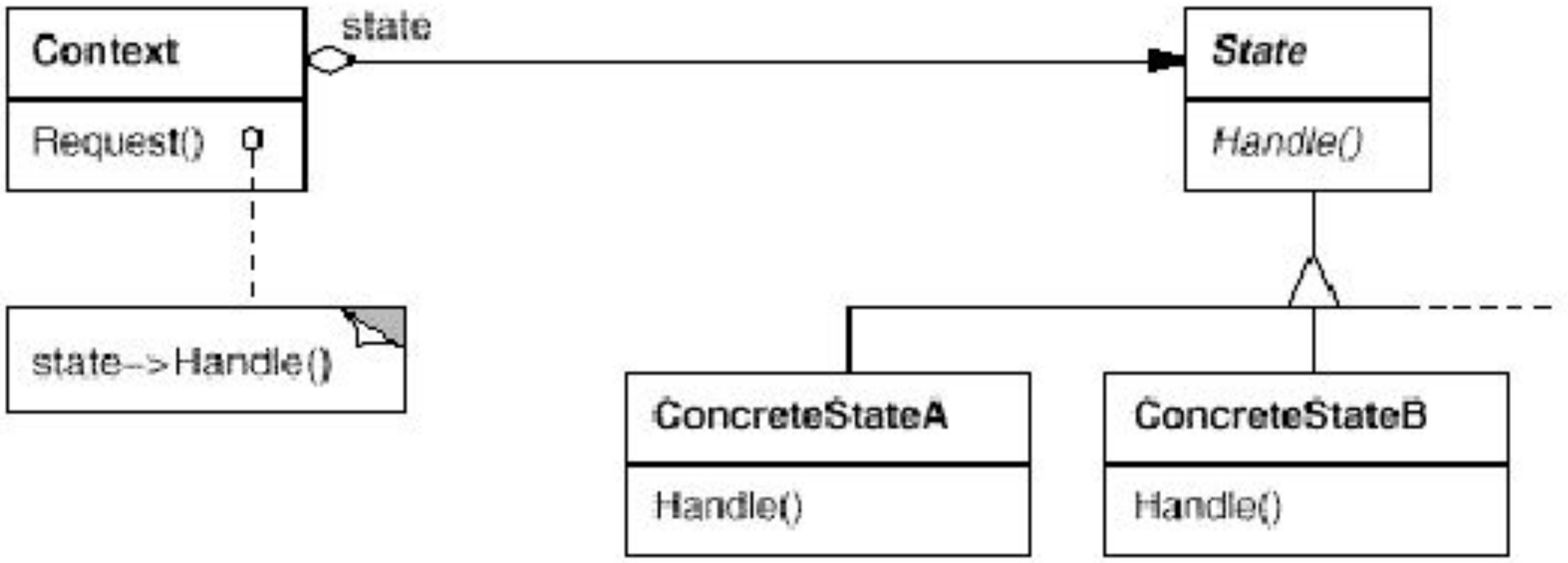
- An object’s behavior depends on its state, and it must change its behavior at run-time depending on that state.
- Operations have large, multipart conditional statements that depend on the object’s state. This state usually represented by one or more enumerated constants. The State pattern puts each branch of the conditional in a separate class. This lets you treat the object’s state as an object in its own right that can vary independently from other objects.

When to use STATE pattern ?

- State pattern is useful when there is an object that can be in one of several states, with different behavior in each state.
- To simplify operations that have large conditional statements that depend on the object's state.

```
if (myself = happy) then
{
    eatIceCream();
    ….
    }
else if (myself = sad) then
{
    goToPub();
    ….
   }
else if (myself = ecstatic)
    then
{
    ….
```

Structure


Context
Request()
state
state->Handle()
State
Handle()
ConcreteStateA
Handle()
ConcreteStateB
Handle()

How is STATE pattern implemented ?

- "Context" class:

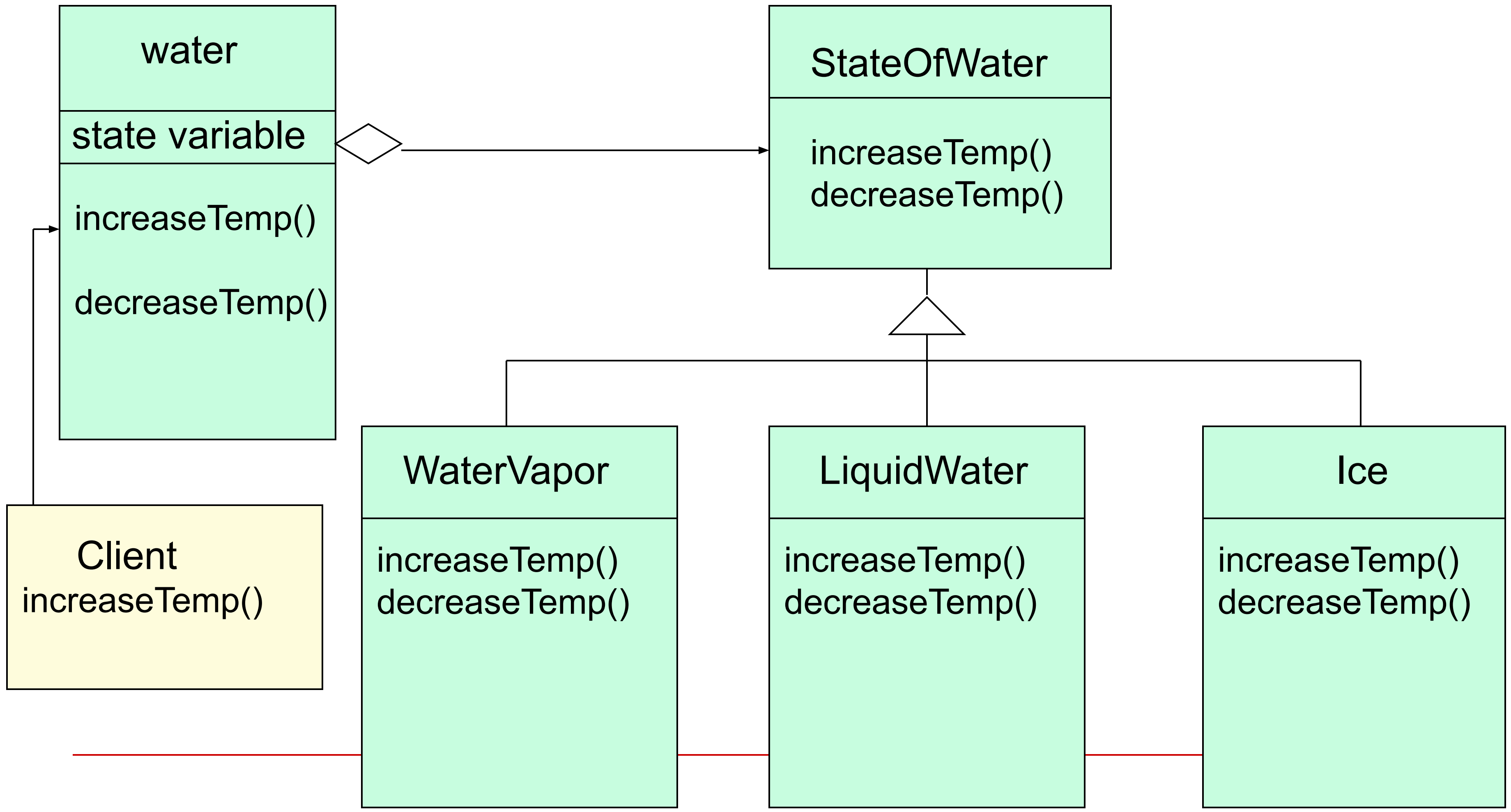
  Represents the interface to the outside world.
- "State" abstract class:

  Base class which defines the different states of the "state machine".
- "Derived" classes from the State class:

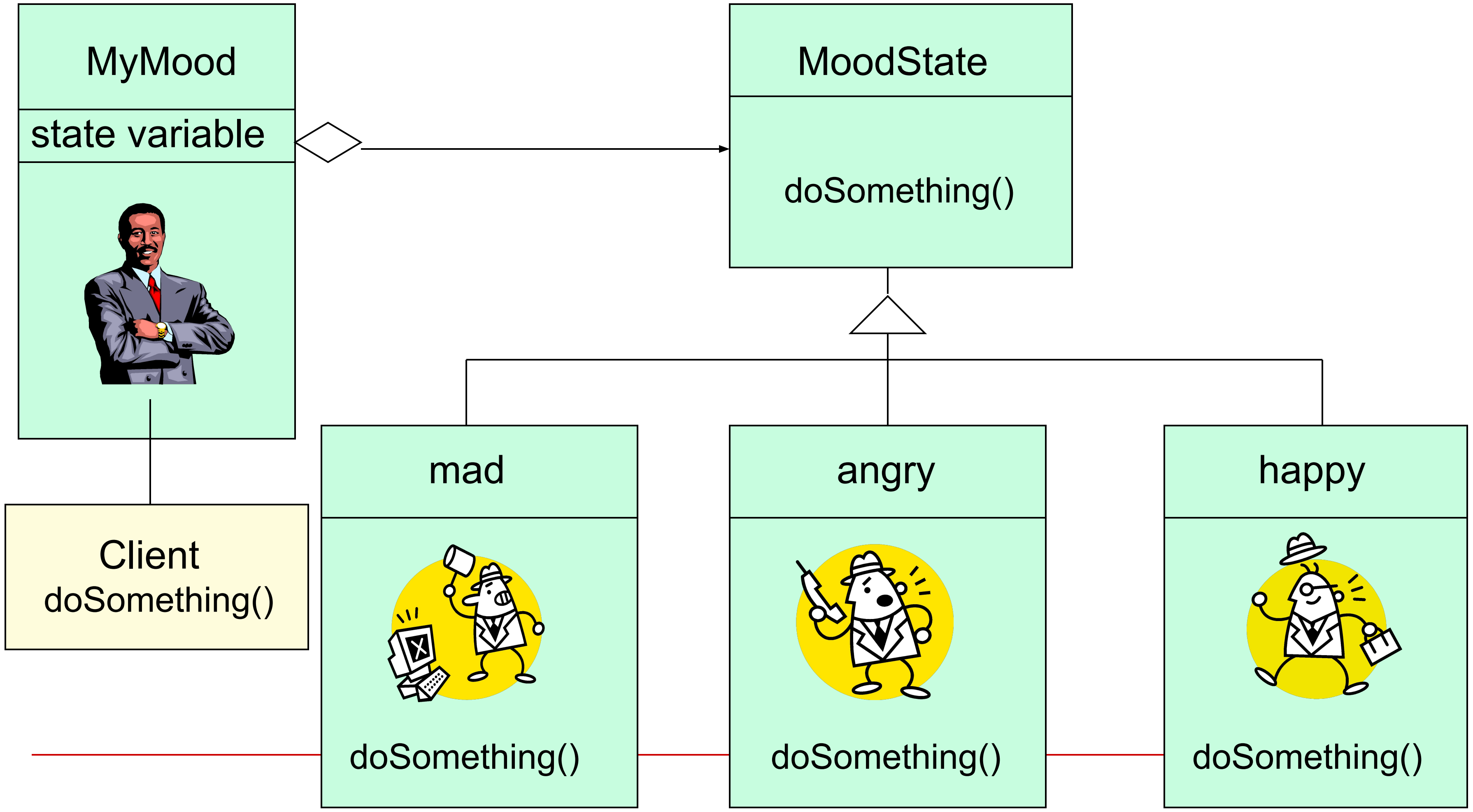
  Defines the true nature of the state that the state machine can be in.

Context class maintains a pointer to the current state. To change the state of the state machine, the pointer needs to be changed.

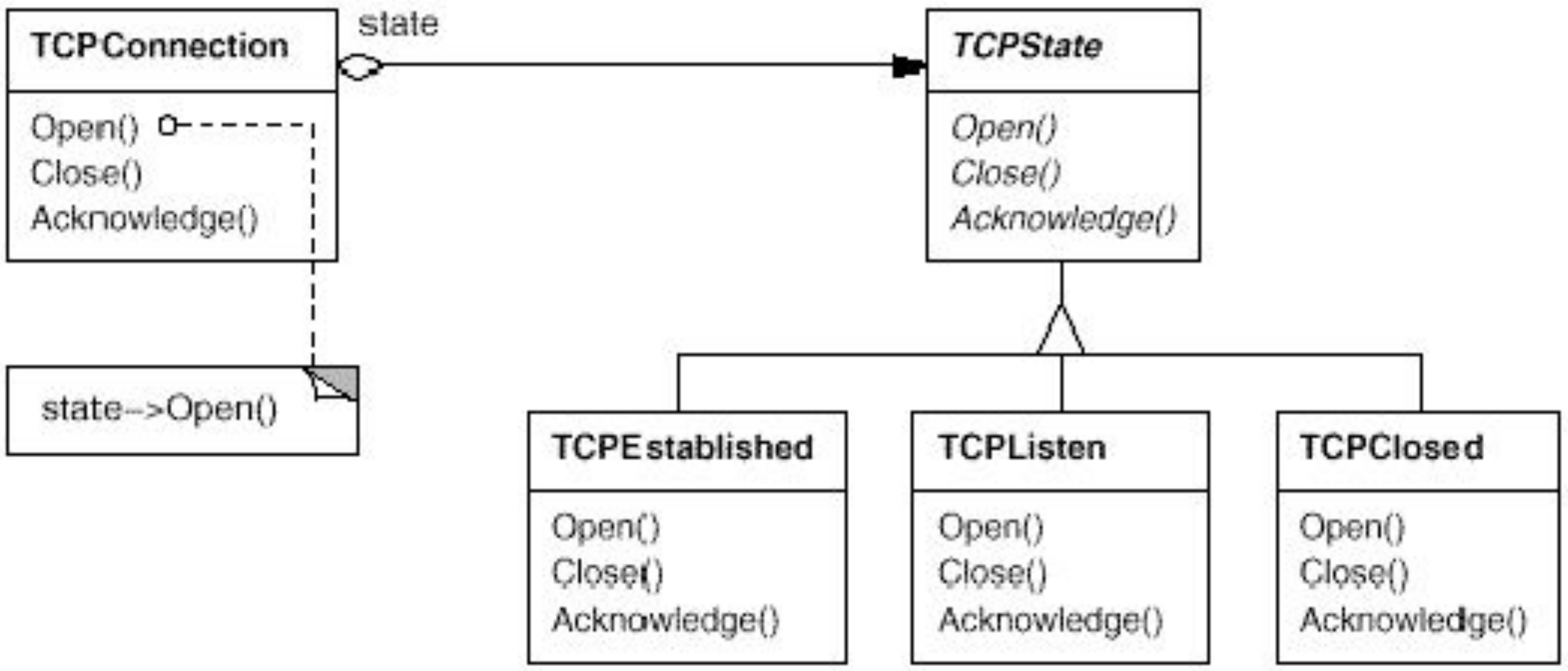
Example I


water
state variable
increaseTemp()
decreaseTemp()
StateOfWater
increaseTemp()
decreaseTemp()
Client
increaseTemp()
WaterVapor
increaseTemp()
decreaseTemp()
LiquidWater
increaseTemp()
decreaseTemp()
Ice
increaseTemp()
decreaseTemp()

Example II


MyMood
state variable
MoodState
doSomething()
Client
doSomething()
mad
doSomething()
angry
doSomething()
happy
doSomething()

Example III


TCPConnection
Open()
Close()
Acknowledge()
state
state->Open()
TCPState
Open()
Close()
Acknowledge()
TCPEstablished
Open()
Close()
Acknowledge()
TCPListen
Open()
Close()
Acknowledge()
TCPClosed
Open()
Close()
Acknowledge()

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. This lets the State object access the context if necessary.
- Context is the primary interface for clients. Clients can configure a context with State objects. Once a context is configured, its clients don’t have to deal with the State objects directly.
- Either Context or the ConcreteState subclasses can decide which state succeeds another and under what circumstances.

Benefits of using STATE pattern

- **Localizes all behavior associated with a particular state into one object.**
  - New state and transitions can be added easily by defining new subclasses.
  - Simplifies maintenance.

- **It makes state transitions explicit.**
  - Separate objects for separate states makes transition explicit rather than using internal data values to define transitions in one combined object.

- **State objects can be shared.**
  - Context can share State objects if there are no instance variables.

Implementation issues

- **To have a monolithic single class or many subclasses ?**
  - Increases the number of classes and is less compact.
  - Avoids large conditional statements.

- **Where to define the state transitions ?**
  - If criteria is fixed, transition can be defined in the context.
  - More flexible if transition is specified in the State subclass.
  - Introduces dependencies between subclasses.

- **Whether to create State objects as and when required or to create-them-once-and-use-many-times ?**
  - First is desirable if the context changes state infrequently.
  - Later is desirable if the context changes state frequently.

Known Uses

- Popular interactive drawing programs use the State pattern to perform specific operations based on which State(tool) is selected.

Related Patterns

- The Flyweight pattern explains when and how State objects can be shared.
- State Objects are often Singletons.

Code – Finite State Machine

```
class FSM {
      State state;
      public FSM(State s) {
            state = s;
      }
      public void move(char c) {
            state = state.move(c);
      }
      public boolean accept() {
            return state.accept();
      }
}
public interface State {
      State move(char c);
      boolean accept();
}
```

Code – Finite State Machine

```
class State2 implements State {
    static State2 instance = new State2();
    private State2() {}
    public State move (char c) {
        switch (c) {
            case 'a': return State1.instance;
            case 'b': return State2.instance;
            default: throw new IllegalArgumentException();
        }
    }
    public boolean accept() {
        return true;
    }
}
```

Code – Finite State Machine

```
class State1 implements State {
    static State1 instance = new State1();
    private State1() {}
    public State move (char c) {
        switch (c) {
            case 'a': return State2.instance;
            case 'b': return State1.instance;
            default: throw new IllegalArgumentException();
        }
    }
    public boolean accept() {
        return false;
    }
}
```