**State**

# Intent

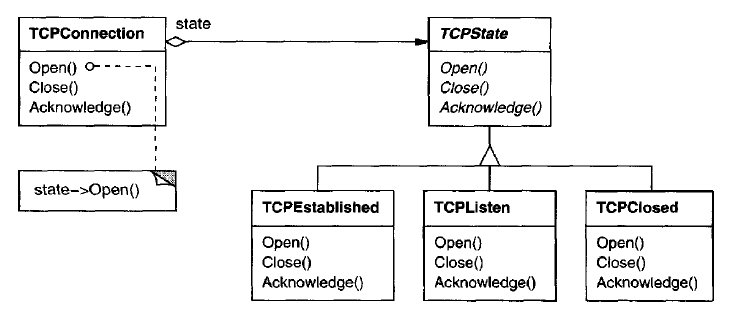
Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.

# Also Known As

Objects for States

# Motivation

Consider a class TCPConnection that represents a network connection. A TCPConnection object can be in one of several different states: Established, Listening, Closed. When a TCPConnection object receives requests from other objects, it responds differently depending on its current state.



The key idea in this pattern is to introduce an abstract class called TCPState to represent the states of the network connection. The TCPState class declares an interface common to all classes that represent different operational states. Subclasses of TCPState implement state-specific behavior.

The class TCPConnection maintains a state object (an instance of a subclass of TCPState) that represents the current state of the TCPconnection. The class Connection delegates all state-specific requests to this state object. TCPConnection uses its TCPState subclass instance to perform operations particular to the state of the connection.

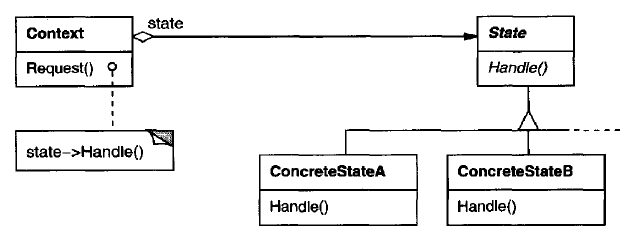
Whenever the connection changes state, the TCPConnection object changes the state object it uses. When the connection goes from established to closed, for example, TCPConnection will replace its TCPEstablished instance with a TCPClosed instance.

# 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 that state.
* Operations have large, multipart conditional statements that depend on the object's state. This state is usually represented by one or more enumerated constants. Often, several operations will contain this same conditional structure. 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.

# Structure



# Participants

**Context (TCPConnection)**

* defines the interface of interest to clients
* maintains an instance of a ConcreteState subclass that defines the current state

**State (TCPState)**

* defines an interface for encapsulating the behavior associated with a particular state of the Context.

**ConcreteState subclasses (TCPEstablished, TCPListen, TCPClosed)**

* each subclass implements a behavior associated with a state of the Context

# 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.

# Consequences

The State pattern has the following consequences:

## It localizes state-specific behavior and partitions behavior for different states.

The State pattern puts all behavior associated with a particular state into one object. Because all state-specific code lives in a State subclass, new states and transitions can be added easily by defining new subclasses.

The State pattern introduce a problem, because the pattern distributes behavior for different states across several State subclasses. This increases the number of classes and is less compact than a single class. But such distribution is actually good if there are many states, which would otherwise necessitate large conditional statements.

## It makes state transitions explicit

When an object defines its current state solely in terms of internal data values, its state transitions have no explicit representation; they only show up as assignments to some variables. Introducing separate objects for different states makes the transitions more explicit.

## State objects can be shared.

If State objects have no instance variables—that is, the state they represent is encoded entirely in their type—then contexts can share a State object. When states are shared in this way, they are essentially flyweights (see Flyweight) with no intrinsic state, only behavior.

# Implementation

The State pattern raises a variety of implementation issues:

## Who defines the state transitions?

The State pattern does not specify which participant defines the criteria for state transitions. If the criteria are fixed, then they can be implemented entirely in the Context.

It is generally more flexible and appropriate, however, to let the State subclasses themselves specify their successor state and when to make the transition. This requires adding an interface to the Context that lets State objects set the Context's current state explicitly. Decentralizing the transition logic in this way makes it easy to modify or extend the logic by defining new State subclasses.

A **disadvantage** of decentralization is that one State subclass will have knowledge of at least one other, which introduces implementation dependencies between subclasses

## A table-based alternative.

In C++ Programming Style, Cargill describes another way to impose structure on state-driven code: He uses tables to map inputs to state transitions. For each state, a table maps every possible input to a succeeding state. In effect, this approach converts conditional code (and virtual functions, in the case of the State pattern) into a table look-up.

The main advantage of tables is their regularity: You can change the transition criteria by modifying data instead of changing program code. There are some **disadvantages**, however:

* A table look-up is often less efficient than a (virtual) function call.
* Putting transition logic into a uniform, tabular format makes the transition criteria less explicit and therefore harder to understand.
* It's usually difficult to add actions to accompany the state transitions.

The table-driven approach captures the state s and their transitions, but it must be augmented to perform arbitrary computation on each transition. The key difference between table-driven state machines and the State pattern can be summed up like this: The State pattern models state-specific behavior, whereas the table-driven approach focuses on defining state transitions.

## Creating and destroying State objects

A common implementation trade-off

1. to create State objects only when they are needed and destroy them thereafter versus
2. creating them ahead of time and never destroying them.

The first choice is preferable when the states that will be entered aren't known at run-time, and contexts change state infrequently. This is important if the State objects store a lot of information.

The second approach is better when state changes occur rapidly, in which case you want to avoid destroying states, because they may be needed again shortly. Instantiation costs are paid once up-front, and there are no destruction costs at all. This approach might be inconvenient, though, because the Context must keep references to all states that might be entered.

## Using dynamic inheritance

Changing the object's class at run-time, is not possible in many object-oriented programming languages. Exceptions include Self and other delegation-based languages.

Objects in Self can delegate operations to other objects to achieve a form of dynamic inheritance. Changing the delegation target at run-time effectively changes the inheritance structure. This mechanism lets objects change their behavior and amounts to changing their class.

# Sample Code

Code file <03_Behavioral_Patterns_09_State.cpp>

#include <iostream>

class Switch;

class State {

public:

virtual ~State() = default;

virtual void setOn(Switch \* swtch) = 0;

virtual void setOff(Switch \* swtch) = 0;

virtual void display() = 0;

virtual void changeState(Switch \* swtch, State & state);

};

class Switch {

State \* swtch\_state;

friend class State;

void setState(State & state) {

swtch\_state = &state;

}

public:

Switch();

void on() {

swtch\_state->setOn(this);

}

void off() {

swtch\_state->setOff(this);

}

void currentState() {

swtch\_state->display();

}

};

class OnState : public State {

OnState() = default;

public:

virtual ~OnState() = default;

static OnState & instance();

virtual void setOn(Switch \* );

virtual void setOff(Switch \* swtch);

virtual void display() { std::cout << "ON\n"; }

};

class OffState : public State {

OffState() = default;

public:

virtual ~OffState() = default;

static OffState & instance();

virtual void setOn(Switch \* swtch);

virtual void setOff(Switch \* );

virtual void display() { std::cout << "OFF\n"; }

};

void State::changeState(Switch \* swtch, State & state) {

swtch->setState(state);

}

OnState & OnState::instance() {

static OnState on\_state;

return on\_state;

}

void OnState::setOn(Switch \* ) {

std::cout << "Already in ON state\n";

}

void OnState::setOff(Switch \* swtch) {

std::cout << "Setting OFF from ON\n";

//swtch->setState(OffState::instance());

changeState(swtch, OffState::instance());

}

OffState & OffState::instance() {

static OffState off\_state;

return off\_state;

}

void OffState::setOn(Switch \* swtch) {

std::cout << "Setting ON from OFF\n";

//swtch->setState(OnState::instance());

changeState(swtch, OnState::instance());

}

void OffState::setOff(Switch \* ) {

std::cout << "Already in OFF state\n";

}

Switch::Switch() : swtch\_state(&OffState::instance()) {}

int main() {

Switch swtch; swtch.currentState();

swtch.on(); swtch.currentState();

swtch.off(); swtch.currentState();

swtch.off(); swtch.currentState();

swtch.on(); swtch.currentState();

swtch.on(); swtch.currentState();

return 0;

}

Output:

OFF

Setting ON from OFF

ON

Setting OFF from ON

OFF

Already in OFF state

OFF

Setting ON from OFF

ON

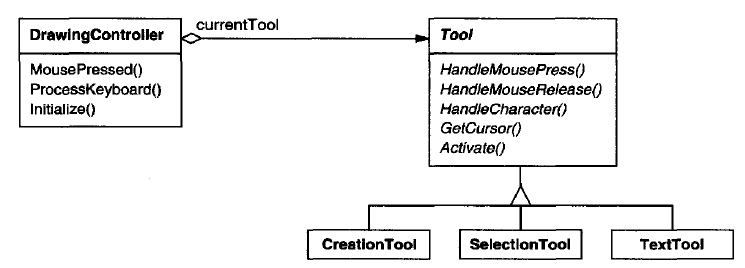
Already in ON state

ON

# Known Uses

Johnson and Zweig characterize the State pattern and its application to TCP connection protocols.

This technique is used in both the HotDraw and Unidraw drawing editor frameworks . It allows clients to define new kinds of tools easily. In HotDraw, the DrawingController class forwards the requests to the current Tool object. In Unidraw, the corresponding classes are Viewer and Tool. The following class diagram sketches the Tool and DrawingController interfaces:



# Related Patterns

The Flyweight pattern explains when and how State objects can be shared.

State objects are often Singletons.

# References

Book: Design Patterns Elements of Reusable Object-Oriented Software