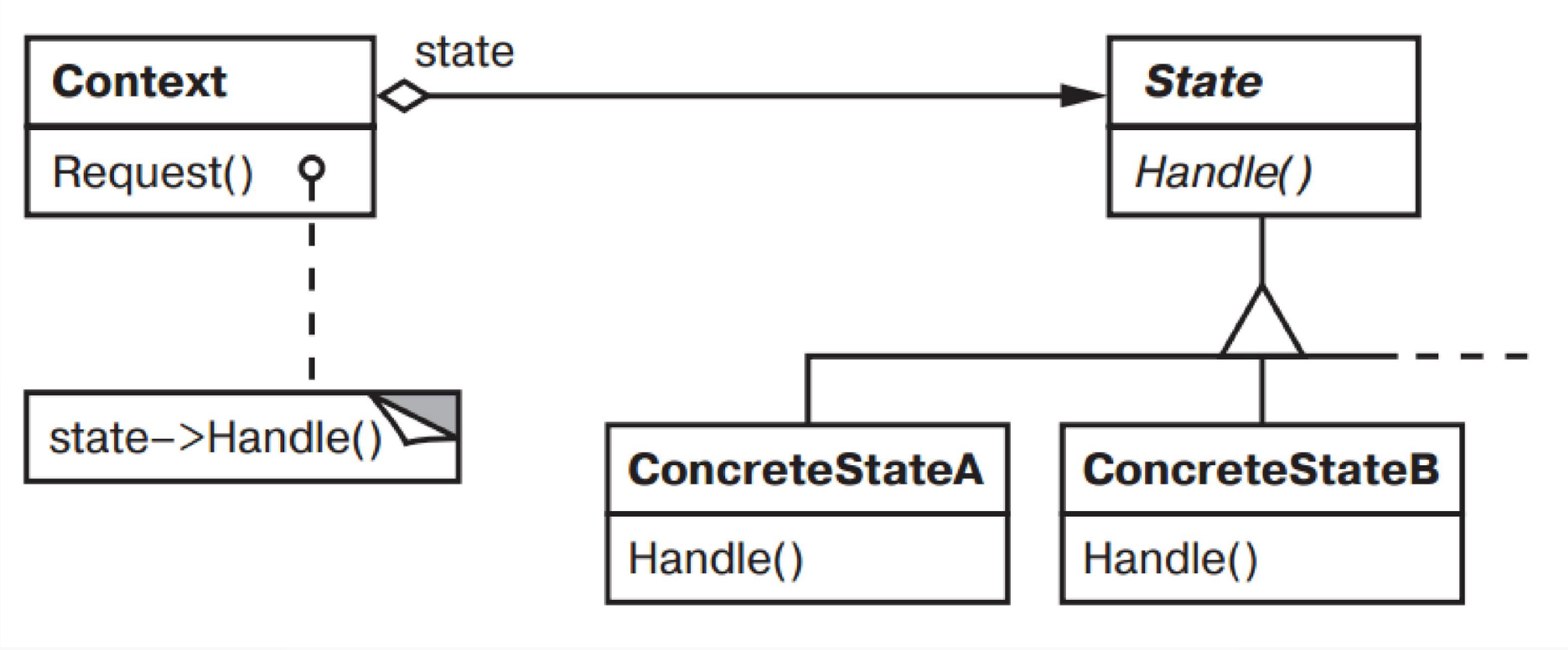
Паттерн State

Выполнил:
студент 1 курса магистратуры
05182м группы
Василевский А.В.

Описание

- Известен как **State (Состояние)**.
- Поведенческий паттерн.
- Позволяет вынести зависящее от конкретного состояния поведение в отдельный класс. Переход между состояниями сводится к замене объекта текущего состояния на другой.

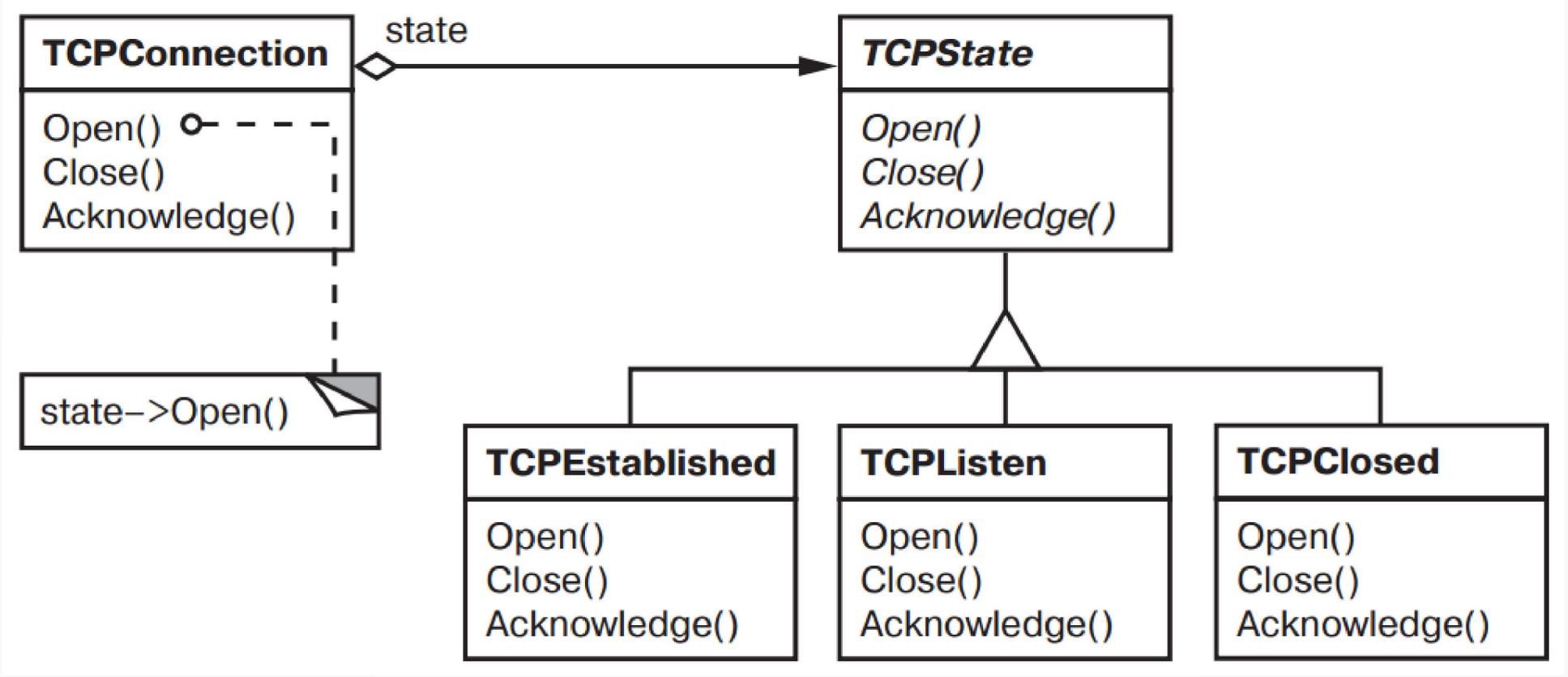
Структура


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

Участники

- **Context** (TCPConnection) – контекст:
  - определяет интерфейс, представляющий интерес для клиентов;
  - хранит экземпляр подкласса ConcreteState, которым определяется текущее состояние;
- **State** (TCPState) – состояние:
  - определяет интерфейс для инкапсуляции поведения, ассоциированного с конкретным состоянием контекста Context;
- Подклассы **ConcreteState** (TCPEstablished, TCPListen, TCPClosed) – конкретное состояние:
  - каждый подкласс реализует поведение, ассоциированное с некоторым состоянием контекста Context.

Пример


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

Результаты

- Локализует зависящее от состояния поведение и делит его на части, соответствующие состояниям.
- Делает явными переходы между состояниями.
- Делает атомарными переходы между состояниями.
- Объекты состояния можно разделять.

Реализация

```
interface TCPOctetStream {...}

interface TCPState {
    TCPState transmit(TCPConnection c, TCPOctetStream s);
    TCPState open(TCPConnection c);
    TCPState close(TCPConnection c);
    TCPState synchronize(TCPConnection c);
    TCPState acknowledge(TCPConnection c);
    TCPState send(TCPConnection c);
}

static class TCPConnection {

    private TCPState state;

    public TCPConnection() {
        this.state = TCPEstablished.getInstance();
    }

    public void open()        { setState(state.open(this)); }
    public void close()       { setState(state.close(this)); }
    public void send()        { setState(state.send(this)); }
    public void acknowledge() { setState(state.acknowledge(this)); }
    public void synchronize() { setState(state.synchronize(this)); }
    public void processOctet(TCPOctetStream s) { /* do something */ }

    private void setState(TCPState s) {
        // atomic store if needed
        this.state = s;
    }
};
```

```
static class TCPEstablished implements TCPState {
    private static final TCPState instance = new TCPEstablished();
    public static TCPState getInstance() { return instance; }

    @Override
    public TCPState transmit(TCPConnection c, TCPOctetStream s) {
        c.processOctet(s);
        return this;
    }

    { implementation }
};

static class TCPListen implements TCPState {
    private static final TCPState instance = new TCPListen();
    public static TCPState getInstance() { return instance; }

    @Override
    public TCPState send(TCPConnection c) {
        // send SYN, receive SYN, ACK, etc.
        return TCPEstablished.getInstance();
    }

    { implementation }
};

static class TCPClosed implements TCPState {
    private static final TCPState instance = new TCPClosed();
    public static TCPState getInstance() { return instance; }

    @Override
    public TCPState open(TCPConnection c) {
        // send SYN, receive SYN, ACK, etc.
        return TCPEstablished.getInstance();
    }

    { implementation }
};
```

Без State

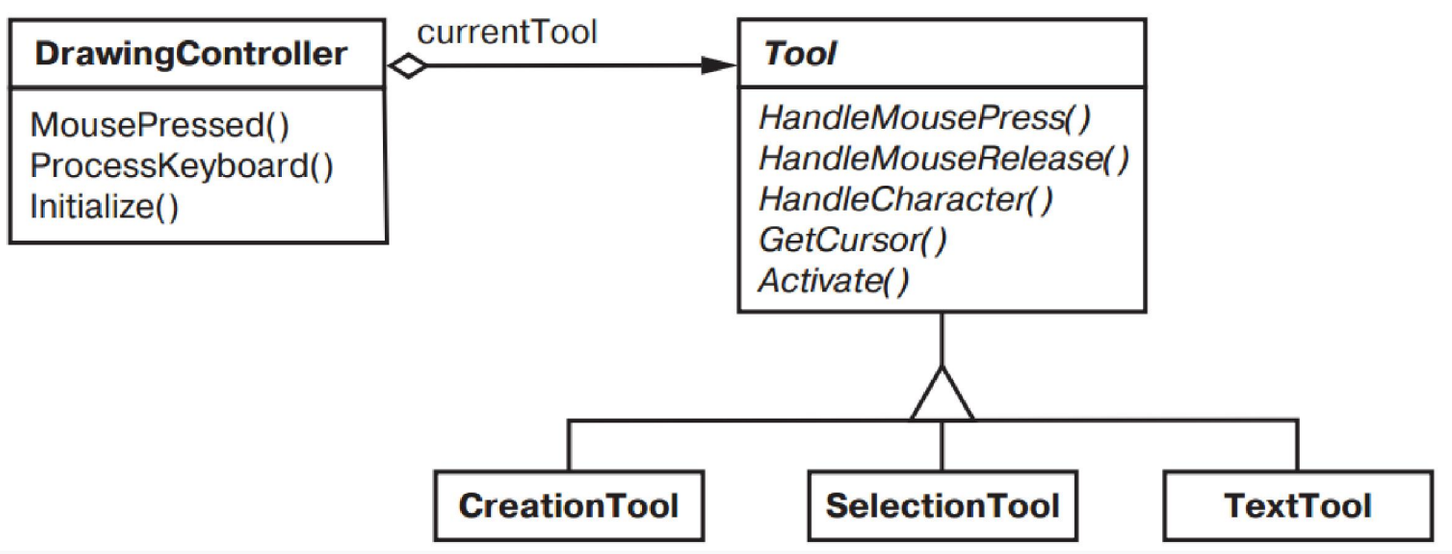
```
interface TCPOctetStream {...}

enum TCPState {
    ESTABLISHED, LISTEN, CLOSED
}

static class TCPConnection {

    private TCPState state;

    public TCPConnection() {
        this.state = TCPState.ESTABLISHED;
    }

    public void open() {
        if (this.state != TCPState.CLOSED)
            throw new IllegalStateException("should be closed");
        // send SYN, receive SYN, ACK, etc.
        this.state = TCPState.ESTABLISHED;
    }
    public void close() {
        if (this.state == TCPState.CLOSED)
            throw new IllegalStateException("should not be closed");
        // actually close connection
        this.state = TCPState.CLOSED;
    }
    public void send() {
        if (this.state != TCPState.LISTEN)
            throw new IllegalStateException("should listen");
        // send SYN, receive SYN, ACK, etc.
        this.state = TCPState.ESTABLISHED;
    }
    public void acknowledge() { /* */ }
    public void synchronize() { /* */ }
    public void processOctet(TCPOctetStream s) { /* do something */
};
```

Вопросы реализации

- Что определяет переходы между состояниями. Варианты: только Context, сами состояния (более гибкий, но появляются зависимости между подклассами состояний).
- Табличная альтернатива. Менее гибкий. Акцент на переходах, а не на зависящем от состояния поведении.
- Создание и уничтожение объектов состояния. Варианты: заранее и навсегда, при необходимости.
- Использование динамического наследования.

Применения

- Работа с TCP и другими stateful-протоколами обмена.
- Графические редакторы. Текущее поведение курсора определяется выбранным из палитры инструментом (состояние – инструмент).



Известные применения

- Ральф Джонсон и Джонатан Цвейг характеризуют паттерн состояние и описывают его применительно к протоколу TCP.
- Графические редакторы предоставляют «инструменты» для выполнения операций прямым манипулированием. Можно определить абстрактный класс Tool (реализация паттерна State), подклассы которого реализуют зависящее от инструмента поведение. Данная техника используется в каркасах графических редакторов HotDraw и Unidraw.

Родственные паттерны

- Паттерн **приспособленец** (**Flyweight**) подсказывает, как и когда можно разделять объекты класса State.
- Объекты класса State часто бывают **одиночками** (**Singleton**).

Литература

- [1] Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. Приемы объектно-ориентированного проектирования. Паттерны проектирования. — СПб.: Питер, 2015. — 368 с.: ил. — (Серия«Библиотека программиста»).