Состояние (State) — это поведенческий паттерн проектирования, который позволяет объектам менять поведение в зависимости от своего состояния. Извне создаётся впечатление, что изменился класс объекта.

Шаблоны бывают следующих трех видов:

1. Порождающие.
2. Структурные.
3. **Поведенческие** — о них мы рассказываем в этой статье.

Простыми словами: Поведенческие шаблоны связаны с распределением обязанностей между объектами. Их отличие от структурных шаблонов заключается в том, что они не просто описывают структуру, но также описывают шаблоны для передачи сообщений / связи между ними. Или, другими словами, они помогают ответить на вопрос «Как запустить поведение в программном компоненте?»

Поведенческие шаблоны — шаблоны проектирования, определяющие алгоритмы и способы реализации взаимодействия различных объектов и классов.

Например. Автомобиль может стоять на месте, а может двигаться по шоссе. В обоих случаях это один и тот же объект (один и тот же автомобиль), но в разных состояниях его поведение будет отличаться.

Пример из жизни: Допустим, в графическом редакторе вы выбрали кисть. Она меняет своё поведение в зависимости от настройки цвета, т. е. рисует линию выбранного цвета.

Простыми словами: Шаблон позволяет менять поведение класса при изменении состояния.

Данный паттерн призван сделать более гибкой архитектуру систем, которые описывают подобные объекты и манипулируют ими.

Проблема

Паттерн Состояние невозможно рассматривать в отрыве от концепции машины состояний, также известной как *стейт-машина* или *конечный автомат*.

Конечный автомат — абстрактный автомат, число возможных внутренних состояний которого конечно.

Основная идея в том, что программа может находиться в одном из нескольких состояний, которые всё время сменяют друг друга. Набор этих состояний, а также переходов между ними, предопределён и *конечен*.

Находясь в разных состояниях, программа может по-разному реагировать на одни и те же события, которые происходят с ней.

Такой подход можно применить и к отдельным объектам. Например, объект Документ может принимать три состояния: Черновик, Модерация или Опубликован. В каждом из этих состоянии метод опубликовать будет работать по-разному:

- Из черновика он отправит документ на модерацию.
- Из модерации — в публикацию, но при условии, что это сделал администратор.
- В опубликованном состоянии метод не будет делать ничего.

Основная проблема такой машины состояний проявится в том случае, если в Документ добавить ещё десяток состояний. Каждый метод будет состоять из увесистого условного оператора, перебирающего доступные состояния. Такой код крайне сложно поддерживать. Малейшее изменение логики переходов заставит вас перепроверять работу всех методов, которые содержат условные операторы машины состояний.

Путаница и нагромождение условий особенно сильно проявляется в старых проектах. Набор возможных состояний бывает трудно предопределить заранее, поэтому они всё время добавляются в процессе эволюции программы. Из-за этого решение, которое выглядело простым и эффективным в самом начале разработки, может впоследствии стать проекцией большого макаронного монстра.

Решение
Паттерн Состояние предлагает создать отдельные классы для каждого состояния, в котором может пребывать объект, а затем вынести туда поведения, соответствующие этим состояниям.

Вместо того, чтобы хранить код всех состояний, первоначальный объект, называемый *контекстом*, будет содержать ссылку на один из объектов-состояний и делегировать ему работу, зависящую от состояния.

Документ делегирует работу своему активному объекту-состоянию. Благодаря тому, что объекты состояний будут иметь общий интерфейс, контекст сможет делегировать работу состоянию, не привязываясь к его классу. Поведение контекста можно будет изменить в любой момент, подключив к нему другой объект-состояние.

Очень важным нюансом, отличающим этот паттерн от Стратегии, является то, что и контекст, и сами конкретные состояния могут знать друг о друге и инициировать переходы от одного состояния к другому.

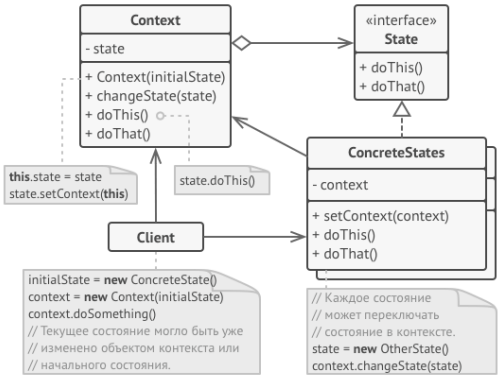
Аналогия из жизни
Ваш смартфон ведёт себя по-разному, в зависимости от текущего состояния:

- Когда телефон разблокирован, нажатие кнопок телефона приводит к каким-то действиям.
- Когда телефон заблокирован, нажатие кнопок приводит к экрану разблокировки.
- Когда телефон разряжен, нажатие кнопок приводит к экрану зарядки.

Преимущества и недостатки

- Избавляет от множества больших условных операторов машины состояний.
- Концентрирует в одном месте код, связанный с определённым состоянием.
- Упрощает код контекста.

- Может неоправданно усложнить код, если состояний мало и они редко меняются.



Контекст хранит ссылку на объект состояния и делегирует ему часть работы, зависящей от состояний. Контекст работает с этим объектом через общий интерфейс состояний. Контекст должен иметь метод для присваивания ему нового объекта-состояния.
Состояние описывает общий интерфейс для всех конкретных состояний.

Конкретные состояния реализуют поведения, связанные с определённым состоянием контекста. Иногда приходится создавать целые иерархии классов состояний, чтобы обобщить дублирующий код.

Состояние может иметь обратную ссылку на объект контекста. Через неё не только удобно получать из контекста нужную информацию, но и осуществлять смену его состояния.

И контекст, и объекты конкретных состояний могут решать, когда и какое следующее состояние будет выбрано. Чтобы переключить состояние, нужно подать другой объект-состояние в контекст.

- Context – контекст. объект, который должен варьировать своё поведение в зависимости от состояния;
- State – состояние. Определяет интерфейс для инкапсуляции поведения соответствующего конкретному состоянию объекта Context. Экземпляр этого класса хранится в закрытом поле state класса Context;
- ConcretteState – конкретное состояние. Классы наследники State, которые реализуют поведения соответствующего конкретному состоянию объекта Context.

Применимость

Когда у вас есть объект, поведение которого кардинально меняется в зависимости от внутреннего состояния, причём типов состояний много, и их код часто меняется.

Паттерн предлагает выделить в собственные классы все поля и методы, связанные с определёнными состояниями. Первоначальный объект будет постоянно ссылаться на один из объектов-состояний, делегируя ему часть своей работы. Для изменения состояния в контекст достаточно будет подставить другой объект-состояние.

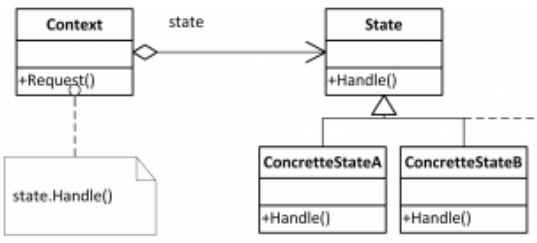
Когда код класса содержит множество больших, похожих друг на друга, условных операторов, которые выбирают поведения в зависимости от текущих значений полей класса.

Паттерн предлагает переместить каждую ветку такого условного оператора в собственный класс. Тут же можно поселить и все поля, связанные с данным состоянием.

Паттерн Состояние позволяет реализовать иерархическую машину состояний, базирующуюся на наследовании. Вы можете отнаследовать похожие состояния от одного родительского класса и вынести туда весь дублирующий код.

Отношения с другими паттернами

- **Мост**, **Стратегия** и **Состояние** (а также слегка и **Адаптер**) имеют схожие структуры классов — все они построены на принципе «композиции», то есть делегирования работы другим объектам. Тем не менее, они отличаются тем, что решают разные проблемы. Помните, что паттерны — это не только рецепт построения кода определённым образом, но и описание проблем, которые привели к данному решению.
- **Состояние** можно рассматривать как надстройку над **Стратегией**. Оба паттерна используют композицию, чтобы менять поведение основного объекта, делегируя работу вложенным объектам-помощникам. Однако в *Стратегии* эти объекты не знают друг о друге и никак не связаны. В *Состоянии* сами конкретные состояния могут переключать контекст.



```
class LampStateBase():
    """Состояние лампы"""

class GreenLampState(LampStateBase):
    def get_color(self):
        return 'Green'

class RedLampState(LampStateBase):
    def get_color(self):
        return 'Red'

class BlueLampState(LampStateBase):
    def get_color(self):
        return 'Blue'

class Lamp():
    def __init__(self):
        self._current_state = None
        self._states = self.get_states()

    def get_states(self):
        return [GreenLampState(), RedLampState(), BlueLampState()]

    def next_state(self):
        if self._current_state is None:
            self._current_state = self._states[0]
        else:
            index = self._states.index(self._current_state)
            if index < len(self._states) - 1:
                index += 1
            else:
                index = 0
            self._current_state = self._states[index]
        return self._current_state

    def light(self):
        state = self.next_state()
        print (state.get_color())

lamp = Lamp()
[lamp.light() for i in range(3)] (# Green Red Blue)
```