

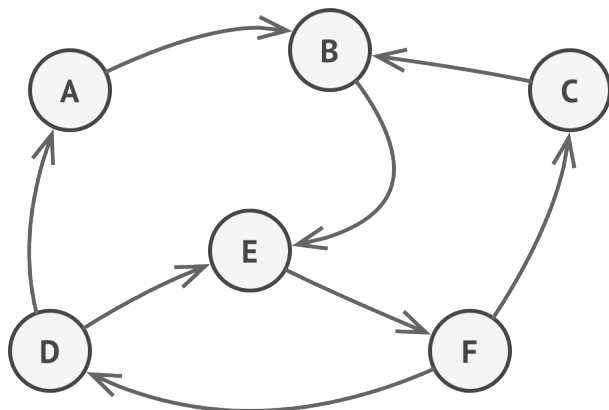
# СОСТОЯНИЕ

*Также известен как: State*

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

## ☹ Проблема

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



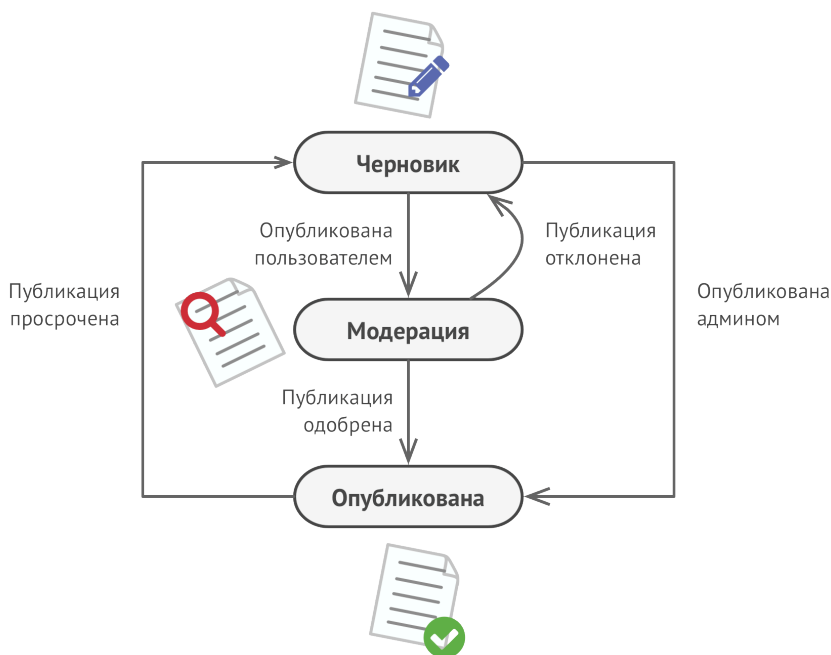
*Конечный автомат.*

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 может быть применён и к отдельным объектам. Например, объект `Документ` может принимать три состояния: `Черновик`, `Модерация` или `Опубликован`. В

каждом из них его метод `опубликовать` будет работать по-разному:

- В первом случае, он отправит документ на модерацию.
- Во втором — отправит документ в публикацию, но при условии, что это сделал администратор.
- А в последнем — и вовсе ничего не будет делать.



*Возможные состояния страницы и переходы между ними.*

Машину состояний чаще всего реализуют с помощью множества условных операторов, `if` либо `switch`, которые проверяют текущее состояние объекта и выполняют соответствующее поведение. Наверняка вы уже реализовали хотя бы одну машину состояний в своей

жизни, даже не зная об этом. Как на счёт вот такого кода, выглядит знакомо?

```
1 class Document
2     string state;
3     // ...
4     method publish() {
5         switch (state) {
6             "draft":
7                 state = "moderation";
8                 break;
9             "moderation":
10                if (currentUser.role == 'admin')
11                    state = "published"
12                break;
13            "published":
14                // Do nothing.
15        }
16    }
17    // ...
```

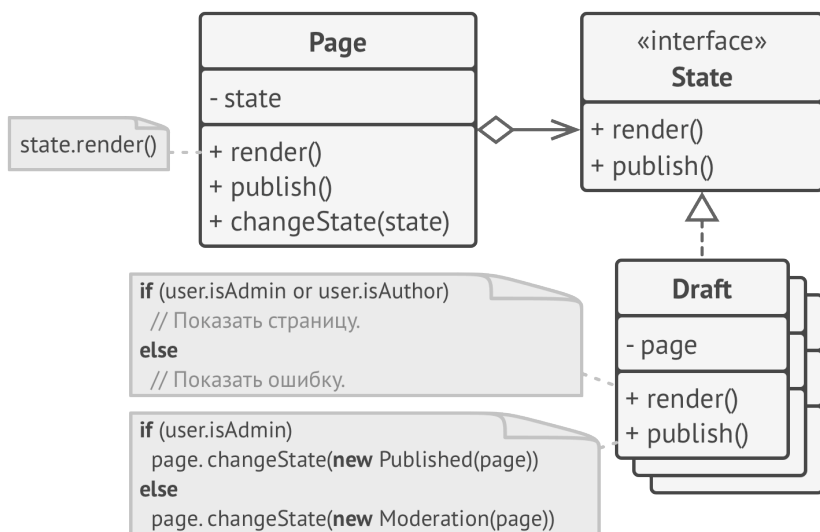
Основная проблема машины состояний построенной таким образом проявится, если в Документ добавить ещё десяток состояний. Каждый метод будет состоять из увесистого условного оператора, перебирающего доступные состояния.

Такой код крайне сложно поддерживать, так как любое изменение логики переходов влечёт за собой путешествие по всем методам и их условным операторам в поисках веток, которые изменились.

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. Таким образом, то, что выглядело простым и эффективным в самом начале, может впоследствии стать проекцией большого макаронного монстра.

## 😊 Решение

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



*Страница делегирует выполнение своему активному состоянию.*

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.

Благодаря тому, что состояния будут иметь общий интерфейс, контекст сможет делегировать работу состоянию, не привязываясь к его классу. Состояние и поведение контекста можно будет изменить в любой момент, подключив к нему другой объект-состояние.

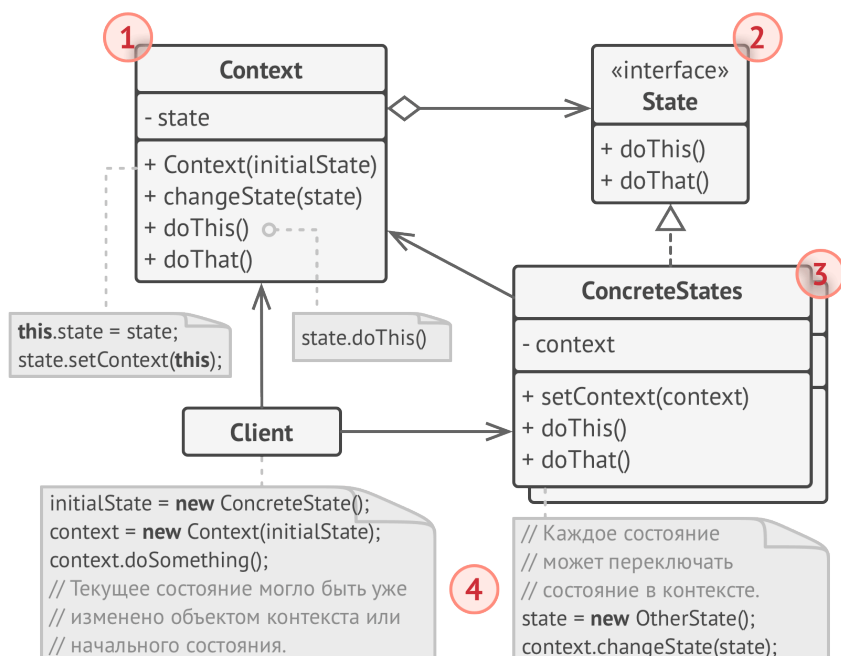
Очень важным нюансом, отличающим этот паттерн от **Стратегии**, является то, что и контекст, и сами конкретные состояния могут знать друг о друге и инициировать переходы от одного состояния к другому.

## 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.

## Структура



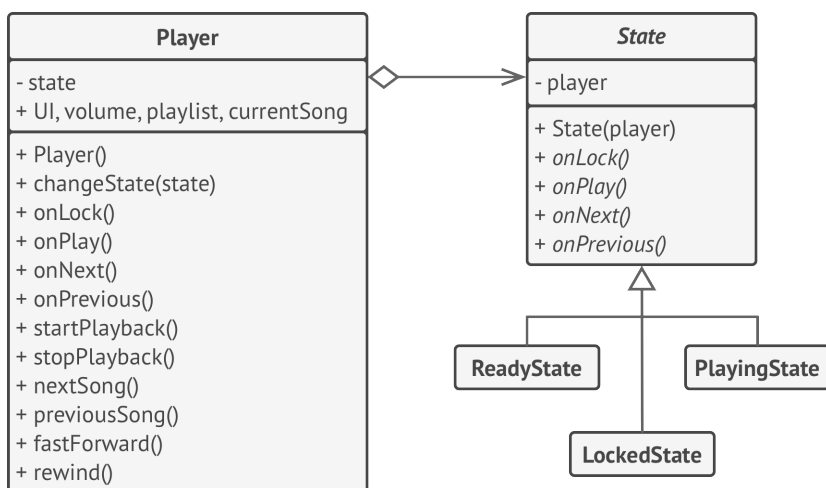
- Контекст** хранит ссылку на объект состояния и делегирует ему работу, зависящую от внутреннего состояния. Контекст работает с этим объектом через общий интерфейс состояний. Контекст должен иметь метод для присваивания ему нового объекта-состояния.
- Состояние** описывает общий интерфейс для всех конкретных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

## # Псевдокод



*Пример изменение поведения проигрывателя с помощью состояний.*



В этом примере паттерн **Состояние** изменяет функциональность одних и тех же элементов управления музыкальным проигрывателем, в зависимости от того, в каком состоянии находится сейчас проигрыватель.

Объект проигрывателя содержит объект-состояние, которому и делегирует основную работу. Изменяя состояния, можно менять то, как ведут себя элементы управления проигрывателя.

```

1  // Общий интерфейс всех состояний.
2  abstract class State is
3      protected field player: Player
4
5      // Контекст передаёт себя в конструктор состояния, чтобы
6      // состояние могло обращаться к его данным и методам в
7      // будущем, если потребуется.
8      constructor State(player) is
9          this.player = player
10
11     abstract method clickLock()
12     abstract method clickPlay()
13     abstract method clickNext()
14     abstract method clickPrevious()
15
16
17     // Конкретные состояния реализуют методы абстрактного
18     // состояния по-своему.
19     class LockedState is
20
21         // При разблокировке проигрователя с заблокированными
22         // клавишами, он может принять одно из двух состояний.
```

```
23     method clickLock() is
24         if (player.playing)
25             player.changeState(new PlayingState(player))
26         else
27             player.changeState(new ReadyState(player))
28
29     method clickPlay() is
30         // Ничего не делать.
31
32     method clickNext() is
33         // Ничего не делать.
34
35     method clickPrevious() is
36         // Ничего не делать.
37
38
39     // Они также могут переводить контекст в другие состояния.
40     class ReadyState is
41         method clickLock() is
42             player.changeState(new LockedState(player))
43
44         method clickPlay() is
45             player.startPlayback()
46             player.changeState(new PlayingState(player))
47
48         method clickNext() is
49             player.nextSong()
50
51         method clickPrevious() is
52             player.previousSong()
53
54
55
56
```






```

57 class PlayingState is
58     method clickLock() is
59         player.changeState(new LockedState(player))
60
61     method clickPlay() is
62         player.stopPlayback()
63         player.changeState(new ReadyState(player))
64
65     method clickNext() is
66         if (event.doubleclick)
67             player.nextSong()
68         else
69             player.fastForward(5)
70
71     method clickPrevious() is
72         if (event.doubleclick)
73             player.previous()
74         else
75             player.rewind(5)
76
77
78 // Проигрыватель играет роль контекста.
79 class Player is
80     field state: State
81     field UI, volume, playlist, currentSong
82
83     constructor Player() is
84         this.state = new ReadyState(this)
85
86     // Контекст заставляет состояние реагировать на
87     // пользовательский ввод вместо себя. Реакция может
88     // быть разной в зависимости от того, какое
89     // состояние сейчас активно.
90     UI = new UserInterface()

```

```
91     UI.lockButton.onClick(this.clickLock)
92     UI.playButton.onClick(this.clickPlay)
93     UI.nextButton.onClick(this.clickNext)
94     UI.prevButton.onClick(this.clickPrevious)
95
96     // Другие объекты должны иметь возможность заменить
97     // состояние проигрывателя.
98     method changeState(state: State) is
99         this.state = state
100
101     // Методы UI будут делегировать работу
102     // активному состоянию.
103     method clickLock() is
104         state.clickLock()
105     method clickPlay() is
106         state.clickPlay()
107     method clickNext() is
108         state.clickNext()
109     method clickPrevious() is
110         state.clickPrevious()
111
112     // Сервисные методы контекста, вызываемые состояниями.
113     method startPlayback() is
114         // ...
115     method stopPlayback() is
116         // ...
117     method nextSong() is
118         // ...
119     method previousSong() is
120         // ...
121     method fastForward(time) is
122         // ...
123     method rewind(time) is
124         // ...
```

## 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 все поля и методы, связанные с определённым состоянием в собственные классы. Первоначальный объект будет постоянно ссылаться на один из объектов-состояний, делегируя ему большую часть работы. Для изменения состояния, в контекст достаточно будет подставляться 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.
-  **Когда вы сознательно используете табличную машину состояний, построенную на условных операторах, но вынуждены мириться с дублированием кода для похожих состояний и переходов.**



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.



## Шаги реализации

1. Определитесь с классом, который будет играть роль контекста. Это может быть как существующий класс, в котором уже есть зависимость от состояния, так и новый класс, если код состояний размазан по нескольким классам.
2. Создайте интерфейс состояний. Он должен описывать методы, общие для всех состояний, обнаруженных в контексте. Заметьте, что не всё поведение контекста нужно переносить в состояние, а только то, которое зависит от состояний.
3. Для каждого фактического состояния, создайте класс, реализующий интерфейс состояния. Переместите весь код, связанный с конкретным состоянием в нужный класс. В конце концов, все методы интерфейса состояния должны быть реализованы.

При переносе поведения из контекста, вы можете столкнуться с тем, что это поведение зависит от приватных полей или методов контекста, к которым нет доступа из

состояния. Есть парочка способов обойти эту проблему. Самый простой — оставить поведение внутри контекста, вызывая его из объекта состояния. С другой стороны, вы можете сделать классы состояний вложенными в класс контекста, и тогда они получают доступ ко всем приватным частям контекста. Но последний способ доступен только в некоторых языках программирования (например, Java, C#).

4. Создайте в контексте поле для хранения объектов-состояний, а также публичный метод для изменения значения этого поля.
5. Старые методы контекста, в которых находился зависимый от состояния код, замените на вызовы соответствующих методов объекта-состояния.
6. В зависимости от бизнес-логики, разместите код, который переключает состояние контекста либо внутри контекста, либо внутри классов конкретных состояний.



## 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.

## ⇔ 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.