



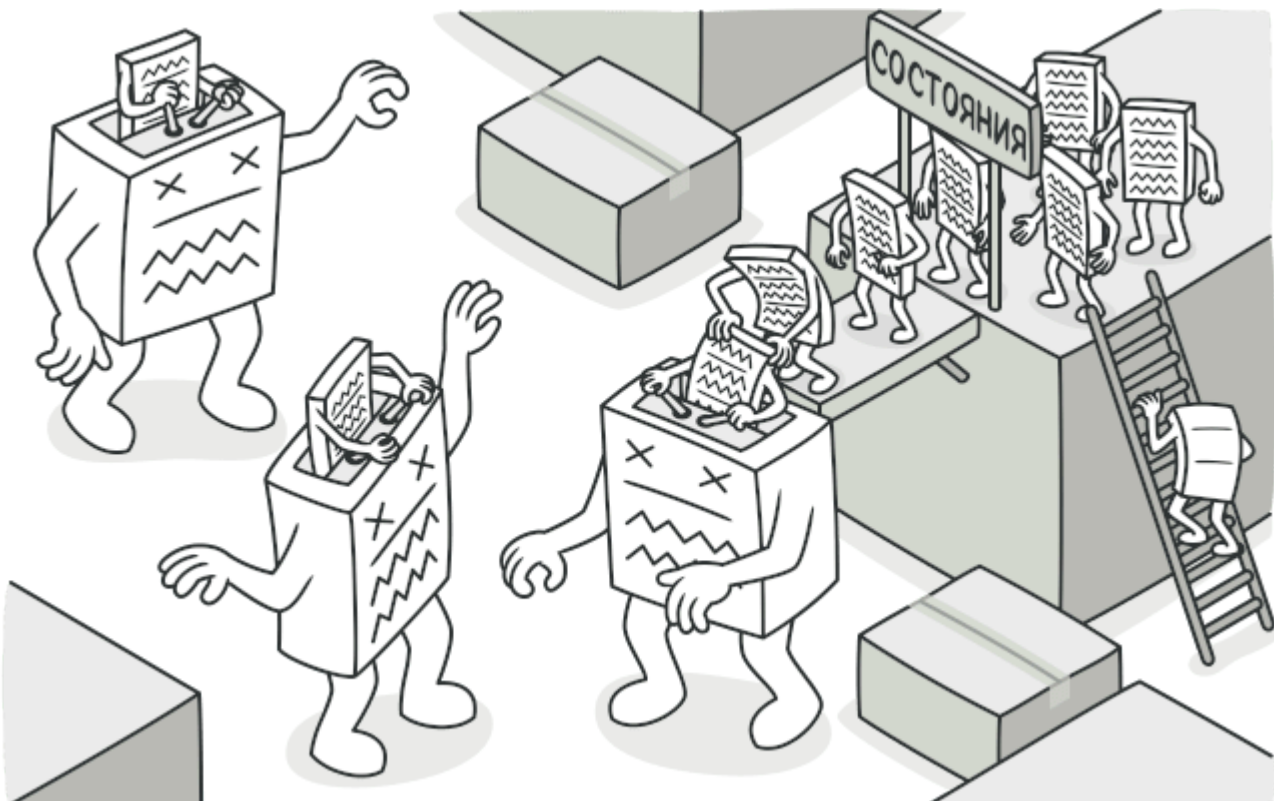
🏠 / Паттерны проектирования / Поведенческие паттерны

Состояние

Также известен как: State

💬 Суть паттерна

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

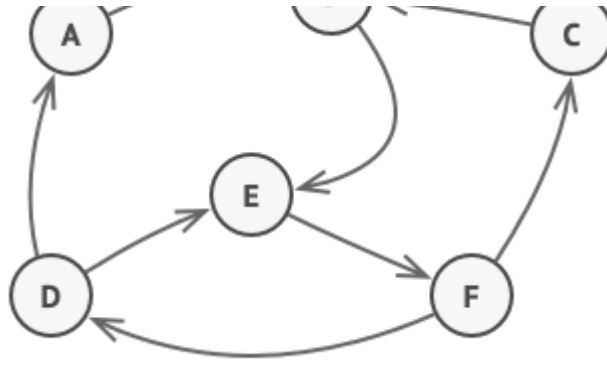


😞 Проблема

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



НОВОГОДНЯЯ РАСПРОДАЖА!

*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 можно применить и к отдельным объектам. Например, объект `Документ` может принимать три состояния: `Черновик`, `Модерация` или `Опубликован`. В каждом из этих состояний 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.



НОВОГОДНЯЯ РАСПРОДАЖА!



Возможные состояния документа 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 насчёт вот такого кода, выглядит знакомо?

```

class Document is
    field state: string
    // ...
    method publish() is
        switch (state)
            "draft":
                state = "moderation"
                break
            "moderation":
                if (currentUser.role == "admin")
                    state = "published"
                break
            "published":
                // Do nothing.
                break
    // ...
  
```



НОВОГОДНЯЯ РАСПРОДАЖА!



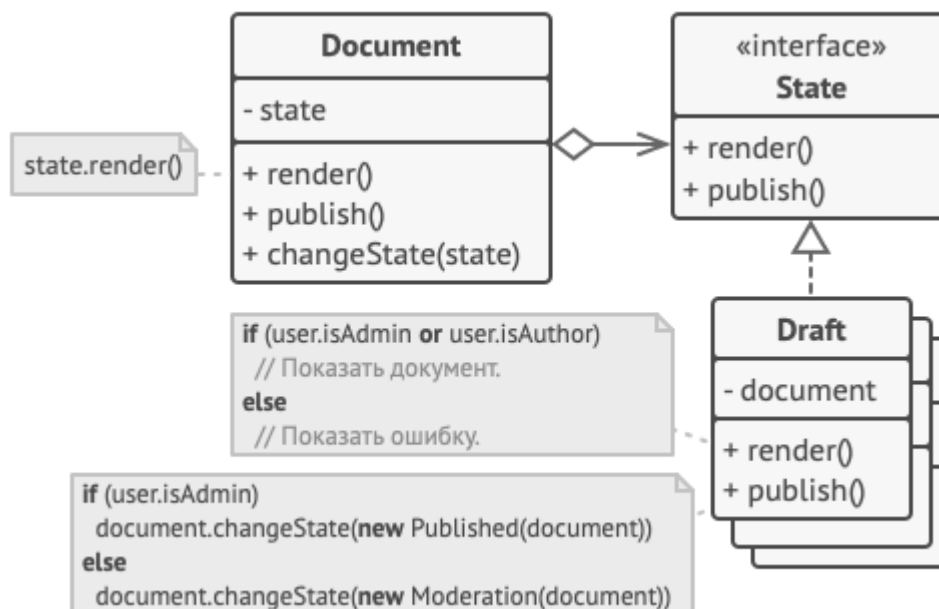
добавить еще десяток состояний. Каждый метод будет состоять из увесистого условного оператора, перебирающего доступные состояния. Такой код крайне сложно поддерживать. Малейшее изменение логики переходов заставит вас перепроверять работу всех методов, которые содержат условные операторы машины состояний.

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



НОВОГОДНЯЯ РАСПРОДАЖА!



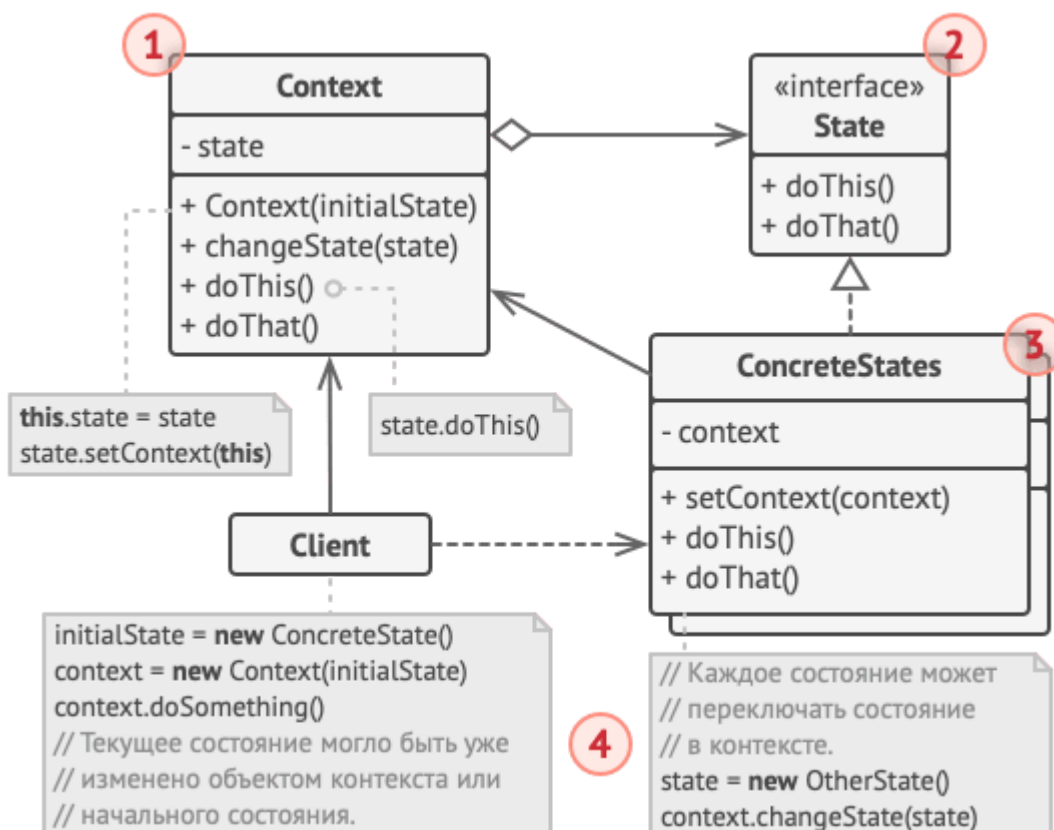
Очень важным нюансом, отличающим этот паттерн от Стратегии, является то, что и контекст, и сами конкретные состояния могут знать друг о друге и инициировать переходы от одного состояния к другому.

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



1. **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.



НОВОГОДНЯЯ РАСПРОДАЖА!



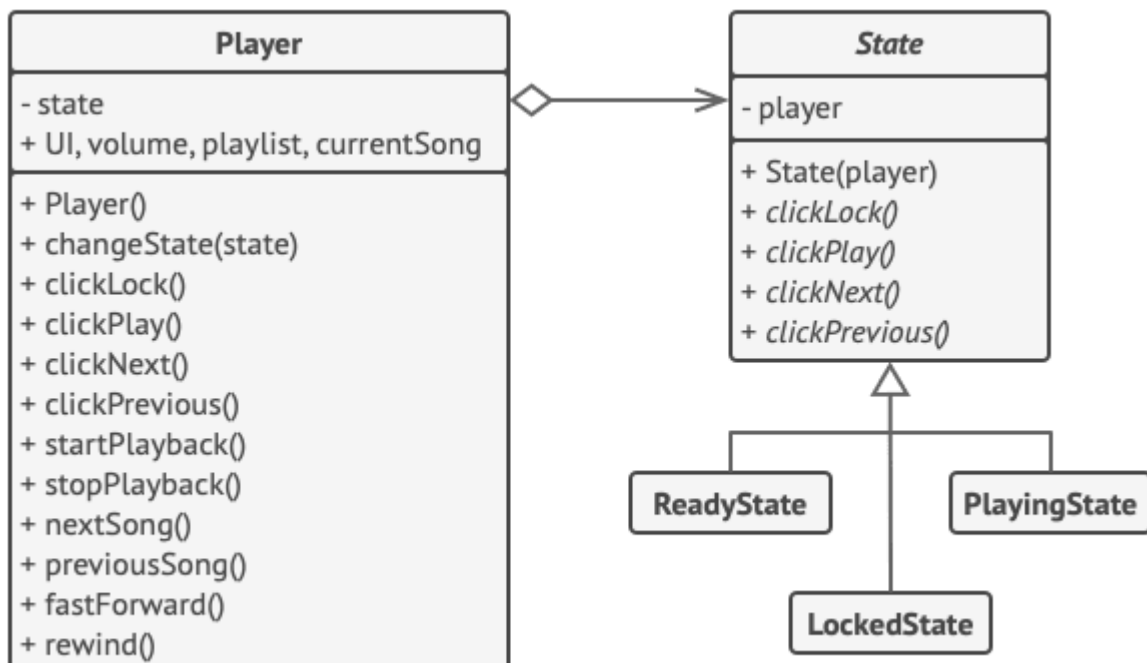
3. **Конкретные состояния** реализуют поведения, связанные с определённым состоянием контекста. Иногда приходится создавать целые иерархии классов состояний, чтобы обобщить дублирующий код.

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

4. 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

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.



Пример изменение поведения проигрывателя с помощью состояний.

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



НОВОГОДНЯЯ РАСПРОДАЖА!

**abstract class State is****protected field** `player`: `AudioPlayer`

```
// Контекст передаёт себя в конструктор состояния, чтобы
// состояние могло обращаться к его данным и методам в
// будущем, если потребуется.
```

constructor `State(player)` **is**`this.player = player`**abstract method** `clickLock()`**abstract method** `clickPlay()`**abstract method** `clickNext()`**abstract method** `clickPrevious()`

```
// Конкретные состояния реализуют методы абстрактного состояния
// по-своему.
```

class LockedState extends State is

```
// При разблокировке проигрователя с заблокированными
// клавишами он может принять одно из двух состояний.
```

method `clickLock()` **is**`if (player.playing)``player.changeState(new PlayingState(player))``else``player.changeState(new ReadyState(player))`**method** `clickPlay()` **is**

```
// Ничего не делать.
```

method `clickNext()` **is**

```
// Ничего не делать.
```

method `clickPrevious()` **is**

```
// Ничего не делать.
```

```
// Конкретные состояния сами могут переводить контекст в другое
// состояние.
```

class ReadyState extends State is**method** `clickLock()` **is**`player.changeState(new LockedState(player))`**method** `clickPlay()` **is**`player.startPlayback()``player.changeState(new PlayingState(player))`**method** `clickNext()` **is**`player.nextSong()`



НОВОГОДНЯЯ РАСПРОДАЖА!



```
class PlayingState extends State is
  method clickLock() is
    player.changeState(new LockedState(player))

  method clickPlay() is
    player.stopPlayback()
    player.changeState(new ReadyState(player))

  method clickNext() is
    if (event.doubleclick)
      player.nextSong()
    else
      player.fastForward(5)

  method clickPrevious() is
    if (event.doubleclick)
      player.previous()
    else
      player.rewind(5)

// Проигрыватель выступает в роли контекста.
class AudioPlayer is
  field state: State
  field UI, volume, playlist, currentSong

  constructor AudioPlayer() is
    this.state = new ReadyState(this)

    // Контекст заставляет состояние реагировать на
    // пользовательский ввод вместо себя. Реакция может быть
    // разной, в зависимости от того, какое состояние сейчас
    // активно.
    UI = new UserInterface()
    UI.lockButton.onClick(this.clickLock)
    UI.playButton.onClick(this.clickPlay)
    UI.nextButton.onClick(this.clickNext)
    UI.prevButton.onClick(this.clickPrevious)

// Другие объекты тоже должны иметь возможность заменять
// состояние проигрывателя.
method changeState(state: State) is
  this.state = state

// Методы UI будут делегировать работу активному состоянию.
method clickLock() is
  state.clickLock()
method clickPlay() is
```







**НОВОГОДНЯЯ РАСПРОДАЖА!**


```
state.clickNext()

method clickPrevious() is
    state.clickPrevious()

// Сервисные методы контекста, вызываемые состояниями.
method startPlayback() is
    // ...
method stopPlayback() is
    // ...
method nextSong() is
    // ...
method previousSong() is
    // ...
method fastForward(time) is
    // ...
method rewind(time) is
    // ...
```

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.
-  Когда вы сознательно используете табличную машину состояний, построенную на условных операторах, но вынуждены мириться с дублированием кода для похожих состояний и переходов.

 **НОВОГОДНЯЯ РАСПРОДАЖА!**
ослезирующуюся на наследовании. Вы можете отнаследовать похужие 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 общий 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 код, связанный с конкретными состояниями в нужные классы. В конце концов, все методы интерфейса состояния должны быть реализованы во всех классах состояний.


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 объекта состояния. Существует 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

 **НОВОГОДНЯЯ РАСПРОДАЖА!**
условных операторов машины
состояний.

 **НОВОГОДНЯЯ РАСПРОДАЖА!**
если состояний мало и они редко
меняются.

- ✓ Концентрирует в одном месте код, связанный с определённым состоянием.
- ✓ Упрощает код контекста.

⇔ 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.

</> Примеры реализации паттерна

