refactoring.guru

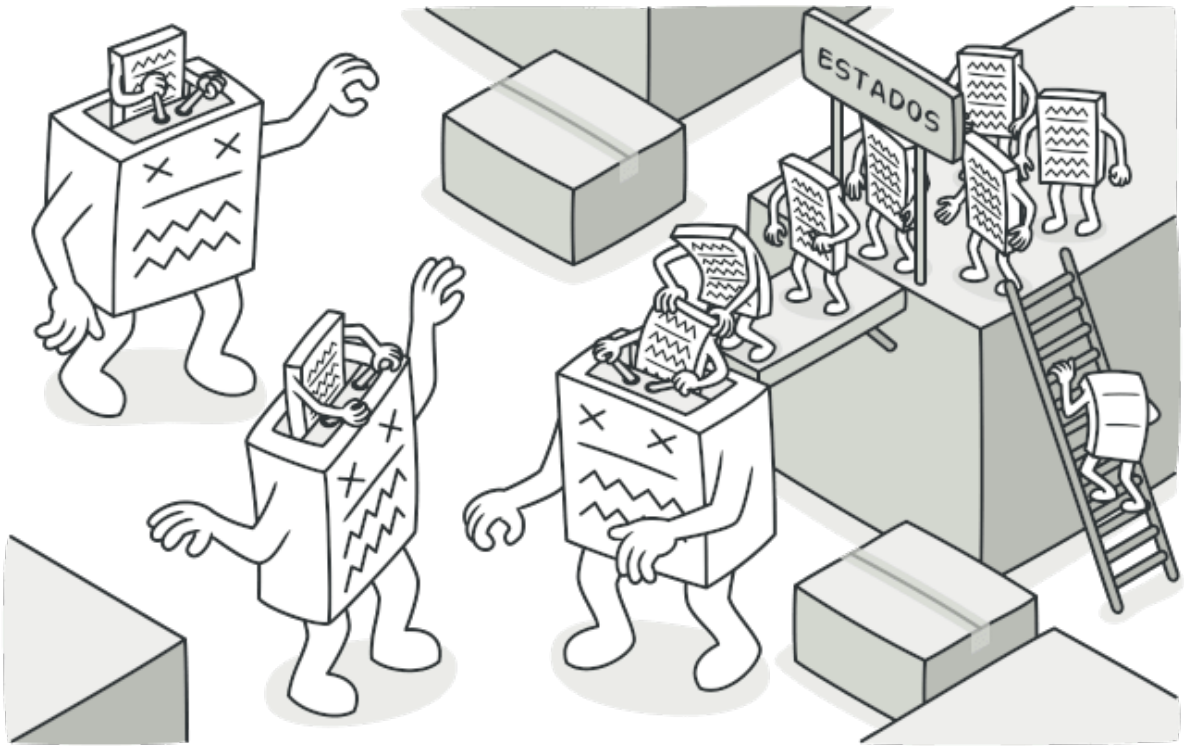
State

14–19 minutes

Também conhecido como: Estado

Propósito

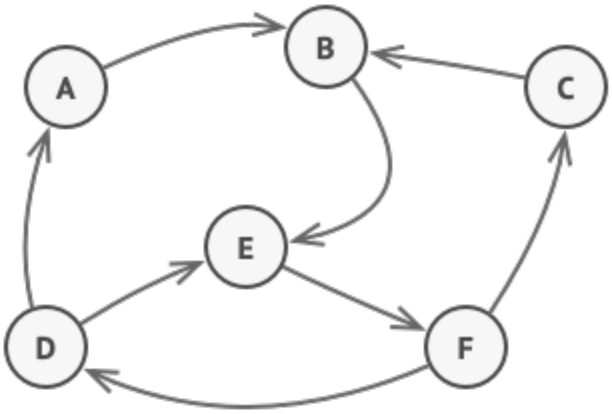
O **State** é um padrão de projeto comportamental que permite que um objeto altere seu comportamento quando seu estado interno muda. Parece como se o objeto mudasse de classe.



Problema

O padrão State é intimamente relacionado com o conceito de uma

Máquina de Estado Finito .

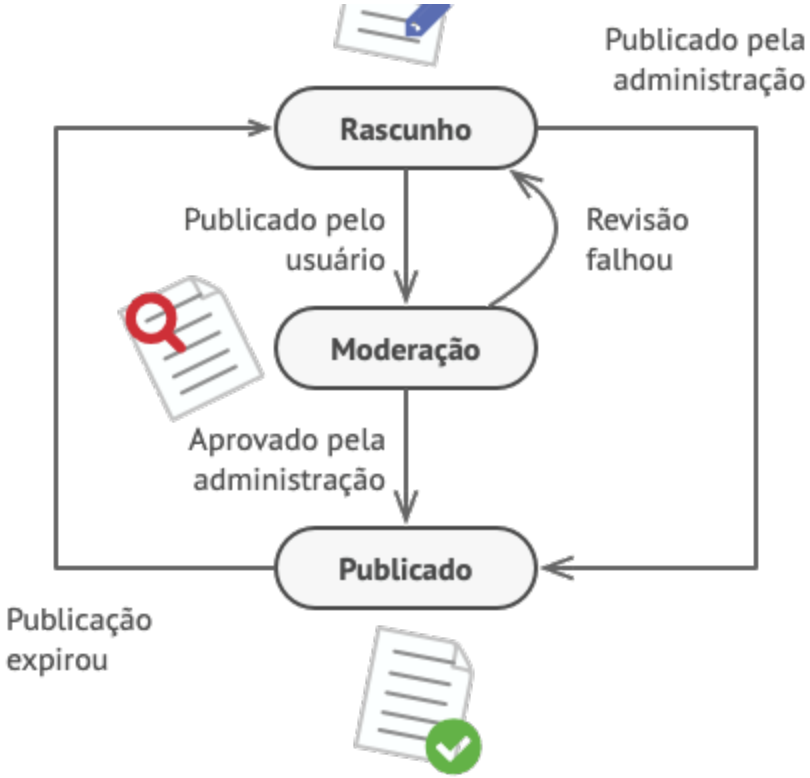


Máquina de Estado Finito.

A ideia principal é que, em qualquer dado momento, há um número *finito* de *estados* que um programa possa estar. Dentro de qualquer estado único, o programa se comporta de forma diferente, e o programa pode ser trocado de um estado para outro instantaneamente. Contudo, dependendo do estado atual, o programa pode ou não trocar para outros estados. Essas regras de troca, chamadas *transições*, também são finitas e pré determinadas.

Você também pode aplicar essa abordagem para objetos. Imagine que nós temos uma classe `Documento`. Um documento pode estar em um de três estados: `Rascunho`, `Moderação` e `Publicado`. O método `publicar` do documento funciona um pouco diferente em cada estado:

- No `Rascunho`, ele move o documento para a moderação.
- Na `Moderação` ele torna o documento público, mas apenas se o usuário atual é um administrador.
- No `Publicado` ele não faz nada.



Possíveis estados e transições de um objeto documento.

Máquinas de estado são geralmente implementadas com muitos operadores de condicionais (`if` ou `switch`) que selecionam o comportamento apropriado dependendo do estado atual do objeto. Geralmente esse “estado” é apenas um conjunto de valores dos campos do objeto. Mesmo se você nunca ouviu falar sobre máquinas de estado finito antes, você provavelmente já implementou um estado ao menos uma vez. A seguinte estrutura de código lembra alguma coisa para você?

```
class Document is
    field state: string

    method publish() is
        switch (state)
            "draft":
                state = "moderation"
```

```
        break
    "moderation":
        if (currentUser.role == "admin")
            state = "published"
        break
    "published":
        // Não fazer nada.
        break
```

A maior fraqueza de uma máquina de estados baseada em condicionais se revela quando começamos a adicionar mais e mais estados e comportamentos baseados em estados para a classe `Documento`. A maioria dos métodos irá conter condicionais monstruosas que selecionam o comportamento apropriado de um método de acordo com o estado atual. Um código como esse é muito difícil de se fazer manutenção porque qualquer mudança na lógica de transição pode necessitar de mudanças de condicionais de estado em todos os métodos.

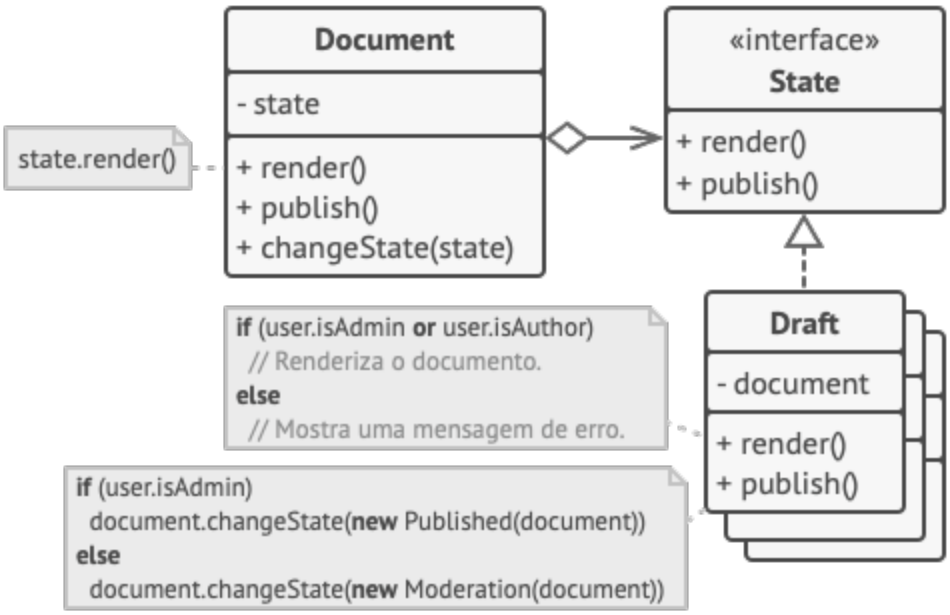
O problema tende a ficar maior a medida que o projeto evolui. É muito difícil prever todos os possíveis estados e transições no estágio inicial de projeto. Portanto, uma máquina de estados enxuta, construída com um número limitado de condicionais pode se tornar uma massa inchada e disforme com o tempo.

Solução

O padrão State sugere que você crie novas classes para todos os estados possíveis de um objeto e extraia todos os comportamentos específicos de estados para dentro dessas

classes.

Ao invés de implementar todos os comportamentos por conta própria, o objeto original, chamado *contexto*, armazena uma referência para um dos objetos de estado que representa seu estado atual, e delega todo o trabalho relacionado aos estados para aquele objeto.



O documento delega o trabalho para um objeto de estado.

Para fazer a transição do contexto para outro estado, substitua o objeto do estado ativo por outro objeto que represente o novo estado. Isso é possível somente se todas as classes de estado seguirem a mesma interface e o próprio contexto funcione com esses objetos através daquela interface.

Essa estrutura pode ser parecida com o padrão Strategy, mas há uma diferença chave. No padrão State, os estados em particular podem estar cientes de cada um e iniciar transições de um estado para outro, enquanto que estratégias quase nunca sabem sobre as outras estratégias.

Analogia com o mundo real

Os botões e interruptores de seu smartphone comportam-se de forma diferente dependendo do estado atual do dispositivo:

- Quando o telefone está desbloqueado, apertar os botões leva eles a executar várias funções.
- Quando o telefone está bloqueado, apertar qualquer botão leva a desbloquear a tela.
- Quando a carga da bateria está baixa, apertar qualquer botão mostra a tela de carregamento.

Estrutura

1. O **Contexto** armazena uma referência a um dos objetos concretos de estado e delega a eles todos os trabalhos específicos de estado. O contexto se comunica com o objeto estado através da interface do estado. O contexto expõe um setter para passar a ele um novo objeto de estado.
2. A interface do **Estado** declara métodos específicos a estados. Esses métodos devem fazer sentido para todos os estados concretos porque você não quer alguns dos seus estados tendo métodos inúteis que nunca irão ser chamados.
3. Os **Estados Concretos** fornecem suas próprias implementações para os métodos específicos de estados. Para evitar duplicação ou código parecido em múltiplos estados, você pode fornecer classes abstratas intermediárias que encapsulam alguns dos comportamentos comuns.

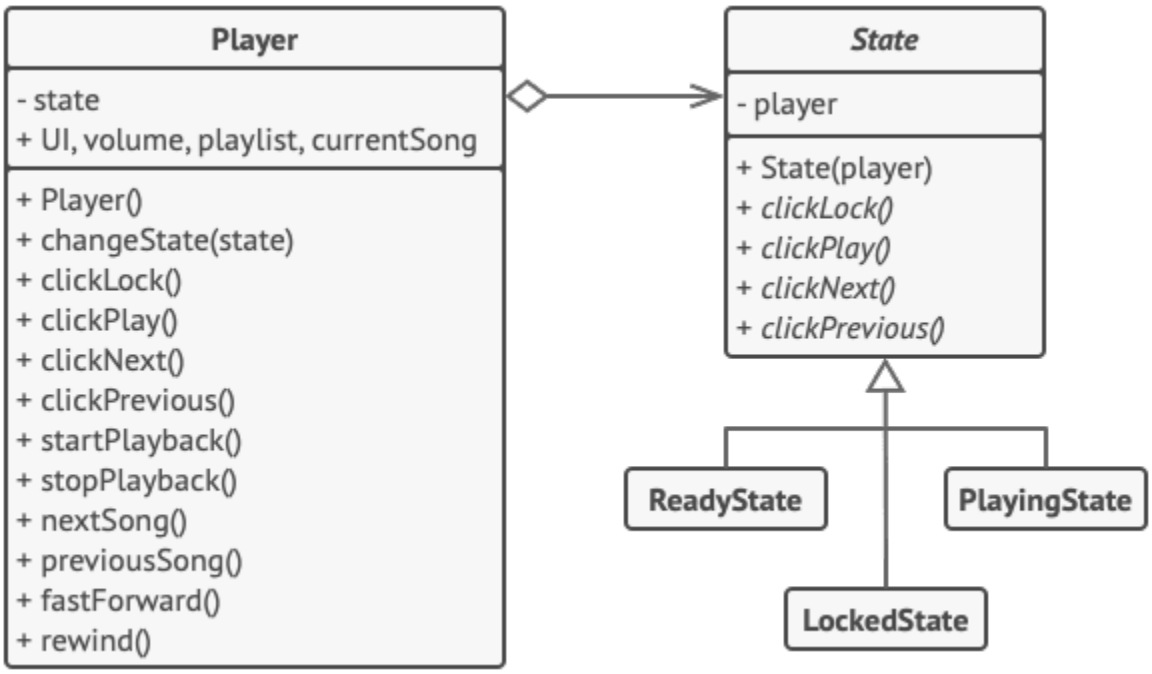
   Objetos de estado podem armazenar referências retroativas para

o objeto de contexto. Através dessa referência o estado pode buscar qualquer informação desejada do objeto contexto, assim como iniciar transições de estado.

4. Ambos os estados de contexto e concretos podem configurar o próximo estado do contexto e realizar a atual transição de estado ao substituir o objeto estado ligado ao contexto.

Pseudocódigo

Neste exemplo, o padrão **State** permite que os mesmos controles de tocador de mídia se comportem diferentemente, dependendo do atual estado do tocador.



Exemplo da troca do comportamento de um objeto com objetos de estado.

O objeto principal do tocador está sempre ligado ao objeto estado que realiza a maior parte do trabalho para o tocador. Algumas ações substituem o objeto do estado atual do tocador por outro, que muda a maneira do tocador reagir às interações do usuário.

```
// A classe AudioPlayer age como um contexto. Ela também
mantém
// uma referência para uma instância de uma das classes de
// estado que representa o atual estado do tocador de áudio.
class AudioPlayer is
    field state: State
    field UI, volume, playlist, currentSong

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

        // O contexto delega o manuseio das entradas do usuário
        // para um objeto de estado. Naturalmente, o resultado
        // depende de qual estado está ativo, uma vez que cada
        // estado pode lidar com as entradas de forma diferente.
        UI = new UserInterface()
        UI.lockButton.onClick(this.clickLock)
        UI.playButton.onClick(this.clickPlay)
        UI.nextButton.onClick(this.clickNext)
        UI.prevButton.onClick(this.clickPrevious)

    // Outros objetos devem ser capazes de trocar o estado ativo
    // do tocador.
    method changeState(state: State) is
        this.state = state

    // Métodos de UI delegam a execução para o estado ativo.
    method clickLock() is
        state.clickLock()
    method clickPlay() is
```

```
        state.clickPlay()
    method clickNext() is
        state.clickNext()
    method clickPrevious() is
        state.clickPrevious()

    // Um estado pode chamar alguns métodos de serviço no
    // contexto.
    method startPlayback() is

    method stopPlayback() is

    method nextSong() is

    method previousSong() is

    method fastForward(time) is

    method rewind(time) is


// A classe de estado base declara métodos que todos os estados
// concretos devem implementar e também fornece uma referência
// anterior ao objeto de contexto associado com o estado.
// Estados podem usar a referência anterior para realizar a
// transição contexto para outro estado.
abstract class State is
    protected field player: AudioPlayer

    // O contexto passa a si mesmo através do construtor do
```

```
    // estado. Isso pode ajudar o estado a recuperar alguns
    // dados de contexto úteis se for necessário.
    constructor State(player) is
        this.player = player

    abstract method clickLock()
    abstract method clickPlay()
    abstract method clickNext()
    abstract method clickPrevious()


// Estados concretos implementam vários comportamentos
// associados com um estado do contexto.
class LockedState extends State is

    // Quando você desbloqueia um tocador bloqueado, ele vai
    // assumir um dos dois estados.
    method clickLock() is
        if (player.playing)
            player.changeState(new PlayingState(player))
        else
            player.changeState(new ReadyState(player))

    method clickPlay() is
        // Bloqueado, então não faz nada.

    method clickNext() is
        // Bloqueado, então não faz nada.

    method clickPrevious() is
```

```
        // Bloqueado, então não faz nada.


// Eles também podem ativar transições de estado no contexto.
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()

    method clickPrevious() is
        player.previousSong()


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)
```

Aplicabilidade

Utilize o padrão State quando você tem um objeto que se comporta de maneira diferente dependendo do seu estado atual, quando o número de estados é enorme, e quando o código estado específico muda com frequência.

O padrão sugere que você extraia todo o código estado específico para um conjunto de classes distintas. Como resultado, você pode adicionar novos estados ou mudar os existentes independentemente uns dos outros, reduzindo o custo da manutenção.

Utilize o padrão quando você tem uma classe populada com condicionais gigantes que alteram como a classe se comporta de acordo com os valores atuais dos campos da classe.

O padrão State permite que você extraia ramificações dessas condicionais para dentro de métodos de classes correspondentes. Ao fazer isso, você também limpa para fora da classe principal os campos temporários e os métodos auxiliares envolvidos no código estado específico.

Utilize o State quando você tem muito código duplicado em muitos

estados parecidos e transições de uma máquina de estado baseada em condições.

O padrão State permite que você componha hierarquias de classes estado e reduza a duplicação ao extrair código comum para dentro de classes abstratas base.

Como implementar

1. Decida qual classe vai agir como contexto. Poderia ser uma classe existente que já tenha código dependente do estado; ou uma nova classe, se o código específico ao estado estiver distribuído em múltiplas classes.
2. Declare a interface do estado. Embora ela vai espelhar todos os métodos declarados no contexto, mire apenas para aqueles que possam conter comportamento específico ao estado.
3. Para cada estado real, crie uma classe que deriva da interface do estado. Então vá para os métodos do contexto e extraia todo o código relacionado a aquele estado para dentro de sua nova classe.

   Ao mover o código para a classe estado, você pode descobrir que ela depende de membros privados do contexto. Há várias maneiras de contornar isso:

- Torne esses campos ou métodos públicos.
- Transforme o comportamento que você está extraindo para um método público dentro do contexto e chame-o na classe estado. Essa maneira é feia mas rápida, e você pode sempre consertá-la mais tarde.
- Aninhe as classes estado dentro da classe contexto, mas apenas

se sua linguagem de programação suporta classes aninhadas.

4. Na classe contexto, adicione um campo de referência do tipo de interface do estado e um setter público que permite sobrescrever o valor daquele campo.
5. Vá até o método do contexto novamente e substitua as condicionais de estado vazias por chamadas aos métodos correspondentes do objeto estado.
6. Para trocar o estado do contexto, crie uma instância de uma das classes estado e a passe para o contexto. Você pode fazer isso dentro do próprio contexto, ou em vários estados, ou no cliente. Aonde quer que isso seja feito, a classe se torna dependente da classe estado concreta que ela instanciou.

Prós e contras

- *Princípio de responsabilidade única*. Organiza o código relacionado a estados particulares em classes separadas.
- *Princípio aberto/fechado*. Introduz novos estados sem mudar classes de estado ou contexto existentes.
- Simplifica o código de contexto ao eliminar condicionais de máquinas de estado pesadas.
- Aplicar o padrão pode ser um exagero se a máquina de estado só tem alguns estados ou raramente muda eles.

Relações com outros padrões

- O Bridge, State, Strategy (e de certa forma o Adapter) têm estruturas muito parecidas. De fato, todos esses padrões estão baseados em composição, o que é delegar o trabalho para outros

objetos. Contudo, eles todos resolvem problemas diferentes. Um padrão não é apenas uma receita para estruturar seu código de uma maneira específica. Ele também pode comunicar a outros desenvolvedores o problema que o padrão resolve.

- O State pode ser considerado como uma extensão do Strategy. Ambos padrões são baseados em composição: eles mudam o comportamento do contexto ao delegar algum trabalho para objetos auxiliares. O *Strategy* faz esses objetos serem completamente independentes e alheios entre si. Contudo, o *State* não restringe dependências entre estados concretos, permitindo que eles alterem o estado do contexto à vontade.