



HELP UKRAINE STOP RUSSIA

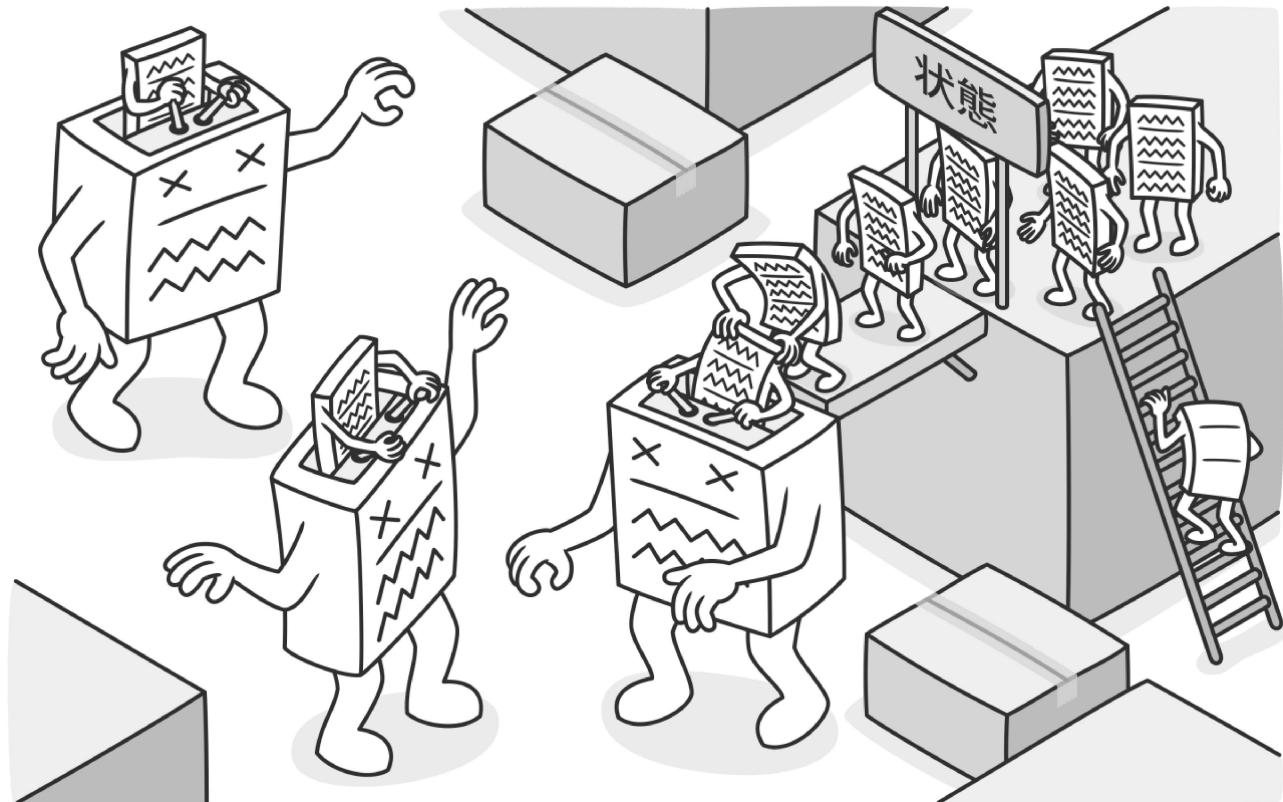
[Home](#) / [Design Patterns](#) / 振る舞いに関するパターン

State

別名：ステート

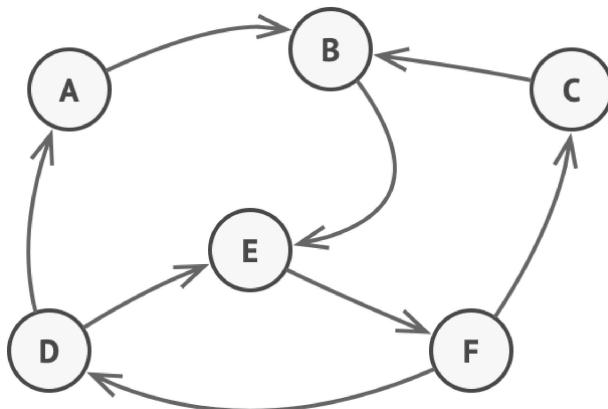
一言でいうと

State（ステート、状態）は、振る舞いに関するデザインパターンの一つで、オブジェクトの内部状態が変化した時に、その挙動を変化させます。それは、あたかもそのオブジェクトのクラスが変わったかのように見えます。



問題

State パターンは、有限オートマトン ①の概念と密接に関連しています。

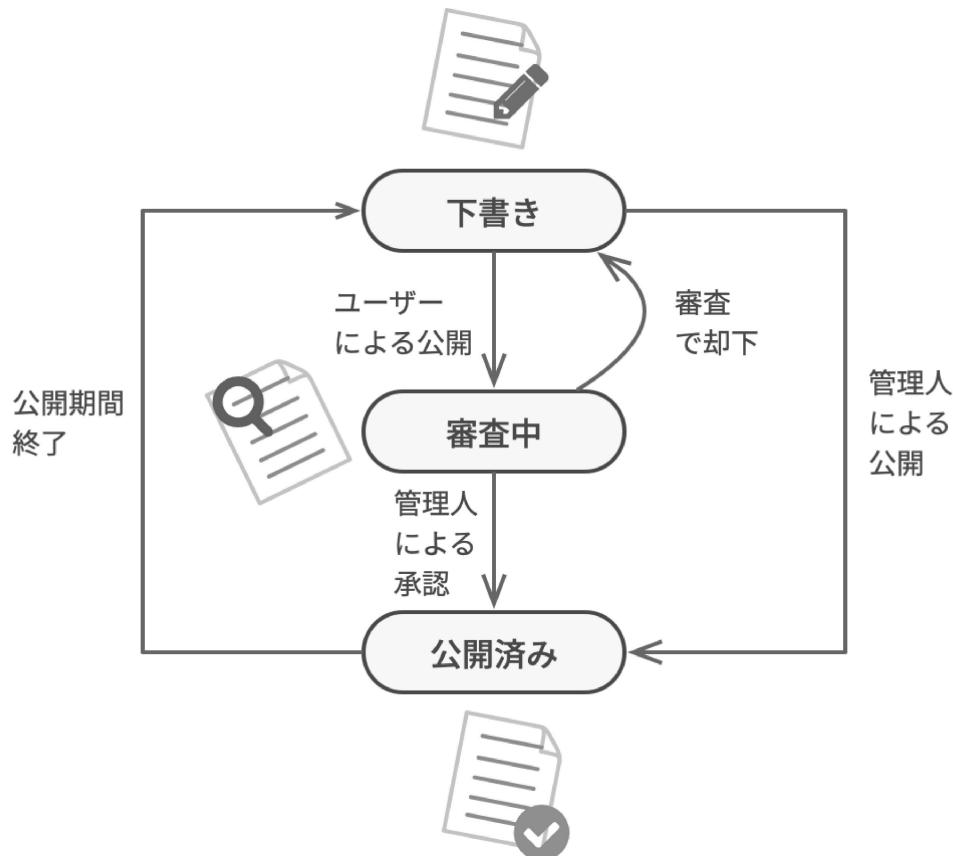


有限状態機械

基本的な考えとしては、いかなる時点でも、プログラムが取ることのできる有限個の状態がある、ということです。ある固有の状態では、プログラムは異なる振る舞いを見せ、一つの状態から別の状態に瞬時に切り替え可能です。しかし、現在に状態によっては、プログラムは特定の別の状態に移行できるかもしれませんし、移行できないかもしれません。この移行の規則は、遷移と呼ばれ、有限個であり、予め決められています。

このやり方をオブジェクトに当てはめることもできます。`Document` (ドキュメント) クラスが一つあると仮定してみてください。ドキュメントは、次の三つの状態を取ることができます：`Draft` (下書き)、`Moderation` (考査中)、`Published` (発行済み)。ドキュメントの `publish` メソッドは、それぞれの状態でちょっとずつ違って機能します。

- `Draft` では、ドキュメントを `Moderation` に移行します。
- `Moderation` では、現在のユーザーが管理者である場合にのみ、ドキュメントを公開にします。
- `Published` では、まったく何もしません。



ドキュメント・オブジェクトが取り得る状態と遷移。

状態機械は通常、オブジェクトの現在の状態に応じて適切な動作を選択する多くの条件付き演算子（`if` または `switch`）で実装されます。通常、この「状態」はオブジェクトのフィールドの値の集合です。仮に今まで有限状態機械なんて聞いたことがないとしても、おそらく少なくとも 1 回は状態を実装したことがあるはずです。次のコードの構造に、何か見覚えないですか？

```

class Document {
    field state: string
    // .....

    method publish() {
        switch (state) {
            case "draft":
                state = "moderation"
                break
            case "moderation":
                if (currentUser.role == "admin")
                    state = "published"
                break
            case "published":
                // 何もしない
                break
        }
        // .....
    }
}
  
```

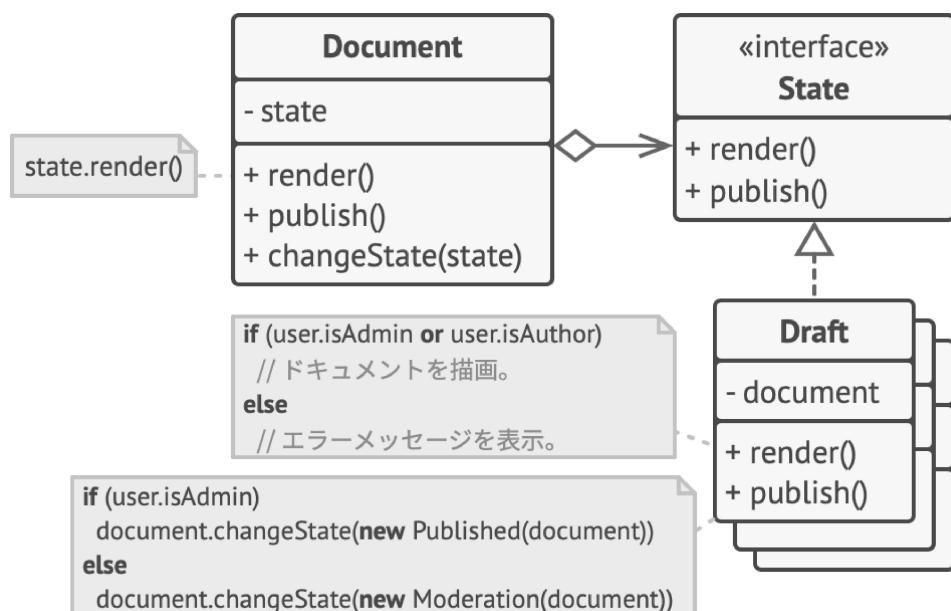
条件文に基づく状態機械の最大の弱点が、`Document` クラスに新しい状態と状態依存の動作とを次々に追加し始めると露呈します。ほとんどのメソッドは、現在の状態に従ってメソッドの適切な動作を選択する退屈な条件文から成ります。遷移ロジックを変更するためには、すべてのメソッドで状態条件を変更する必要があり、コードを保守が非常に困難になります。

プロジェクトが進展するにつれて問題は大きくなりがちです。設計段階ですべての可能な状態や遷移を予測することは非常に困難です。したがって、限られた条件文で作られた無駄のない状態機械は、時間の経過とともに膨張した化け物になりがちです。

😊 解決策

`State` パターンに従うと、オブジェクトのすべての可能な状態に対して新しいクラス（訳註：一つの状態ごとに一つのクラス）を作成し、状態固有の振る舞いをこれらのクラスに抽出します。

すべての振る舞いを単独で実装する代わりに、コンテキスト（context、前後関係）と呼ばれる元々のオブジェクトが、現在の状態を表す状態オブジェクトの一つへの参照を持ち、そのオブジェクトにすべての状態関連の作業を委任します。



ドキュメントは、その作業を状態オブジェクトに委任。

コンテキストを別の状態に遷移するには、現在有効な状態オブジェクトを、新しい状態を表す別のオブジェクトで置き換えます。これは、すべての（具象）ステート・クラスが同じインターフェースに従っていて、コンテキスト自体がそのインターフェースを介し、これらのオブジェクトとやりとりする場合にのみ可能です。

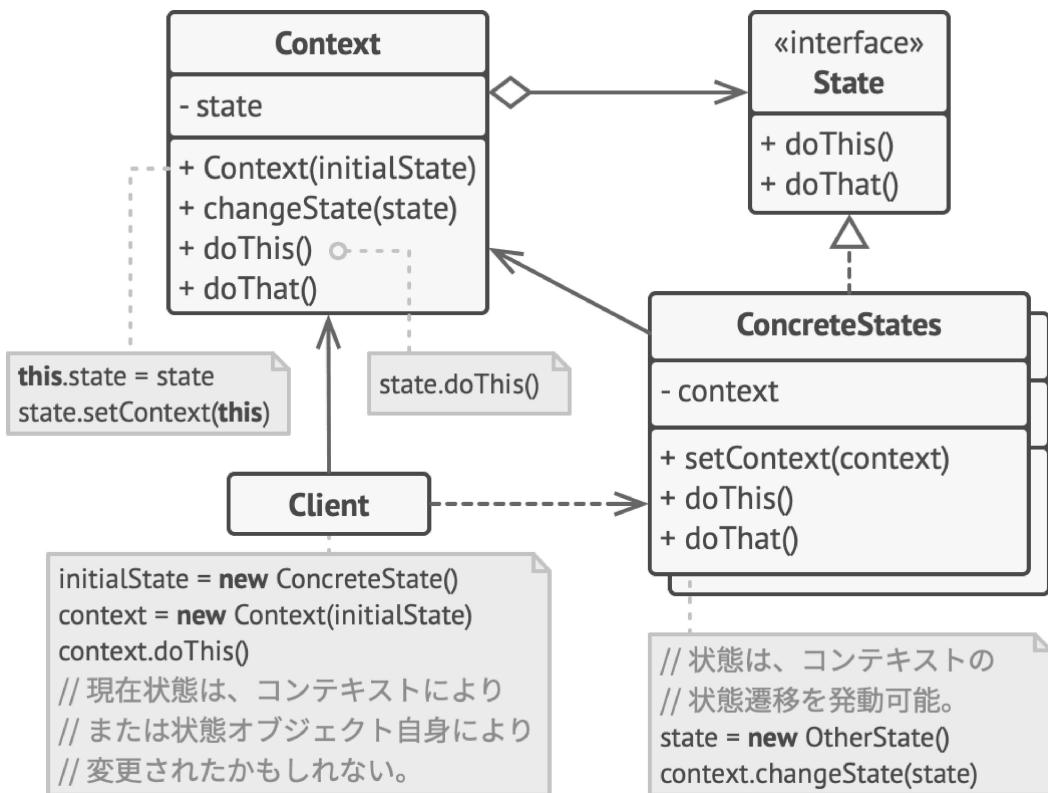
この構造は、**Strategy** パターンに似ているように見えますが、重要な違いが一つあります。State パターンでは、特定の状態同士はお互いを知っており、ある状態から別の状態への遷移を開始しますが、ストラテジーは、お互いを知っていることはまずありません。

🚗 現実世界でのたとえ

スマートフォーンのボタンとスイッチは、現在の機器の状態によって異なる振る舞いをします。

- 携帯電話がロックされてない時は、ボタンを押すと様々な機能が実行されます。
- 電話がロックされていると、どのボタンを押してもロック解除画面が表示されます。
- 携帯電話の充電量が少ないと、どのボタンを押しても充電画面が表示されます。

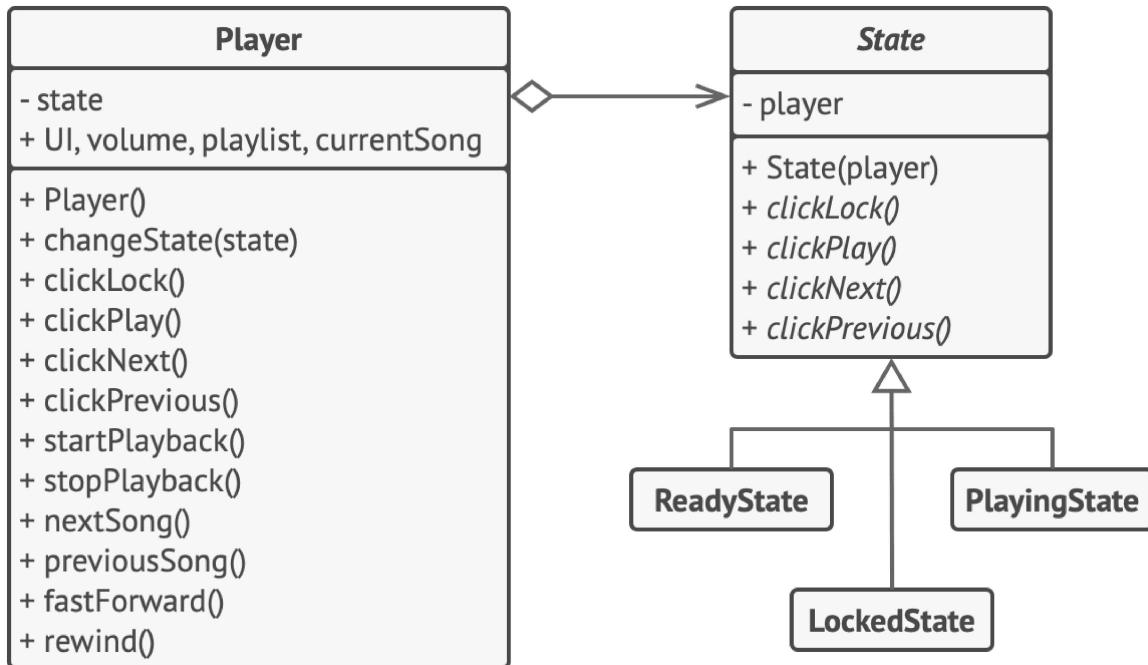
嚚 構造



1. コンテキスト (Context) は、複数個ある具象状態クラスのオブジェクトのいずれか一つへ参照を保存し、状態に固有の作業は、すべてそのオブジェクトに委任されます。コンテキストは、状態インターフェースを介して状態オブジェクトと通信します。コンテキストには、新しい状態オブジェクトを渡すための公開の setter があります。
 2. ステート (State) インターフェースは状態固有のメソッドを宣言します。これらのメソッドはすべての具象ステートに意味のあるものでなければなりません。いくつかの状態では絶対に呼ばれない無駄なメソッドがないようにしたいからです。
 3. 具象ステート (Concrete State、具象状態) は、状態固有のメソッドに対して独自の実装を行います。似たようなコードが複数の状態で重複することを避けるために、共通の振る舞いを内包した中間層の抽象クラスを作ることもできます。
- ステート・オブジェクトは、コンテキスト・オブジェクトへの逆参照を格納するようにもできます。この参照を通して、ステートはコンテキスト・オブジェクトから必要な情報を取りしたり、状態遷移を開始したりできます。
4. コンテキストと具象ステートの両方とも、コンテキストの次の状態の設定が可能で、コンテキストにリンクされたステート・オブジェクトを置き換えることにより実際の状態遷移を行います。

擬似コード

この例では、**State** パターンにより、メディア・プレーヤーの現在の再生状態に応じて、同じコントロール（訳注：UI のボタンなど）に異なる振る舞いをさせます。



オブジェクトの振る舞いを状態オブジェクトで変更する例。

プレーヤーの主オブジェクトは、プレーヤーのほとんどの作業を実行する状態オブジェクトのどれか一つに常にリンクされています。ユーザーの行う操作によっては、プレーヤーの現在の状態オブジェクトを別のものに置き換え、プレーヤーがユーザーとの対話方法を変更します。

```

// AudioPlayer クラスはコンテキストとして機能する。また、オーディオ・プレーヤーの現在の状態に対応する状態クラスのインスタンスへの参照も維持する。
// る。
class AudioPlayer is
    field state: State
    field UI, volume, playlist, currentSong

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

        // コンテキストは、ユーザー入力の処理をステート・オブジェクトに委任。状態=ステートによって、入力を違うように扱うため、結果は現在有効な状態に当然依存する。
        UI = new UserInterface()
  
```

```

UI.lockButton.onClick(this.clickLock)
UI.playButton.onClick(this.clickPlay)
UI.nextButton.onClick(this.clickNext)
UI.prevButton.onClick(this.clickPrevious)

// 他のオブジェクトがオーディオ・プレーヤーの現在有効な状態を変更する
// ことを可能とするメソッド。
method changeState(state: State) is
    this.state = state

// UI メソッドは、現在有効なステートに実行を委任。
method clickLock() is
    state.clickLock()
method clickPlay() is
    state.clickPlay()
method clickNext() is
    state.clickNext()
method clickPrevious() is
    state.clickPrevious()

// ステートは、コンテキスト上のサービス・メソッドをいくつか呼び出すか
// もしれない。
method startPlayback() is
    // .....
method stopPlayback() is
    // .....
method nextSong() is
    // .....
method previousSong() is
    // .....
method fastForward(time) is
    // .....
method rewind(time) is
    // .....

// ステートの基底クラスは、全部の具象ステートが実装すべきメソッドを宣言し、
// ステートと関連するコンテキスト・オブジェクトに対する逆参照を提供する。
// ステートは、コンテキストを別の状態に遷移するために逆参照を使用できる。
abstract class State is
    protected field player: AudioPlayer

// コンテキストは、ステートのコンストラクターに渡される。これは、もし
// 必要ならば、ステートが有用なコンテキスト・データを取得する助けとな
// る。
constructor State(player) is
    this.player = player

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

```

```
// 具象ステートは、コンテキストの状態に結びついた種々の振る舞いを実装。
class LockedState extends State is

    // ユーザーがロックされていたプレーヤーのロックを解除すると、二つの状
    // 態のうちの一つになる。
    method clickLock() is
        if (player.playing)
            player.changeState(new PlayingState(player))
        else
            player.changeState(new ReadyState(player))

    method clickPlay() is
        // ロックされているので何もしない。

    method clickNext() is
        // ロックされているので何もしない。

    method clickPrevious() is
        // ロックされているので何もしない。

// コンテキスト中の状態遷移の引き金ともなり得る。
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)
```

💡 適応性

💡 現在の状態に応じて異なる振る舞いをするオブジェクトがあり、状態数が多く、状態固有のコードの変更が頻繁な場合に、State パターンを使用します。

⚡ このパターンに従うには、すべての状態固有のコードを抽出し、複数の個別のクラスに入れます。その結果、新しい状態を追加したり、既存の状態を変更したりすることが独立して可能になり、保守コストを削減できます。

💡 クラスのフィールドの現在の値に応じて振る舞いを変えるために、クラスが膨大な数の条件文であふれている場合に、このパターンを使用します。

⚡ State パターンでは、これらの条件文の分岐を、対応する状態クラスのメソッドとして抽出します。これを行う際には、主クラス中の状態固有コードに関連した一時的なフィールドやヘルパー・メソッドを一掃することもできます。

💡 似たような複数の状態間や条件文に基づく状態機械の状態遷移に多数の重複したコードがある場合、State パターンを使用します。

⚡ State パターンを使用すると、状態クラスの階層を作成し、共通のコードを基底抽象クラスに抽出して重複を減らすことができます。

▣ 実装方法

1. コンテキストとして適切なクラスを決定します。状態依存のコードがある既存のクラスがそれかもしれません。もし状態固有のコードが複数のクラスに分散している場合は、新規クラスが必要となります。

2. 状態インターフェースを宣言します。コンテキスト内で宣言されたメソッド全部を反映させることもできますが、ステート固有の振る舞いを持つものだけに絞ることを目指してください。

3. 実際の状態ごとに、状態インターフェースを実装するクラスを作成します。次に、コンテキストのメソッドを見渡し、その状態に関連するすべてのコードを新しく作成したクラスに抽出します。

コードを状態クラスに移動する際に、それがコンテキストの非公開メンバーに依存していることを発見するかもしれません。これにはいくつかの回避策があります：

- これらのフィールドまたはメソッドを公開にする。
- 抽出中の振る舞いをコンテキスト中の公開メソッドとし、状態クラスから呼び出す。醜いが、即効性があり、後で修正可能。
- ステート・クラスをコンテキスト・クラスにネスト。ただし、プログラミング言語がクラスのネストをサポートしている場合に限定。

4. コンテキスト・クラスに、状態インターフェースの型の参照フィールドと、そのフィールドの値を変更できる公開 setter を追加します。

5. コンテキストのメソッドを再度一つずつ検証し、空（から）の状態条件文を状態オブジェクトに対する対応したメソッドの呼び出しと置き換えます。

6. コンテキストの状態を切り替えるには、状態クラスのどれか一つのインスタンスを作成し、コンテキストに渡します。これは、コンテキスト自身の内部でも、種々の状態内部、あるいはクライアント内でもできます。どこで行うにせよ、クラスは、それがインスタンスを作成した具象状態クラスに依存するようになります。

△ 長所と短所

- ✓ 単一責任の原則。特定の状態にまつわるコードを別クラスにして整理。
- ✓ 開放閉鎖の原則。既存の状態クラスやコンテキストを変更せずに新しい状態を導入。
- ✓ 状態機械にある、かさばった条件分を削減し、コンテキストのコードを簡素化。
- ✗ 状態機械に数個しか状態がない、またはめったに変更がない場合、このパターンの適用は過剰。

⇄他のパターンとの関係

- **Bridge**、**State**、**Strategy**（と限られた意味合いでは、**Adapter**も）は、非常に似た構造をしています。実際のところ、これらの全てのパターンは、合成に基づいており、仕事を他のオブジェクトに委任します。しかしながら、違う問題を解決します。パターンは、単にコードを特定の方法で構造化するためのレシピではありません。パターンが解決する問題に関して、開発者同士がするコミュニケーションの道具でもあります。
- **State** は、**Strategy** の拡張と考えることができます。どちらのパターンも合成（コンポジション）に基づいており、コンテキストの振る舞いの変更を、ヘルパー・オブジェクトに仕事の一部を委任することにより行います。**Strategy**では、これらのオブジェクトは完全に独立しており、互いを意識しません。しかし、**State**では、具象状態間の依存関係を制限せず、コンテキストの状態を自由に変更できます。

</> コード例



この電子書籍をベッドで読むべき理由は？

- Netflix より知的刺激に溢れています。
- 他のことをやるよりもリラックスできます。
- 実りある時間を過ごせます！
- 読書モードが選べて目に優しい。
- 全デバイスをサポート：PDF、EPUB、MOBI、KFX フォーマットに対応

詳細

ホーム



リファクタリング



デザインパターン



プレミアムコンテンツ

フォーラム

お問い合わせ

© 2014-2022 Refactoring.Guru. All rights reserved.

イラスト : Dmitry Zhart

ウクライナ Khmelnitske shosse 19 / 27, Kamianets-Podilskyi 32305

Email: support@refactoring.guru

利用規約

プライバシー・ポリシー

コンテンツ利用ポリシー