**🎙️ ステートパターン講義スクリプト（90分）完全台本＋詳細補足**

**🕐 0〜10分｜導入「ゲームキャラの“状態”ってなんだろう？」**

「こんにちは、今日は\*\*ステートパターン（State Pattern）\*\*について学んでいきます。

まず質問です。ゲームでプレイヤーがジャンプしているとき、しゃがんでいるとき、立っているとき…それぞれで動きや操作感って変わりますよね？」

「この“今の状態によって、動きが変わる”という仕組みをコードで表現したい。そういうときに使うのが、ステートパターンなんです。」

**🕐 10〜20分｜ステートパターンの概要と定義**

「ステートパターンとは、“オブジェクトの状態によって、ふるまい（処理）を切り替える”ためのデザインパターンです」

**🔑 用語整理**

* **State（状態）**：立ち・しゃがみ・ジャンプなど
* **Context（文脈・本体）**：状態を持っていて、それによって振るまいを切り替えるオブジェクト（プレイヤーなど）

「ポイントは、“今の状態を知って、それに応じて動く”という設計が、状態クラスごとにきれいに分けられるところです」

**🕐 20〜35分｜ゲームにおける使いどころとステートなし設計との違い**

**💬 「状態を持たない設計だったらどうなるか？」**

「もし、プレイヤーの動作をすべて1つのクラスに書いていたらどうなるでしょうか？

if (isJumping) { ... }

else if (isAttacking) { ... }

else if (isRunning) { ... }

このような条件分岐がどんどん増えていって、クラスが肥大化します。保守性が落ち、バグも入りやすくなります」

「ステートパターンを使えば、ジャンプ状態はJumpState、待機状態はIdleStateというように**責任を分けて整理**できます」

**🧠 状態ごとに整理して書くとどうなるか？（ゲーム例）**

|  |  |  |
| --- | --- | --- |
| **状況** | **状態** | **動作の変化** |
| プレイヤー移動中 | 待機・歩き・ジャンプ | 移動速度、アニメーション、ジャンプ可能か |
| 敵AI行動 | パトロール・警戒・攻撃 | 動きのパターンが全く変わる |
| ステージの状態 | 通常・イベント中・ゲームオーバー | 入力受付・画面遷移・BGMなど |

**🕐 35〜50分｜クラス図の構造と考え方**

**📌 Mermaid図で視覚化**

classDiagram

class State {

<<interface>>

+handle()

}

class IdleState {

+handle()

}

class JumpState {

+handle()

}

class Player {

-State\* state

+setState(State\*)

+update()

}

State <|.. IdleState

State <|.. JumpState

Player --> State

「このように、“状態”はインターフェースとして共通のhandle()を持ち、Context（プレイヤー）が持っている“今の状態”が実際の振る舞いを決めます」

**🕐 50〜70分｜C++実装で体感しよう**

「では、実際にC++でコードを書いてみましょう」

**🛠️ 実装構成**

1. State インターフェースを作成（純粋仮想関数）
2. IdleState, JumpState, AttackState のような状態クラスを用意
3. Player クラスで現在の状態を持ち、 setState() と update() を作る

**💻 実行例（出力）**

* プレイヤーが歩いています
* プレイヤーがジャンプしています！
* プレイヤーが攻撃しています！

「ここまでできれば、どんな新しい状態が来てもクラスを追加するだけで動作が広がります」

**🕐 70〜80分｜Strategyパターンとの違いと比較**

「よく似たデザインパターンに**Strategyパターン**があります」

|  |  |  |  |
| --- | --- | --- | --- |
| パターン | 目的 | 状態 or 戦略？ | 例 |
| State | 状態による動作切替 | 今の内部状態に応じて変化 | プレイヤーの立ち/ジャンプ/攻撃など |
| Strategy | 戦略の差し替え | 外部から“やり方”を変える | プレイヤーの攻撃方法（剣・弓・魔法） |

「Strategyは“どの戦い方にするか”を選ぶ。 Stateは“今どういう状態か”で自動的に切り替わる。 この違いを明確にしておくと、設計がうまくいきます」

**💡 違いのポイント整理**

* Strategy：プレイヤーが「剣」や「魔法」を選ぶ → 攻撃のやり方
* State：今は「ジャンプ中」か「攻撃中」か → 状態の違い

「**Strategyは外部から選ぶ、Stateは内部で変わる**、この意識が大切です」

**🕐 80〜90分｜演習・発展・まとめ**

**✍️ 演習課題（説明＋宿題）**

1. プレイヤーが「歩く」「ジャンプ」「攻撃」できるようにStateクラスを作る
2. ランダムに状態を切り替えて、それぞれの動作が正しく呼ばれることを確認する
3. Strategyパターンで“攻撃方法”の切り替えも組み合わせてみる（例：近距離 or 遠距離）

**💡 発展的な話題**

**🔄 状態遷移図を描いてみよう**

* 実際のゲームでは「ジャンプ中は攻撃できない」などの制約がある
* → 状態同士の関係を図で表すことで、全体の設計が見えてくる

**🔁 ステート遷移をコードで表現するには？**

* State クラス内に canTransitionTo(State\*) などを定義し、遷移を制限することもできる

**🧩 他のパターンとの組み合わせ**

* **Observerパターン**：状態が変わったら他クラスに通知（UI更新や音など）
* **Singletonパターン**：状態クラスを1インスタンスで使い回したいときに使える
* **Commandパターン**：状態によってプレイヤーの操作をコマンドとして扱うと拡張性が増す

**✅ まとめ**

* ステートパターンは「状態によって行動が変わる」設計に最適
* 条件分岐を減らし、責任を状態クラスに分けられる
* 状態を増やしても他のクラスに影響を与えず拡張できる（OCP）
* Strategyとの違いも押さえて、適材適所で使い分けよう
* 他のパターンと組み合わせることで、より強力な設計ができる

💡 状態と戦略は違う。両方知っていれば、もっとスマートにゲームが作れる！ 🎮 状態の数が増えれば増えるほど、Stateパターンの力が発揮されます。実務でもぜひ使ってみましょう！