デザインパターンについて

—

Singletonパターン
Commandパターン

目次

デザインパターンとは

デザインパターンとは、オブジェクト指向プログラミングのテクニックです。

例えば、クラスやオブジェクトの関係性を改善するための手法や、

オブジェクトの要求をカプセル化し、

メソッドの呼び出しとして表現する手法があります。

デザインパターンを理解し、適切に利用することで、より効率的に

ソフトウェアの開発を行うことができます。

デザインパターンとは

デザインパターンの大まかな種類には、

- **生成（Creational）**
- **構造（Structural）**
- **振る舞い（Behavioral）**

の3つがあります。

今回のスライドでは、

生成から「**Singletonパターン**」（たまに記事で見るから）

振る舞いから「**Commandパターン**」（今後に活かせそうだから）

をご紹介します。

メリット

1. 再利用性の向上

デザインパターンを適用することで、
設計上の問題に対する一般的な解決策を提供することができます。
これにより、特定の問題に直面した際に、過去の成功したパターンを再利用することが可能になります。

メリット

2. 柔軟性とメンテナンス性の向上

デザインパターンは優れた設計原則に基づいている為、
コードをより柔軟でメンテナンスしやすくします。
パターンによって変更に対する影響を最小限に抑え、
コードの拡張や修正がしやすくなります。

メリット

3. 統一した開発手法の促進

デザインパターンはチーム全体で共有される開発手法を促進します。
パターンを利用することで、
チームメンバー間で共通言語や設計原則を使用することができ、
コード全体の一貫性が向上します。

デメリット

1. 過度な複雑さ

デザインパターンを過剰に使用すると、
システムの複雑さが増してしまい、
コードの可読性や理解性が低下する可能性があります。
そのため、過剰に使用すると、
逆に開発プロセスを混乱させることになります。

デメリット

2. 柔軟性の欠如

過度なデザインパターンの使用は、
固定化された設計や過剰な抽象化につながる可能性があります。
これにより、将来の変更や拡張に対応する柔軟性が制限され、
新しい要件に対応することが難しくなるリスクが生じてしまいます。

デメリット

3. 過度な抽象化

過度なデザインパターンの使用は、余分な抽象化を導入する傾向があります。
過剰な抽象化は、コードの理解を難しくし、
開発者同士のコミュニケーションを困難にします。

Singletonパターンとは

ここから「Singletonパターン」について見ていきます。

Singletonパターンの特徴は、

クラスのインスタンスが一つしかない状態を保証するパターンです。

「インスタンスが一つしかない」という前提を生み出す事で、

複数のインスタンスを用いる事によるバグの発生リスクを防ぎます。

また、一つしかインスタンスを作成しない事で、メモリやCPUのリソースを

抑えることができます。

Singletonパターンとは

Singletonパターンに則ったクラスの条件は

- publicのコンストラクタではなく、privateのコンストラクタを使用
- private変数として自身のクラスインスタンスを所持
- public関数として、自身のクラスインスタンスを返すgetterメソッドを持っている。

という3点があります。

Singletonパターンとは

なぜクラスを作成するのに条件が必要なのか

上記については、外部からのインスタンス生成を制限するために

コンストラクタでprivateを使用しています。

また、子クラスといった派生クラスを作成しないようにするためでもあります。

その為、Singletonのクラスはメソッドを通してインスタンスにアクセス

するしかありません。

Singletonパターンとは

どういう時にSingletonを使用するのか

例えば、データベースの接続に使用したり、

設定管理、ログ管理に使用することがあります。

Laravelの場合だと、サービスコンテナにsingletonメソッドがあり、

1回依存解決を行うと、今後の呼び出しでは同じオブジェクトインスタンスが

返却されます。

メリット

1. **リソース効率:**

   プログラム内で唯一のインスタンスを共有するので、

   リソースの効率的な利用が可能となります。

2. **設計の一貫性:**

   同じクラスから生成されたインスタンスは常に同一であるため、

   システム内でのオブジェクトの一貫性が保たれます。

   これにより、意図しない状態遷移や不整合を防ぐことができます。

コード例

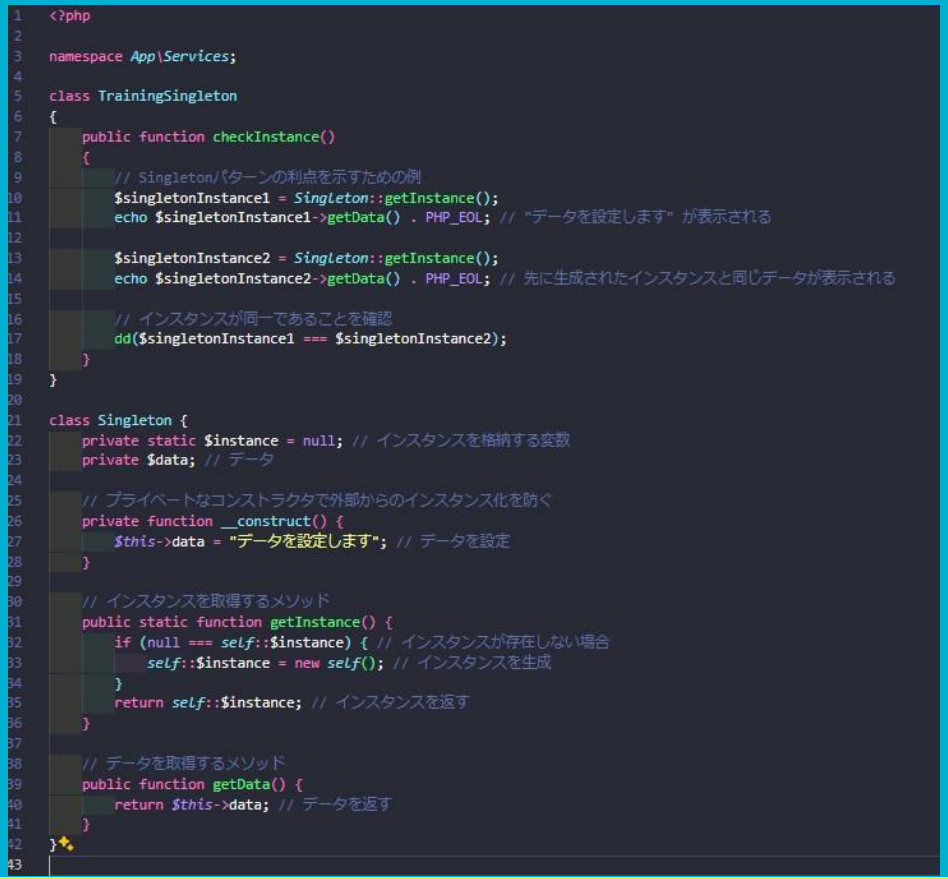
実際にコードでSingletonについて

説明していきます。

用意するのは

- TrainingSingleton
- Singleton

上記2つのクラスです。



```php
<?php

namespace App\Services;

class TrainingSingleton
{
    public function checkInstance()
    {
        // Singletonパターンの利点を示すための例
        $singletonInstance1 = Singleton::getInstance();
        echo $singletonInstance1->getData() . PHP_EOL; // "データを設定します" が表示される

        $singletonInstance2 = Singleton::getInstance();
        echo $singletonInstance2->getData() . PHP_EOL; // 先に生成されたインスタンスと同じデータが表示される

        // インスタンスが同一であることを確認
        dd($singletonInstance1 === $singletonInstance2);
    }
}

class Singleton {
    private static $instance = null; // インスタンスを格納する変数
    private $data; // データ

    // プライベートなコンストラクタで外部からのインスタンス化を防ぐ
    private function __construct() {
        $this->data = "データを設定します"; // データを設定
    }

    // インスタンスを取得するメソッド
    public static function getInstance() {
        if (null === self::$instance) { // インスタンスが存在しない場合
            self::$instance = new self(); // インスタンスを生成
        }
        return self::$instance; // インスタンスを返す
    }

    // データを取得するメソッド
    public function getData() {
        return $this->data; // データを返す
    }
}
```

コード例

TrainingSingletonクラスでは、
生成したインスタンスが
同一のインスタンスか確認します。
Singletonクラスでは、
コンストラクタをprivateにし、
一つ以上のインスタンスが生成
されないようにしています。

```php
<?php

namespace App\Services;

class TrainingSingleton
{
    public function checkInstance()
    {
        // Singletonパターンの利点を示すための例
        $singletonInstance1 = Singleton::getInstance();
        echo $singletonInstance1->getData() . PHP_EOL; // "データを設定します" が表示される

        $singletonInstance2 = Singleton::getInstance();
        echo $singletonInstance2->getData() . PHP_EOL; // 先に生成されたインスタンスと同じデータが表示される

        // インスタンスが同一であることを確認
        dd($singletonInstance1 === $singletonInstance2);
    }
}

class Singleton {
    private static $instance = null; // インスタンスを格納する変数
    private $data; // データ

    // プライベートなコンストラクタで外部からのインスタンス化を防ぐ
    private function __construct() {
        $this->data = "データを設定します"; // データを設定
    }

    // インスタンスを取得するメソッド
    public static function getInstance() {
        if (null === self::$instance) { // インスタンスが存在しない場合
            self::$instance = new self(); // インスタンスを生成
        }
        return self::$instance; // インスタンスを返す
    }

    // データを取得するメソッド
    public function getData() {
        return $this->data; // データを返す
    }
}
```

コード例

Singletonクラスでは、

コンストラクタをprivate化

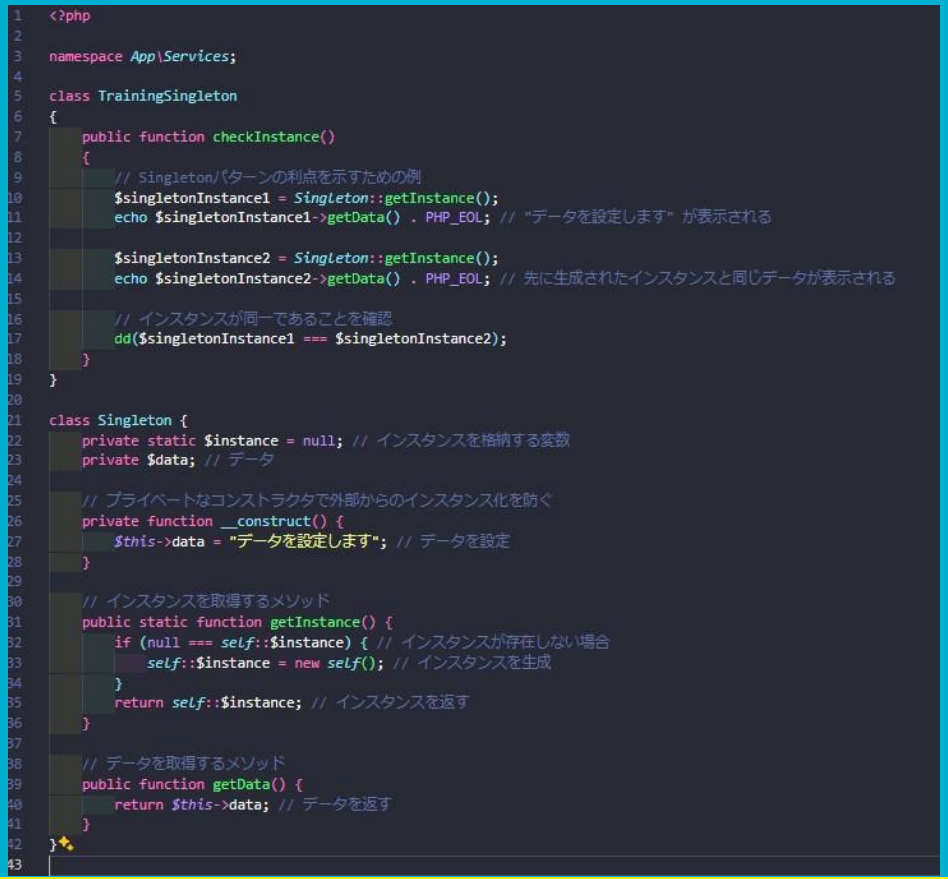
している為、クラス自体を

インスタンス化できません。

その為、getInstance()メソッドで、

インスタンスを作成 / 呼び出し

を行い、インスタンス返却を行います



```php
<?php

namespace App\Services;

class TrainingSingleton
{
    public function checkInstance()
    {
        // Singleton/パターンの利点を示すための例
        $singletonInstance1 = Singleton::getInstance();
        echo $singletonInstance1->getData() . PHP_EOL; // "データを設定します" が表示される

        $singletonInstance2 = Singleton::getInstance();
        echo $singletonInstance2->getData() . PHP_EOL; // 先に生成されたインスタンスと同じデータが表示される

        // インスタンスが同一であることを確認
        dd($singletonInstance1 === $singletonInstance2);
    }
}

class Singleton {
    private static $instance = null; // インスタンスを格納する変数
    private $data; // データ

    // プライベートなコンストラクタで外部からのインスタンス化を防ぐ
    private function __construct() {
        $this->data = "データを設定します"; // データを設定
    }

    // インスタンスを取得するメソッド
    public static function getInstance() {
        if (null === self::$instance) { // インスタンスが存在しない場合
            self::$instance = new self(); // インスタンスを生成
        }
        return self::$instance; // インスタンスを返す
    }

    // データを取得するメソッド
    public function getData() {
        return $this->data; // データを返す
    }
}
```

コード例

CheckInstanceメソッドで

Singletonクラスを呼び出し、

$singletonInstance1と

$singletonInstance2の

2つの変数にSingletonインスタンスを

代入します。

次に、変数同士を比較し、

本当に同じインスタンスか確認します

```php
<?php

namespace App\Services;

class TrainingSingleton
{
    public function checkInstance()
    {
        // Singletonパターンの利点を示すための例
        $singletonInstance1 = Singleton::getInstance();
        echo $singletonInstance1->getData() . PHP_EOL; // "データを設定します" が表示される

        $singletonInstance2 = Singleton::getInstance();
        echo $singletonInstance2->getData() . PHP_EOL; // 先に生成されたインスタンスと同じデータが表示される

        // インスタンスが同一であることを確認
        dd($singletonInstance1 === $singletonInstance2);
    }
}

class Singleton {
    private static $instance = null; // インスタンスを格納する変数
    private $data; // データ

    // プライベートなコンストラクタで外部からのインスタンス化を防ぐ
    private function __construct() {
        $this->data = "データを設定します"; // データを設定
    }

    // インスタンスを取得するメソッド
    public static function getInstance() {
        if (null === self::$instance) { // インスタンスが存在しない場合
            self::$instance = new self(); // インスタンスを生成
        }
        return self::$instance; // インスタンスを返す
    }

    // データを取得するメソッド
    public function getData() {
        return $this->data; // データを返す
    }
}
```

コード例

比較した結果がこちらです。

```
root@5b645c211561:/var/www/html/src# php artisan tinker
Psy Shell v0.12.0 (PHP 8.2.1 — cli) by Justin Hileman
> $i = app(TrainingSingleton::class)
[!] Aliasing 'TrainingSingleton' to 'App\Services\TrainingSingleton' for this Tinker session.
= App\Services\TrainingSingleton {#5352}

> $i->checkInstance()
データを設定します
データを設定します
true // app/Services/TrainingSingleton.php:17
root@5b645c211561:/var/www/html/src#
```

比較を行った結果「**true**」が返却されました。

このようにして、インスタンスを一つしか生成せず、

作業を行うことが可能となります。

Singleton まとめ

・クラスのインスタンスが1つしか存在しないことを保証するデザインパターン。

・Singletonクラス内部にプライベートな静的変数を持ち、
そのクラスの唯一のインスタンスを格納する。

・Singletonパターンは、インスタンスを1つだけ生成し、
そのインスタンスへの唯一のアクセスポイントを提供することで、
システム全体の整合性やセキュリティを確保する役割を果たす。

Commandパターンとは

Commandパターンとは、

特定の操作をオブジェクトとしてカプセル化し、

操作を呼び出すためのコマンドとして表現するデザインパターンです。

このパターンによって、命令の送信者と受信者を切り離し、

柔軟性を高めることができます。

Commandパターンとは

Commandパターンは下記の要素で構成されています。

1. **Command（コマンドインターフェース）**
2. **ConcreteCommand（具象コマンド）**
3. **Invoker（呼び出し）**
4. **Receiver（受信）**

次のスライドより、一つずつ説明をしていきます。

Commandパターンとは

1. Command（コマンドインターフェース）

実行される操作を表すインターフェースを定義します。

これにより、異なるコマンドを同じように扱うことができます。

2. ConcreteCommand（具象コマンド）

Command インターフェースを実装し、

実際に操作を実行する具象クラスです。

Commandパターンとは

3. Invoker（呼び出し）

コマンドを受け取り、コマンドの実行を要求するクラスです。

4. Receiver（受信）

実際の操作を行うクラスであり、命令の受信と実行を担当します。

メリット

1. **柔軟性:**

   コマンドパターンは、異なるコマンドを追加したり、既存のコマンドを変更したりすることが容易であり、システムの柔軟性を高めます。

2. **拡張性:**

   新しいコマンドを追加する際には既存のコードに手を加える必要がなく、新しいコマンドクラスを作成するだけで済むため、システムの拡張性が向上します。

コード例

実際にコードで動作を見ていきましょう。

今回のコードを見ていく上でテレビのリモコンを例に進めます。

まずは要素の確認です。

コード例

1. Command（Commandインターフェース）

リモコンのボタンを表すインタフェースです。

Command インターフェースがこの要素にあたります。

```php
// Command Interface
interface Command {
    public function execute();
}
```

コード例

2. Concrete Command（具象クラス）

リモコンの各ボタンを

具体的に表現するクラスです。

TurnOnCommand クラスと

TurnOffCommand クラスが

これに該当します。

```php
// Concrete Commands
class TurnOnCommand implements Command {
    private $device;

    public function __construct($device) {
        $this->device = $device; // デバイス（TV）を設定
    }

    public function execute() {
        $this->device->turnOn(); // テレビをオンにする
    }
}

class TurnOffCommand implements Command {
    private $device;

    public function __construct($device) {
        $this->device = $device; // デバイス（TV）を設定
    }

    public function execute() {
        $this->device->turnOff(); // テレビをオフにする
    }
}
```

コード例

3. Receiver（受信）

テレビ本体が実際にコマンドを受け取って操作を行うオブジェクトです。

TV クラスがこの役割を果たします。

```php
// Receiver (TV)
class TV {
    public function turnOn() {
        echo "電源を点ける\n"; // テレビをオンにする
    }

    public function turnOff() {
        echo "電源を切る\n"; // テレビをオフにする
    }
}
```

コード例

4. Invoker（呼び出し）

実際の操作を行うクラスです。

このクラスにリモコンのコマンドを設定し、実行を行います。

```php
// Invoker (Remote Control)
class RemoteControl {
    private $command;

    public function setCommand(Command $command) {
        $this->command = $command; // コマンドを設定
    }

    public function pressButton() {
        $this->command->execute(); // ボタンを押したら対応するコマンドを実行
    }
}
```

コード例

今回の実行クラスはこちらです。

```php
public function checkInstance()
{
    // Let's test the Command Pattern
    $tv = new TV();

    $turnOnCommand = new TurnOnCommand($tv); // テレビをオンにするコマンドを生成
    $turnOffCommand = new TurnOffCommand($tv); // テレビをオフにするコマンドを生成

    $remoteControl = new RemoteControl(); // リモコンを生成

    // Turning on the TV
    $remoteControl->setCommand($turnOnCommand); // リモコンにテレビをオンにするコマンドを設定
    $remoteControl->pressButton(); // リモコンのボタンを押す

    // Turning off the TV
    $remoteControl->setCommand($turnOffCommand); // リモコンにテレビをオフにするコマンドを設定
    $remoteControl->pressButton(); // リモコンのボタンを押す
}
```

コード例

実行結果がこちらになります。

今回の実行では、電源を付けるクラス、電源を消すクラスを実行している為、

両方出力されていますが、これにより、同じ動線の中で

リモコンの操作を切り替える事に成功しました。

```
root@5b645c211561:/var/www/html/src# php artisan tinker
Psy Shell v0.12.0 (PHP 8.2.1 — cli) by Justin Hileman
> $i = app(TrainingCommand::class)
[!] Aliasing 'TrainingCommand' to 'App\Services\TrainingCommand' for this Tinker session.
= App\Services\TrainingCommand {#5352}

> $i->checkInstance()
電源を点ける
電源を切る
= null
```

Command まとめ

・Commandパターンは、操作や要求をオブジェクトにカプセル化し、操作を呼　び出すクラスと実行するクラスを分離するデザインパターンである。

・Commandパターンの要素には、Command（インターフェース）、Concrete Command（具象コマンド）、Receiver（受信）、Invoker（呼び出し）。

・Commandパターンは、オブジェクト指向プログラミングにおいて操作のカプ　セル化と柔軟性を提供する重要なデザインパターンであり、システムの保守性　や拡張性を高める際に役立つ。

まとめ

・デザインパターンは、コードの最適化、再利用、理解、アップグレードと修復　を容易にするのに役立ちます。

・適切なデザイン パターンを理解して適用すれば、多くの時間と労力を節約し、 開発、拡張、保守が容易になります。

・覚えたての状態で何でも使いすぎるのは良くなく、
　使うべきタイミングをちゃんと見極める。

ご清聴
ありがとうございました。