Template Methodパターン教材

授業計画

授業の目標

- Template Methodパターンの基本的な概念を理解する。
- 簡単なC++のプログラムを通じて、Template Methodパターンの使い方を学ぶ。

授業の流れ

1. **イントロダクション**
    - 授業の目標と概要を説明。
    - デザインパターンとは何かを簡単に説明。
2. **Template Methodパターンの概念**
    - Template Methodパターンの基本的な説明。
    - 身近な例を使ってイメージしやすく説明。
3. **簡単なコード例の解説**
    - シンプルなC++のコード例を使ってTemplate Methodパターンの実装方法を説明。
    - コードの逐次説明と質疑応答。
4. **実習**
    - 学生にTemplate Methodパターンを用いた簡単なプログラムを作成させる。
    - テーマ例: 簡単な料理のレシピ、ゲームキャラクターの動作など。
5. **まとめ**
    - 授業の内容を振り返り、重要ポイントを再確認。
    - 質疑応答の時間を設ける。

イントロダクション

デザインパターンとは？

デザインパターンは、プログラミングにおける一般的な問題を解決するためのテンプレートです。これを使うと、コードが再利用しやすくなり、保守もしやすくなります。

Template Methodパターンの概要

Template Methodパターンは、処理の流れを親クラスで決めておき、詳細な処理を子クラスに任せる方法です。これは、料理のレシピのようなものです。料理の手順（流れ）は決まっていますが、具体的な食材や調理方法はレシピごとに異なります。

Template Methodパターンの概念

意図

Template Methodパターンの意図は、共通の処理の流れを親クラスで定義し、具体的な処理を子クラスに任せることです。これにより、共通のアルゴリズム構造を保持しながら、具体的な処理を変更できます。

例: 料理のレシピ

料理の準備手順は共通ですが、料理の種類によって具体的な調理方法が異なります。例えば、「料理を作る」という流れは共通ですが、「パスタを作る」と「スープを作る」の具体的な手順は異なります。

メリット

- **再利用性の向上**: 共通の処理を親クラスにまとめることで、コードの再利用性が向上します。
- **柔軟性**: サブクラスで具体的な処理を変更できるため、柔軟性があります。

デメリット

- **複雑性の増加**: クラスの数が増えると、全体の構造が複雑になることがあります。
- **依存関係**: サブクラスが親クラスに依存するため、変更が伝播する可能性があります。

使用する場面

- **共通の処理があるが、特定のステップが異なる場合**: 例えば、レポート生成やデータのフォーマット。
- **複数のアルゴリズムが共通のステップを持つ場合**: 例えば、ゲームキャラクターの動作。

使用しない場面

- **全てのステップが異なる場合**: Template Methodパターンのメリットが薄れる。
- **シンプルな処理の場合**: 不必要に複雑化する可能性があります。

簡単なコード例の解説

共通の料理の流れ

1.　Meal.hクラスの作成

Visual Studioを立ち上げたら、まずはMeal.hクラスを作成し、 以下のコードを書いてみてください。

```
#ifndef MEAL_H
#define MEAL_H

#include <iostream>


// 料理の基本的な流れを定義するクラス
 class Meal {
public:
    virtual ~Meal() {}
```

```
    // 料理を準備するためのテンプレートメソッド
    void prepareMeal() {       //重要！
        prepareIngredients(); // 材料を準備する
        cook();               // 料理を調理する
        serve();              // 料理を提供する
    }

protected:
    // 材料を準備するための抽象メソッド
    virtual void prepareIngredients() = 0;

    // 料理を調理するための抽象メソッド
    virtual void cook() = 0;

    // 料理を提供するためのメソッド
    void serve() {
        std::cout << "料理を提供します。" << std::endl;
    }
};

#endif // MEAL_H
```

ここのコードの[prepareMeal()]は、関数が呼び出されたら、

1.材料を準備するコードをまとめた、prepareIngredients()が実行される。
2.調理するコードをまとめた、cook()が実行される。
3.料理を提供するコードをまとめた、serve()が実行される。

という、流れに基づいて実行されるコードになります。
ただし、このコードだけでは関数と実行する流れ作っただけなので、
継承先の子クラスに、具体的な処理内容を実装していきます。

2.　Pastaクラスの作成

Meal.hを作成した後、それを継承した1つめのクラス、PASTAクラスを作成します。
Meal.hクラスで継承した、prepareIngredients()とcook()クラスの中身を実装します。
下記のコードを書いてみましょう。

```
#ifndef PASTA_H
#define PASTA_H

#include "Meal.h"

// パスタ料理の具体的な実装クラス
class Pasta : public Meal {
protected:
    // パスタの材料を準備する
    void prepareIngredients() override {
        std::cout << "パスタの材料を準備します。" << std::endl;
    }
```

```
    // パスタを調理する
    void cook() override {
        std::cout << "パスタを調理します。" << std::endl;
    }
};

#endif // PASTA_H
```

このコードにより、継承元を材料を準備するクラスである、prepareIngredients()関数と、調理をするcook()関数の 具体的な中身の処理が実装できるようになりました。
もう一つ、具体的な子クラスを作っていこうと思います。

3.　SOUP.hクラスの作成

Meal.hを作成した後、それを継承した2つめのクラスSOUPクラスを作成します。
こちらもMeal.hクラスで継承した、prepareIngredients()とcook()クラスの中身を実装します。
下記のコードを書いてみましょう。

```
#ifndef SOUP_H
#define SOUP_H

#include "Meal.h"

// スープ料理の具体的な実装クラス
class Soup : public Meal {
protected:
    // スープの材料を準備する
    void prepareIngredients() override {
        std::cout << "スープの材料を準備します。" << std::endl;
    }

    // スープを調理する
    void cook() override {
        std::cout << "スープを調理します。" << std::endl;
    }
};

#endif // SOUP_H
```

こちらも、継承元を材料を準備するクラスである、prepareIngredients()関数と、調理をするcook()関数の 具体的な中身の処理が実装できるようになりました。

ここまで実装できたら、最後にこれらのコードを呼び出すためのMainクラスのコードを作成します。

4.　Main.cppの作成

```
#include "Pasta.h"
#include "Soup.h"

int main() {
    // パスタ料理の準備
    Meal* pasta = new Pasta();
    pasta->prepareMeal();
    delete pasta;

    // スープ料理の準備
    Meal* soup = new Soup();
    soup->prepareMeal();
    delete soup;

    return 0;
}
```

ここまで書いたら、F5ボタンを押してビルドしてみてください。
パスタクラス、スープクラスともに、親クラスで作成したprepareMeal()関数の流れを汲んで
それぞれのクラスで具体的に設定した処理が行われていることを確認できていたらOKです。

まとめ　Template Methodパターンを学ぶメリット

コードの再利用性が向上する

共通の処理を親クラスにまとめることで、同じ処理を複数のサブクラスで共有できるため、コードの重複を避けることができます。

- **例**：料理の準備手順のように、共通の手順がありつつ、具体的な内容は異なる場合に便利です。

柔軟性が高まる

サブクラスで具体的な処理を変更することができるため、新しい要件や異なる仕様に対して柔軟に対応できます。

- **例**：新しい料理のレシピを追加する際に、既存のコードを変更せずに対応できます。

コードの保守性が向上する

共通の処理が親クラスに集約されるため、修正が必要な場合でも、親クラスのコードを変更するだけで済みます。

- **例**：料理の提供手順を変更する場合、親クラスの`serve`メソッドを修正するだけで、全ての料理に反映されます。

具体的な例

レポート生成システム

例えば、レポート生成システムで、共通のレポート生成手順がありつつ、具体的なデータ取得方法やフォーマットが異なる場合にTemplate Methodパターンを使用できます。

- 共通の生成手順を親クラスにまとめ、データ取得とフォーマットをサブクラスで実装することで、異なるレポートを簡単に作成できます。

ゲームキャラクターの動作

ゲームで、キャラクターの動作（移動、攻撃など）の共通の流れを定義し、具体的な動作をサブクラスで実装することで、異なるキャラクターを簡単に作成できます。

- **例**：共通の`move`と`attack`メソッドを持つ親クラスを作り、キャラクターごとの具体的な動作をサブクラスで実装します。

使用する場面

共通の処理があり、特定のステップが異なる場合

- **例**：レポート生成、データのフォーマット、料理のレシピなど。

複数のアルゴリズムが共通のステップを持つ場合

- **例**：ゲームキャラクターの動作、ファイル処理の手順など。

使用しない場面

全てのステップが異なる場合

Template Methodパターンのメリットが薄れるため。

シンプルな処理の場合

不必要に複雑化する可能性があるため。