템플릿 메서드 패턴 (Template Method Pattern)

Wikipedia의 Template Method pattern 문서를 한글로 번역한 뒤 스터디 발표를 위해 정리한 문서입니다. 👉 https://en.wikipedia.org/wiki/Template_method_pattern

개요

템플릿 메서드 패턴(Template Method Pattern)은 객체 지향 프로그래밍에서 알고리즘의 골격을 기본 클래스에서 정의하고, 서브클래스가 특정 단계의 구체적인 구현을 제공할 수 있도록 하는 행위 디자인 패턴입니다.

이 패턴은 Gang of Four의 "Design Patterns" 책에서 소개되었으며, 소프트웨어 엔지니어링에서 널리 사용되는 디자인 패턴 중 하나입니다.

주요 특징

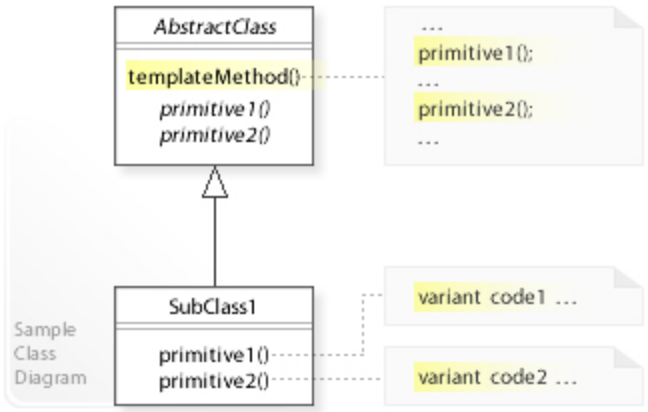
- **상속을 통한 알고리즘 변형**: 상속을 사용하여 알고리즘의 일부를 변경합니다
- **추상 슈퍼클래스에서 템플릿 메서드 구현**: 알고리즘의 골격을 정의합니다
- **상위 수준의 단계 정의**: 헬퍼 메서드를 통해 알고리즘의 주요 단계를 정의합니다
- **훅 메서드와 추상 메서드를 통한 커스터마이징**: 서브클래스가 특정 단계를 재정의할 수 있습니다

핵심 구조

템플릿 메서드 패턴은 다음과 같은 주요 구성 요소로 이루어져 있습니다:

1. **추상 기본 클래스**: 템플릿 메서드를 포함합니다
2. **템플릿 메서드**: 알고리즘의 골격을 정의하고 추상 메서드나 훅 메서드를 호출합니다
3. **추상 메서드**: 서브클래스에서 반드시 구현해야 하는 메서드입니다
4. **훅 메서드**: 서브클래스에서 선택적으로 재정의할 수 있는 메서드입니다
5. **구체적인 서브클래스**: 추상 메서드를 구현하여 특정 변형을 제공합니다

템플릿 메서드 패턴의 UML 구조



Java 예제 : 알고리즘의 골격(템플릿 메소드)이 구현되어 있는 추상 클래스

```java
/**
 * 추상 클래스는 템플릿 메서드를 구현하고, 추상 메서드를 선언합니다.
 */
abstract class Game {
    // 템플릿 메서드 - 알고리즘의 골격을 정의
    public final void play() {
        initialize();
        startPlay();
        endPlay();
    }

    // 추상 메서드 - 서브클래스에서 구현
    abstract void initialize();
    abstract void startPlay();
    abstract void endPlay();
}
```

Java 예제 : 추상 클래스인 Game의 Cricket 구현클래스

```
class Cricket extends Game {
    @Override
    void initialize() {
        System.out.println("Cricket Game Initialized! Start playing.");
    }

    @Override
    void startPlay() {
        System.out.println("Cricket Game Started. Enjoy the game!");
    }

    @Override
    void endPlay() {
        System.out.println("Cricket Game Finished!");
    }
}
```

Java 예제 : 추상 클래스인 Game의 Football 구현 클래스

```
class Football extends Game {
    @Override
    void initialize() {
        System.out.println("Football Game Initialized! Start playing.");
    }

    @Override
    void startPlay() {
        System.out.println("Football Game Started. Enjoy the game!");
    }

    @Override
    void endPlay() {
        System.out.println("Football Game Finished!");
    }
}
```

Java 예제 : 클라이언트 코드

```
/**
 * 클라이언트 코드
 */
public class TemplatePatternDemo {
    public static void main(String[] args) {
        Game game = new Cricket();
        game.play();
        game = new Football();
        game.play();
    }
}
```

장점

1. **코드 중복 제거**: 공통 코드를 부모 클래스에 두어 중복을 피합니다
2. **제어 지점 명확화**: 서브클래스가 커스터마이징할 수 있는 지점을 명확히 합니다
3. **프레임워크에 유용**: 프레임워크 개발 시 커스터마이징 포인트를 제공하는 데 유용합니다
4. **할리우드 원칙**: "우리가 당신을 호출할 것입니다, 당신이 우리를 호출하지 마세요" (Hollywood Principle - "Don't call us, we'll call you")
5. **알고리즘 구조 재사용**: 동일한 알고리즘 구조를 여러 구현에서 재사용할 수 있습니다

단점

1. **상속의 제약**: 상속을 사용하므로 유연성이 제한될 수 있습니다
2. **리스코프 치환 원칙 위반 가능성**: 잘못 설계하면 LSP를 위반할 수 있습니다
3. **템플릿 메서드가 많아질 수 있음**: 복잡한 알고리즘의 경우 관리가 어려워질 수 있습니다

사용 사례

템플릿 메서드 패턴은 다음과 같은 상황에서 유용하게 사용됩니다:

1. **프레임워크 개발**: 사용자가 특정 단계를 커스터마이징할 수 있도록 합니다
2. **코드 생성**: 공통 구조를 유지하면서 다양한 출력을 생성합니다
3. **데이터 처리 파이프라인**: 동일한 처리 단계를 거치지만 각 단계의 구현이 다른 경우
4. **테스트 프레임워크**: 테스트 실행의 공통 구조를 정의합니다
5. **문서 생성**: 동일한 문서 구조를 유지하면서 내용을 변경합니다

실제 적용 예시

1. 데이터 마이닝 애플리케이션

```
abstract class DataMiner {
    // 템플릿 메서드
    public void mine(String path) {
        File file = openFile(path);
        Data rawData = extractData(file);
        Data processedData = parseData(rawData);
        Data analyzedData = analyzeData(processedData);
        sendReport(analyzedData);
        closeFile(file);
    }

    abstract File openFile(String path);
    abstract Data extractData(File file);
    abstract Data parseData(Data rawData);

    // 훅 메서드 - 선택적 재정의
    protected Data analyzeData(Data data) {
        // 기본 분석 구현
        return data;
    }

    abstract void sendReport(Data data);
    abstract void closeFile(File file);
}

class PDFDataMiner extends DataMiner {
    // PDF 특화 구현...
}

class CSVDataMiner extends DataMiner {
    // CSV 특화 구현...
}
```

2. UI 컴포넌트 렌더링

```
abstract class Widget {
    // 템플릿 메서드
    public final void render() {
        beforeRender();
        doRender();
        afterRender();
    }

    // 훅 메서드
    protected void beforeRender() {}
    protected void afterRender() {}

    // 추상 메서드
    abstract void doRender();
}
```