TEMPLATE METHOD DESIGN PATTERN

7.2.1 Refactoring

- issue) when the common code segments are intermixed with context-specific code.

```
Class COntextA  {
        void method1(…) {
          (common code segment1)
          (Context-Specific code A)
          (common code Segment2)
        }
        //…
}
```

```
Class COntextB  {
        void method2(…) {
          (common code segment1)
          (Context-Specific code B)
          (common code Segment2)
        }
        //…
}
```

- Refactoring of Common Code Segments Intermixed with Context Specific Code
  - method1() and method2() … have some common code
  - some of them…. Intermixed with some context specific code
  - How Can the common code segments be refactored?
    - 1) to refactor each common code segment into
    - 2) to extract the entire method and then refactor

factoring

```
Class Common   {
          void commonCOde1() {
            (common code segment1)
          }
          void commonCOde2() {
            (common code segment1)
          }
}
```

```
Class ContextA   extends Common
{
          void method(….) {
             commonCOde1()
             (Context-Specific code A)
             commonCode2()
          }
         //
}
```

```
Class ContextB   extends Common
{
          void method(….) {
             commonCOde1()
             (Context-Specific code B)
             commonCode2()
          }
         //
}
```

- first approach) each common code into a separate method
  - refactoring by inheritance

7.2.1 Refactoring

- perfectly good if the two common code ... relatively independent
- error-prone if .... Closely related. And are merely pieces of a larger process
- breaks the logical flow
- hampers the readability of the code
- If one of them is omitted or they are invoked in a different order, unpredictable outcomes may result.

7.2.1 Re

- Second approach) more generic approach
  - extract the entire method that contains both common and context specific code to a superclass
  - and then refactor the context specific code by introducing a new method as which is intended to be overridden and customized in each subclass.

```
Class Common   {
        void method(…) {
          (common code segment1)
          contextSpecificCOde();
          (common code segment2)
        }
       void contextSpecificCOde(){}
            // No implementation
}
Class COntextA extends Common{
        void contextSpecificCode(){
          (Context specific code A)
        }
        //…
}
Class COntextB extends Common{
        void contextSpecificCode(){
          (Context specific code B)
        }
        //…
}
```

7.2.1 Refactoring

- declared to serve as a placeholder for the context specific code
- no implementation
- abstract superclass

- .

```
Class Common   {
        void mehtod(…) {
          (common code segment1)
          contextSpecificCOde();
          (common code segment2)
        }
       abstract void contextSpecificCOde();

}
```

7.2.1 Refactoring

- An abstract superclass
  - when refactoring recurring code segments intermixed with context-specific code.
  - the recurring code is extracted, as before, and an abstract method is declared to serve as a placeholder for the context-specific code.
  - High-level of encapsulation
    - the designer of subclass may focus their attention to the context-specific code.

7.2.2 Design Pattern : Template Method

- Design Pattern:
  - an abstract class *[DBAnimationApplet]*
    - serves as a template for classes with shared functionality.
    - contains behavior that is common to all its subclass
    - 1)
      - common behavior is encapsulated in nonabstract methods
    - 2)
      - require that context-specific behavior be implemented for each concrete subclass
      - ex) paintFrame()
      - Acts as a placeholder for the behavior that is implemented differently for each specific context. ➔

7.2.2 Design Pattern : Template Method

- Hook method
  - upon which context-specific behavior may be hung, or implemented
- Template method
  - Methods containing hooks where hook is placed, invoked.
- Ex) the double-buffered generic animation applet
  - the abstract method : paintFrame()
    - represents the behavior that is changeable
    -
  - update() method
    - using the hook method, we are able to define the update() method.
    - (uses hook methods to define a common behavior)

7.2.2 Design Pattern : Template Method

- Frozen spots / hot spots
  - frozen spots : (template methods)
    - describe the fixed behaviors of a generic class
  - hot spots : hook methods
    - changeable behaviors of a generic class

7.2.2 [

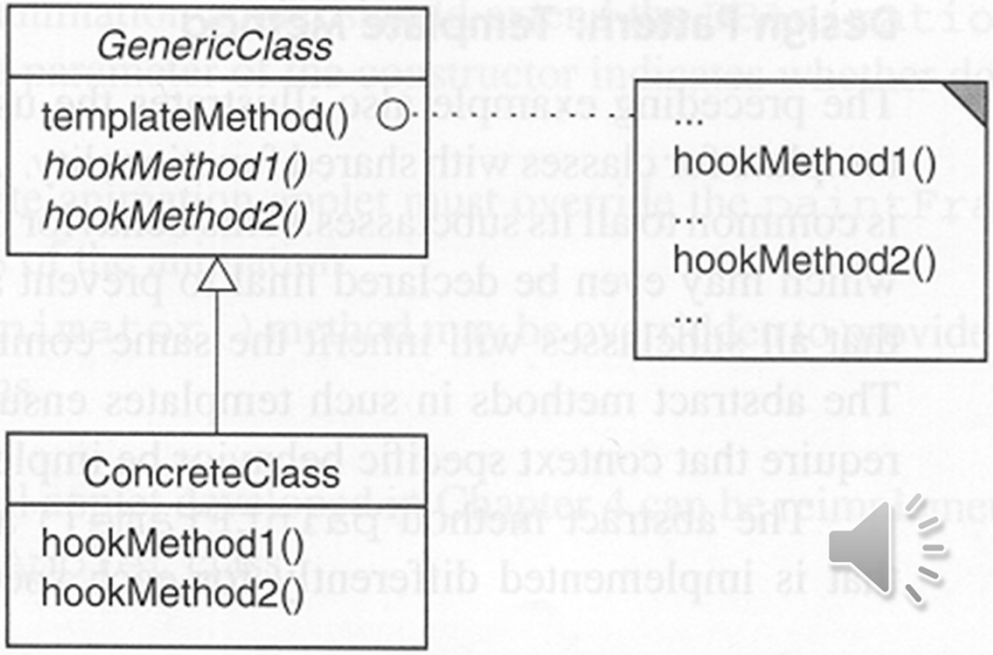
Design Pattern ***Template Method***

Category: Behavioral design pattern.

Intent: Define the skeleton of an algorithm in a method, deferring some steps to subclasses, thus allowing the subclasses to redefine certain steps of the algorithm.

Applicability: The Template Method pattern should be used

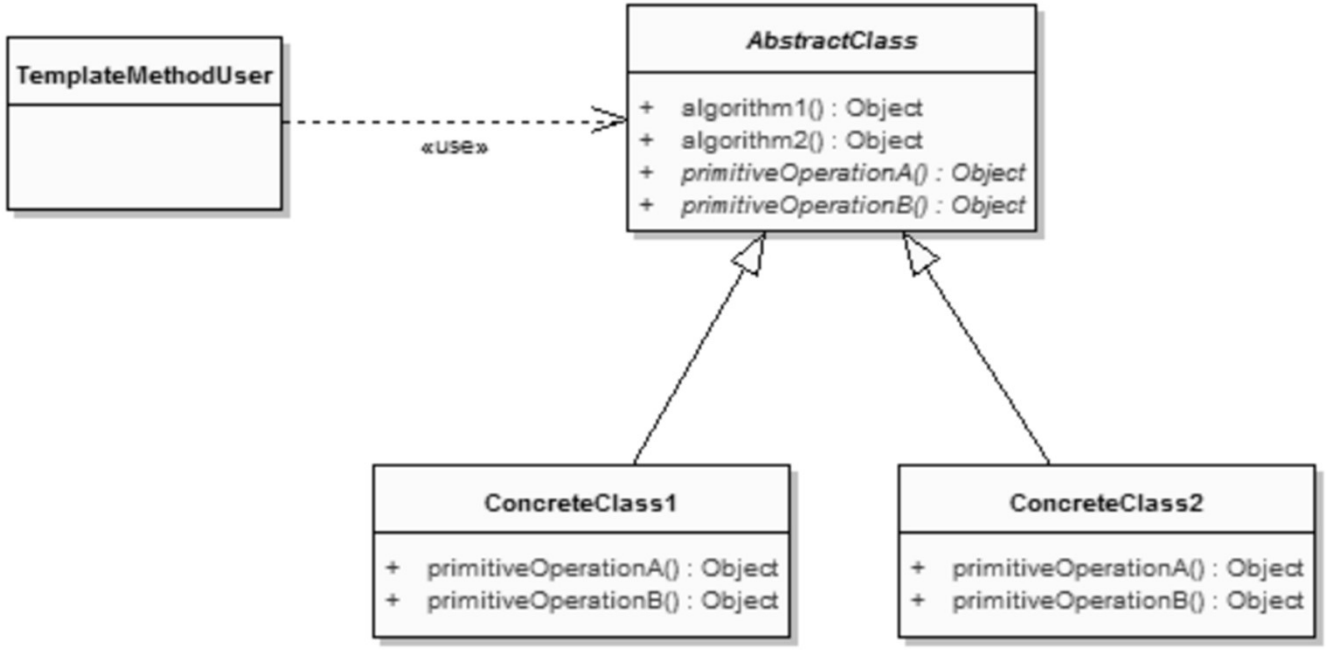
- to implement the invariant parts of an algorithm once and leave it to the subclasses to implement behavior that can vary and
- to factorize and localize the common behavior among subclasses to avoid code duplication.

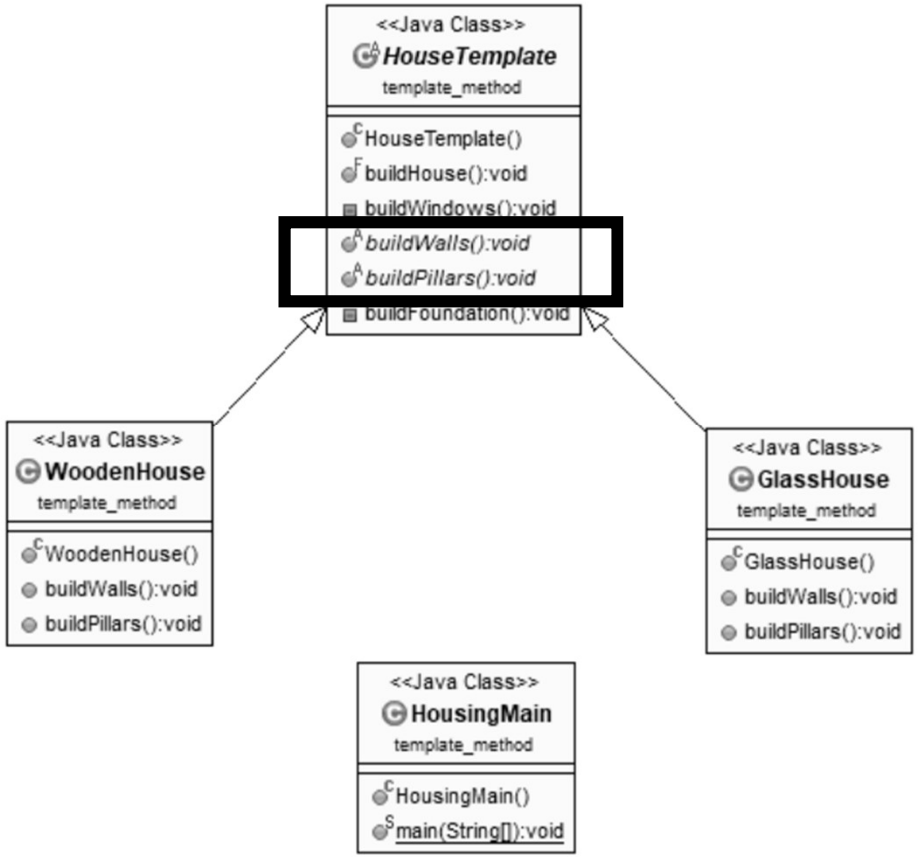


TEMPLATE METHOD 예제

HTTPS://NICEMAN.TISTORY.COM/142

템플릿 메소드 패턴 구조(Structure)




<<Java Class>>
HouseTemplate
template_method
HouseTemplate()
buildHouse():void
buildWindows():void
buildWalls():void
buildPillars():void
buildFoundation():void
<<Java Class>>
WoodenHouse
template_method
WoodenHouse()
buildWalls():void
buildPillars():void
<<Java Class>>
GlassHouse
template_method
GlassHouse()
buildWalls():void
buildPillars():void
<<Java Class>>
HousingMain
template_method
HousingMain()
main(String[]):void

```java
public abstract class HouseTemplate {

    //final 선언으로 Override 방지
    public final void buildHouse(){
        buildFoundation();
        buildPillars();
        buildWalls();
        buildWindows();
        System.out.println("House is built.");
    }

    //기본으로 구현
    private void buildWindows() {
        System.out.println("Building Glass Windows");
    }

    //서브클래스에서 직접 구현 할 메소드
    public abstract void buildWalls();
    public abstract void buildPillars();

    private void buildFoundation() {
        System.out.println("Building foundation with cement,iron rods and sand");
    }
}
```

```java
public class GlassHouse extends HouseTemplate {

    @Override
    public void buildWalls() {
        System.out.println("Building Glass Walls");
    }

    @Override
    public void buildPillars() {
        System.out.println("Building Pillars with glass coating");
    }

}
```

```java
public class WoodenHouse extends HouseTemplate {

    @Override
    public void buildWalls() {
        System.out.println("Building Wooden Walls");
    }

    @Override
    public void buildPillars() {
        System.out.println("Building Pillars with Wood coating");
    }

}
```

```java
public class HousingMain {

    public static void main(String[] args) {

        //템플릿 메소드 사용(Wooden House)
        HouseTemplate houseType = new WoodenHouse();

        houseType.buildHouse();

        //구분 선 삽입
        System.out.println();
        System.out.println("************");
        System.out.println();

        //템플릿 메소드 사용(Glass House)
        houseType = new GlassHouse();

        houseType.buildHouse();
    }
}
```


Problems
Javadoc
Declaration
Console
Progress
<terminated> HousingMain [Java Application] C:₩Django₩lib₩jdk1.8.0_121₩bin₩javaw.exe (2018. 5. 18. 오후 2:51:54)
Building foundation with cement,iron rods and sand
Building Pillars with Wood coating
Building Wooden Walls
Building Glass Windows
House is built.
****
Building foundation with cement,iron rods and sand
Building Pillars with glass coating
Building Glass Walls
Building Glass Windows
House is built.

장점

- 코드 중복 감소
- 자식 클래스의 역할(롤)을 감소시키면서 핵심로직 관리 용이
- 객체 추가 및 확장 쉽게 가능

단점

- 추상메소드가 너무 많아지면 클래스 관리가 복잡
- 추상클래스와 구현클래스간 복잡성 증대