

Design Patterns

Introduction, Template Hook and Factory Method Patterns

School of Computing
Clemson University
Clemson, SC 29634 USA

CPSC 2150: Monday, Mar. 27, 2023

Slide Sources:

- 1 Previous/Current CPSC 2150 Instructors
- 2 The lecture slides have also benefitted from instructors at Ohio State: Paolo Bucci, Wayne Heym, Paul Sivilotti, and Bruce Weide.

Last Class

- 1 Recap: Verification vs. Testing
 - ▶ Primitive Types and Abstraction
- 2 String Theory
 - ▶ Empty String
 - ▶ Denoting a Specific String
 - ▶ Concatenation
 - ▶ Length
 - ▶ `Reverse` Function
 - ▶ `Prt_Btwn` Function
 - ▶ `Occurs_Ct` Function

Homework Assignment

- 1 Programming assignment 3 is posted on your lab Canvas page.

What Are Design Patterns?

“a general reusable solution to a commonly occurring problem within a given context in software design”

- ▶ Design patterns represent the collective experience of decades of object-oriented software development experience.

Why Design Patterns?

- ▶ Design patterns are elegant solutions to common coding problems
 - ▶ Don't reinvent the wheel
- ▶ Design patterns are common knowledge among programmers
 - ▶ Easier for others to read and understand your code
 - ▶ Easier for other team members to build off of your code
- ▶ Design patterns come with some confidence
 - ▶ There are a lot of examples of implementations of a design pattern
 - ▶ Their common use means we are familiar with pros and cons

Design Patterns vs. Architectural Patterns

- ▶ **Design Patterns**

- ▶ Specify a solution for a particular problem within a larger program.

- ▶ **Architectural Patterns**

- ▶ Specify a solution for the entirety of a program (or at least a significant portion of it).

Design Pattern Categories¹

- ▶ **Creational**
 - ▶ Solutions involving the **creation**/instantiation of objects
- ▶ **Structural**
 - ▶ Solutions involving the **relationships** between objects
- ▶ **Behavioral**
 - ▶ Solutions involving the **interactions** between objects

¹It is not important to know the differences for this course. . .

Where to Find Design Patterns?

- ▶ The seminal work on design patterns is the 1994 book, **Design Patterns: Elements of Reusable Object-Oriented Software**
 - ▶ **Written by four authors:** Erich Gamma, John Vlissides, Ralph Johnson, and Richard Helm
 - ▶ commonly known as the *Gang of Four*
 - ▶ Contains all of the most popular design patterns still relevant today
- ▶ Design patterns have been documented and written about extensively online
 - ▶ As usual, Wikipedia is a useful starting point
- ▶ O'Reilly Learning, a better source
 - ▶ <https://learning.oreilly.com/library/view/design-patterns-elements/0201633612/>

Design Patterns and Languages

- ▶ Design patterns are meant to work with any **Object Oriented Programming (OOP)** language
 - ▶ C++, Java, C#, VB.NET, etc
- ▶ However, the syntax of each language is different
 - ▶ Different languages may have different constructs to make it easier to implement these design patterns
- ▶ Design patterns have to stay general, so they avoid using language-specific terms

Design Patterns and Languages

- ▶ All OOP languages have classes, inheritance, and abstract classes
 - ▶ So we can use these constructs in our design patterns
- ▶ Not all OOP languages have interfaces
 - ▶ So interfaces are not referred to in our design patterns
 - ▶ They usually have an abstract class concept
 - ▶ However, an interface and an abstract class are similar

Interfaces and Abstract Classes

Interfaces and abstract classes are similar:

- ▶ Neither can be the dynamic type, just the declared type
 - ▶ The dynamic type of an abstract class would be a class that **extends** the abstract class
 - ▶ The dynamic type of an interface would be a class that **implements** the interface
- ▶ Both can have abstract methods that are implemented by another class

Interfaces and Abstract Classes

Interfaces and abstract classes are similar:

- ▶ Both can have abstract methods that are implemented by their dynamic type
 - ▶ All methods in an interface are either **default** or **abstract**
 - ▶ Abstract methods must be implemented by the implementations of the interface.
 - ▶ Default methods can be overridden or inherited
 - ▶ An **abstract** class can declare a method as **abstract** and provide no body for it
 - ▶ Classes that **extend** the **abstract** class provide code for the abstract methods
 - ▶ Classes that **extend** the **abstract** class can inherit other methods or override them

Interfaces and Abstract Classes

Differences between interfaces and abstract classes:

- ▶ Abstract classes can have data members, while interfaces can only have a data member if it is `public`, `static` and `final` (constant)
 - ▶ Because of that, abstract classes can have non-abstract methods that refer to `private` data, while interfaces do not
 - ▶ Default methods in interfaces still cannot refer to data members unless they are `public`, `static` and `final` (constant)

Interfaces and Abstract Classes

- ▶ Consider an abstract class without any data members except for `public`, `static`, and `final` ones (constant)
 - ▶ It would be functionally the same as an interface
- ▶ You could have multiple classes that **extend** the abstract class with their own private data representations
- ▶ You could **program to the interface** by having the declared type always be an abstract class
- ▶ This is how to add “interfaces” to C++

Design Patterns and Languages

- ▶ Since not all OOP languages have interfaces, design patterns do not refer to them
- ▶ Design patterns can refer to abstract classes
- ▶ If those abstract classes don't require specific `private` data members, we can replace them with interfaces in Java

Template Method Pattern

- ▶ The **Template Method** design pattern is a common design pattern for working with multiple implementations.
- ▶ You divide the methods of your object into two categories, *template* and *hook* methods
 - ▶ Template methods don't refer to any `private` data
 - ▶ Hook methods do refer to `private` data

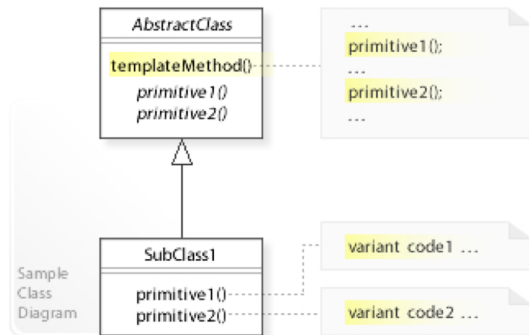
Template Method Pattern

Steps to Follow

- 1 Create an abstract class
- 2 Hook methods are declared as `abstract`
- 3 Template methods are not `abstract`
 - ▶ They access any `private` data by calling hook methods
- 4 A class that **extends** the abstract class provides code for the hook methods
 - ▶ Can just inherit the implementation of the template methods
 - ▶ Factors out the common code among many implementations

Template Method Pattern

UML Diagram



Source: By Vanderjoe - Own work, CC BY-SA 4.0,

<https://commons.wikimedia.org/w/index.php?curid=63155402>

Sound familiar?

- ▶ It should!
 - ▶ Replace Template with Secondary
 - ▶ Replace Hook with Primary
 - ▶ Change the abstract class to an interface
- ▶ O'Reilly Learning: <https://learning.oreilly.com/library/view/design-patterns-elements/0201633612/ch05.html#ch05lev1sec10>

Template Method Pattern

Steps to Follow (Java)

- 1 Create an **interface**
- 2 Primary methods are **abstract**
- 3 Secondary methods are **default** implementations
 - ▶ They access any **private** data by calling primary methods
- 4 A class that **implements** the interface provides code for the primary methods
 - ▶ Can just inherit the implementation of the secondary methods
 - ▶ Factors out the common code among many implementations

Factory Method Pattern

- ▶ Creates an indirection
 - ▶ Instead of calling the constructor, we call the factory
 - ▶ The factory calls our constructor
 - ▶ We factor out the creation of our objects to one place in the code
- ▶ Leverages **programming to the interface** to do so
 - ▶ Our factory returns objects of the interface/abstract class type
 - ▶ So it can call any constructor that implements that interface/extends the abstract class
 - ▶ If we want to change the implementation, we only need to change the constructor call

Why Factory Method Pattern?

Suppose you have some code like this:

```
void oneMethod() {  
    IntegerStack s = new Stack1(100);  
    IntegerStack t = new Stack1(20);  
    ...  
}  
  
void anotherMethod() {  
    IntegerStack u = new Stack1(50);  
    ...  
}
```

Why Factory Method Pattern?

Suppose you want `Stack2` instead of `Stack1` everywhere. . .

Why Factory Method Pattern?

Suppose you want `Stack2` instead of `Stack1` everywhere...

```
void oneMethod() {  
    IntegerStack s = new Stack2(100);  
    IntegerStack t = new Stack2(20);  
    ...  
}  
  
void anotherMethod() {  
    IntegerStack u = new Stack2(50);  
    ...  
}
```

- Need to make many changes

Factory Method Pattern

Steps to Follow

- 1 Create a new interface (`IFactory`) that is a “Factory” for interface `I`
- 2 All the factory interface does is make objects
 - ▶ The objects returned all implement interface `I`
- 3 The factory has a “make” method
 - ▶ Returns an object of type `I`
- 4 Make implementations of the factory interface
 - ▶ Each implementation makes a different implementation of `I`

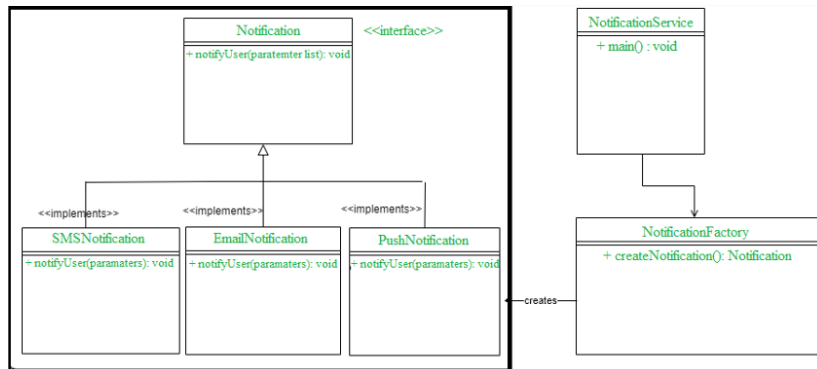
Factory Method Pattern

Example

- ▶ **Problem Statement:** Consider we want to implement a notification service through email, SMS, and push notification. Let's try to implement this with the help of the factory method design pattern. First, we will design a UML class diagram for this.
- ▶ <https://www.geeksforgeeks.org/factory-method-design-pattern-in-java/>

Factory Method Pattern

Example - UML Diagram



Factory Method Pattern

- ▶ The solution strategy creates an indirection
- ▶ In each implementation of the factory, the code for “make” operation calls an appropriate constructor
- ▶ So, from the client side, the same exact “make” operation is called
- ▶ But underneath, different constructors are invoked, depending on the factory implementation

Factory Method Pattern

Stacks Example

- 1 Create a new interface (`IStackFactory`) that is a “Factory” for interface `IntegerStack`
 - ▶ One operation: `public IntegerStack makeStack();`
- 2 Make implementations of `IStackFactory` for each implementation of `IntegerStack`
 - ▶ `makeStack` calls the different constructors
 - ▶ **Implementation 1: `Stack1Factory`**
 - ▶ `makeStack` calls constructor for `Stack1`
 - ▶ **Implementation 2: `Stack2Factory`**
 - ▶ `makeStack` calls constructor for `Stack2`

Factory Method Pattern

Stacks Example

Change localized to one place:

```
private IStackFactory sf = new Stack1Factory();  
  
...  
  
void oneMethod() {  
    IntegerStack s = sf.makeStack(100);  
    IntegerStack t = sf.makeStack(20);  
    ...  
}  
  
void anotherMethod() {  
    IntegerStack u = sf.makeStack(50);  
    ...  
}
```

Factory Method Pattern

Stacks Example

Change localized to one place:

```
private IStackFactory sf = new Stack2Factory();
```

```
...
```

```
void oneMethod() {  
    IntegerStack s = sf.makeStack(100);  
    IntegerStack t = sf.makeStack(20);  
    ...  
}
```

```
void anotherMethod() {  
    IntegerStack u = sf.makeStack(50);  
    ...  
}
```

Another Factory Method Pattern Use

- 1 Recall JUnit tests
 - ▶ Constructors like: `new Stack2(100);` are hardcoded
- 2 If you want to test a different stack implementation, you will have to edit many places
- 3 Factory method pattern can be used so your JUnit tests will work for all implementations of an `interface`

Factory Method Pattern in JUnit

- ▶ In our JUnit code, we don't even have to make the factory interface
 - ▶ In fact, that doesn't work as well in JUnit
 - ▶ In our previous example the factory was a `private` member of the class and initiated in the constructor
 - ▶ Our JUnit code isn't production code, so we can simplify the factory method pattern
 - ▶ Create a `private` method for our factory that takes in the constructor arguments and returns the `interface` type
 - ▶ Now we can just change the constructor called in the `interface` method to test a new implementation

Factory Method Pattern

Another Example

- ▶ See this wiki page: https://en.wikipedia.org/wiki/Factory_method_pattern
- ▶ O'Reilly Learning: <https://learning.oreilly.com/library/view/design-patterns-elements/0201633612/ch03.html#ch03lev1sec3>

Homework Assignment

- 1 Programming assignment 3 is posted on your lab Canvas page.

Summary

1 Design Patterns

- ▶ Introduction
- ▶ Interfaces and Abstract Classes
- ▶ Design Patterns and OOP Languages

2 Template Method Pattern

3 Factory Method Pattern