Megan Landau
CSCI 360 - Software Architecture
Dr. Bowring

Factory Method Design Pattern: Application, Advantages and Example

The Factory Method design pattern is a commonly used behavioral design pattern found often in software frameworks. Design patterns are programming techniques or methods which provide a solution for common problems found in object-oriented software design. There are 23 classic software design patterns; each were introduced and discussed by a group of authors known as the Gang of Four (GoF) in their famous 1994 publication *Design Patterns: Elements of Reusable Object-Oriented Software* [2]. The authors do not claim ownership of the designs and instead declare that the book is intended as a catalog of collected design patterns “that have been applied more than once in different systems” and are delivered to the reader in a consistent, readable format [3].

The GoF detail the intention, structure, implementation, and application for each pattern, dividing them into three types based on use cases: Creational, Structural, and Behavioral. According to the GoF, creational patterns “abstract the instantiation process. They help make a system independent of how its objects are created, composed, and represented” [3]. As a behavioral design pattern, the Factory Method allows a system to create objects without specifying the exact class of object that will be instantiated. They allow the subclasses to instantiate objects. On a high-level, the Factory Method abstracts the creator of an object from the client who is calling for the object. The caller does not need to know the exact type of object they are asking for, they just need to know the abstract type they are using and the subclass will instantiate the object via the Factory Method.

Backing up a little bit: there are five basic principles of good design practice in software which are intended to make designs more understandable. To remember the principles, the acronym “SOLID” is commonly used. The Factory Method design pattern caters to the “S” and “D” principles, mostly. “S” stands for the Single Responsibility Principle (SRP) - the idea that classes should have only a single responsibility and if a class needs to be changed, only that responsibility should need to be changed within the code. The “D” stands for the Dependency Inversion Principle (DIP) - a code object should not depend upon a concrete instantiation of an object, but rather on an abstraction of it [8]. Knowing this, it is easier to explain the need for the Factory Method.

The Factory Method comes into play when there is a need to create many objects that could potentially be subclasses of the same superclass. In this example, an IceCreamShop is the abstract parent class of the CharlestonIceCreamShop and the SavannahIceCreamShop. Each IceCreamShop subclass can instantiate objects of the IceCream class according to what flavor of ice cream is ordered. Both the Charleston and Savannah locations have the same flavors, but the sauces and toppings are different - there is the SavannahStrawberry subclass and the CharlestonStrawberry subclass. Each can be created only by its corresponding shop.

The following is a scenario of a system without the Factory Method design pattern. If the system had a client that called upon an IceCreamShop class to createIceCream(“strawberry”), then the IceCreamShop would have to have a bunch of if-else statements to check what location it was set to and what flavor of ice cream was served at that location and finally be able to create a *new* IceCream object and return it. In this situation, the decision of which ice cream to instantiate is made at runtime depending on a set of conditions such as location and flavor. Code like this is often repeated in different parts of the system and makes maintaining and updating the system more difficult and prone to error. This could be a violation of the “Don’t Repeat Yourself” principle since there is duplication of code. What if something changed in the ice cream flavors later and now the programmer has to go looking for those if-then statements all over the system? The Factory Method design pattern is a solution to these issues and will also promote SRP and DIP. This can be better explained after discussing *how* the factory method is implemented.

According to the 2004 O’Reilly Media publication, *Head First Design Patterns*, “the Factory Method Pattern encapsulates object creation by letting subclasses decide what objects to create” [7]. The Factory Method itself has this basic signature: abstract Product factoryMethod(String type), where the Product being returned would be the IceCream class in the example before. The IceCream class would need to be abstract and the instantiated object would actually be a subclass which extends the abstract IceCream class, such as CharlestonStrawberry. The client, IceCreamDriver, has no idea how the ice cream is instantiated, it just knows that when it calls the factory method createIceCream(“strawberry”) on the IceCreamShop that is a CharlestonIceCreamShop, it is returned an instance of IceCream which happens to be CharlestonStrawberry.

The GoF define the structure of the Factory Method in *Design Patterns*. The four components are: Product abstract class, ConcreteProduct class which extends the

Product interface, the ConcreteCreator which is the factory class and contains the factory method, and the abstract Creator class which contains the abstract factory method and full implementations of other methods that it might need [3]. For an IceCreamShop, this could mean serveIceCream() is implemented within the class so that all IceCream is served in the same manner - this ensures that there is consistency between shops. In the ice cream example, the Product is the IceCream abstract class, the ConcreteProduct would be a subclass of IceCream such as CharlestonVanilla or SavannahStrawberry, and the Creator class is the abstract IceCreamShop. Finally, the concrete subclass of IceCreamShop such as CharlestonIceCreamShop or SavannahIceCreamShop is the “factory” where the Factory Method createIceCream(String flavor) is implemented. At runtime, only CharlestonIceCreamShop or SavannahIceCreamShop knows what ice cream will be created when a certain flavor is asked for; when createIceCream(“strawberry”) is called from the IceCreamDriver, it does not know that a StrawberryIceCream object was returned, all it knows and cares about is that the IceCreamShop subclass (either CharlestonIceCreamShop or SavannahIceCreamShop in this example) returned a class that implements the IceCream interface and therefore, it can use it the way it would use any class which implements the IceCream interface.

Previously mentioned, the benefits of the factory method include not needing to change multiple classes if there is ever a change to the flavors of ice cream needed in the system. One would merely need to alter the specific subclass’s (ex: CharlestonIceCreamShop) factory method and the system would continue to run. No other dependencies need to be altered. This promotes maintainability of the system and ensures that duplication of code is avoided. Also, the Dependency Inversion Principle has been followed; clients callers depend only on interfaces to instantiate objects. Therefore the system is programmed to the interface and not the implementations and the use of the Factory Method has allowed the system to be more flexible and extensible [7]. It exploits the benefits of the OOP principle of Inheritance and promotes loose coupling by reducing the dependency of the system on concrete classes [5].

A similar design pattern to Factory Method is the Abstract Factory design pattern and the two are often confused. Without going into detail, the Abstract Factory pattern is a pattern which utilizes the Factory Method pattern, but uses it multiple times to create groupings of similar classes. While the Factory Method is a *class* creational pattern which uses inheritance to have variance in the subclasses that are instantiated, the Abstract Factory pattern is instead an *object* creational pattern that delegates instantiation to a different object [3]. Factory Methods are also often called within implementations of the Template Method design pattern.

Use of the Factory Method is often seen in software development kits and other frameworks where there is the need to choose a subclass from a long list of subclasses given a certain input or condition. The Factory Method makes maintenance on a system easier by only necessitating changes in a single class when requirements change for the system (for example, if a CharlestonBlueberry flavor needed to be added, it only needs to be added to the CharlestonIceCreamShop class within the createIceCream() Factory Method). Other clarifying examples could include the use of the Factory Method in a game where many different enemy types need to be spawned. The factory method could have a long switch statement of different enemy subclasses which could be instantiated via a random number generator, allowing for variance in enemy characters during runtime. There are many benefits to this design pattern and its application is sure to be found in a variety of systems.

Bibliography

(1) Banas, Derek. "Factory Design Pattern Tutorial." Factory Design Pattern Tutorial, New Think Tank, 1 Sept. 2012, www.newthinktank.com/2012/09/factory-design-pattern-tutorial/.

(2) "Design Patterns." Wikipedia, Wikimedia Foundation, 10 Feb. 2018, en.wikipedia.org/wiki/Design_Patterns.

(3) Gamma, Erich, et al. Design Patterns: Elements of Reusable Object-Oriented Software. Pearson Education, 2015.

(4) Javin, Paul. "What Is Factory Method Design Pattern in Java with Example." Javarevisited, javarevisited.blogspot.com/2011/12/factory-design-pattern-java-example.html.

(5) Kanjilal, Joydip. "The Factory Method and Abstract Factory Design Patterns: Managing Object Creation Efficiently." InfoWorld, InfoWorld, 5 June 2015, www.infoworld.com/article/2931441/c-sharp/the-factory-method-and-abstract-factory-design-patterns-managing-object-creation-efficiently.html.

(6) Mitra, Shamik. "Factory Method vs. Simple Factory - DZone Java." DZone, 8 Jan. 2018, dzone.com/articles/factory-method-vs-simple-factory-1.

(7) Sierra, Kathy, et al. "Head First Design Patterns." O'Reilly | Safari, O'Reilly Media, Inc., 2018, www.safaribooksonline.com/library/view/head-first-design/0596007124/ch04.html.

(8) "SOLID (Object-Oriented Design)." Wikipedia, Wikimedia Foundation, 10 Feb. 2018, en.wikipedia.org/wiki/SOLID_(object-oriented_design).