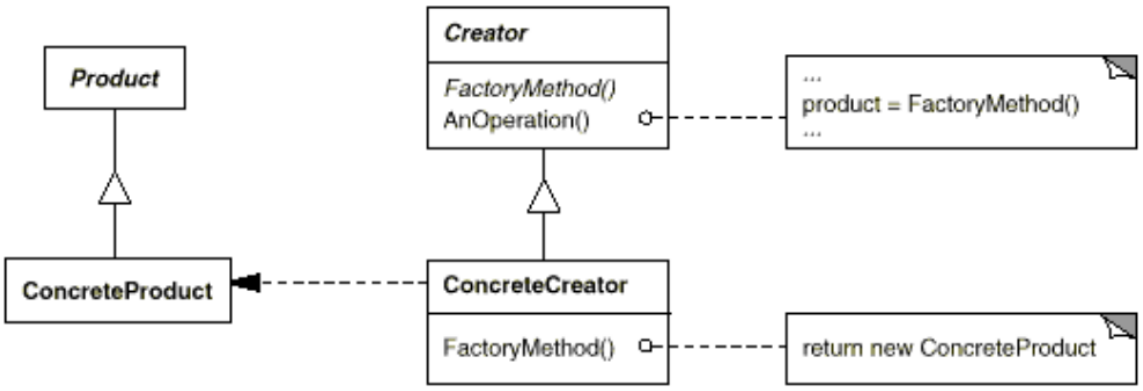
Creational pattern: trừu tượng hóa việc khởi tạo các objects

1. factory method: Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

Structure



Use the Factory Method pattern when

- a class can't anticipate the class of objects it must create.
- a class wants its subclasses to specify the objects it creates.
- classes delegate responsibility to one of several helper subclasses, and

you want to localize the knowledge of which helper subclass is the delegate.

Abstract Factory (99) is often implemented with factory methods. The Motivation example in the Abstract Factory pattern illustrates Factory Method as well.

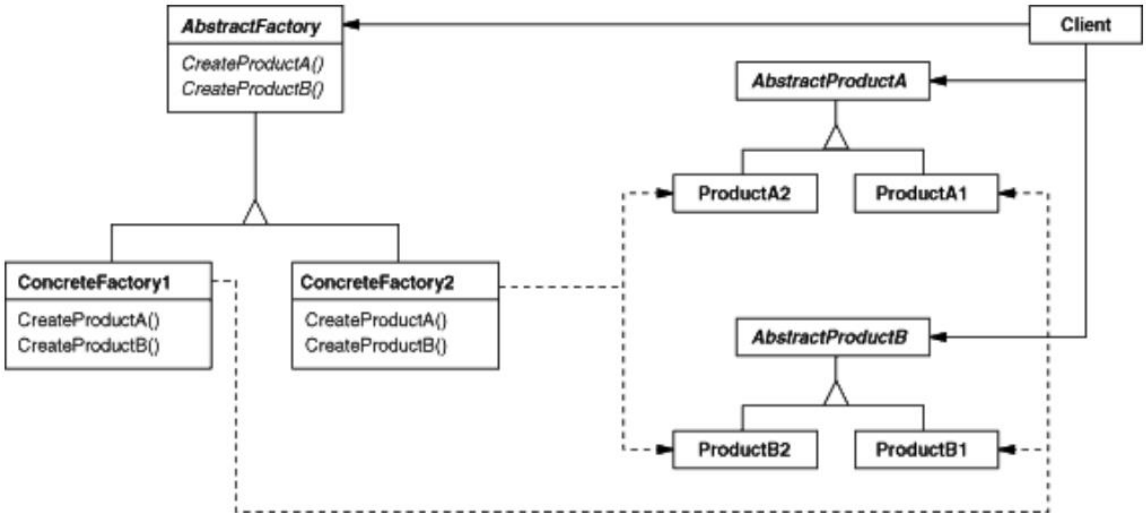
Factory methods are usually called within Template Methods (360). In the document example above, NewDocument is a template method.

Prototypes (133) don't require subclassing Creator. However, they often require an Initialize operation on the Product class. Creator uses Initialize to initialize the object. Factory Method doesn't require such an operation

2. abstract factory:

Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

Structure



Use the Abstract Factory pattern when

· a system should be independent of how its products are created, composed, and represented.

· a system should be configured with one of multiple families of products.

· a family of related product objects is designed to be used together, and you need to enforce this constraint.

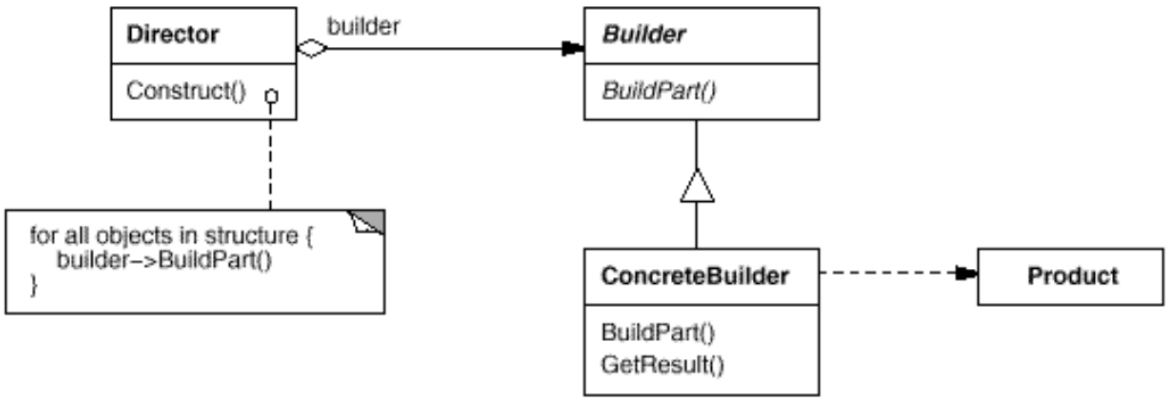
· you want to provide a class library of products, and you want to reveal just their interfaces, not their implementations.

AbstractFactory classes are often implemented with factory methods (Factory Method (121)), but they can also be implemented using Prototype (133).

A concrete factory is often a singleton (Singleton (144)).

3. builder pattern: Separate the construction of a complex object from its representation so that the same construction process can create different representations.

Structure



Use the Builder pattern when

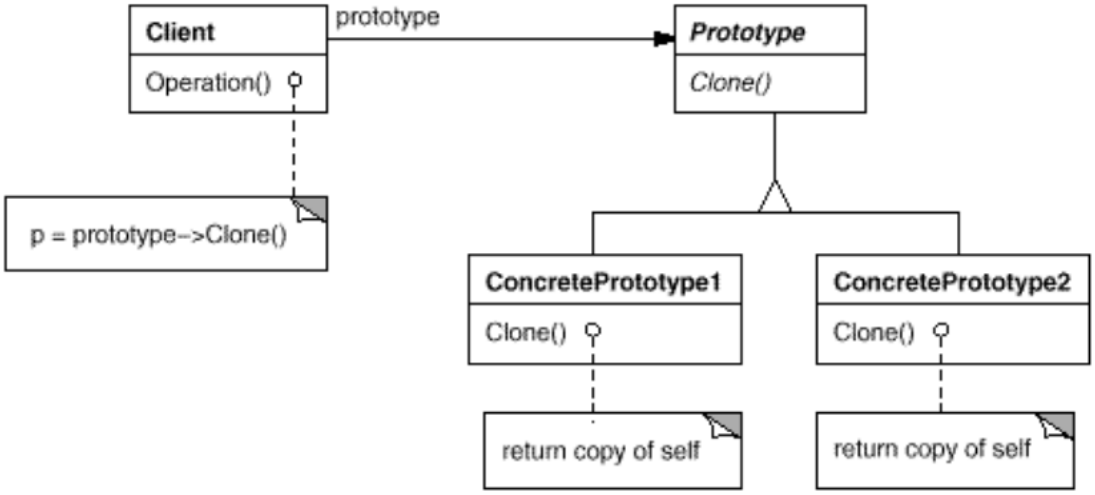
· the algorithm for creating a complex object should be independent of the parts that make up the object and how they're assembled.

· the construction process must allow different representations for the object that's constructed.

Abstract Factory (99) is similar to Builder in that it too may construct complex objects. The primary difference is that the Builder pattern focuses on constructing a complex object step by step. Abstract Factory's emphasis is on families of product objects (either simple or complex). Builder returns the product as a final step, but as far as the Abstract Factory pattern is concerned, the product gets returned immediately.

4. prototype: Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

Structure



Use the Prototype pattern when a system should be independent of how its products are created, composed, and represented; **and**

· when the classes to instantiate are specified at run-time, for example, by dynamic loading; **or**

· to avoid building a class hierarchy of factories that parallels the class hierarchy of products; **or**

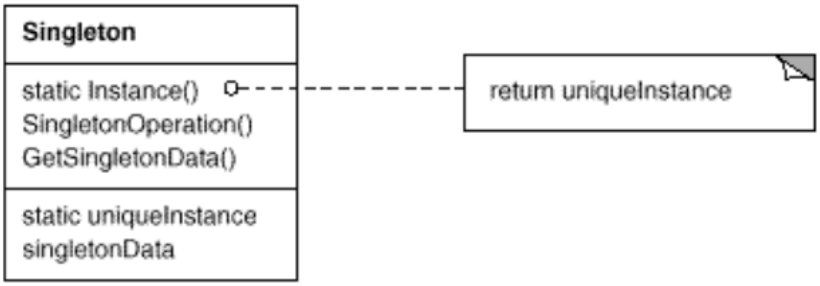
· when instances of a class can have one of only a few different combinations of state. It may be more convenient to install a corresponding number of prototypes and clone them rather than instantiating the class manually, each time with the appropriate state.

Prototype and Abstract Factory (99) are competing patterns in some ways, as we discuss at the end of this chapter. They can also be used together, however. An Abstract Factory might store a set of prototypes from which to clone and return product objects.

Designs that make heavy use of the Composite (183) and Decorator (196) patterns often can benefit from Prototype as well.

5. Singleton: Ensure a class only has one instance and provide a global point of access to it.

Structure



Use the Singleton pattern when

· there must be exactly one instance of a class, and it must be accessible to clients from a well-known access point.

· when the sole instance should be extensible by subclassing, and clients should be able to use an extended instance without modifying their code.

Many patterns can be implemented using the Singleton pattern. See Abstract Factory (99), Builder (110), and Prototype (133).

Reference: Design Patterns: Elements of Reuseable object-oriented software, Gangs of four. https://github.com/media-lib/prog_lib/blob/master/general/Gang%20of%20Four%20-%20Design%20Patterns%20-%20Elements%20of%20Reusable%20Object-Oriented%20Software.pdf