Design Pattern Report

Factory method (creational pattern)

Motivation: Also known as Virtual Constructor, it defines an interface for creating an object, but leaves the choice of its type to the subclasses, creation being deferred at run-time.

* **Intent** :Defines an interface for creating objects, but let subclasses to decide which class to instantiate

**Description**

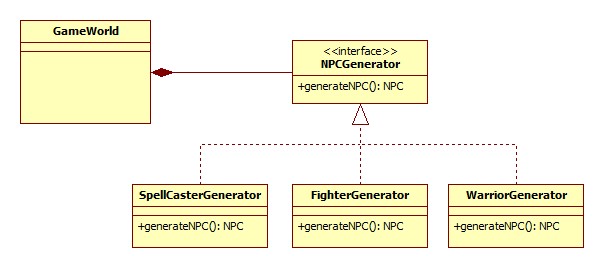
Factory method design pattern defines an interface in order to create objects. However, the type of class to be instantiate is decided by subclasses of the interface.

**Example**

A game needs to implement a class to generate NPCs. However, there are different types of NPCs that can be created, for example: spell caster, fighter and warrior. In order to meet this requirement, the game uses factory method. An interface NPCGenerator is with 3 different implementations: SpellCasterGenerator, FighterGenerator, WarriorGenerator is created.

Each concrete subclasses of the interface will decide which type of NPCs to be instantiated by NPCGenerator.

**Diagram**



**Advantages**

* **Factory** **method**s eliminate the need to bind application-specific classes into your code.
* The code only deals with the product interfaces; therefore, it can work with any user-defined concrete product classes.
* **Factory** **method**s provide hooks for sub-classes to create different concrete products. In the example below, the**Factory** **method** MakeUISpecificCtrls provides the hook for creating the UI component specific controls. In the default CUIComponent, a simple edit control is created; however, we can change this behavior in the derived class to create an edit control which accepts only floating point values.
* **Factory** **method**s connect parallel class hierarchies in such a way that it localizes the knowledge of which classes belong together. In the example below, the CUIComponent and CEditCtrl class hierarchies can be connected with each other. Also notice how the **Factory** **method** defines the connection between the two class hierarchies.

The pattern decouples the use of concrete objects from their implementations.

For example: GameWorld class can create any types of NPC (Spell Caster, Fighter or Warrior) by the help of NPCGenerator without knowing how each NPC is created. The instantiation of each NPC is implemented by these concrete classes: SpellCasterGenerator, FighterGenerator and WarriorGenerator.

If another type of the product is going to be added, then users are only required to add one more implementation of the interface. It is not necessary to change the other classes.

For example: In order to add Sniper in to the game, another implementation of NPCGenerator, such as SniperGenerator class, is required to be included.

**Disadvantages**

* A potential disadvantage of **Factory** **method**s is that clients might have to sub-class the creator class just to create a particular concrete product object.
* Subclassing is fine when the client has to subclass the creator class anyway, but otherwise, the client now must deal with another point of evolution.
* In **Factory** **Method** **pattern**, the **factory** used for creating the objects is bound with the client code, i.e., it is difficult to use a different **factory** for creating objects.

It is cruel to add more implemented class in order to create more different products, even if these products are the same in type and several properties. This means that factory method is not scalable and extensible.

Related pattern

Decorator pattern

**Alternate pattern/Why it is a better choice**

Factory

If adding more products is unlikely to happen, then hardcoding the instantiation of objects will make the runtime faster.

Abstract factory

Abstract factory is useful when several types of products have their own set of similar objects. In this situation, abstract factory enable users to decouple the instantiation of each product and its set of objects from the use of these classes.