Object Oriented Software Design

Tutorial 6: Creational Patterns

1a) What is the difference between Factory Method and AbstractFactory pattern.

The best way to understand these patterns is to start from the perspective of what BENEFIT we want to get from it (e.g. encapsulation or reuse).

Factory Method : we want to REUSE some code that is 90% the same the only difference being which concrete class to instantiate. This usually happens when we’re writing a reusable framework that manages generic components, some or all of which are specialised outside the framework (i.e. the application using the framework) : the framework knows WHEN to instantiate a component but not WHICH class to instantiate.

So we provide the common code utilising an abstract method who’s contract it is to instantiate the appropriate class. The subclasses that inherit this common code can implement the “factory” method instantiating different classes.

Abstract Factory : we want to ENCAPSULATE the implementation of some components that have a common behaviour but multiple implementations at runtime. We want the source code to be independent of what implementation is being used at runtime.

The first part of the solution is obvious : each implementation is a concrete subclass of an abstract class (or a class that implements an interface). The next problem is how do clients construct these components? The clients can use the components via their abstract interface but they can’t construct them because the concrete class is hidden from them (intentionally). The consequence of this is we need to abstract the construction process as well. To do this we create an abstract class (called an abstract factory) that just contains abstract methods who’s contract it is to instantiate one component. Then each subclass of this abstract factory implements all the construction methods using some concrete class for each component type.

b) Give two examples where Factory method is suitable

1) REUSE common code that opens new and old documents as part of a document-oriented application. It knows when to instantiate a document but not what type. {from Go4)

2) An point of sale (POS) framework that manages orders and invoices but the application defines it own product class. We know when to instantiate a product but not which type.

Note: Scala negates the need for this pattern with the use of generic type parameters that allow concrete instantiation at runtime. Java’s generics are broken but it is still possible with some convoluted meta-programming

c) Give two examples where AbstractFactory pattern is suitable

1) ENCAPSULATE the UI widgets for many platforms (from Go4)

2) ENCAPSULATE serial vs parallel vs distributed implementations of collections like lists and maps. We want the source code to be unaware of which implantation is being used at runtime because this should be configurable in deployment depending on whether we deploy to a single core desktop, muti-core machine or the cloud with massive distribution of compute nodes.

Note: Scala negates the need for this pattern with the use of generic type parameters that allow concrete instantiation at runtime. Java’s generics are broken but it is still possible with some convoluted meta-programming.

2a) What is the main purpose of a singleton pattern

ensure only one instance of a class and provide a way for clients to reference it.

b) Why is the constructor made private in a Singleton pattern?

To prevent clients making multiple instances. If it was public, any client could construct one.

c) Give one advantage of a Singleton over global variable?

The instance is only constructed if its needed which is more performant especially if the singleton object construction is expensive in cpu and memory resources.

The implementation if the single object is kept hidden from its client's. The global variable is open to corruption from clients. More specifically, the singleton property can be enforced with the pattern, but a global variable can be overwritten with another object bu one of the clients which breaks the singleton property.

d) What will be output of the program below?

class X {

private static X uniqueInstance;

static int count = 0;

private X() { count++; }

public static X getInstance()

{ if (uniqueInstance == null)

uniqueInstance = new X();

return uniqueInstance;

}

}

public class TestX {

public static void main(String args[])

{ X s1 = X.getInstance();

X s2 = X.getInstance();

System.out.println("Number of Xs = " + X.count);

}

}

1. Number of Xs = 0  
   Number of Xs = 1 (b) correct : count = 1
2. Number of Xs = 2
3. None of the above

3. Think about how you would write an XML parser that works for both constructing a document object model (DOM) and SAX events so it can be piped into another XML processor. What design pattern will you use?

**BUILDER**

(a) Draw the UML for the entities you’ll need and their relationships.

Entities are :

Document, Element, Attribute, Comment, ProcessingInstruction etc.

Relationships are :

Document has many Element, Comment, ProcessingInstruction

Element has many Element (reflexive)

Element has many Attribute, Comment, ProcessingInstruction

(b) Sketch out a sequence diagram for the parser showing both generic and specific objects as it reads this XML :

<designer name=”Martin Odersky”>

<language name=”Pizza” />

<language name=”Scala” />

</designer>

4. An accounting application that has a feature to compute company tax is to be developed. It will be used in all of the states in the US. The head office of each state runs its own instance of the application, but they all must share the common source code. The types of taxes applicable vary across the US which includes GST, asset tax, carbon tax, capital gains tax, etc. The rates of each tax varies across states and some states don’t implement some types of tax at all.  
  
Suggest an appropriate pattern giving two reasons. Explain briefly how this pattern helps compute the total company tax effectively for different states.

Abstract Factory

Reasons :

1) We need to encapsulate the different ways of calculating tax for each state.

2) We want to reuse the generic total tax calculation code for every state.

Explanation :

Each tax type will be represented as an object. (1) requires an abstract factory because there are many different tax type implementations. (2) requires a few factory methods and we already have them in the abstract factory so the total tax algorithm can live in a class associated to the abstract factory.