David Schirduan CSCI360 Computer Architecture and Design

Bowring 2/15/11

Factory Method

Computers are simple machines, in theory at least. Programmers make all of the important decisions for the computer to execute: which numbers to add, multiply, subtract; where to store those numbers; in what order operations should be executed, etc. Everything is laid out very plainly, and in theory all the computer has to do is execute the directions exactly as the programmer has laid out. Every once in a while though, programmers expect more of computers. Sometimes decisions must be made by the computer, very quickly and accurately, which will then carry out those decisions. Fortunately for the computer, programmers make it very clear the conditions and requirements for this decision to be executed by the computer. One of these decisions involves something called the “Factory Method”.

Sometimes a program is required to manipulate or create multiple types of objects. Programmers call these “Abstract Classes”, which are created to interface with and manipulate a broad range of objects in different states. These abstract classes are generalists, very rarely performing specific functions to one type of object or another. When these abstract classes are expected to create objects, there arises a problem inherent to abstract classes. Since abstract classes are generic, and essentially type-less, they cannot create multiple types of objects. An abstract class may be able to create one type of object, but not multiple types. When the program is running, the class knows when to create a new object, but it cannot know which type of object must be created. This is the problem that is solved by the Factory Method.

Although there is no concrete history of the development of this method, it is easy to

see how this method may have become a common, accepted solution to the problem that it solves. When software was still in it's infancy, problems could be directly solved, and there were simple solutions to simple problems. As programs became more complex, and faster, they required more maintenance and thus more complexity. Since computers were moving at the speed of light, they had to start making their own complex decisions at those same speeds, instead of relying on user input. Instead of making huge and complex if-then statements, it is easier to split programs into different classes that may call upon or override one another. This way each individual class can be relatively simple, and also be interchangeable. This idea is called coupling. The Factory Method applies the principle of coupling to object creation.

Basically, the Factory Method is an expanded, outsourced if-then statement. When a program needs to decide which type of class it will create, it allows the Factory Method to make that determination. Although in reality, the Factory Method just allows the sub-clases to override the object creation, and take control of the creation process. This is accomplished through the use of several specialized sub-classes. These sub-classes contain the details of the possible objects that may be created. The Factory Method decides which sub-classes is required for this specific situation, and calls on that class to create the needed object that is covered by that subclass. After the correct object is created, the program can continue to the next command.

The main benefit for this method is that it allows for expansion. By adding additional sub-classes, and modifying the parameters for the decision of class creation, a programmer can effectively expand the types of objects that this Factory Method can create. This can be extremely useful when first designing a program that may require expansion over time or alteration.

The biggest problems with this method are the obvious ones. It is not very useful when

there are only one or two types of objects that are necessary. Creating separate subclasses for each one is a waste of resources. Also, some programming languages handle object creation differently. For example, in an assembly language, such as MIPS, it would be much better to have an extended if-then statement than to require the jumping to multiple loops, and usage of several registers. The Factory Method also requires fairly consistent and expanded naming conventions. Often times there are multiple abstract classes, concrete objects, and methods all interacting with one another; resulting in even more objects being created and added to the fray. It is essential to name classes appropriately according the language and the specific characteristics and functions of those classes or methods.

The factory method is a very effective way to solve the problem of abstract class creation. This allows you to have one creation method or function for multiple types of objects, as opposed to simply replicating code for different objects. When used correctly, it allows for easy expansion of classes, and keeps coupling low. Depending on the complexity of the sub-classes, this can significantly reduce your overhead, with entire sections of code being skipped over, or unused except in the rare case that the particular object is needed.

Works Cited

Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John (1994). *Design Patterns: E Elements of Reusable Object-Oriented Software*. Addison-Wesley. ISBN 0-201-63361-2.

"Factory Method Design Pattern." *Design Patterns and Refactoring*. Web. 16 Feb. 2011. <http://sourcemaking.com/design_patterns/factory_method>.

"The Factory Method Design Pattern by Gopalan Suresh Raj." *A Component Engineering Cornucopia by Gopalan Suresh Raj*. Web. 17 Feb. 2011. <http://gsraj.tripod.com/design/creational/factory/factory.html>.

```
public interface Barracks
{
	public static needTroops createTroop(input type);
	{
	neededtroop Troop = whichTroop(type)

		switch (Troop)
		{

			case Mage:
				return new MakeMage(type);
			case Warrior:
				return new MakeWarrior(type);
			case Archer:
				return new MakeArcher(type);

		}
	}
}


public class MakeWarrior implements Barracks
{
	public needTroops createTroop()
	{
		//code here...
		return troop
	}
}

public class MakeArcher implements Barracks
{
	public needTroops createTroop()
	{
		//code here...
		return troop
	}
}

public class MakeMage implements Barracks
{
	public needTroops createTroop()
	{
		//code here...
		return troop
	}
}
```