VOL. 4, NO. 12, December 2014 ISSN 2225-7217
ARPN Journal of Science and Technology
©2011-2014. All rights reserved.

http://www.ejournalofscience.org

Simplifying the Abstract Factory and Factory Design Patterns

[1] **Egbenimi Beredugo Eskca,** [2] **Sandeep Bondugula,** [3] **El Taeib Tarik**
[1, 2, 3] Department of Computer Science, University of Bridgeport, Bridgeport Connecticut

[1] eberedug@my.bridgeport.edu, [2] sbondugu@my.bridgeport.edu, [3] teltaeib@my.bridgeport.edu

ABSTRACT

The goal of this research paper is to make a clear and simple distinction between the factory and the abstract factory patterns. As a result of the similarities between these two patterns, understanding the unique and appropriate design requirement that calls for the application of each of these patterns has been a challenge to inexperience object oriented programming (OOP) programmers, as well as experienced OOP programmers that are new to the idea of design patterns in functional software development.

Our objective in this research paper is to clarify the meaning as well as the usage of these two very important design patterns, in a manner that would make sense to new comers in the design pattern arena. We commence by giving a descriptive definition of both patterns followed by a more specific definition of the patterns in line with OOP concepts. Following the definition, is an illustration of each of the design pattern in simple diagrammatic form.

Furthermore, we have made an effort to help the new comers to design patterns to quickly and easily identify scenarios and problems definitions where the adoption of each of the design pattern is necessary and appropriate. We have also attempted to ensure that, new comers, on identifying the need to apply either the factory or the abstract factory pattern in the solution to a problem, should also match these requirements with the core OOP concepts of encapsulation, inheritance and polymorphism.

Following the preceding sections, we have included a dialogue between members of a small software development team in the identification and analysis stage of a software solution that requires the implementation of the factory as well as the abstract factory patterns.

We finalize the paper with a C++ code section to illustrate the implementation of the abstract factory pattern.

We conclude with a word of advice for new programmers and programmers that are unfamiliar with the concept of design patterns, as well as suggestions for future research that would attempt to simplify and demystify the use of design patterns for new comers to this arena.

Keywords: *Factory, Abstract factory, Pattern, Base, Interface, Concrete, Class, Implementation*

1. FACTORY PATTERN

Factory pattern is the design pattern of choice, when there exist the need to centralize the collection of resources that are required to construct, produce, or create an object. The factory method's fundamental function is to isolate, decouple or specialize the construction of an object from the eventual use of the object. The purpose of the factory pattern is not to avoid the creation of objects, but to eliminate the creation of objects within the same client code that uses the object. By employing the factory design pattern, the resultant client code, is not only clean, but also clear and more readable than a traditional code that mixes up the creation of objects and their use in the same code. The client code only requests the object of interest, and leaves the detail of its construction to the object factory. New comers to design pattern can envision this process as akin to the manufacture of an A380 for instance, where Boeing (the client) would call on General Electric Co. (the factory) to produce, construct or create a jet engine for Boeing, to be used for the production of the A380 jet.

In the preceding instance, Boeing has been able to isolate or specialize the creation of the required jet engine from the overall manufacture of the A380 jet, thus making its assembly lines leaner, cleaner and more focused on its end game.

The text book and classic definition of the Factory Method is, a pattern intended to help assign responsibility for the creation of objects. According to the Gang of Four in their classic publication: "Design patterns, elements of reusable objected oriented software"; the intent of the Factory Method is to define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

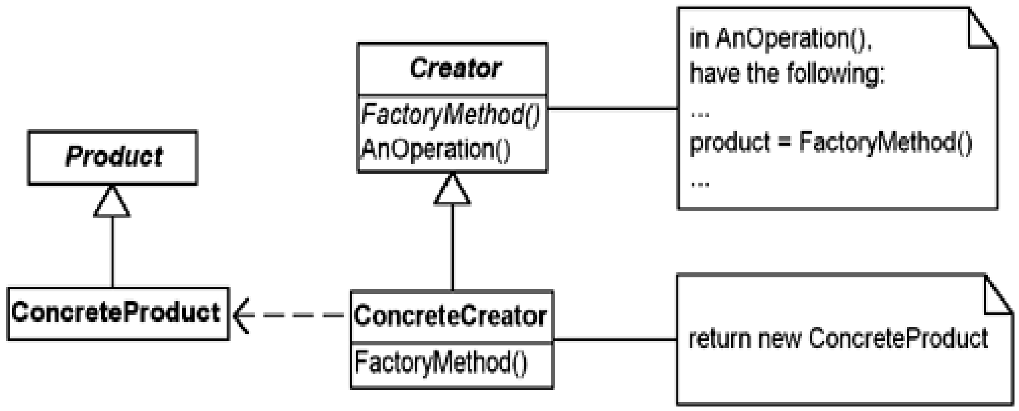

Fig 1: Above is a simplied diagram of the inside view of the factory pattern [Alan Shalloway, et al]

1.1 The (Factory Pattern) Distinctive Characteristics

Approach	▪ Define an interface to create the required object. ▪ Specify the method that creates the object in the interface. ▪ Define subclasses of the interface. ▪ Let subclasses decide which class to instantiate. Defer instantiation to subclasses.
Situation	▪ A class needs to instantiate a derivation of another class, but doesn't know which one. ▪ Separate the creation of an object from its use. ▪ The Factory Method allows a derived class to make this decision.
Solution	▪ A derived class makes the decision on which class to instantiate and how to instantiate it.
Participants and Collaborators	▪ Product is the interface for the type of object that the Factory Method creates. ▪ Creator is the interface that defines the Factory Method.
Consequences	▪ Clients can use objects created by the interface. ▪ Client can extend the interface by overriding the create method of the interface via subclasses of the create interface.
Implementation	▪ Use the create method in the interface to create objects as required. ▪ Use the overridden version of the create method in the subclasses to create objects as desired.

2. ABSTRACT FACTORY PATTERN

When you need to create a collection or set of related objects without an explicit specification of their concrete classes, the abstract factory is the design pattern of choice. When you encounter the necessity to choose between two or more objects that are closely related, there usually exists the requirement to implement the abstract factory to simplify the logic of selection or switching between these objects. In a system where a group or related objects are required to perform distinct task; if these objects are not created in a proper and timely manner, the resulting solution would be one that places more than necessary responsibility on some objects and less than necessary responsibilities others. These misplaced responsibilities would eventually lead to a low cohesion and tight coupling among the various classes that constitutes the subsystem of related objects.

Implementing an abstract factory ensures that objects are constructed and used in a manner that reflects the true structure and hierarchy of the system.

We would attempt to describe the necessity of an abstract factory using a student admission system. The student admission system that follows, awards scholarships and determine student tuition based on whether the student is a graduate or an undergraduate student, or whether the student is a campus based or online student. In this scenario, the admission system has the responsibility of deciding if a student object is of type graduate or undergraduate, and then determining if the student type is a campus base or distant student; before applying the rules for scholarship and tuition. In this situation, it will be appropriate to separate the task of applying the rules of admission from that of determining and creating the type of student. This is where the need for the abstract factory arises. The admission subsystem is better off delegating the duty of determining and creating the student type to the abstract factory, and concentrating on the application of the admission rules to the student type that are received from the abstract factory.

As shown in figure 2 below, the abstract factory which we called Student Factory, will have two methods: get Graduate Student (); and get Un Grad Student ();. It will also have two concrete classes namely: Online Student Factory and Campus Student Factory classes. The concrete classes will have methods that override those in the abstract base class. The benefit of this approach is that, the admission system ask the abstract factory for the desired student type, and apply the admission rules to the received student type without course to worry about the manner in which the student object was derived. The abstract factory makes the decision of which of the classes of related student object to create at a given point in time.

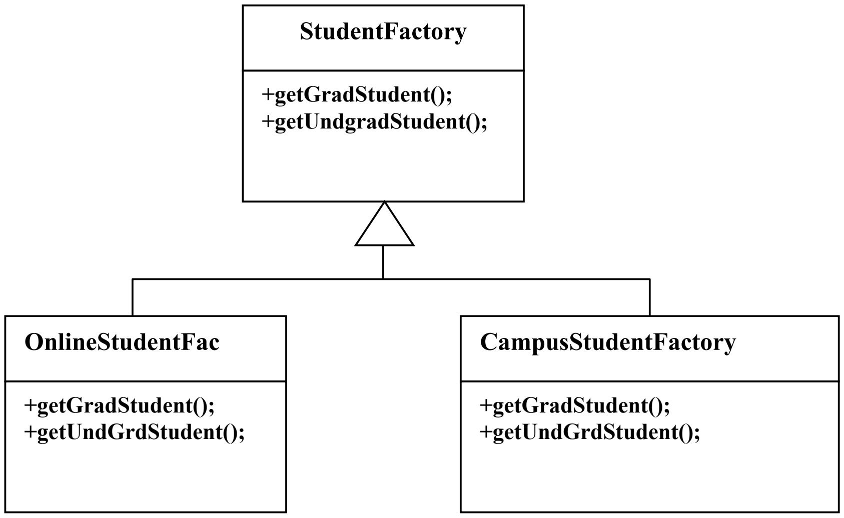

Fig 2: Basic Abstract Factory pattern

Below are three key strategies involved in the Abstract Factory.[Alan Shalloway, et al]

Strategy	**Shown in the Design**
Find what varies and encapsulate it.	▪ The choice of which student type to apply which scholarship and tuition rule varies. ▪ So we encapsulated the determination of the student type in the abstract factory.
Favor composition over inheritance.	▪ We have embed the varying student objects in one Student Factory object. ▪ The admission Rule can extract and use the embedded student objects from the Student Factory object, as opposed to having different admission objects for different student type.
Design to interfaces, not to implementations.	▪ Admission Rule only need to ask the Student Factory to instantiate student objects. ▪ It does not need to know or does not care how the Student Factory actually construct the student objects.

VOL. 4, NO. 12, December 2014 ISSN 2225-7217
ARPN Journal of Science and Technology
©2011-2014. All rights reserved.

http://www.ejournalofscience.org

See figure 3 below for a full diagrammatic section of the interaction of the subsystems of the admission system.

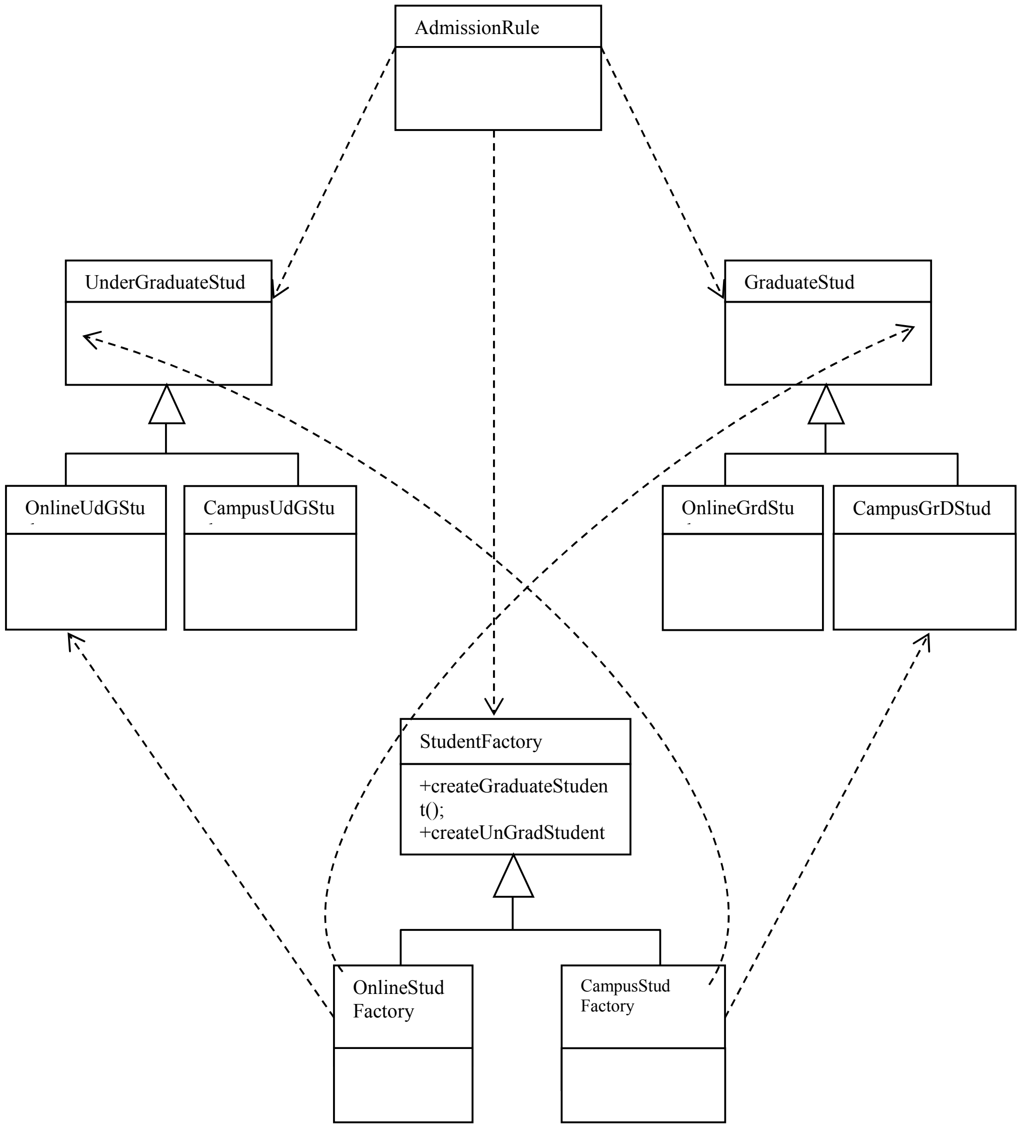

Fig 3

2.1 Classic Definition of the Abstract Factory

According the text book of design patterns (Elements of Reusable Object-Oriented Software) the definition of Abstract Factory pattern is to “Provide an interface for creating families of related (or) dependent objects without specifying their concrete classes.”[Eric Gamma, et al]It is also known as KIT.

The abstract factory pattern can be used in the scenarios where

- 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.
- When we want to provide a class library of products, and when we want to reveal just their interfaces, not their implementations.

Deciding which factory object is needed is really the same as determining which family of objects to use.

2.2 The Abstract Factory Pattern: Characteristics

Intent	▪ You want to have families or sets of objects for particular clients.
Situation	▪ Families of related objects need to be instantiated.
Solution	▪ Coordinates the creation of families of objects. ▪ Gives a way to take the rules of how to perform the instantiation out of the client object that is using these created objects. ▪ Allow concrete factory class instantiate appropriate objects
Participants and Collaborators	▪ The **Abstract Factory** defines the interface for how to create each member of the family of objects required. ▪ Typically, each family is created by having its own unique **Concrete Factory**.
Consequences	▪ The pattern isolates the rules of which objects to use from the logic of how to use these objects.
Implementation	▪ Define an abstract class that specifies which objects are to be made. ▪ Then implement one concrete class for each family.

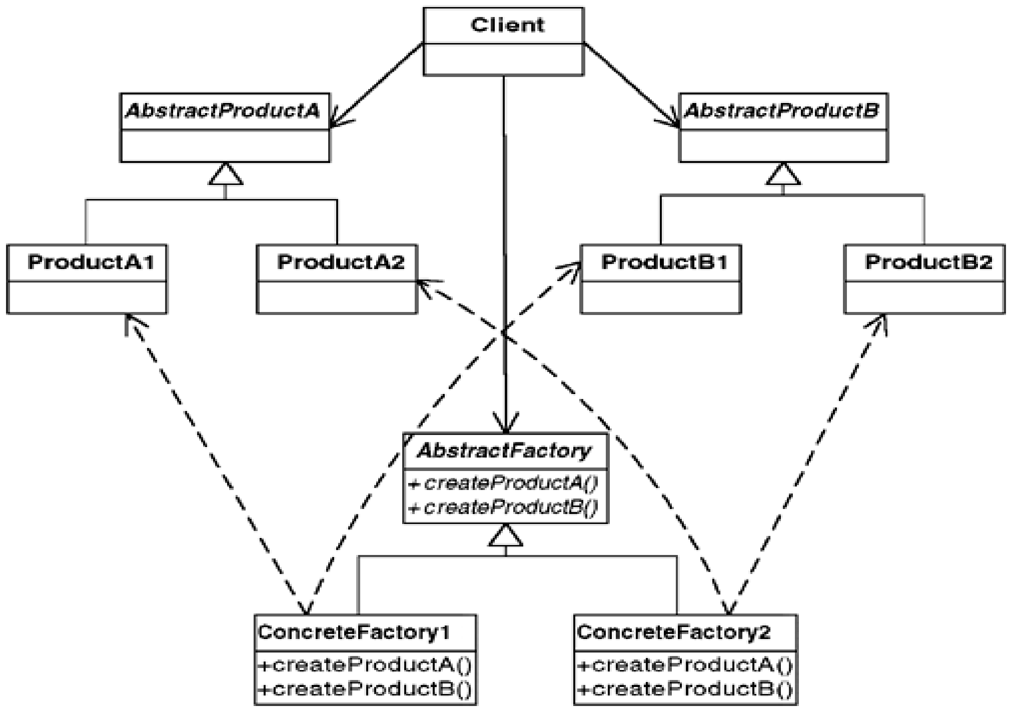

Fig 4: Standard, simplified view of the Abstract Factory pattern

Here is an excerpt of a discussion between two programmers during the design analysis stage of software development project.

Eskca: Hey Sandeep! How do we address the need to specialize the creation of the student objects in the admission process?

Sandeep: Since we have identified the uniqueness of the objects, I think we have to resort to one of the factories.

Eskca: You are right on that. Graduate online student, graduate campus student, under graduate online student, under graduate campus student? This is a multiple switch scenario. I think the Abstract Factory is the way to go.

Sandeep: Cool! We'll delegate the creation of the student objects to a Student Factory, and free up the admission Process to perform its very specialized functions of assigning admission rules.

Eskca: Right. All the admission Process need to do, is ask the factory to create whatever student object type it needs to process.

Example 10-6 C++ Code Fragments: Implementation of Res Factory

```
Class Student Factory {public:
Virtual Graduate Stud* get Grad Stud()=0;
Virtual Under graduate Stud* get Under grad Stud ()=0;}

Class Online Stud Factory : public Student Factory
{public:
Graduate Stud* get Grad Stud ();
Under graduate Stud* get Under grad Stud ();}

Graduate Stud* Online Stud Factory::get Grad Stud ()
{return new Online Grd Stud;}

Under graduate Stud* Online Stud Factory::get Under
grad Stud () {return new Online Und Stud;}

Class Campus Stud Factory : public Student Factory
{public:
Graduate Stud* get Grad Stud ();
Under graduate Stud* get Undergrad Stud ();}

Graduate Stud* Campus Stud Factory::get Grad Stud ()
{return new Campus Grd Stud;}

Under graduate Stud* Campus Stud Factory::get Under
grad Stud () {return new Campus Und Stud;}
```

3. CONCLUSION

We have made an attempt to simplify some of the very technical areas of the factory and abstract factory patterns, in definition as well as in implementation. It is our opinion that a lot of work still need to be done in this area, as understanding design patterns and applying that knowledge into practical software solution presents an enormous challenge to new comers to the idea of design pattern.

First new programmers have to grabble with the necessity of absorbing the object oriented programming mentality and then apply a new way of thinking (pattern base thinking) into their everyday programming solutions. It will be very helpful if more practical and simplified research work is carried out in this direction towards producing a sort of design pattern for dummy material.

ACKNOWLEDGEMENT

Our immense appreciation goes to Professor Tarik El Taeib for his support and encouragement.

REFERENCES

[1] Design Patterns Elements of Reusable Object-Oriented Software Eric Gamma, Richard Helm, Ralph Johnson, John Vlissides

[2] Design Patterns Explained: A New Perspective on Object-Oriented Design Alan Shalloway, James R. Trott

[3] Object-Oriented Design Case Studies with Patterns & C++Douglas C. Schmidt