VOL. 3, NO.10 Oct, 2012 ISSN 2079-8407
Journal of Emerging Trends in Computing and Information Sciences

http://www.cisjournal.org

An Approach to Reducing Complexity in Abstract Factory Design Pattern

[1] Aleksandar Bulajic, [2] Slobodan Jovanovic
[1] Faculty of Information Technology, Metropolitan University, 11000 Belgrade, Serbia
[1] aleksandar.bulajic.1145@fit.edu.rs, [2] slobodan.jovanovic@fit.edu.rs

ABSTRACT

It is well known that the Abstract Factory design pattern defines a new Abstract Product Factory for each family of products. Adding a new family of products affects any existing class that depends on it, and requires complex changes in the existing Abstract Factory code, as well as changes in the application or client code. Many papers discuss this issue and the issues related to increasing the number of classes. Often these papers recommend the Prototype pattern, or another solution is to add a parameter to operations that creates objects. This paper offers another approach to this issue—it employs a specially designed database, and in this way it reduces the number of Abstract Product factories and Concrete Product classes to one per Abstract Factory family. Also, adding a new product is comprehensible for the existing code, and in cases when it is necessary to add a new product, all changes are implemented without any actions related to alterations or updates of existing code.

Keywords: *Design-Patterns, Abstract Factory, Patterns Best Practice, Creational Patterns, Patterns by Example*

1. INTRODUCTION

Design Patterns are common patterns used in the Object-Oriented Software Design process. The primary goal of the Design Patterns is to reuse good design practice. Another important reason for using Design Patterns is to improve common application design understanding and reduce communication overhead by reusing the same generic names for the implemented solutions. Patterns are designed to portray the best practice in a specific domain. A pattern is supposed to present a problem and a solution that is supported by an example. It is always worth listening to expert advice, but keep in mind that common sense should decide a particular implementation, even in cases when already proven Design Patterns are being used. Critical views and frequent, well-designed testing provide the best answer about a design's validity and quality

Abstract Factory design pattern covers the instantiation of the concrete classes behind two kinds of interfaces, where the first interface is responsible for creating a family of related and dependent products, and the second interface is responsible for creating concrete products. The client is using only the declared interfaces and is not aware which concrete families and products are created.

Adding a new family of products affects any existing class that depends on it and requires complex changes in the existing Abstract Factory pattern code, as well as changes in the application or client code. "Design Patterns: Elements of Reusable Object-Oriented Software" discusses this issue and the issues related to increasing the number of classes, and recommends the Prototype pattern in cases when many product families are possible. Another discussed solution, one that is more flexible and less safe, is to add a parameter to operations that create objects. However, both of these solutions can create many product classes.

This paper proposes a new approach to this issue, and it reduces the number of classes and code complexity in the Abstract Factory design pattern. Namely, it reduces number of Abstract Product factories and Concrete Product factories to one per Abstract Factory family. Also, adding a new product is clear for existing code, and in cases when it is necessary to add a new product, all changes are implemented without any actions related to changes or updates of existing code.

The "Abstract factory Design Pattern" section introduces the Abstract Factory design pattern and the relations between Abstract Factory, Concrete Factory, Abstract Products and Concrete Products, interfaces and classes, and the Client class that is using this pattern. This section also introduces common terminology used in the rest of this paper. Understanding the differences between interfaces and concrete classes, as well as understanding the differences between a family of related and dependent products and the product itself, is very important for a proper understanding of the Abstract Factory Design pattern.

The "Related Work" section describes known issues related to reducing the number of classes and reducing Abstract Factory pattern complexity when changes are necessary to implement.

In the "Reducing Complexity When Supporting New Kinds of Product Family" section, the paper describes a new solution that reduces the number of product classes and reduces code complexity in situations where the Abstract Factory design pattern needs to be modified.

The "Illustrative Example" section, together with "Appendix Code Example" section, demonstrates this paper's solution by illustrative example, and provides a fully working code example implemented using Java language.

The "Discussion" section compares the advantages and disadvantages of this paper's solution to those of already known solutions. The "Conclusion" section

contains a short summary of authors' final conclusions and recommendations.

2. ABSTRACT FACTORY DESIGN PATTERN

Abstract Factory design pattern is described as an interface for creating families of related or dependent products without specifying a concrete class. [1] A client application in the Abstract Factory pattern uses generic interfaces to access a concrete class, therefore the client is not aware of which product is being used. Instantiation of concrete class is hidden inside the Abstract Factory. A Factory in the Design Pattern's vocabulary describes a place where other products are constructed. A Factory usually lets subclasses decide which class should be instantiated and this technique is also known as a deferred instantiation. The following classes are involved in designing the Abstract Factory pattern:

a. Abstract Factory – declares an interface for operations that create a family of related or dependent products.
b. Concrete Factory – implements the operations to create a concrete family of related or dependent products.
c. Abstract Product – declares an interface for a type of product object.
d. Concrete Product – implements the Abstract Product interface and creates a product object.
e. Client – uses only interfaces declared by the Abstract Factory and Abstract Product classes. The Client is not aware which concrete families and products are created.

The following picture (Fig. 1), which is largely a copy of the original Abstract Factory Design Pattern [1], illustrates this pattern by a class diagram:

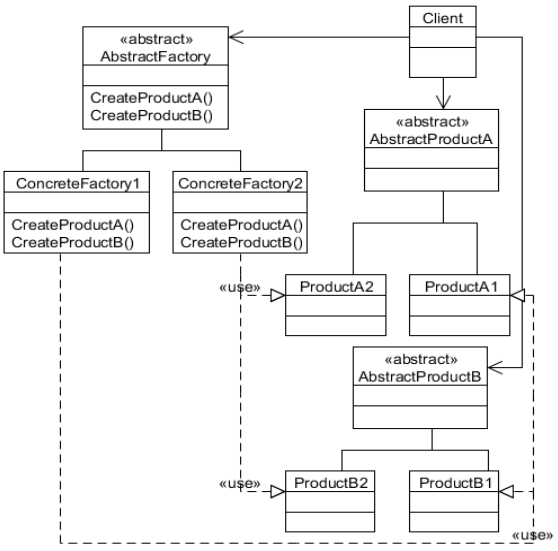

Fig 1: Abstract Factory Classic Design

From Figure 1, the UML class diagram, we can read that:

- The Abstract Factory pattern involves one Abstract Factory abstract class and multiple Abstract Product abstract classes. Actually, in the case of the Abstract Factory design pattern, an abstract class is created for each product.
- The Client class uses abstract classes and does not have knowledge regarding the concrete product.
- Each Concrete Factory is related to a particular set of Concrete Product classes. Products in this case are separated and cannot be mixed. In Figure 1, ConcreteFactory1 is always related to the ProductA1 and ProductB1, but ConcreteFactory2 is always related to the ProductA2 and ProductB2.
- Between Concrete Factory classes and Product classes exist weak relationships that are described by the dashed arrows. According to IBM Rational Software Architect [2], dashed arrows and the "<<use>>" word describe usage relationships "in which one model element (the client) requires another model element (the supplier) for full implementation or operation" and "does not specify how the client uses the supplier." [2]

Usage relation is described further as "a dependency relationship in which one model element requires the presence of another model element (or set of model elements) for its full implementation or operation. The model element that requires the presence of another model element is the client, and the model element whose presence is required is the supplier. Although a usage relationship indicates an ongoing requirement, it also indicates that the connection between the two model elements is not always meaningful or present." [2]

The following describes when this pattern can be applied [1]:

- a system should be independent of how its products are created, composed, and represented,
- a system should be configured with one of the multiple families of products.
- a family of related and dependent products 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.

Most of the current literatures about Abstract Factory design pattern use a slightly adapted Look & Feel example; the original example "consider a user interface toolkit that supports multiple look-and-feel standards, such as Motif and Presentation Manager" [1] is often replaced by Windows or Mac GUI interfaces, or especially today by the very popular Linux GUI Interface. Another very often used example is called the Pizza Factory example, and uses

different kind of pizzas to demonstrate the Abstract Factory design patterns.

These examples demonstrate the use of Abstract Factory with an advanced known number of products, but such implementation examples in the real world are very seldom and unusual. It is difficult to calculate whether sooner or later it will be necessary to add another product or even a family of related and dependent products, and it is even harder to imagine that the design can predict future changes in a particular GUI design, like the Windows or Linux front end interfaces and those front end interfaces related to mobile device screen design. It is just as difficult to imagine future implementation in the case of business applications and even such trivial examples like the Pizza Factory example.

3. RELATED WORK

Presented in Figure 1 are Abstract Product A and Abstract Product B, two abstract product factories, as well as two Concrete Product classes for each abstract product factory—ProductA1 and ProductA2 classes and ProductB1 and ProductB2 classes. The UML class diagram in the Figure 1 is easily readable as long as there are a limited number of products. But how many real-life implementations have such a limited number of products? It is more convenient to have a huge number of products. Each new product in this case would create three classes, one Abstract Product class and two Concrete Product classes, in those cases when only two families of related or dependent products are available. In situations when the Abstract Factory works with more than two families of related or dependent products, one Abstract Product class and three Concrete Product classes will be created for each product.

The Maze Factory example [1] defines families of related or dependent products, like Room, Wall, or Door, but the number of product can explode if a room contains more details, such as pictures, furniture, flowers, carpets, stairs, windows, curtains, traps and pitfalls, and whatever else. The Room, the Wall, and the Door can be of different types, for example, Dining Room, Bedroom, Enchanted Room, Red Wall, Green Wall, Iron Door, Wooden Door, Front Door, Back Door, Bedroom Door, Door Needing Spell, etc. This will increase the number of families of related and dependent products as well as the number of products. What about the warehouse that sells various consumer products—food, clothes, machines, pesticides, alcohol, and who knows whatever else? Each product family in this case can have a huge number of products.
A solution to this problem is adding a parameter to the operations that create products. This parameter specifies the kinds of product to be created. A shortcoming of this solution is that client is unable to see the differences between classes of a product and cannot subclass specific operations through an abstract interface [1].

Another possible solution to reducing the number of classes when many product families are possible is the use of the Prototype pattern [1], [3] to avoid sub classing and the expensive creation of a new object using a “new” keyword. In Prototype patterns the objects are cloned and the parameters are used for specific attributes. However, Prototype is best applicable when class differences are very small and the main part of each class attributes is the same or very similar, or “when instances of a class can have one of only a few different combinations of state” [1]. Using Prototype introduces another problem because all classes are returned to a client with the same abstract interface type. “But even when no coercion is needed, an inherent problem remains: All products are returned to the client with the same abstract interface as given by the return type. The client will not be able to differentiate or make safe assumptions about the class of a product. If clients need to perform subclass-specific operations, they won't be accessible through the abstract interface” [1].

When using the Prototype solution, there is no need for a new concrete factory class for each new product [1]. Instead, there are prototypical instances of products initialized in the concrete factory. New products are created by cloning the corresponding prototype [1].

The first question that can be raised is what number of product prototypes need to be initialized? Other important questions when Prototype pattern is used are: “Who configures Concrete Factory, and with what prototypes and when? Is the configuration carried out when the Concrete Factory is instantiated? If so, does its constructor allow the client to supply prototypes of the client’s choosing? Is there an interface for setting up the prototypes after Concrete Factory instantiation? If so, does that interface include operations for getting the prototypes? And are there safeguards against mixing prototypes from different families? How do they work?” [3].

When a Prototype pattern is used, the mixing of products from different families should be safeguarded and implemented by code. The Abstract Factory pattern does not allow the mixing of products from different families and this is a default implementation by this pattern design.

Using a Prototype solution requires only one Concrete Factory class “and there’s no Abstract Factory class at all [3]. Can we call this design pattern an Abstract Factory design pattern when the Abstract Factory class is removed and replaced by a single Concrete Factory class? We believe this can be better described as a Prototype and Factory Method design pattern combination.

Supporting new kinds of products is difficult in the case of classic Abstract Factory design pattern when supporting a new product family requires changes in the Abstract Factory interface. “Supporting new kinds of products is difficult. Extending abstract factories to produce new kinds of Products isn't easy. That's because the Abstract Factory interface fixes the set of products that can be created. Supporting new kinds of products requires extending the factory interface, which involves changing the Abstract Factory class and all of its subclasses” [1].

A solution to this problem in real-world examples is replacing Abstract Factory with the Prototype or Composite pattern [1], [3]. The simple reasons are flexibility and a better handling of tree data structures, especially in case using the Composite pattern. However, flexibility has a price, and while the product from different families are separated by default implementation in Abstract Factory families of related and dependent products, in the case of a Composite design pattern the products from different families can be mixed. Here it is the developer's responsibility to keep these products separated.

Hierarchical data should not be underestimated. One example of how it can explode and how the design can affect number of objects is the "Lexi" document processor [1]. The components are described at the level of the individual characters that would, in the case of a typical book, contain 5000 characters per page and have 200 pages, thus allocating one million page objects [3].

4. REDUCING COMPLEXITY IN ABSTRACT FACTORY DESIGN PATTERNS

This paper presents an approach to reduce the number of classes and the code complexity in an Abstract Factory design pattern by employing a specially designed database. The approach in this paper differs from all of the above mentioned ones, and creates one-to-one relations between a family of related and dependent products and the product class. This means that each concrete family of products has one and only one corresponding concrete product class. This class implements operations that are responsible for the creation of all products that are related to the family of related and dependent products. There is only one Abstract Product class, and all products are implementing the same Abstract Product interface. The picture in Figure 2 illustrates this new design.

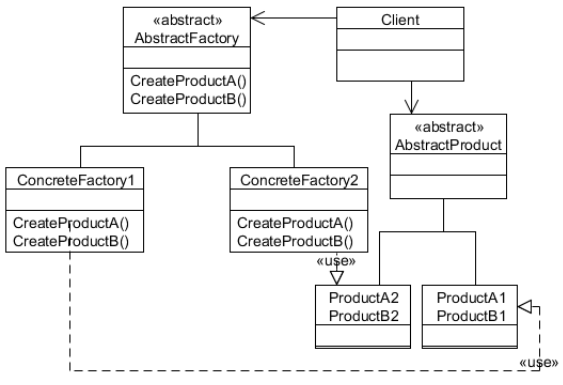

Fig 2: New Abstract Factory Design (with Reduced Number of Abstract Product Classes)

The most important difference seen in this solution is the removal of the Abstract Product class for each new product. The single Abstract Factory is good enough to provide a sufficient level of indirection so the client can still use only abstract classes without knowing how the concrete product has been created. Another very important difference from the previous solutions is that here each family of related or dependent products, in this case defined as a Concrete Factory, has one and only one corresponding product class that knows where the information about each product is stored and how to use it to create the desired product.

In this solution, products are initialized/created on demand. This means that products that are unnecessary are not created until they are needed. When it becomes necessary to modify an existing product or create an enhanced product, the concrete product class can be extended by a new class, and the abstract method is overridden and provided with enhanced product attributes or functionality.

If Figure 2 is compared to Figure 1, the Abstract Product B class is removed and the concrete classes Product B2 and ProductB1 are attached to the Abstract Product class that is used to create all concrete products. Each family of products, in this case ConcreteFactory1 and ConcreteFactory2, has only one product class for creating all products. This approach will significantly reduce the number of product classes; the number of product classes will be the same as the number of product families classes.

Figure 3 below illustrates in more detail this new kind of design, where a specially designed database is used. What is important to note in this solution is that a parameter solution is not used to create products. Instead of using a parameter, all information about a product is stored in the permanent memory outside of the application. In this case it is not important if the permanent memory is a database, file system, or XML document. All the product class needs to know is where this information is stored. The whole process is transparent from the Client's point of view and Client is unaware how the concrete class creates a product.

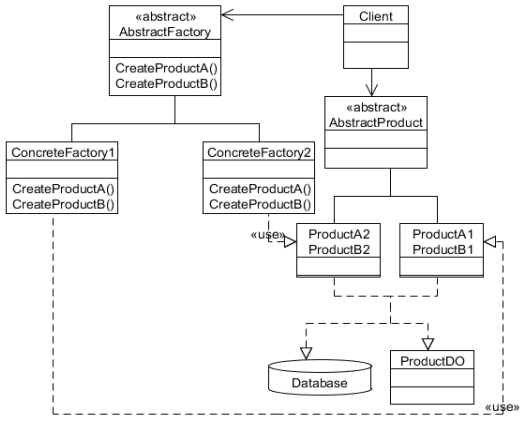

Fig 3: New Abstract Factory Implementation (with Reduced Number of Abstract Product Classes)

Information about each product is stored in the database, and a Product DO, (Product Data Object) class is used for mapping the database product information to the Java object. This means that, for example, all database product table columns are mapped to the Java variables

defined inside of the Product DO class. In this case, product information can be stored and updated or replaced without any changes in the existing code.

If a database contains a number of different products that belongs to the same family of products, all of these products can be easily retrieved and created at once. When it is necessary to add a new product, there is no need to make any changes in the existing code. All that is necessary is to simply add a new product to the database record. For example, if the database record contains information about the product family, product type, and product description, that information is sufficient to create the product. The database record can easily be extended by other product-related information. The Java code in the "Appendix Code Example" section demonstrates this solution.

This solution also solves the inheritance issue that exists when the parameter solution is used, where all the products are returned to the client with the same abstract interface type and the client is unable to differentiate between the classes of products [1]. In this solution, each concrete class belongs to a specific product; with a simple class-casting or class-type operator, like for example a Java instance of operator, it is easy to distinguish between product classes and implement specific operations solely to a particular product class.

In the proposed solution, adding a new family of product is as simple as adding a new product family implementation class and new product concrete class. There is no need to change each single class that implements the Abstract Factory interface because all methods in this class return the same type—the Abstract Product class type. This means that existing abstract methods can be reused and the implementation class needs only to provide implementation that can support a new family of products.

The cost of implementing this solution is mainly in additional CPU time used to access the database and retrieve product-related information. In this case, the major advantage is its generic approach and the use of common procedures. Access to the product information and products' attributes as well as the procedures for product creation can be highly generalized. For example all products in the warehouse share common attributes, such as product code, name, description, price, quantity at stock, availability, etc. A second advantage of this solution is its initialization on demand, which can help better utilize computer resources.

It can be argued that accessing a database is never simple and involves user identification as well as authorization, and network infrastructure and costs related to network transportation. However, today it is difficult to imagine that products are still hard-coded and stored inside of a class. Products are usually stored in permanent storage, such as a file system or a database. This provides one more argument for using this solution, which optimizes use of available resources, such as memory. The network speed is no longer a limitation factor as it was some years ago.

5. ILLUSTRATIVE EXAMPLE

The example discussed in this section is based on a requirement to provide a menu list and furniture vendor list for different kind of restaurants. Restaurants are classified as Italian, Indian, and Chinese. The class diagram in Figure 4 illustrates how this would look if it is implemented using the original Abstract Factory design pattern described in Figure 1. Here the Manufactory and Furniture Factory corresponds to the Abstract Product A and Abstract Product B classes (from Figure 1), and Restaurant Factory corresponds to the Abstract Factory class.

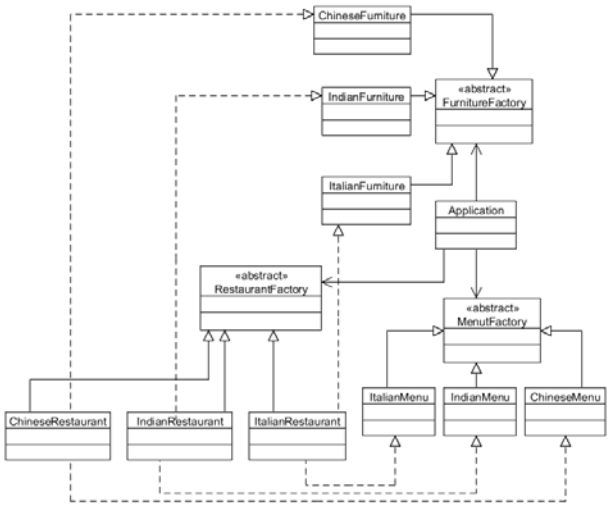

Fig 4: Abstract Factory Classic Design Example

The class diagram in Figure 5 illustrates the new solution that is implemented in this paper. When these two class diagrams (Fig. 4 and Fig. 5) are compared, the differences are obvious and easy to spot. The most important difference in this case is that Menu Factory and Furniture Factory, the classes that are used for managing product creation, are now represented by only one class called the Product Factory. The Product Factory class is responsible to create the menu and furniture, and any other product that belongs to a particular restaurant family.

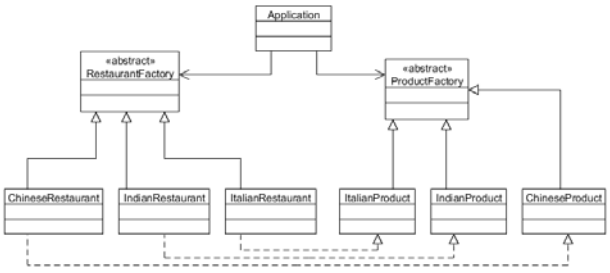

Fig 5: Design Example with Reduced Number of Abstract Product Classes

What is also very clear from these two pictures is that when it is necessary to create more various products, such as decorations, dishes, machines, tools, and whatever else that can be specific to each restaurant type, the class

diagram will become very crowded and impossible to draw, or to follow and understand using a human being's built-in sensors like eyes and a brain. The class diagram in Figure 5 will be the same without any differences, whether it has only one product or huge number of products. The code examples presented in the "Appendix Code Example" demonstrate this solution. The most important example to note is the Product Factory class, and Italian Product or Chinese Product classes that create the concrete products.
For example, the Italian Product. Create Product () method ensures that all products belonging to the Italian family products are selected and created. Italian Product. Display Product Type () displays each product. The code example does not include the Indian restaurant and Indian products, but can be easily created by copying existing classes; for example, by copying Italian Restaurant class to Indian Restaurant class and Italian Product class to Indian Product class, and by simply correcting the Italian names to Indian names.

6. DISCUSSION

If this paper's solution is compared to a parameter solution [1], then the first noticeable difference is that this solution does not use a parameter to create a product. In this case, a parameter is not necessary because a concrete product class belongs to each family of related or dependent products. This concrete product class is responsible for product creation.

Another important difference from the parameter solution is that this solution solves the inheritance issue that exists in the parameter solution, where all products are returned to the client with the same abstract interface type and the client is not able to differentiate between the classes of products [1].

While the parameter solution for reducing the number of classes does not provide a mechanism for establishing variations between different products and all of the products are of the same class type, the paper's proposed solution enables differentiates between the product classes and enables implementation that is specific only for a particular class type. Although all of the products are using the same Abstract Product interface, by using a simple class-type operator like Java instance of operator, for example, or by casting an Abstract Product class to a particular concrete product class type, it is possible to distinguish between product classes and implement specific operations only to a particular product class.

The parameter solution requires only one concrete product class exist, but this solution provides concrete classes for each family of related or dependent products.
While the Prototype solution requires initialization of the each product prototype, the proposed solution does not require any kind of initialization. Products are created on demand, when each product is requested. The Prototype solution can be compared to the parameter solution because cloned products need to be adapted and updated using parameterized methods (setter methods).

The Prototype solution is applicable in cases when products are slightly different. When products differences are significant, a prototype model can require a lot of methods and parameters to adapt s cloned product, which can create a code that is difficult to understand and follow.
The Prototype introduces another problem because all products are returned to a client with the same abstract interface type. "But even when no coercion is needed, an inherent problem remains: All products are returned to the client with the same abstract interface as given by the return type. The client will not be able to differentiate or make safe assumptions about the class of a product. If clients need to perform subclass-specific operations, they won't be accessible through the abstract interface" [1].

When the Prototype pattern is used, the mixing of products from different families should be safeguarded and implemented by code. In this paper's proposed solution, the mixing of products from different families is prevented by design.

While the solution demonstrated in this paper is applicable to examples where products can be described by attributes, in instance of the Look & Feel examples where each product creation requires a different API call or different set of API calls, this solution would not be an appropriate one.

Design Patterns becomes synonymous with good design and the implementation of best practice, as well as a common mechanism for recording and sharing design knowledge and experience. However, one should never forget that Design Patterns are templates. Each Design Pattern implementation should be adapted to the particular context. There is no available single solution. Rather, different solutions are offered that are dependent on a problem's complexity. As the development is an iterative process, so too is its use of patterns. As Martin Fowler stated: "Once you've made it, your decision isn't completely cast in stone, but it is trickier to change. So it's worth some upfront thought to decide which way to go" [4].

7. CONCLUSION

This paper demonstrates a solution where adding a new product class does not require complex changes in existing code, and the number of product classes is reduced to one product class per family of related or dependent products. Besides this, this paper offers a solution that can automate the adding of new products without changing any line of existing code.

The solution demonstrated in this paper and illustrated by the code example has the following advantages:

- Significantly reduces the number of Concrete Products classes,
- Removes complexity in cases when new kinds of Product need to be introduced,
- Flexibility that enables adding new product

without changing any line of existing code. Adding new product can be as simple as adding a new row to a database table, or adding a row to a file, or adding a new entry in an XML file,

- The Abstract Factory design pattern structure is not changed and client is working with abstract classes unaware how concrete families of related or dependent products, as well as concrete products are created.

If products are stored inside of database table, then selecting all the products that belong to a particular family of products is a very simple operation. If products are stored inside of an XML file, then the operation to select all of the products is more complex and first requires XML file parsing.

However, there is also a price that in this case is identified as an additional use of CPU time to access the database and retrieve product information and network overhead.

This solution's complexity of adding a new product, or changing an existing one, is moved outside of the Abstract Factory design pattern and replaced with storing product information in the simple database table.

Although it can be argued that accessing a database is never simple and involves user identification, as well as authorization and involvement in the network infrastructure today is difficult to imagine that products are still hard-coded inside of a class. Products are usually stored in permanent storage, like a file system or a database. This is one more argument for applying this solution that optimizes the usage of available resources, such as memory resources.

REFERENCES

[1] Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995), Design Patterns Elements of Reusable Object-Oriented Software, Boston: Addison-Wesley. Partially available online (http://c2.com/cgi/wiki?DesignPatternsBook) (17 March 2003)

[2] IBM_RSA (2004, 2005), "Rational Software Architect," IBM, available at Internet (http://publib.boulder.ibm.com/infocenter/rtnlhelp/v6r0m0/index.jsp?topic=%2Fcom.ibm.xtools.modeler.doc%2Ftopics%2Frreltyp.html)

[3] Vlissides, John (1998), "Pattern Hatching: Composite Design Pattern: They Aren't What You Think," available at Internet, (http://http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.88.7306)

[4] Fowler, Martin (1999), Refactoring: Improving the Design of Existing Code, Boston: Addison-Wesley.

APPENDIX

Java Code Example

```
// Application class:
Public class Application {public enum Restaurant Type {Indian, Italian, Chinese}
    Public static void main (String [] args) {Restaurant Factory restaurant Factory = null;
        Restaurant Type restaurant Type = Restaurant Type. Italian;
        Switch (restaurant Type) {case Indian: restaurant Factory = new Indian Restaurant (); break;
    Case Italian: restaurant Factory = new Italian Restaurant (); break;
    Case Chinese: restaurant Factory = new Chinese Restaurant (); break ;}
    Product Factory product Factory = restaurant Factory. Create Product ();
    Product Factory. Display Product Type ();}}
Restaurant Factory class:
Public abstract class Restaurant Factory {public abstract Product Factory create Product ();}
// Italian Restaurant class:
Public class Italian Restaurant extends Restaurant Factory {@Override
Public Product Factory create Product () { return new Italian Product ().create Product (); } }
Chinese Restaurant class:
Public class Chinese Restaurant extends Restaurant Factory {@Override
Public Product Factory create Product () {return new Chinese Product ().create Product ();}}
// Product Factory class:
Public abstract class Product Factory {public abstract void display Product Type ();
    Public abstract Product Factory create Product ();
}
// Italian Product class:

Public class Italian Product extends Product Factory {private Array List<Product> product List = new Array List<Product> ();
    @Override

Public void display Product Type () {for (Product product: product List) {System. out. Print ln (product. Get Product Description ()); }}
    @Override

Public Product Factory creates Product () {select Product (); return this ;}

Private void select Product () {Product menu = new Product ();
Menu. Set Product Class ("ItalianMenu");
menu.setProductDescription ("Italian Menu\n\n" + "Italian Appetizer1\n"
```

```
    + “Italian Appetizer2\n\n” +” Italian MainCourse1\n”
+ " Italian MainCourse2\n\n"
     + " Italian Desert1\n"  + " Italian Desert2\n\n" );
          menu.setProductfamily                ("Italian");
menu.setProductType ("menu");
 productList.add (menu);
          Product    furniture    =    new    Product    ();
        furniture.setProductClass ("Italian Furniture");
          furniture.setProductDescription          ("Italian
Furniture Vendors\n" + “Italian Vendor1\n"
                  + " Italian Vendor2\n"  +  " Italian
Vendor3\n\n");
          furniture.setProductfamily           ("Italian");
furniture.setProductType ("furniture");
        productList.add (furniture); } }
// Chinese Product class:

Public class Chinese Product extends Product Factory {

Private Array List<Product> product List = new Array
List<Product>();
        @Override

Public void displayProductType () {
            For (Product product:  product List) {
                        System.out.println
(product.getProductDescription ());} }
        @Override

Public Product Factory create Product () {select Product
();  return this; }

Private void select Product () {
        Product    menu    =    new    Product    ();
menu.setProductClass ("Chinese Menu");
        menu.setProductDescription ("Chinese Menu\n\n"
+ “Chinese Appetizer1\n"
        + “Chinese  Appetizer2\n\n”  +  "  Chinese
MainCourse1\n"
+ " Chinese MainCourse2\n\n” + " Chinese Desert1\n"
+ " Chinese Desert2\n\n");
menu.setProductfamily                ("Chinese");
menu.setProductType ("menu");
productList.add (menu);
Product    furniture    =    new    Product();
furniture.setProductClass("ChineseFurniture");
furniture.setProductDescription    ("Chinese    Furniture
Vendors\n" + “Chinese Vendor1\n"
                  + " Chinese Vendor2\n" + " Chinese
Vendor3\n\n");
furniture.setProductfamily           ("Chinese");
        furniture.setProductType ("furniture");
productList.add (furniture); } }
// Product class:
Public class Product {

Private String product Family;  private  String  product
Type;

 Private String product Class; private String product
Description;
Public void setProductfamily (String s) {  productFamily
= s;}

Public void setProductType (String s) {  product  Type
= s;}

 Public void setProductClass (String s) {product Class =
s;}

Public  void  setProductDescription  (String  s)  {product
Description = s;}

Public String getProductDescription () {  return  product
Description; }

Public String getProductClass() {return product Class;}

Public String getProductType() {return product Type;}

Public  String  getProductFamily()  {return  product
Family; }}
```