DOI 10.34185/1562-9945-1-132-2021-09
УДК 614.2+574/578+004.38

A.A. Litvinov

ON THE VARIATION OF ABSTRACT FACTORY PATTERN

Annotation. The work is devoted to revision of Abstract Factory Pattern for its practical use in information system development. Provided variation of Abstract Factory Pattern significantly simplifies the process of information system development making the system more flexible, robust to change and maintainable.
Key words: software development, design patterns, Factory pattern.

Modern information system can be represented as a set of interconnected components organized to resolve a computing problem. To resolve the problem, components work cooperatively interacting each other. Generally, the interconnections between the components can be divided into two large classes: static which means that the connections between the components are defined before the system starts working and cannot be changed while the system is running; dynamic means that the connections between the components are not predefined and can be changed without stopping and rebooting the system, as a rule in such systems the interaction between the components is controlled by the subsystem-mediator (e.g. event bus, blackboard architectures [1]).

In reality, in most cases, there is no need to make all the connections dynamic, but, it is not an exaggeration to say that the flexibility of the system depends on the ability to replace one component by another. Such ability provided by late binding mechanism and polymorphism. Late binding/typing means that the types of all names (variables and expressions) are not known until runtime [2]. Polymorphism represents a concept in which a single name (e.g., variable declaration) may denote objects of many different classes that are related by some common superclass or interface. In modern systems interface polymorphism is considered preferable. Interfaces allow to define polymorphism in a declarative way, unrelated to implementation. Two objects are polymorphic with respect to a set of behaviors if they realize the same interfaces.

Interface-based polymorphism allow to build information systems out of loosely coupled components depended on interfaces instead of other components.

© Litvinov A.A., 2021

Such system becomes interface-based, because instead of a system composed of real components now it is being transformed to the system where dependencies (dependency components) substituted by the interfaces [3]. We can see the same approach in all spheres of real life starting from automobiles and airplanes and ending with business companies and social organizations. That is one of the best ways to make a system flexible and consequently robust to change.

Thus, when we use interface-based approach the main goal we are intended to achieve is to increase the flexibility and changeability of the system by constructing it as a set of loosely coupled components.

The next question is, how to resolve which of the components should realize the interface in the current assembly of the system (the system configured and ready to run)? More precisely, how the system can make the choice of the components should work in the current assembly substituting the interfaces they realize? To make the process of building the components flexible the system needs mediators able to create the realizations of the interfaces. That mediators called factories [4][5]. The main idea is as follows:

We need to produce a realization of an interface considering that there may be several variations of the realization.

To produce the realization of the interface we need a component which knows how to create that realization. In case of several realizations there is a need of several factories producing different realizations of the same interface. It is notable that the creation process can be nontrivial (e.g., can relate to resolving dependencies etc.)

If we have several different creators-factories, we need to introduce an abstract one realized by the set of concrete creators-factories. And when the system starts it should simply choose the factory realization and then use it to create interface realizations.

Abstract factory can be realized as an abstract class or as an interface. Interface variant seems more conformable for the factories depended on other factories.

The next question related to the task of creating a huge number of different objects from the same level of abstraction (in reality, in most cases, the level is equal to layer). In that case there is no need to create a huge number of factories, each per interface (Fig. 1). That is the idea of the abstract factory pattern which offers the interface for creating a family of related objects (Fig. 2) [6]. The client chooses the realization of an abstract factory and gets a number of products produced by the chosen factory which satisfy the client's needs.

ISSN 1562-9945 (Print)
ISSN 2707-7977 (Online)

Task definition. The problem of classical abstract factory solution is that the adding of a new product to the domain implies the need of adding a method to all the factories. When the number of types of products is high it causes maintenance problems.

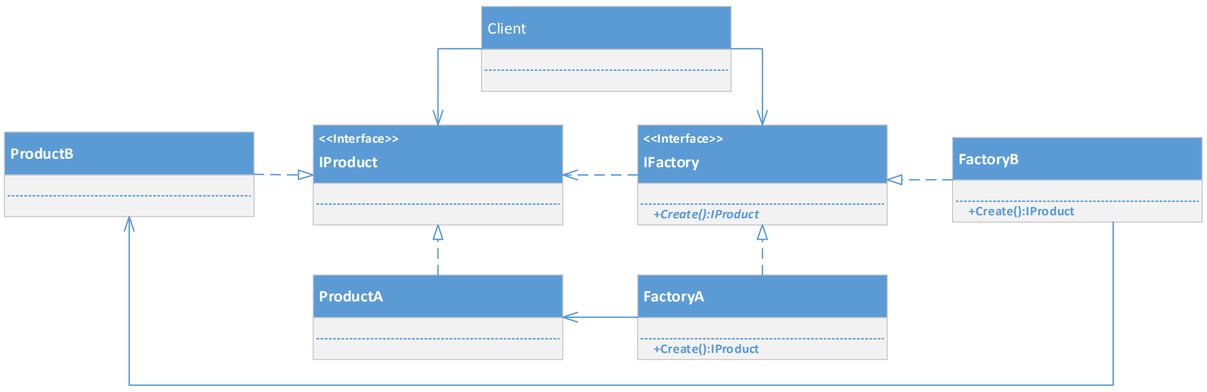

Figure 1 – Factory method pattern

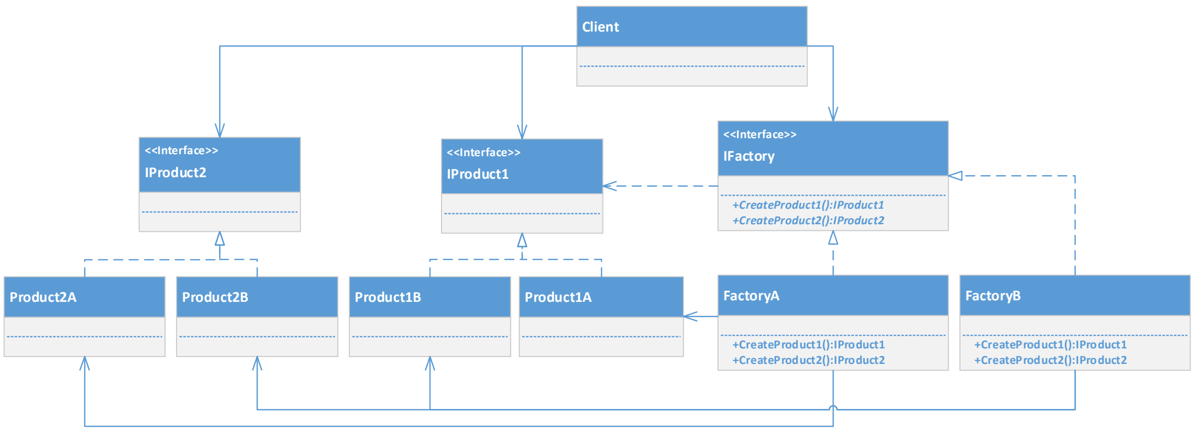
Client
<<Interface>>
IProduct2
<<Interface>>
IProduct1
<<Interface>>
IFactory
+CreateProduct1():IProduct1
+CreateProduct2():IProduct2
Product2A
Product2B
Product1B
Product1A
FactoryA
+CreateProduct1():IProduct1
+CreateProduct2():IProduct2
FactoryB
+CreateProduct1():IProduct1
+CreateProduct2():IProduct2

Figure 2 – Abstract factory pattern

Main part. To reduce the resistance of the infrastructure we can introduce instead of huge number of methods (each per interface) one generic Create method which takes as a parameter an interface and returns an appropriate realization of that interface or throws an exception when it is not able to resolve the binding (Fig.3).

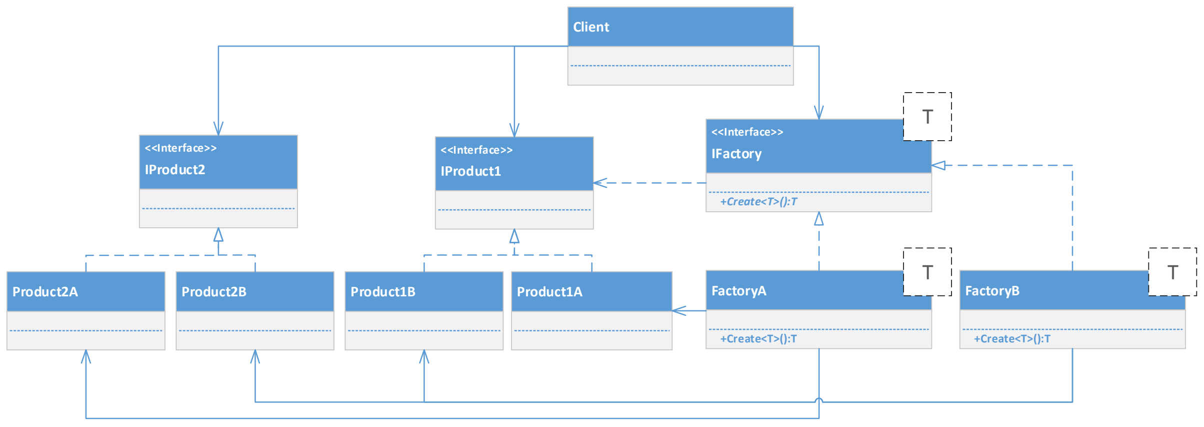

Figure 3 – Abstract factory pattern with generic method

The simple implementation of the pattern is shown in listing 1. The collection of “interface - realization” bindings represented within the constructor defines the only one area of the class opened to change (open/close principle of SOLID) [7]. We can make an abstract class Factory with the collection and generic Create method, making all other factories its successors (Fig. 4). We can also add Create method with parameters needed to create the object, but in our opinion, it is better to avoid creating an object with passing the parameters, we think it is better to use factories interaction instead. Factories interaction means that in case of factories dependency we can inject the dependency factory into dependent one according to the dependency injection principle of SOLID. In other words, in case when we need to create an object produced by the first factory which is dependent on an object produced by the second factory, it would be better to resolve the situation by injecting factory responsible for creation of dependency object into the factory used to create dependent one. For example, the creation of use cases objects depends on repository objects and to resolve such dependency the factory responsible for repository objects creation should be injected to factory responsible for use cases objects creation. The code shown in listing 1 suggests “single object per multiple creation calls” model, but it can be easily modified to realize “create object per each call” model by the deep cloning of created objects. To make lazy loading variant of the solution, we can use two dictionaries (one for “type of interface – type of realization” pairs and one for “type of interface - realization” one) and we should also rebuild the logic of Create method to examine whether the realization of the requested interface type has already built, using “type of interface - realization” dictionary, and, if not, make it using reflection API (i.e., Activator.CreateInstance method), using “type of interface – type of realization” dictionary.

ISSN 1562-9945 (Print)
ISSN 2707-7977 (Online)

Listing 1

```
public class ValidationRuleFactory : IValidationRuleFactory
{
    readonly Dictionary<Type, object> collection =
                                   new Dictionary<Type, object>();
    public ValidationRuleFactory()
    {
    this.collection.Add(typeof(IEnterOperationValidationRule),
                                new EnterOperationValidationRule());
    }
     public T Create<T>()
    {
        Type type = typeof(T);
         if (!this.ruleCollection.ContainsKey(type))
        {
            throw new MissingMemberException(type.ToString() +
                           "is missing in the rule collection");
        }
         return  (T)this.ruleCollection[type];
    }
}
```

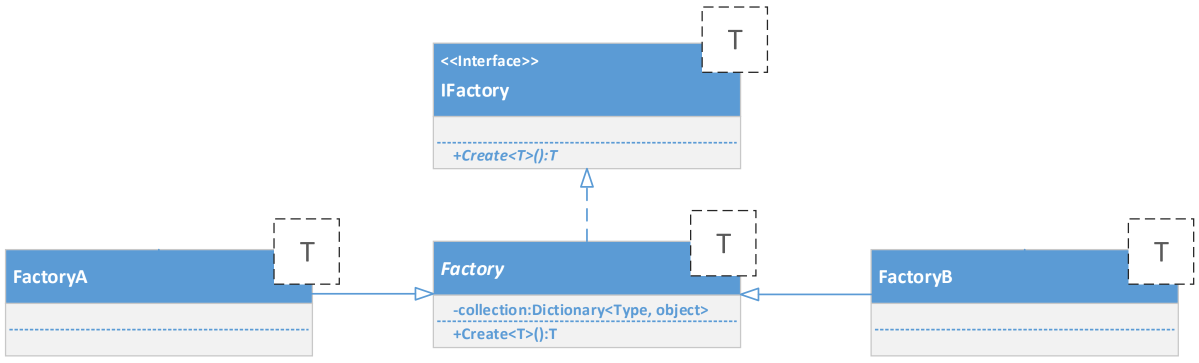

Figure 4 – Introducing generic factory responsible for object creation

The disadvantage of provided solution is additional effort connected to adding new interface and its realization causing the modification of the factory constructor. The code provided in the listing 2 could help solving that problem. Using reflection mechanism, we can get all the classes of the assembly and bind them with the interfaces from the assembly responsible for the definition of interfaces. Thus, in the result we have the collection of "interface - realization" bindings filled automatically by the factory. It is remarkable that classes and interfaces should follow predefined naming conventions, but it could be considered as a positive point rather than something to avoid.

Listing 2

```
private void ConfigureMapping()
        {
            Type currentClassType = Method-
Base.GetCurrentMethod().DeclaringType;
            string classAssemblyName = current-
ClassType.Assembly.GetName().Name;
            Assembly contractAssembly =
                      this.GetAssemblyByName(this.contractAssemblyNa
                      me);
            Assembly classAssembly =
this.GetAssemblyByName(classAssemblyName);
            List<Type> classTypeCollection =
                      classAssembly.GetTypes().Where(t => t.IsClass
                      &&
                      t.Name != this.factoryName).ToList();
            List<Type> interfaceCollection =
                       contractAssembly.GetTypes().Where(t =>
                                         t.IsInterface).ToList();
            foreach (Type interfaceType in interfaceCollection)
            {
                string name = interfaceType.Name;
                try
                {
                    Type classType =
                         classTypeCollection.Find(t => t.Name ==
                                                 name.Substring(1));
                    object classInstance =
                         Activator.CreateInstance(classType);
                    this.collection.Add(interfaceType, classIn-
stance);
                }
                catch (Exception ex)
                {
                    Logger.Log.ErrorFormat("Interface {0} is not
valid", name);
                }
            }
        }
        Assembly GetAssemblyByName(string name)
        {
            return AppDomain
                .CurrentDomain
                .GetAssemblies()
```

ISSN 1562-9945 (Print)
ISSN 2707-7977 (Online)

```
                .SingleOrDefault(assembly => assembly.GetName().Name
== name);
        }
```

According to the provided strategy, each solution become composed of three basic types of projects-packages:

✓ The projects-packages contained only the interfaces representing the contracts of the components to be realized, including abstract factory interface.

✓ The packages contained the realizations of the interfaces. Each realization package has its own factory.

✓ The projects responsible for gluing the system of the units-realizations using the factories.

An example of the solution is shown in Fig.5. Service package is responsible for system assembly, Contract packages responsible for interfaces definition.

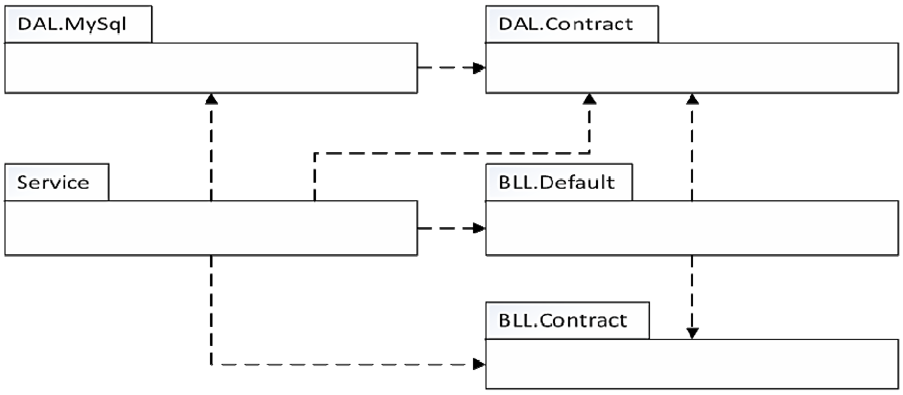

Figure 5 – An example of the solution according to the strategy provided

As we can see Service package depends on all other packages to glue the system of components. And in case when the number of variations increases, the dependency of Service package on all the existed packages will cause the problem. To make the solution more flexible we can use Inversion of Control (IoC) Container that is a framework used to manage automatic dependency injection throughout the application. For .NET platform there are a plenty of IoC Container realizations (Castle Windsor, Spring.NET, Autofac, Unity). Thus, we can use IoC Container for resolution of factories realizations based on configuration xml-file without recompilation of the solution. In our opinion, factories resolution is the best and most usable practice of resolution using IoC allowed to reduce expenses related to the activities to be applied in case of modifications (adding new types and interfaces etc.).

Conclusions. The suggested solution significantly reduces the resistance of the system to change making it highly flexible. Each time when we add new interface and its realization to the appropriate package(assembly) we do not need to correct the factory by adding new methods and correcting the logic within its constructor.

REFERENCES

1. R. N. Taylor, N. Medvidovic, E. M. Dashofy. Software Architecture: Foundations, Theory, and Practice. Wiley, 2009. 750 p.

2. Booch, Grady. Object Oriented Analysis And Design With Applications. Addison-Wesley, 2007. 691 p.

3. L. Robert Varney. Interface-Oriented Programming. University of California, Los Angeles Computer Science Department Technical Report TR-040016. March 29, 2004 Revised: September 17, 2004.

4. C.Larman. Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development. Pearson; 3rd edition. 2004. 736 p.

5. Eric Evans. Domain-driven design: tackling complexity in the heart of software. Addison-Wesley Professional. 2004. 560 p.

6. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software.Addison-Wesley Professional. 1994. 416 p.

7. Martin C. Robert Agile Principles, Patterns, and Practices in C#. / C. Robert Martin, Micah Martin // Prentice Hall. – 2006. – 768 p.

Received 16.01.2021.
Accepted 20.01.2021.

Варіація шаблону Abstract Factory

В роботі розглядається варіант вдосконалення шаблону Abstract Factory для практичної розробки сучасних інформаційних систем.

Запропонований варіант відрізняється тим, що замість декількох методів, які відповідають за створення реалізації певних інтерфейсів в класі Abstract Factory, вводиться один узагальнений метод. Метод отримує тип інтерфейсу реалізацію якого треба отримати. Крім того, розглядається варіант автоматичного створення словника, котрий містить пари «інтерфейс-реалізація» з використанням рефлексії. В висновках розглядаються обмеження та різноманітні модифікації наведеного розв'язання.

On the variation of Abstract Factory Pattern

Modern information system can be represented as a set of interconnected components organized to resolve a computing problem. To resolve the problem, components work cooperatively interacting each other. Generally, the interconnections between the components can be divided into two large classes: static which means that the connections between the components are defined before the system starts working and cannot be changed while the system is running; dynamic means that the connections between the components are not predefined and can be

ISSN 1562-9945 (Print)
ISSN 2707-7977 (Online)

changed without stopping and rebooting the system, as a rule in such systems the interaction between the components is controlled by the subsystem-mediator.

The work is devoted to revision of Abstract Factory Pattern for its practical use in information system development. Provided variation of Abstract Factory Pattern significantly simplifies the process of information system development making the system more flexible, robust to change and maintainable.

Литвинов Олександр Анатолійович - кандидат технічних наук, доцент кафедри електронних обчислювальних машин Дніпропетровського національного університету ім. О. Гончара.

Литвинов Александр Анатольевич - кандидат технических наук, к-цент кафедры электронных вычислительных машин Днепропетровске кого национального университета им. О. Гончара.

Litvinov Alexander Anatolievich — candidate of technical sciences, associate Professor of Computer Systems Engineering Department of DNU.