DYNAMIC FACTORY

New Possibilities for Factory Design Pattern

Dawid R. Ireno
Jagiellonian University,
6 Profesora Stanisława Łojasiewicza Street, Kraków, Poland
alamandra007@gmail.com, dawid.ireno@uj.edu.pl, ireno@ii.uj.edu.pl

KEYWORDS

Software development; design patterns; dynamic factory; dynamic languages; intermediate language; byte code; just in time compilation; runtime environment.

ABSTRACT

Software design patterns have been within developers' realm of influence for several years now. They come from every possible direction, indicating the best courses of action for problem-solving, and are well documented in numerous articles, magazines, and books. Some are corner stones, constituting the foundation of software development. Others are highly evolved complex constructions using other patterns as building blocks to bring about higher quality in more challenging situations. After years of experience in the Information Technology industry, every experienced developer has his own way of perceiving certain design patterns which he has used, and heard, read or talked about. But as the years pass, technology evolves, software design pattern knowledge is still not yet finally distilled, and new design patterns are created. In this article a new design pattern, which the author has called the dynamic factory, is explained. The new type of factory enhances the design pattern possibilities known so far. It creates new object types according to the situation, containing just what is needed, and nothing redundant.

INTRODUCTION

The factory design pattern in software manufacturing is a way to implement the object creation process in a situation where a constructor is not preferred. It is the standard manner of encapsulating logic that lies behind an object's construction. Although this seems somewhat straightforward, it can bring about a great deal of misunderstanding.

SOFTWARE DESIGN PATTERNS

A design pattern is a well-known and technologically independent way to solve a family of problems and is carefully documented, proven to be effective and recommended for use in developed projects.

CONSTRUCTORS

As software trends evolved and procedural languages changed to object ones, there becomes a need to construct objects. The foremost method in which to do this are constructors - factory functions, which reside in a class definition and always return objects of the class they are placed in. Constructor methods may be parameter-free, but may also have various arguments. But one thing is certain; it can never return an object of any derived class. Constructors can only produce instances of the exact class in which they are contained.

Let us suppose there is a class named Shape. The line initializing a new instance and assigning it to a variable would look similar to the following in most of today's known languages:

```
var shape = new Shape(); // Declare the shape
// variable, and initialize it with a new Shape
// class instance.
```

This generally means that an instance is created using default initialization logic. If a developer wants to pass some logic to an initialization block it is still as simple as it seems, assuming the class supports it.

```
var color = Color.Blue;
// Built-in enumeration of colors is used
// to initialize the variable.
// Then the color is passed to the Shape
// constructor.
var shape = new Shape(color);
```

If, on the other hand, a developer wants to create a circle, knowing that circle is a shape in its nature, meaning in the inheritance chain, a special convention is needed.

```
// Enumeration of shape types was
// previously declared in the code.
if (type == ShapeType.Circle)
  var shape = new Circle(color);
else
  ; // TODO: Handle other shape types here.
// Developer cannot write
// var shape = new Shape(type);
// because it might only return Shape, but not
Circle.
```

Proceedings 28th European Conference on Modelling and Simulation ©ECMS Flaminio Squazzoni, Fabio Baronio, Claudia Archetti, Marco Castellani (Editors)
ISBN: 978-0-9564944-8-1 / ISBN: 978-0-9564944-9-8 (CD)

In [5] one can read about still more dangerous creation scenarios.

> *When the knowledge for creating an object is spread out across numerous classes, you have creation sprawl: the placement of creational responsibilities in classes that ought not to be playing any role in an object's creation.*

Therefore, in the following chapter, it is demonstrated how to manage all types of the aforementioned situations, in a more elegant manner - a way in which to omit logical comparisons and the need to possess knowledge of what the derived class type is. In reality, the only necessity is merely to create a derived class instance while having only some parameterization knowledge. Additionally, all the creation logic will be situated in a single location within the code.

FACTORY

An object factory is the simplest manner of solving the aforementioned problem. As mentioned in [1]

> *A Factory pattern is one that returns an instance of one of several possible classes depending on the data provided to it.*

A factory constructs objects of well-known types. Using the factory, the construction logic mentioned in the previous chapter would be much simpler.

```
var type = ShapeType.Circle;
var color = Color.Blue;
var shape = Shape.Create(type, color);
```

Now in the factory construction method, the logical comparisons are undertaken bearing in mind the need to know what the derived class types are. The factory method code would look similar to the following.

```
public Shape Create(ShapeType type, Color color)
{
  var shape = null;
  if (type == ShapeType.Circle)
    shape = new Circle(color);
  else
    ; // TODO: Handle other shape types here.
  return shape;
}
```

The factory method may be static, but that is not a given. It is usually static if placed in a class of which the method produces instances. If not, the class containing the given factory method usually implements an interface defining the factory method. Despite the applied approach, it is better than in the previous example as the object construction code is encapsulated in a single method body. However, in reality this is the solitary advantage of this approach.

It is worth mentioning is that in a factory design pattern, constructors of types returned by the factory are sometimes intentionally not made publicly available. In this case, the factory method acts as a gateway for creating objects of a certain type. Construction logic is not divided into several classes and is thus much easier to maintain.

However, when the factory method supports an ever-increasing number of creational options because of growing business requirements, factory methods start producing various, only partially similar object sets, so-called object families.

For example, when not only the color of the shape is important, but also its size, dimensionality, and for 3D shapes the density and friction, there would be a resultant factory method constituting numerous arguments; of which some would be useful for all families while others would be used only for a single family. As a consequence, most arguments would have null value, and few would have a value assigned at the same time.

```
var ct = ShapeType.Circle;
var st = ShapeType.Sphere;
var density = 0,7;
// Variables can be of various types,
var friction = 0,1;
// also floating point numbers.
// Use factory methods to create objects.
var circle = Shape.Create
  (ct, color, null, null);
var sphere = Shape.Create
  (st, color, density, friction);
```

However, the factory pattern is commonly overused, if not understood correctly, as described in [2].

> *I've seen numerous systems in which the Factory pattern was overused. For example, if every object in a system is created by using a Factory, instead of direct instantiation (e.g., new StringNode(..)),the system probably has an overabundance of Factories.*

In the given example, the actual goal was to have different factories produce 2D and 3D objects. It is possible in this situation to have factory methods with different signatures. In this example a 3D factory will have the same base arguments as a 2D factory, but additionally will add its own arguments.

```
var f2d = new TwoDimFactory();
var f3d = new ThreeDimFactory();
var ct = ShapeType.Circle;
var st = ShapeType.Sphere;
var density = 0,7; var friction = 0,1;
var circle = f2d.Create(ct, color);
var sphere = f3d.Create(st, color, density,
friction);
```

Although the proposed code gives us a straightforward implementation process, 2D and 3D factories' codes become separated. Let us observe that factories have the same arguments in part; with that knowledge in mind, a more appropriate solution may arise.

ABSTRACT FACTORY

The abstract factory patterns come to the rescue here. The difference is that there is an abstract class with the factory

method as yet not implemented, but said to produce some type of objects. Shapes will be the example given in this case. The abstract factory is said to produce families of related objects, and the concrete family creation method is implemented by the concrete factory. In [2] an explanation is provided.

> *If the creation logic inside a Factory becomes too complex, perhaps due to supporting too many creation options, it may make sense to evolve it into an Abstract Factory. Once that's done, clients can configure a system to use a particular Concrete Factory (i.e., a concrete implementation of an Abstract Factory) or let the system use a default Concrete Factory.*

In [6] the authors describe the abstract factory pattern using a straightforward example with two distinct factory methods so as to better understand the difference.

> *If the abstract factory has two methods CreateProductA and CreateProductB, than one subclass of factory (Factory1) will create ProductA1 and ProductB1, and the other subclass (Factory2) will create ProductA2 and ProductB2 because the factory always produces families of related products.*

As the abstract factory defines the factory method signature, all concrete factories must maintain compatibility. In this situation, literals such as connection strings in database development, simple object arrays, key-value dictionaries or dynamic objects are used. Let us demonstrate an example using the dynamic objects approach.

```
var f2d = new TwoDimFactory();
var f3d = new ThreeDimFactory();
var ct = ShapeType.Circle;
// Sample 3D shapes listed in figure 1 below.
var st = ShapeType.Sphere;
var circle = f2d.Create(ct,
  new {Color = color});
var sphere = f3d.Create(st, new {Color = color,
  Density = 0.7, Friction = 0.1});
```

Thus, the given solution has evolved into an abstract factory. Different concrete factories work in rather different ways while maintaining the abstract factory method signature. The natural thing is that the difference lies in the handling of the arguments. If a concrete factory receives an argument that it does not understand, it may neglect it or consider it to be an error, throwing exception, what for user means interrupting program execution. In given example, a 2D factory wishes to have a flat shape and color passed to the factory method. The 3D factory, on the other hand, wants to receive a 3D shape of type, color, density and friction.

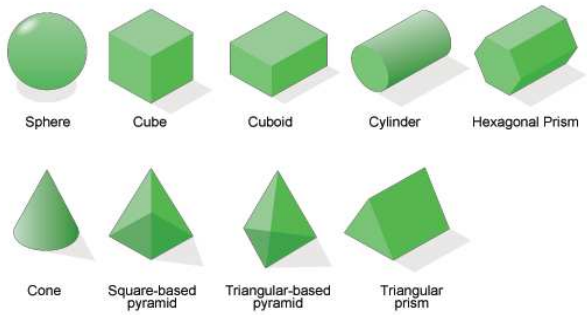

Fig. 1 Sample non-textured 3D shapes

In the example provided, variable information was passed to the factory method using the last argument. This is not obligatory in the case of abstract factories. Usually they have fixed argument vectors and their internal processes are the only factor that differentiates one concrete factory from another, much in the same way that Italian vegetable soup differs from Croatian despite being composed of identical ingredients.

The example given using shapes is extremely simple and does not usually cover the real world system requirements. Therefore one should focus on the 3D factory and assume that he or she wants to change the simple information like shape color to more complex one like shape fixture. To demonstrate the nature of the problem, a fixture will be the complex structure containing information about the texture (image) covering the shape and its luminescence. Let us also assume fixtures are singletons and each fixture points to all shape instances that use this fixture. Although this task seems to be nothing more than that which has been previously mentioned, in the software development industry this topic is covered by a special variation of factory pattern.

COMPLEX FACTORY

Creating objects with an advanced structure is covered by the complex factory design pattern. This type of factory may be abstract, though this is not necessarily the case. Most importantly it creates complex structures of objects, hiding the logic that lies behind object graph initialization. In this section, therefore, the factory function will be described in detail.

```
// Simplified notation
// ReturnedType ClassType.MethodName(Parameters)
// {} is used to denote static methods.
public Shape ThreeDimFactory.Create(type,
 density, friction, texture, luminescence)
{
  var fixture = null;
  // Fixture.All holds all Fixture
  // instances. Check if instance with given
  // parameters was already created or not.
  // If not, create one.
  if (Fixture.All.Contains(texture,
      luminescence))
  {
    fixture = Fixture.All.Get
              (texture, luminescence);
  }
  else
```

```
  {
    fixture = new Fixture
                  (texture, luminescence);
    Fixture.All.Add(fixture);
  }
  var shape = null;
  if (type == ShapeType.Sphere)
    shape = new Sphere();
  else
    ; // TODO: Handle other shape types here.
  shape.Density = density;
  shape.Friction = friction;
  shape.Fixture = fixture;
  fixture.Shapes.Add(shape);
  return shape;
}
```

Complicated logic has been enclosed here into a single method body for the 3D shape factory. All creations, calculations and object manipulations are done exactly here.

DYNAMIC FACTORY

In this chapter a new design pattern is proposed: The Dynamic Factory. Key ideas about this pattern are explained first. Sample implementation is also provided.

Just in Time compilation

As computer programming languages evolved and virtualized high abstraction environments were created, a need arose to dynamically compile parts of algorithms just before execution. This somewhat lazy method of executable code production was called JIT (Just In-Time) compilation, and also labelled "code jitting" by developers. Various technologies implemented it in different forms. Popular approaches in this area were class and method level compilation types. Among these, the more granular compilation proved to be more useful.

In further considerations, using virtualized runtime environments, so-called virtual machines, will be assumed. This is the key issue while planning dynamic factory implementation in one's algorithms.

Code templates, static code and runtime types

For further analysis, one has to investigate the purposes for which JIT is utilised. One of many places it has proven to be useful was in template type and method production. When template types or methods are defined, they usually reside in its code base file as parameterized code blocks, useless until the template parameter vector is applied. This static template code cannot be executed earlier than the point at which the type becomes concrete in the virtual machine memory. In such a situation it is desirable to have the code base file is as small as possible, but at the same time containing all important information needed for post processing by the Just in Time compilation engine. This way, the static template code is post processed by the Just in Time engine and becomes concrete runtime code in the virtual machine memory. In the case of the template type, it can be instantiated and executed, and is as useful as any other that was non-template type in a code base file. A similar set of circumstances can be viewed in the case of template methods, but on a slightly more granular level.

Dynamic type construction

As Just in Time code compilation proved to be effective, the world became hungry for new methods of runtime type production. This way, dynamic types were created and developers gained the ability to utilize the Just in Time engine to produce type in a way that was previously unknown.

The basic idea is to deliver a way of telling runtime environment to produce a runtime type with given name, which extends the desired base type, implements certain interfaces, and has exactly defined constructors, methods and properties. Although this idea seems rather difficult to cover logically, it turns out the implementation process is not as difficult as had been previously expected.

Dynamic languages

In this article, dynamic languages and interpreters that execute code line by line will not be discussed in any detail, as they are much too slow to meet real business requirements and exhibit poor syntax checking, if there is any at all. These languages are more applicable for dynamic construction types, although their disadvantages place them outside the sphere of author's interest.

Reflection and emission

When a compiler produces a code base file, it sometimes can prove useful to possess knowledge of how the code works without having the source code itself. This method of browsing outputted files is called reflection, and is usually used for educational purposes.

If a developer does not want anybody to browse his code base files, he uses an obfuscation mechanism to protect them. However, let us assume that the developer will reflect his own files, to learn what assembler instructions and arguments the compiler produces while outputting the code base files. These instructions with arguments are called intermediate code or byte code in environments with a Just in Time-capable virtual machine.

From now on, further investigations are provided using .NET technology and C# language, which is one of many that meet software requirements. Similar implementations can be done in other languages, such as Managed C++, VB.NET, Iron Python, Iron Ruby, Delphi Prism, Oxygene, F-Sharp, J-Sharp or even in other technology such as Java. Single technology is chosen to provide a strict view of the most important factors in implementing a dynamic factory.

It is even suggested that the reader try implementing the pattern on his own, for example in Java. In such a situation, Class Reader, Class Visitor, Annotation Visitor, Field Visitor, Method Visitor, Class Writer, Opcodes and other surrounding built-in types could all prove useful.

To browse non-obfuscated files prepared in .NET technology, one can use Telerik Just Decompile (*), IL Spy (figure 2 below, *), Red Gate Reflector (figure 3 below, *) or Jet Brains Dot Peek.

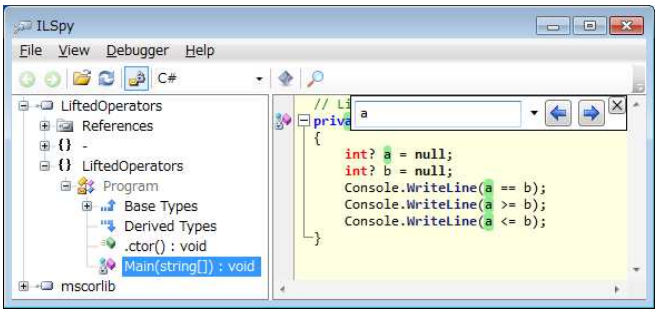

Fig. 2 IL Spy

Tools marked with an asterisk (*) in their current versions have the ability to display intermediate code version of code base files. This is desirable for further investigations. Among interesting tools, Telerik Just Decompile and IL Spy are free decompiling software usable for .NET. Only some versions of Red Gate Reflector are freely available.

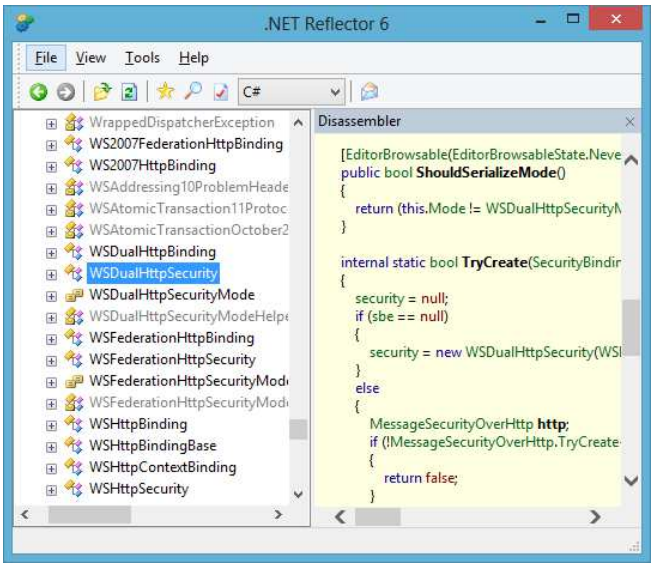

Fig. 3 Red Gate Reflector

An interesting step at this point is to gather knowledge which intermediate code is used to call base methods, pass method arguments, and undertake other low level operations. For sample class with zero-parameter constructors, the code looks relatively simple.

```
namespace SomeNamespace
{
  // Derived type extends base type.
  public class DerivedType : BaseType
  {
    // Derived type constructor calls
    // the base one.
    public DerivedType() : base() { }
  }
}
```

Although using any reflection tool generates intermediate code, it appears to be a bit more complicated than c#.

```
  .class public auto ansi beforefieldinit
DerivedType
      extends SomeNamespace.BaseType
  {
    .method public hidebysig specialname
rtspecialname
    instance void .ctor() cil managed
    {
      .maxstack 8
      L_0000: ldarg.0
      L_0001: call instance void
        SomeNamespace.Base::.ctor()
      L_0006: nop
      L_0007: nop
      L_0008: nop
      L_0009: ret
    }
  }
```

Code emission is the process of generating intermediate code instruction by instruction, just as the compiler had done for the sample code in C# given previously. Code emission allows for the creation of method bodies, constructors, indexers, properties getter and setter logic, indeed everything supported by the given technology.

Implementing Dynamic Factory

Some approaches to building a Dynamic Factory use built-in dynamic variable type. In .NET the type is called ExpandoObject. The sample factory provided below shows how dynamic objects with tracked properties are constructed.

```
public class Factory
{
  public static dynamic CreateTrackedObject()
  {
    // Class ExpandoObject represents dynamic
    // objects in .NET. It is also referenced
    // using keyword "dynamic".
    dynamic result = new ExpandoObject();
    // Tell the runtime engine to track
    // property changes.
    ((INotifyPropertyChanged)result)
      .PropertyChanged +=
        new PropertyChangedEventHandler
          (Factory.HandlePropertyChanged);
    // Property change will be intercepted.
    result.Name = "John Smith";
    return result; // Return dynamic object.
  }
  // Method to handle property changes,
  private static void HandlePropertyChanged
    (object sender, PropertyChangedEventArgs e)
  {
    // Write the name of changed property.
    Console.WriteLine("{0} has changed.",
      e.PropertyName);
  }
}
```

Somewhat similar possibilities are delivered for Java Script developers, although in Java Script all objects exhibit similar behavior. As mentioned in previous chapters, there is actually an entire group of languages with dynamic syntax support, or languages that can only interpret code line by line.

Although this construction pattern is relatively simple, in that the developer does not have to know or understand intermediate language, there are nonetheless some major disadvantages. There are issues with speed; constructed

objects are volatile; and they can be broken at any point in the process.

```
dynamic employee = new ExpandoObject();
// Declare and initialize new property.
employee.Name = "John Smith";
// Detach property from its parent object.
((IDictionary<String, Object>)employee)
  .Remove("Name");
```

In the example above, the dynamic object gains a new property, in that it is assigned a value after creation, and finally the dynamic object loses the new property and its value in a single shot. There is no bake method, to say that dynamic object implementation is final, and the object should be made unbreakable from this point.

A much better approach is provided by utilizing code emission, which means producing intermediate code on the fly. In this kind of factory implementation, the developer needs to know what kind of intermediate code the compiler is producing. To gather this knowledge, reflection is used, as explained previously.

In order to demonstrate dynamic factory strength, new runtime type that derives the given base type, has exactly the same public constructors, and appends few desired attributes to the class definition, will be constructed. It is worth mentioning that this is merely an example of the power provided to the developer; pure proof of the concept, although taken from the real-world application.

First of all, the code emission infrastructure must be configured.

```
IEnumerable<CustomAttributeBuilder> cubs = null;
Type baseType = null;
string fullName = null;
// TODO: Fill in custom attribute requirements,
// base type and dynamic type full name from
// factory method arguments.
AssemblyName an = new AssemblyName();
an.Name = "DynamicAssembly";
// Current application domain will load
// new type.
AppDomain ad = Thread.GetDomain();
// Define in-memory dynamic assembly.
AssemblyBuilder ab = ad.DefineDynamicAssembly
  (an, AssemblyBuilderAccess.Run);
ModuleBuilder mb = ab.DefineDynamicModule
  ("DynamicModule");
// Define new type deriving from base one.
TypeBuilder tb = mb.DefineType(fullName,
TypeAttributes.Public, baseType);
ConstructorInfo[] cis = baseType.GetConstructors
  (BindingFlags.Public | BindingFlags.Instance);
```

Afterwards, the constructors' intermediate code must be emitted. For simplicity it is assumed that derived classes only call base constructors and does nothing more. Knowledge from the chapter on reflection will be utilized herein.

```
// Iterate through base type public
// constructors.
foreach (ConstructorInfo ci in cis)
{
  // Gather constructor argument type
  //collection.
  Type[]constructorArgumentTypes=
    ci.GetParameters().Select
   (pi => pi.ParameterType).ToArray();
  // Define public constructor with the same
  // arguments.
  ConstructorBuilder cb = tb.DefineConstructor
   (MethodAttributes.Public,
    CallingConventions.Standard,
    constructorArgumentTypes);
  ILGenerator il  = cb.GetILGenerator();
  // Emit intermediate language line by line.
  il.Emit(OpCodes.Ldarg_0);
  int parameters = ci.GetParameters().Count();
  // Load constructor arguments onto the stack.
  if (parameters >= 1) il.Emit(OpCodes.Ldarg_1);
  if (parameters >= 2) il.Emit(OpCodes.Ldarg_2);
  if (parameters >= 3) il.Emit(OpCodes.Ldarg_3);
  for (byte i = 4; i <= parameters; i++)
    il.Emit(OpCodes.Ldarg_S, i);
  // Call the base constructor.
  il.Emit(OpCodes.Call, ci);
  il.Emit(OpCodes.Nop);
  il.Emit(OpCodes.Nop);
  il.Emit(OpCodes.Nop);
  il.Emit(OpCodes.Ret); // Return the derived
// type instance.
}
```

What remains is to append the desired attributes to derived class definition and bake the new runtime type. This type has all the required features, is extremely quick and non-volatile.

```
if (cubs != null)
  foreach (CustomAttributeBuilder cub in cubs)
    tb.SetCustomAttribute(cub);
derivedType = tb.CreateType();
```

Note that merely compiling the code does not automatically mean that it will work as expected. Dynamic factories should be rigorously tested before use in business environments.

In order to further improve performance, new dynamic types should be cached using concrete vector of parameterization arguments. In the given example it would be the vector <Base Type, Full Name, Type Attributes>. Although it is important to note that two types with the same Full Name of the type cannot exist in one application domain. For reasons of simplicity, the type Full Name will be used as a key in the cache dictionary. The dynamic factory will be able to produce instances after the dynamic type is retrieved from cache or created and baked on the fly. When creating instances, the desired constructor will be automatically best fitted investigating constructors argument types, as if instances of base type were being created.

```
public static Dictionary<string, Type> Cache =
     new Dictionary<string, Type>();
public static T Create<T>( string fullName,
  IEnumerable<CustomAttributeBuilder> cubs,
  params object[] constructorArguments)
{
  Type baseType = typeof(T);
  Type derivedType = null;
  // Check if cache dictionary contains desired
type.
  if (Cache.ContainsKey(fullName))
    derivedType = Cache[fullName];
```

```
    else
    {
      // TODO: Use mentioned logic to create new
      // type. New type is baked and therefore
      // non-volatile.
      // Store the new type in cache dictionary.
    }
    // Create instance using desired constructor.
    T result = Activator.CreateInstance
      (derivedType, constructorArguments);
    return result;
  }
```

This kind of dynamic factory implementation was utilized in two business scenarios by the author. In both cases, modified versions were used so as to fit specific business needs, while maintaining the core principles as discussed. The most interesting case was the need to create transactional objects that implemented certain behaviors. A dynamic factory was used to provide object types with the desired structure and functionalities, and change tracking/recording code injected into the implemented mechanisms of the instances of constructed type. Tracking functionalities were then utilized to implement transactional behavior. While recording changes to the state of the objects, they offered the ability to roll-back all actions up to a certain point in time – namely the moment when all previous transactions have been successfully committed. In both business scenarios mentioned by the author, the dynamic factory proved to be extremely useful.

Inversion of Control

Last but not least, it is worth mentioning what Inversion of Control means. Basically, it is a method of constructing and utilizing objects. In this technique the most important factor is that object coupling is bound at the time of code execution. Using object reference analysis in code, it cannot be predicted which objects will cooperate. In [7] the authors write:

> *The function of IoC is transferring the control from code to external container. [..] Relationship between the components is specified by the container in the runtime.*

An Inversion of Control container is a design pattern used to localize objects that should be used in certain situations. It may be utilised in conjunction with patterns that construct instances of new objects if they do not yet exist in the container.

CONCLUSION

Let us summarize the collected knowledge and think over the final outcome of conducted reasoning.

Constructor

Constructors should be always used whenever possible, assuming that no complicated logic lies behind object construction. Constructors also always return the type they reside in, with no possibility of producing derived type instances.

Factory

A factory is commonly used when there is a need to construct objects according to the environment state. This state is passed on to the factory as a set of variables. Depending on the provided argument values, the factory produces the desired object. Factories can also produce derived type instances. All the construction logic of a factory resides in a single location. This design pattern is commonly known to be overused.

Abstract Factory

This variation of factory proved to be useful when creating object families when objects are similar in some aspects, but differ in others. A situation arises when different factories have factory methods with the same signature, but which work in a different manner. Additional logic for creation is passed on using strings, array, dictionaries and dynamic objects.

Complex Factory

A complex factory is a means of creating object graphs. It covers all constructors and object connections logic, and is used in difficult projects to organize structure and make code easier to maintain and enhance. A complex factory is in some aspects similar to the facade design pattern, which will be described in further detail. However, a complex factory does not have to conceal any disadvantages of code. In [1] we can read

> *Facade is a way of hiding a complex system inside a simpler interface. [..]This simplification may in some cases reduce the flexibility of the underlying classes, but usually provides all the function needed for all but the most sophisticated users.*

Dynamic Factory

This kind of factory is used when a developer does not know exactly what the restrictions for object types or functionalities will be. This knowledge is gathered and utilized at the time of program execution. It is the most advanced factory design pattern variation, which creates new object types with only what is required in a certain situation, and instantiates them on the fly. It may be implemented in two ways. The first is when the factory produces fully dynamic but volatile objects; the second, when the factory produces brand new first-baked and non-volatile type instances. The second one is more difficult for the developer, requiring intermediate language knowledge for coding, and the source code is less readable for humans. Although the constructed objects are without the disadvantages of fully dynamic objects, they have therefore proved to be adequate in business environment edge-cutting software constructions.

Summary

Depending on the situation, each factory variation design pattern is considered useful. The more difficult the situation

which is encountered, the stronger the tools which are used. Among them is the proposed dynamic factory pattern with intermediate code emission which gives us the widest range of possibilities. It is more difficult to implement and technology prerequisites are high, met only by modern languages. However, this is the type of technology which will be used in large solutions in upcoming years, and new projects can benefit from this design pattern.

Using a factory pattern always includes the requirement to access factory methods. They are usually implemented as either static or interface. Sometimes objects containing factory methods are implemented as singletons. Another practiced approach is using an Inversion of Control container with a finder method, used to localize the right factory in a given situation. In this case, the factory is implemented inside the Inversion of Control container, as its creational mechanism. It produces objects in concrete situations using the implemented set of rules, which will be described more precisely in an upcoming chapter. Of course, if chosen, the factory used by Inversion of Control may be any kind of factory described in the article.

APPENDIX

Please note that there exist numerous misunderstandings about dynamic factory in literature. For example [3] describes the Inversion of Control container automatic initialization mechanism, using type attributes to localize types that should be instantiated. Although Inversion of Control container with such a mechanism instantiates new objects, it should not be mistaken for any of kind factory pattern. There might be a factory hidden inside an Inversion of Control container, as mentioned previously. It may have some creation rules for certain desired situations. But that does not make such a factory dynamic in any aspect. In [4] on the other hand, the authors propose a factory that reads static types to be instantiated from code base files indicated in XML files or data bases. It is worth mentioning that simple factory variation proposed by the authors has been used in business products like Microsoft Visual Studio or Microsoft Windows Explorer for many years. Of course, although with a substantially different meaning to that in [3], [4] also has nothing to do with dynamic creating new types.

Builder and Complex Factory design patterns are also commonly confused. The builder pattern mentions nothing about the complexity of objects. The most straightforward example is the String Builder commonly known from languages like C-Sharp, C++, Delphi, Java and Java Script. On the other hand, while hierarchically nesting many builders into others, one can obtain an organized structure for creating complex objects. Enclosing this complex creational structure in one easy-to-handle factory method means creating a Complex Factory. Although creating a complex factory does not mean that numerous builders are required, but rather relates to building a facade for the creation of a complex structure.

Further investigations are planned for new constructions and applications of dynamic factory design pattern. Research will be also conducted in order to support the dynamic creation of new static types using lambda expressions, and anonymous types, methods and delegates. This will partially alleviate the need to code factory methods in pure intermediate language.

ACKNOWLEDGMENT

Special thanks to my students who directed me to cover factory design patterns in more detail – as such patterns are well documented but also rather superficially understood. Otherwise it may have not been quite so clear that covering the deficiencies in the literature was a worthwhile exercise.

AUTHOR BIOGRAPHIES

DAWID R. IRENO was born in Kraków, Poland and attended the Jagiellonian University, where he majored in computer science and earned his master's degree in 2007. During the following years, he worked on business projects for Roche, Microsoft, Comarch and other large companies in various cities around Poland, utilizing .NET, ASP.NET, JavaScript, ASP.NET MVC, Ext.NET, PowerShell, SharePoint, SQL, LINQ, WCF, WPF and Silverlight technologies. In 2012 he began his Ph.D. studies at the Jagiellonian University, where he is researching stream databases. His webpage can be found at *http://www.powershell.pl/.*

REFERENCES

[1] James W Cooper "Java Design Patterns" by Addison-Wesley.
[2] Joshua Kerievsky "Refactoring To Patterns" by Addison-Wesley.
[3] Romi Kovacs "Design Patterns: Creating Dynamic Factories in .NET Using Reflection" from MSDN Magazine, March 2003.
[4] León Welicki, Joseph W. Yoder, Rebecca Wirfs-Brock "The Dynamic Factory Pattern", Proceedings of the 15th Conference on Pattern Languages of Programs, 2008.
[5] Joshua Kerievsky "Refactoring to Patterns" by Addison-Wesley, 2004.
[6] A. A. Nykonenko "Using design patterns in computer linguistics: Creational patterns. Part I: Abstract Factory and Builder", Cybernetics and Systems Analysis, Vol. 48, No. 1, January, 2012, pages 138-145, by Springer Science+Business Media, Inc.
[7] Ke Ju and Jiang Bo 'Applying IoC and AOP to the Architecture of Reflective Middleware', 2007 IFIP International Conference on Network and Parallel Computing - Workshops, pages 903 to 908.

COPYRIGHTS