The Dynamic Factory Pattern

León Welicki
ONO (Cableuropa S.A.)
lwelicki@acm.org

Joseph W. Yoder
The Refactory, Inc.
joe@refactory.com

Rebecca Wirfs-Brock
Wirfs-Brock Associates
rebecca@wirfs-brock.com

The DYNAMIC FACTORY pattern describes how to create a factory that allows the creation of unanticipated products derived from the same abstraction by storing the information about their concrete type in external metadata.

Context A software system uses a framework (a set of classes that embodies an abstract design for solutions to a family of related problems, and supports reuse at a larger granularity than classes [JF98]), where collaborations between high-level abstractions determine the execution flow.

Individual solutions are created by extending existing classes and combining these extensions with other existing classes [Foote88]. The configuration of these combinations of implementers of abstractions should be done without the need of modifying the application.

New implementations of the established abstractions can be created by as long as they conform to established protocols. The system should be able to inject these new abstractions into the framework without the need to modify its core. These new abstractions can be created after the system has been delivered.

Example A workflow system has a rule evaluation module. Each rule implements a well defined interface, and is injected into a container that evaluates it. The rules can be simple or composite (using the COMPOSITE [GoF95] and INTERPRETER [GoF95] patterns) allowing the creation of complex expressions by composing finer-grained elements.

The creation of the rules is delegated to a factory that has a standard interface for creating instances of the abstractions. The clients of the rules request an instance of the rule and the factory provides it.

The workflow system vendor supplies a fixed set of rules. New rules could be added by simply implementing the rule interface. The problem comes at rule instantiation, since the factories that contain the logic for creating instances of the rules may need to be modified to support new types of rules.

Problem **How can we define an interface for creating objects that implement a given contract without tying it to concrete implementations of these contracts?**

Forces

- *Flexibility*. The implementers of the products should be easily modifiable, even when the system is running, allowing the injection of new product types into an existing system.
- *Extensibility / Evolvability*. New product types should be easily added without requiring neither a new factory class nor modifying any existing one.
- *Controlled Evolution*: users can create new types of products conforming to the product interface, but providing unanticipated behavior or features.
- *Agility*. New types of products should added to the system in a quick and agile manner, avoiding reworking of a factory class any time a new concrete product is created.
- *Simplicity*. The client interface should be simple, hiding from the client the complex details of dynamic product creation.

Solution **Establish an interface for creating objects that implement a specific contract, and store the concrete type information of the instances to be created in metadata**.

The DYNAMIC FACTORY is a generalized implemetation that is responsible for creating instances (a single well-known location for creating instances of a general type, similar to a REGISTRY [Fowler03]), while not making any a priori decisions about the concrete types of those instances. Some default types may be provided (in the form of base cases or default implementations) but a hook for extensibility must be always provided.

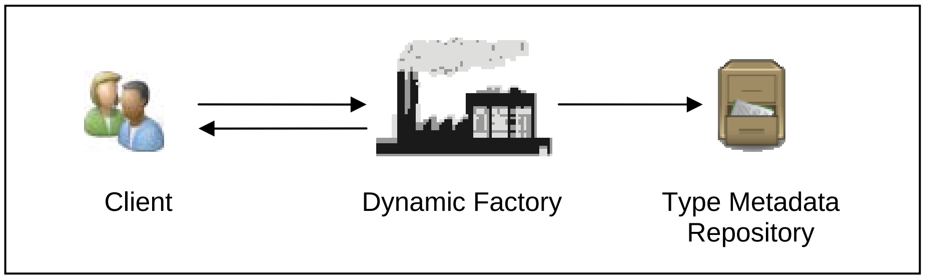


Figure 1 – The Dynamic Factory

The dynamic factory alone is not enough to create the instances of the concrete products: the factory provides the "production engine", but the type repository metadata provides the “raw material”.

The information about the concrete types is persisted in secondary memory storage (xml file, database, plain file, etc.). The concrete type information of an entity may contain the fully qualified name of the type and the physical container where the type is contained (allowing the creation of instances using reflection). The information about the concrete type information may vary according with the implementation platform.

Adding a new implementation of a Product interface to the system is very simple: it only requires implementing the abstraction (which is likely to be implemented just once to support many different product instantiations), and adding a line in the configuration file of the factory indicating how to load it (for example, the assembly and full qualified name of the concrete product in the case of a .NET application).

This follows the principle “p*ut abstractions in code and details in metadata*” [HT00]. The DYNAMIC FACTORY pattern establishes an interface for creating instances of abstractions, but puts the information about the concrete implementation of a product abstraction in metadata.

Structure

The following participants form the structure of the DYNAMIC FACTORY pattern:

- A *DynamicFactory* is a class that creates instances of other classes using metadata at runtime to determine the concrete type of the class to be created.

- A *MetadataReader* reads type metadata from a configuration repository and delivers it to the *DynamicFactory* in the form of an instance of *ProductTypeInfo*.

- *ProductTypeInfo* contains the type metadata about a concrete product. The instance definitions are fairly constant and thus rarely change.

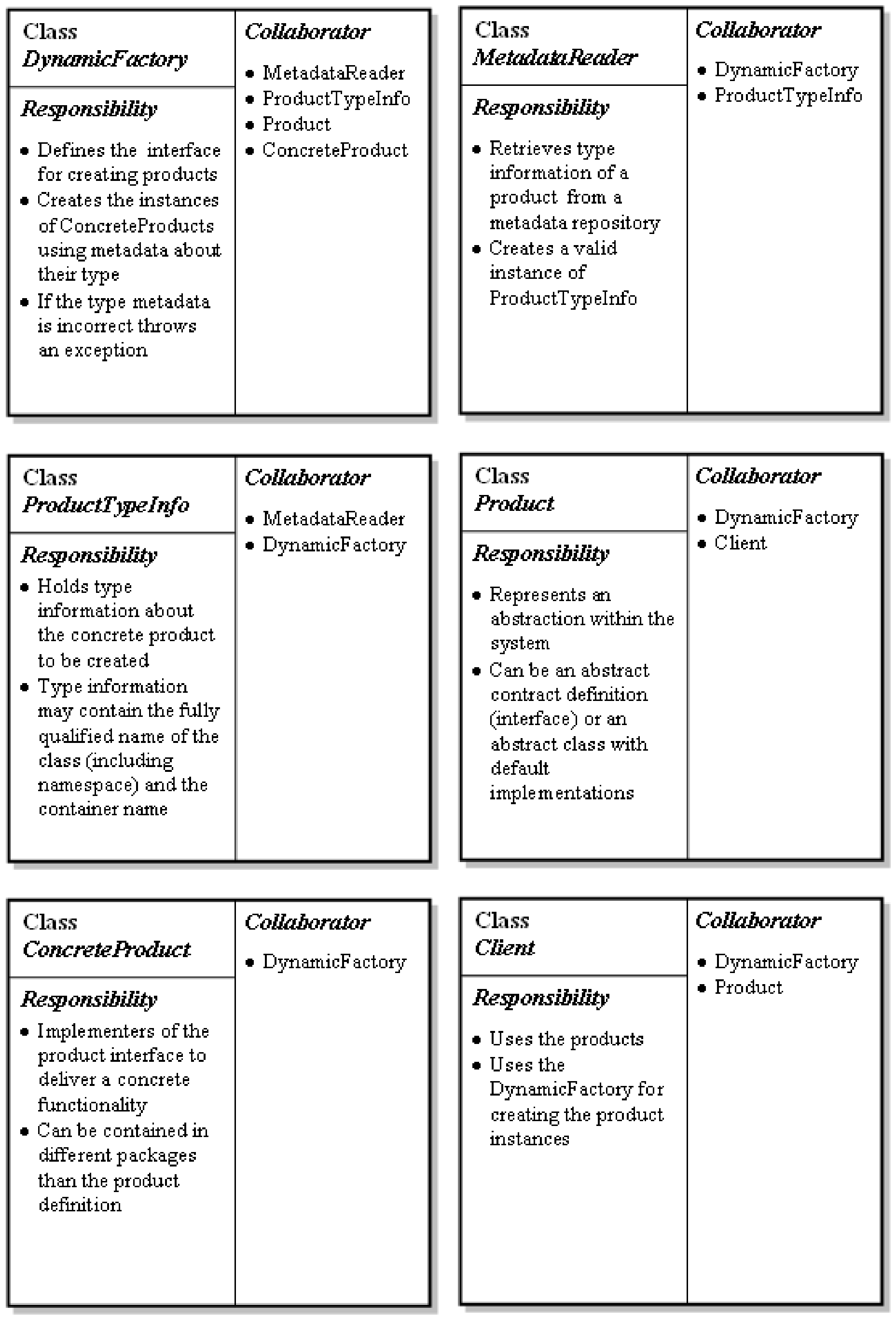
- A *Product* represents a general abstraction in a software system. This abstraction can often be in the form of an interface, an abstract class, or a concrete class with virtual methods.

- A *ConcreteProduct* is an implementation of the Product abstraction that provides concrete functionality or semantics within a software system.

- A *Client* uses instances of *ConcreteProducts* through the

Product abstraction (abstract coupling[GoF95]). The instances of the products are created using the *DynamicFactory*.

The following CRC cards show how the participants interact with each other:



Class *DynamicFactory*	*Collaborator*
Responsibility • Defines the interface for creating products • Creates the instances of ConcreteProducts using metadata about their type • If the type metadata is incorrect throws an exception	• MetadataReader • ProductTypeInfo • Product • ConcreteProduct

Class *MetadataReader*	*Collaborator*
Responsibility • Retrieves type information of a product from a metadata repository • Creates a valid instance of ProductTypeInfo	• DynamicFactory • ProductTypeInfo

Class *ProductTypeInfo*	*Collaborator*
Responsibility • Holds type information about the concrete product to be created • Type information may contain the fully qualified name of the class (including namespace) and the container name	• MetadataReader • DynamicFactory

Class *Product*	*Collaborator*
Responsibility • Represents an abstraction within the system • Can be an abstract contract definition (interface) or an abstract class with default implementations	• DynamicFactory • Client

Class *ConcreteProduct*	*Collaborator*
Responsibility • Implementers of the product interface to deliver a concrete functionality • Can be contained in different packages than the product definition	• DynamicFactory

Class *Client*	*Collaborator*
Responsibility • Uses the products • Uses the DynamicFactory for creating the product instances	• DynamicFactory • Product

Figure 2 – CRC Diagram

The following class diagram illustrates the structure of the Dynamic

Factory pattern.

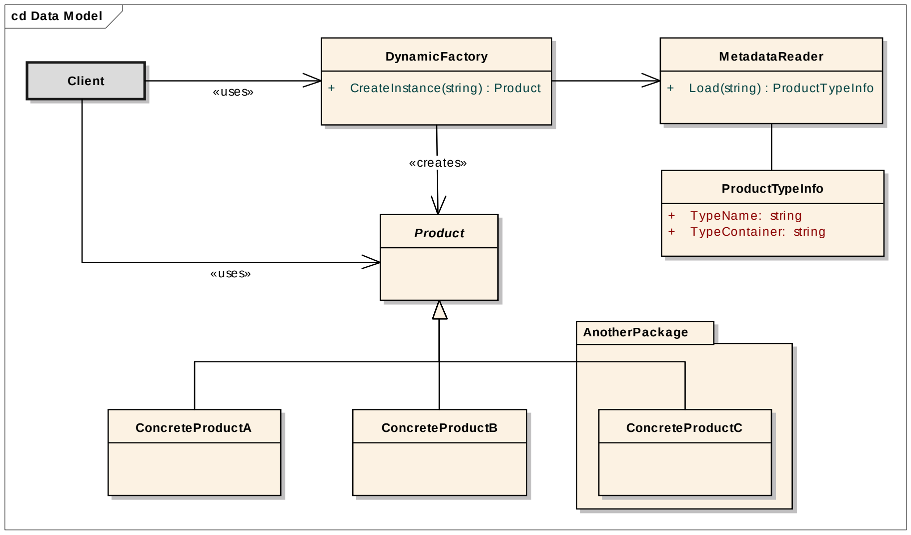


Figure 3 - Dynamic Factory Class Diagram

The `Product` participant defines the general abstraction of the products to be created by the factory. It can be implemented using an interface, abstract class, or any similar mechanism depending on the target implementation language.

The `DynamicFactory` creates instances of realizations of the product abstraction. In the simplest case, a `DynamicFactory` creates instances of a single type of `Product`. However, this can be extended using generics [GenJava], [GenNet] to create a generic dynamic factory for any kind of product. The ABSTRACT DYNAMIC FACTORY, variant of this paper (listed later in the Variants section) also creates instances of multiple types of products.

The type information metadata of the implementations of the `Product` (the `ConcreteProducts`) is stored in a type metadata repository (xml file, relational database, plain text file, or any suitable mechanism for storing configuration data). The `MetadataReader` reads this information from the repository and returns the type information as an instance of `ProductTypeInfo`. This helps to decouple the `DynamicFactory` and the type metadata repository, since it accesses it using the `Load` method `MetadataReader` without regard of the underlying storage support of the type metadata repository (it doesn't have to do any distinction whether is an xml file or a relational database).

Consequences

The DYNAMIC FACTORY adds flexibility and better modularity to an application by abstracting the creation process of instances, exposing it through a well-known entity in the system, and storing all the details about the concrete implementers of the products in metadata. This removes the creation code from the application, hiding it in the factory and the creation metadata.

Additionally, it makes easier to introduce new implementers of a contract in a system, since the location implementation details are encoded in metadata.

The process of dynamically creating the entities is complex, but the proposed solution in this pattern hides all this complexity from the end users. Taking this idea to the extreme, the *DynamicFactory* can be a well-known static class, easily accessible with very simple semantics.

This helps to put in practice the principle "*put abstractions in code and details in metadata*" [HT00] (in this case metadata refers to configuration information). This also builds on the "Dependency Inversion Principle" and "Open Closed Principle" [Martin02].

Creating the instances dynamically can bring a notorious performance overhead. This can be reduced by applying the CACHING pattern combined with some of the other resource management pattern presented in [POSA3].

There are several **benefits** of this pattern:

- *Extensibility*. Adding new concrete products is a relatively simple task consisting of two steps: implementing the concrete product and adding the type information of this new implementer to the metadata repository.

- *Flexibility*. Existing concrete products can be modified or removed and new products can be added dynamically. This can be done at run-time, since the creation of instances is done dynamically using REFLECTION [POSA1] (or any other similar technique available).

- *Configurability*. We can change the behavior of an application by just changing its configuration information. This can be done without the need to change any source code (just change the descriptive information about the type in the metadata repository) or to restart the application (if caching is not used – if caching is used the cache will need to be flushed).

- *Agility*. New concrete products can be, added quickly having a guiding procedure that leverages existing architectural decisions.

There are several **liabilities** of using this pattern:

- *Run-time errors*. Run-time errors may appear when using this pattern. At compile time a good test suite can prevent them, but when adding or modifying type metadata at runtime unexpected errors can occur (for example, very simple typos can lead to creation errors). A good error handling strategy should be established at the architectural level to cope with these kinds of errors. In some cases, default implementations can be provided when the type metadata is incorrect (using a variant of the CHAIN OF RESPONSIBILITY pattern [GoF95])

- *Complexity*. The solution hides the complexity from the clients, but is still complex. The internals of the factory are more complex than having a "case" statement or directly invoking a constructor. This complexity increases significantly when CACHING is added (for a discussion on this liabilities see [POSA03] and [Welicki06]).

- *Possible "over-engineering"*. If new types are not going to be added or current implementations are never going to be modified or switched at runtime, using this pattern is a clear over-effort that should be avoided. A good way of avoiding this situation is starting with simpler options (like a static class, or [GoF95] creational patterns, or just using a constructor). You can then evolve to use of a Dynamic Factory when it is warranted (using an evolutionary redesign approach like in [Kerievsky03]).

Example Resolved

All the rules in the workflow system are derived from a basic abstraction (the `Rule` interface). There is hierarchy of different rule types, but all the hierarchy shares a common ancestor (the `Rule` interface).

To remove all concrete type information from the code, a DYNAMIC FACTORY for creating `Rule` implementations is created.

A metadata format for specifying the types of the rules is established. This format includes an identifier for the rule and its type information (the container and class name of the implementer of the rule). Moreover, the format supports composition (following the COMPOSITE [GoF95] pattern) and the instances of the compositions are loaded dynamically at runtime by a combination of the BUILDER [GoF95], INTERPRETER [GoF95], and DYNAMIC FACTORY patterns.

By doing this we remove the concrete rule types from the source code, allowing for change and extension of the workflow system through the creation of new rules (implementations of the Rule interface) that are added to the type metadata repository.

Sample Code

In this section, we will present a simple implementation of this pattern as presented previously in figure 3. Our sample implementation is written in .NET using C#.

Canonical implementation: creating instances of a single product

The next code snippet shows the product interface. Usually, the implementation of this patterns starts with the definition of the `Product` abstraction (which can be an interface, an abstract class, or any similar mechanism depending on the implementation language).

```
public interface IProduct
{
    void DoSomething();
}
```

This abstraction should be implemented by all the `ConcreteProducts` to be created by the `DynamicFactory`. The implementers of the abstraction may not be known at design time or may change at runtime. Therefore, the next step in the implementation of the pattern is to define the metadata format to store the type information for each realization of the formerly defined interface. The next code snippet shows a sample xml file with type information. Each `product` node contains an identifier of the concrete product (`id` attribute) and the type information for dynamically creating the class (`type` attribute).

```
<typeInfo>
  <products>
    <product
         id="product1"
         type="DynamicFactorySample,
               DynamicFactorySample.ConcreteProducts.ProductA"/>
    <product
         id="product2"
         type="DynamicFactorySample,
               DynamicFactorySample.ConcreteProducts.ProductB"/>
    <product
         id="product3"
         type="AnotherAssembly,
               DynamicFactorySample.ConcreteProducts.ProductC"/>
  </products>
</typeInfo>
```

This metadata can also be stored in a relational database, plain files, etc. To hide the storage implementation details to the factory we use the `MetadataReader` and `ProductTypeInfo` classes. The first is responsible for accessing the type metadata repository (the xml file defined above) and the last is a container for the type information of a requested `ConcreteProduct`.

```
public class ProductTypeInfo
{
    private string productTypeCode;
    private string assemblyName;
    private string className;

    public string ProductTypecode
    { get { return this.productTypeCode; } }

    public string AssemblyName
    { get { return this.assemblyName; } }

    public string ClassName
    { get { return this.className; } }

    public ProductTypeInfo(
                string productTypeCode,
                string assemblyName,
                string className)
    {
        this.productTypeCode = productTypeCode;
        this.assemblyName = assemblyName;
        this.className = className;
    }
}

public class MetadataReader
{
    public ProductTypeInfo Load(string typeName)
    {
        XmlDocument doc = new XmlDocument();
        doc.Load(AppSettings["rootPath"]);

        XmlNode node = doc.SelectSingleNode(
          "/typeInfo/products/product[@id='" + typeName + "']");

        if (node == null)
            return null;

        return new
             ProductTypeInfo(
                   typeName,
                   node.Attributes["type"].Value.Split(',')[0],
                   node.Attributes["type"].Value.Split(',')[1]);
    }
}
```

Following, a simple implementation of the `DynamicFactory` class is presented. The `Create` method creates and returns an instance of an implementer of the `IProduct` interface.

```
public static class DynamicFactory
{
    public static IProduct Create(string productTypeCode)
    {
        MetadataReader metadataReader = new MetadataReader();
        ProductTypeInfo typeInfo =
                          metadataReader.Load(productTypeCode);

        ObjectHandle obj =  Activator.CreateInstance(
                                                typeInfo.AssemblyName,
                                                typeInfo.ClassName);
         return (IProduct)obj.Unwrap();
    }
}

public class SampleClient
{
    public void Main()
    {
       IProduct product = DynamicFactory.Create("product1");
       product.DoSomething();

       rule = DynamicFactory.Create("product2");
       product.DoSomething();
    }
}
```

Extending the factory with Generics

Our implementation of the `DynamicFactory` shown above is limited to creating instances of `IProduct` interface. If we want to make it more general, we can use generics, as shown in the piece of code below.

```
public class GenericDynamicFactory<T>
{
    public T Create(string productTypeCode)
    {
        MetadataReader metadataReader = new MetadataReader();
        ProductTypeInfo typeInfo =
                           metadataReader.Load(productTypeCode);

        ObjectHandle obj =  Activator.CreateInstance(
                                        typeInfo.AssemblyName,
                                        typeInfo.ClassName);
        return (T)obj.Unwrap();
    }
}

public class SampleClient
{
    public void Main()
    {
        DynamicFactory<IProduct> dynamicFactory =
                new DynamicFactory<IProduct>();

        IProduct product = dynamicFactory.Create("product1");
        product.DoSomething();

        product = dynamicFactory.Create("product2");
        product.DoSomething ();

        dynamicFactory = new DynamicFactory<IAnotherProduct>();
        product = dynamicFactory.Create("anotherTypeName");
        product.DoSomething ();
    }
}
```

Another static implementation using generics

Below, another implementation using generics is shown. In this case, the `DynamicFactory` is a static class and the creation method [Kerievsky03] is generic [MSCPG], [SJPL].

```
public static class GenericDynamicFactory
{
    public static T Create<T>(string productTypeCode)
    {
        MetadataReader metadataReader = new MetadataReader();
        ProductTypeInfo typeInfo =  metadataReader.Load(
                                            productTypeCode);

        ObjectHandle obj = Activator.CreateInstance(
                                        typeInfo.AssemblyName,
                                        typeInfo.ClassName);
        return (T)obj.Unwrap();
    }
}

public class SampleClient
{
    public void Main()
    {
        IProduct product = DynamicFactory.
                              Create<IProduct>("product1");
        product.Execute();

        IProduct product = DynamicFactory.
                              Create<IOtherProduct>("otherProd");
        product.Execute();
    }
}
```

The implementations shown above are simplified and don't take into account critical issues like exception handling, caching, security, advanced configuration management setups, etc. More sample implementations of this pattern can be found in [VanDeursen06], [SunDevForum], [MK06], and [Kovacs03].

Variants

- **Cached Dynamic Factory**: the dynamic factory can be combined with the CACHING pattern [POSA3] (or the CONFIGURATION DATA CACHING pattern [Welicki06]) to improve run-time efficiency caused by repetitious acquisition of resources. There are two main points where caching can be introduced: the retrieval of the metadata for a type of concrete product (in this case the CONFIGURATION DATA CACHING may be used) or directly caching the concrete products. The first case is very simple to implement, since the *ProductTypeInfo* are often inmutable. The last case is more difficult and is feasible only when the *ConcreteProducts* are stateless [POSA3].

  If a cache is used, an EVICTOR [POSA3] (or similar) may be necessary to unload outdated or not needed instances from memory. In some scenarios the system can become more complex if you have to introduce a synchronization mechanism: a means to verify that the cached information is synchronized with the contents of the type metadata repository. When the type information is updated in the metadata repository, some synchronization mechanism is needed to synchronize the system with the new versions of the type definitions. An easy way to do this would be to simply restart the system or flush the cache.

- **Parameterized Dynamic Factory**: this variation receives information as a parameter that is used to create the instances. There are several options in this case and the parameter can be the type information of the product to be created or an alias to search for it in a type metadata repository (database, file, etc.).

- **Dynamic Abstract Factory**: in this case, the interface is very simple, containing several methods to create instances of concrete products. The type metadata about the concrete type of the instances to be created are stored in a type metadata repository (database, file, etc.). In this case, the *DynamicFactory* establishes an interface for creating a family of products, but the details about the family member types is stored in metadata. Therefore, flexibility and extensibility is not based in static composition and inheritance. Instead, it is achieved by dynamic interpretation of metadata.

- **Adaptive Object-Model Dynamic Factory**: in AOM based architectures [YBJ01; YJ02; FY97; WYWJ07], the DYNAMIC FACTORY can be used to create the instances of the PROPERTIES, ENTITIES, ACCOUNTABILITIES, and RULE OBJECTS (and their corresponding TYPE OBJECTS [JW98]).

Known Uses

- **Microsoft ASP.NET** uses this pattern to configure its extensibility features and its internal working. HttpHandlers and HttpModules are configured using type metadata and created at runtime using this type info. Taking this model further, there is a dynamic factory for the factories (HttpHandlerFactories) that uses the same mechanism.

- **Adaptive Object-Models**. An Adaptive Object-Model is a system that represents user-defined classes, attributes, relationships, and behavior as metadata [YBJ01; YJ02]. The system is a model based on instances rather than classes. Users change the metadata (object model) to reflect changes in the domain. These changes modify the system's behavior. AOM-based architectures use extensively the dynamic creation of its building blocks based on metadata.

- **Rule based systems.** The rules are configured using a VISUAL LANGUAGE [RJ98], where they can be combined to be applied to a wide variety of contexts. Moreover, new rules can be added, deleted, and changed at runtime. To add new rules, usually a general abstraction (e.g. interface or abstract class) must be implemented and its type information must be registered within a type metadata repository.

- **Spring XT Modeling Framework** provides components for helping develop rich domain models and making them collaborate with other application layers without violating Domain Driven Design principles, including the Dynamic Factory Generator that lets you generate factory objects on the fly, providing only the factory interface [SpringModules].

Related Patterns

FACTORY METHOD [GoF95] and ABSTRACT FACTORY can be evolved to DYNAMIC FACTORY. Since both establish an interface for creating products (single in the first or families in the last), they can be evolved to use metadata.

The DYNAMIC FACTORY can use the CACHING pattern [POSA3] to hold the configuration data (XML metadata), a prototypical instance or the instance itself (in case that the product is statelesss).In this case an EVICTOR [POSA3] may be used for eviction of cached instances of concrete products. For example, never accessed or old values may be evicted periodically.

The DYNAMIC FACTORY can be a SINGLETON [GoF95] and can also be a dynamic REGISTRY [Fowler03]

STRATEGY [GoF95] may be used to change the configuration storage access strategy to fetch data (a provider may have several strategies aimed to fetch data from different kinds of repository, e.g., XML, relational database, flat file, etc.).

References

[BR98] Bäumer, D ; D. Riehle. *Product Trader*. Pattern Languages of Program Design 3. Edited by Robert Martin, Dirk Riehle, and Frank Buschmann. Addison-Wesley, 1998.

[Foote88] Foote, Brian. Designing to Facilitate Changes with Object-Oriented Frameworks. MSc Thesis. University of Illinois at Urbana-Champaign. 1988.

[Fowler 03] Fowler, M. *Patterns of Enterprise Application Architecture*. Addison-Wesley. 2003

[FY98] Foote B, J. Yoder. *Metadata and Active Object-Models*. Proceedings of Plop98. Technical Report #wucs-98-25, Dept. of Computer Science, Washington University Department of Computer Science, October 1998.

[GenJava] Sun Microsystems. *Java Programming Language. Enhancements in JDK 5: Generics*. http://java.sun.com/j2se/1.5.0/docs/guide/language/generics.html

[GenNet] Microsoft Developers Network. *Generics (C# Programming Guide)*. http://msdn.microsoft.com/en-us/library/512aeb7t(VS.80).aspx

[GoF95] Gamma, E.; R. Helm, R. Johnson, J. Vlissides. *Design Patterns: Elements of Reusable Object Oriented Software*. Addison-Wesley. 1995.

[HT00] Hunt, Andrew; David Thomas. *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley. 2000.

[JF98] Johnson, Ralph; Brian Foote. Designing Reusable Classes. Journal of Object-Oriented Programming June/July 1988, Volume 1, Number 2, pages 22-35. http://www.laputan.org/drc/drc.html

[JW98] Johnson, R., R. Wolf. *Type Object*. Pattern Languages of Program Design 3. Addison-Wesley, 1998.

[Kerievsky03] Kerievsky, J. Refactoring to Patterns. Addisson-Wesley. 2003.

[Kovacs03] Kovacs, R. Creating Dynamic Factories in .NET Using Reflection. MSDN Magazine. March 2003. http://msdn.microsoft.com/en-us/magazine/cc164170.aspx

[MK06] Miller, R.; R. Kasparian. *Java For Artists: The Art, Philosophy, and Science of Object-Oriented Programming*. Pulp Free Press, 2006

[MSNET] Microsoft .NET Framework. http://www.microsoft.com/net/

[POSA1] Buschman, F. et al. *Pattern Oriented Software Architecture, Volume 1: A System of Patterns*. Wiley & Sons. 1996

[POSA3] Kircher, M.; P. Jain. *Pattern Oriented Software Architecture, Volume 3: Patterns for Resource Management*. Wiley & Sons. 2004.

[RJ98] Roberts, D.; Johnson, R.: *Evolving Frameworks: A Pattern Language for Developing Object-Oriented Frameworks*.

[RTJ05] Riehle D., M. Tilman, and R. Johnson. "Dynamic Object Model." In *Pattern Languages of Program Design 5*. Edited by Dragos Manolescu, Markus Völter, James Noble. Reading, MA: Addison-Wesley, 2005.

[RY01] Revault, N, J. Yoder. *Adaptive Object-Models and Metamodeling Techniques Workshop Results*. Proceedings of the 15th European Conference on Object Oriented Programming (ECOOP 2001). Budapest, Hungary. 2001.

[SpringModules] Spring Modules. *Chapter 18. XT Framework*. https://springmodules.dev.java.net/docs/reference/0.8/html/xt.html

[SunDevForum] Sun Developer Forums. *Reflections & Reference Objects - Dynamic Factory Method Pattern*. http://forums.sun.com/thread.jspa?threadID=573494

[VanDeursen06] van Deursen, S. *A Fast Dynamic Factory Using Reflection.Emit*. September 2006. http://www.cuttingedge.it/blogs/steven/pivot/entry.php?id=9

[Welicki06] Welicki, L.. *The Configuration Data Caching Pattern*. 14th Pattern Language of Programs Conference (PLoP 2006), Portland, Oregon, USA, 2006.

[WYWJ07] Welicki, L.; J. Yoder; R. Wirfs-Brock; R. Johnson. Towards a Pattern Language for Adaptive Object-Models. Companion of the ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA 2007), Montreal, Canada, 2007.

[YBJ01] Yoder, J.; F. Balaguer; R. Johnson. *Architecture and Design of Adaptive Object-Models*. Proceedings of the ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA 2001), Tampa, Florida, USA, 2001.

[YJ02] Yoder, J.; R. Johnson. *The Adaptive Object-Model Architectural Style*. IFIP 17th World Computer Congress - TC2 Stream / 3rd IEEE/IFIP Conference on Software Architecture: System Design, Development and Maintenance (WICSA 2002), Montréal, Québec, Canada, 2002

[YR00] Yoder, J.; R. Razavi. *Metadata and Adaptive Object-Models*. ECOOP Workshops (ECOOP 2000), Cannes, France, 2000.