The Dynamic Factory Pattern

León Welicki
ONO (Cableuropa S.A.)
Basauri, 7-9
28023, Madrid, Spain
+34 637 879 258
lwelicki@acm.org

Joseph W. Yoder
The Refactory, Inc.
7 Florida Drive
Urbana, Illinois USA 61801
1-217-344-4847
joe@refactory.com

Rebecca Wirfs-Brock
Wirfs-Brock Associates
24003 S.W. Baker Road
Sherwood, Oregon USA
1-503-625-9529
rebecca@wirfs-brock.com

Abstract

The DYNAMIC FACTORY pattern describes a factory that can create product instances based on concrete type definitions stored as external metadata. This facilitates adding new products to a system without having to modify code in the factory class.

Categories and Subject Descriptors

D.1.5 [**Programming Techniques**]: Object-oriented Programming; D.2.2 **[Design Tools and Techniques**]: Object-oriented design methods; D.2.11 **[Software Architectures]**: Patterns

General Terms

Design

Keywords

Factory Objects, Adaptive Object-Models, Creational Patterns

1. Introduction

The DYNAMIC FACTORY pattern describes a factory that can create product instances based on concrete type definitions stored as external metadata. This facilitates adding new products to a system without having to modify code in the factory class.

2. Context

You are working with a software system, possibly 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 [9])—where collaborations between high-level abstractions determine the execution flow.

New functionality is added by extending existing classes and combining new extensions with existing classes [1]. However, to increase flexibility, configuration of how classes are instantiated can be done dynamically. Thus, the types of objects to be dynamically instantiated can be parameterized and changed as needed. This enables new implementations of established framework abstractions to be added as long as they conform to pre-established protocols. Additionally, the system should be able to incorporate these new implementations without coding changes to core framework classes.

3. 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 [7] and INTERPRETER [7] patterns) allowing for the creation of complex expressions by composing finer-grained elements.

Creation of rules is delegated to a factory class that has a standard interface. 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 can be added by simply providing an implementation of the rule interface. The problem comes at rule instantiation, since any factory that contains the logic for creating rule instances may need to be modified to support these new rule types.

4. Problem

How can we define an interface for creating new types of products that implement a given interface without tying it to a concrete implementations?

5. Forces

- *Extensibility / Evolvability*. New product types should be easily added without requiring a new factory class or modification of an existing one.
- *Controlled Evolution*. New types of products that conform to the product interface should be capable of providing different behaviors or new features.
- *Agility*. New types of products should be added to the system quickly, avoiding reworking of a factory class every time a new concrete product is created. It is important to support new versions and quick releases.
- *Simplicity*. The client interface for creating product instances should be simple, hiding from the client the complex details of dynamic product creation.
- *Debugging*. When dynamically creating objects based upon metadata specifications, it can be more difficult to debug since it is not known ahead of time what objects might be instantiated.

- *Security*. Externally storing product definitions expose a potential security risk. It is important to protect product metadata repositories from malicious users.

6. Solution

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

The DYNAMIC FACTORY is a generalized implementation that is responsible for creating instances. It provides a single well-known location for creating instances of a general type, similar to a REGISTRY [2]), 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 or default implementations, but a hook for extensibility must be always provided.

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".

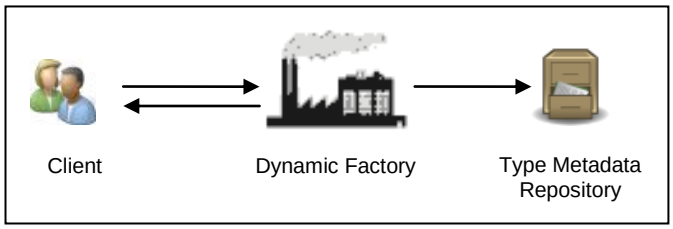

Figure 1 – The Dynamic Factory

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

Adding a new implementation of a Product interface to the system is relatively simple: it requires implementing the product abstraction which is likely to be implemented once and then used by 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).

The following participants form the structure of the DYNAMIC FACTORY pattern as shown in Figure 2:

- A *DynamicFactory* is a class that creates instances using metadata at runtime to determine the concrete type of product to be created.
- A *MetadataReader* reads type metadata from a configuration repository and delivers it to the *DynamicFactory* in an instance of *ProductTypeInfo*.
- *ProductTypeInfo* contains the type metadata about a concrete product. These definitions are fairly constant and rarely change.
- A *Product* represents a general abstraction in a software system. This abstraction can be in the form of an interface, an abstract class with virtual methods, or a class providing default implementations.
- A *ConcreteProduct* is an implementation of the Product abstraction that provides a concrete, specific implementations.
- A *Client* uses instances of *ConcreteProducts* through the *Product* abstraction (abstract coupling [7]). The instances of the products are created using the *DynamicFactory*.

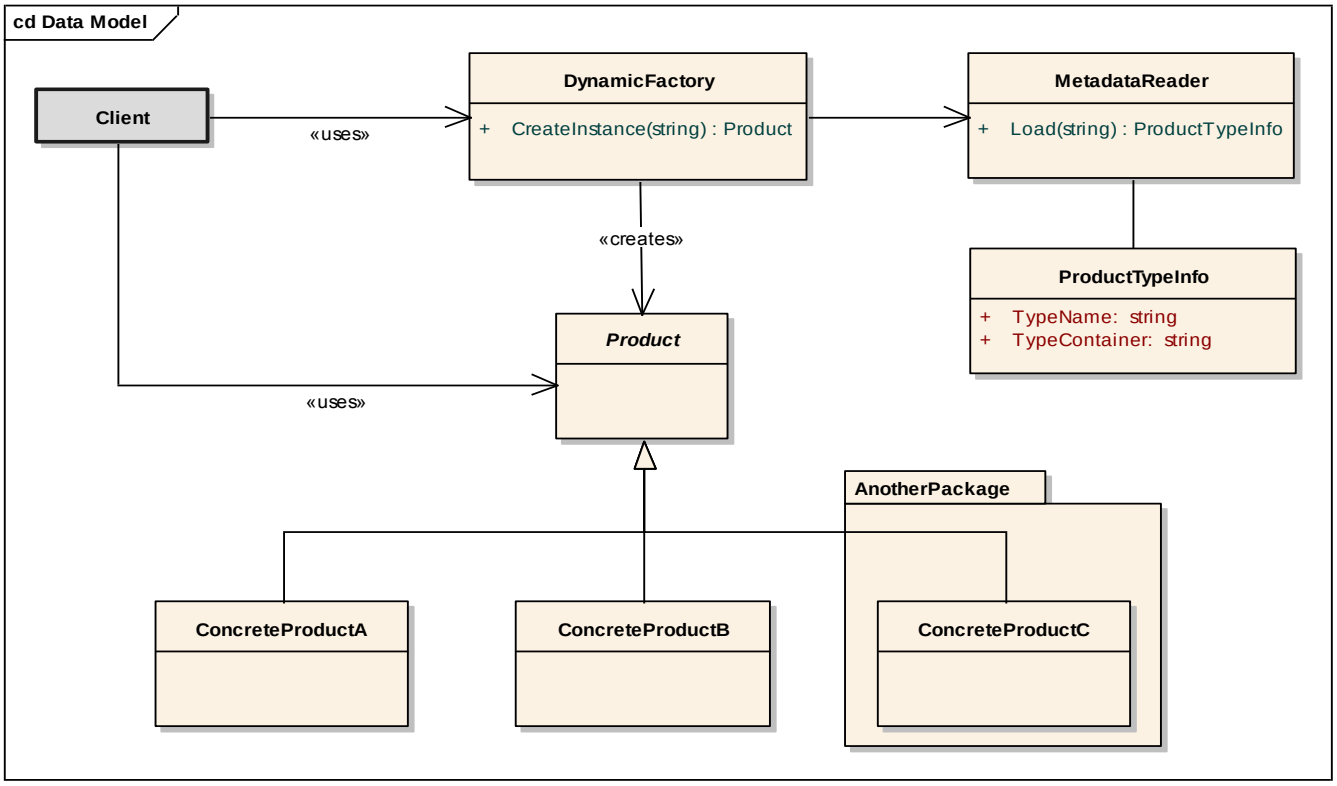

Figure 2 - Dynamic Factory Class Diagram

The following CRC cards (Figure 3) describe the participants' responsibilities and how they interact:

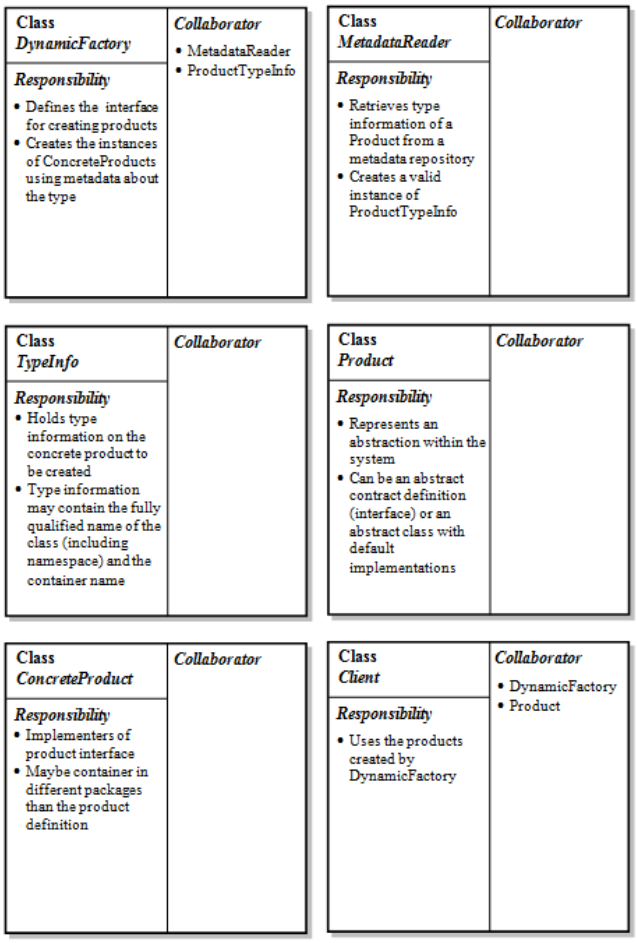

Figure 3 – Abstract Factory CRC Cards

The `Product` defines the general abstraction of 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 the product abstraction. In the simplest case, a `DynamicFactory` creates instances of a single type of `Product`. However, this can be extended by using generics [4], [6] to create a dynamic factory for any kind of product. The ABSTRACT DYNAMIC FACTORY variant described in this paper (see Variants section) creates instances of multiple types of products.

The type information metadata of `Product` implementations, e.g. the `ConcreteProducts`, is stored in a type metadata repository (e.g. an xml file, relational database, plain text file, or any suitable means for storing configuration data). The `MetadataReader` reads and interprets this information and returns the type information as an instance of `ProductTypeInfo`. This decouples the `DynamicFactory` from the metadata repository and product type external representations, since it only accesses `ProductTypeInfo`.

7. Consequences

The DYNAMIC FACTORY improves flexibility and provides better modularity by abstracting the creation process of product instances. Product creation is provided by a well-known entity, Additionally, it makes it easier to introduce new implementers of a product into a system, since the product type specification details are encoded in metadata.

The process of dynamically creating product instances is complex, but this complexity is hidden from other application code. The *DynamicFactory* can be a well-known static class with a simple interface.

This helps to put in practice the principle "*put abstractions in code and details in metadata*" [8]. This also builds on the Dependency Inversion Principle and Open Closed Principle [5].

Creating product instances dynamically can cause significant performance problems. These can be mitigated by applying the CACHING pattern combined with other resource management patterns described in [16].

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 class and adding its type declaration to the metadata repository.
- *Flexibility*. Existing concrete products can be modified or removed and new products can be added dynamically. This can even be done at run-time since the creation of instances is done dynamically using REFLECTION [15] or similar techniques.
- *Configurability*. You can change the behavior of an application without changing any source code. Just change the descriptive information about the type in the metadata repository. If caching is used the cache will need to be flushed.
- *Agility*. New concrete products can be added quickly following a recognized procedure that leverages existing architectural decisions.

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

- *Run-time errors*. It is not enough to write correct product code, you must also define the metadata correctly. At compile time a good test suite can help validate the metadata definitions, but when adding or modifying the type metadata, unexpected runtime errors can occur. Very simple typos in metadata can lead to product instantiation 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 [7].
- *Complexity*. The solution hides the complexity from the clients, but it is still complex. The internals of the factory are more complex than directly invoking a product constructor. This complexity increases significantly when CACHING is added. For a more detailed discussion see [16] and [23].
- *Possible "over-engineering"*. If new product types are not going to be added frequently or current product implementations are rarely modified or switched at

runtime, using this pattern is an overly complex solution. A good way to avoid unnecessary complexity is to start with simpler options like using a static class or simpler creational patterns [7]. You can then evolve to using a Dynamic Factory when it is warranted, following an evolutionary design approach [11].

- *Performance*. Using reflection and dynamically reading product type definitions can cause the system to perform slowly. If product instantiation performance becomes a problem, well known caching techniques can be applied to improve performance.
- *Security*. Security may need to be specially considered, since new concrete products may contain threats to the host system. This could be mitigated in several ways, e.g. running in partial trust or having a strong runtime policy compliance verification mechanism (like .NET's Code Access Security [14]).
- *Debugging*. Debugging of systems using Dynamic Factory may be harder since the new components may introduce unanticipated errors. Another important issue regarding debugging is the configuration differences between production, staging, and development scenarios (each one may be running different implementations of the product interface).

8. Example Resolved

All the rules in the workflow system are derived from a basic abstraction (the `Rule` interface). To remove all concrete type information from rule creation code, a DYNAMIC FACTORY for creating `Rule`s is defined.

A metadata format for specifying the types of the rules is also established. This format includes an identifier for the rule and its type information (e.g. the container and class name of the implementer of the rule). Moreover, the format supports composition following the COMPOSITE [7] pattern). Instances of composed rules are loaded dynamically at runtime by a combination of the BUILDER [7], INTERPRETER [7], and DYNAMIC FACTORY patterns.

By doing so we remove references to concrete rule types from the source code of the factory. This allows for change and extension of the workflow system through the definition of new rules in the metadata repository.

9. 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 single products

The following 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`. Since implementers of the abstraction may not be known upfront, the next step in the implementation of the pattern is to define the format of metadata which is used to declare the type information for each realization of the product interface. The following code snippet shows a sample xml file with type information. Each declaration 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 could also be stored in a relational database, plain files, etc. To hide the storage implementation details from the factory use the `MetadataReader` to access the type metadata repository (the xml file defined above) and the `ProductTypeInfo` to hold 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)
    {
        // fetch concrete product info
        XmlDocument doc = new XmlDocument();
        doc.Load(AppSettings["rootPath"]);
        XmlNode node = doc.SelectSingleNode(
```

```
          "/typeInfo/products/product[@id='" +
                              typeName + "']");

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

A simple implementation of the `DynamicFactory` class is presented below. 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)
    {
        // create the reader and retrieve the
        // requested ProductTypeInfo
        MetadataReader metadataReader =
             new MetadataReader();
        ProductTypeInfo typeInfo =
             metadataReader.Load(productTypeCode);

        // create the instance of concrete product
        // for info about ObjectHandle see [29]
        // for info about Activator see [30]
        ObjectHandle obj =
             Activator.CreateInstance(
                  typeInfo.AssemblyName,
                  typeInfo.ClassName);
        return (IProduct)obj.Unwrap();
    }
}

public class SampleClient
{
    public void Main()
    {
       // create the product an do something
       IProduct product =
         DynamicFactory.Create("product1");
       product.DoSomething();

       // create another product an do something
       product =
         DynamicFactory.Create("product2");
      product.DoSomething();
    }
}
```

Extending the factory with Generics

The implementation of the `DynamicFactory` shown above is limited to creating instances of `IProduct` interface. To make it more general, you can use generics, as shown below.

```
public class GenericDynamicFactory<T>
{
    public T Create(string productTypeCode)
    {
        // create the reader and retrieve the
        // requested ProductTypeInfo
        MetadataReader metadataReader =
             new MetadataReader();
        ProductTypeInfo typeInfo =
             metadataReader.Load(productTypeCode);

        // create the instance of concrete product
        // for info about ObjectHandle see [29]
        // for info about Activator see [30]
        ObjectHandle obj =
            Activator.CreateInstance(
                typeInfo.AssemblyName,
                typeInfo.ClassName)
       return (T)obj.Unwrap();
    }
}

public class SampleClient
{
    public void Main()
    {
        // create and use the DynamicFactory
        DynamicFactory<IProduct> dynamicFactory =
                new DynamicFactory<IProduct>();

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

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

        // create and use another DynamicFactory
        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 is generic.

```
public static class GenericDynamicFactory
{
    public static T Create<T>
                  (string productTypeCode)
    {
        // create the reader and retrieve the
        // requested ProductTypeInfo
        MetadataReader metadataReader =
             new MetadataReader();
        ProductTypeInfo typeInfo =
             metadataReader.Load(productTypeCode);

        // create the instance of concrete product
        // for info about ObjectHandle see [29]
        // for info about Activator see [30]
        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();

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

The implementations shown are greatly simplified. They don't take into account critical issues like exception handling, caching, security, or configuration management. More sample implementations of this pattern can be found in [22], [20], [13], and [12].

10. Variants

Following are brief characterizations of some known variants of the DYNAMIC FACTORY pattern.

- **Cached Dynamic Factory**: the DYNAMIC FACTORY can be combined with the CACHING pattern [16] or the CONFIGURATION DATA CACHING pattern [23] to improve run-time efficiency. 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 when directly caching the concrete products. The first case is very simple to implement, since the *ProductTypeInfo* are often immutable. The last case is more difficult and is feasible only when the *ConcreteProducts* are stateless [16].

  If a cache is used, an EVICTOR [16] or similar approach may be necessary to unload outdated or unused product instances from memory. When the type information is updated in the metadata repository, some 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 a parameter it uses when creating product instances. There are several options for what the parameter represents: it can be the type information of the product to be created or an alias to search for it in the product type metadata repository.

- **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 are declared in a product type metadata repository. 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 achieved by dynamic interpretation of metadata.

- **Adaptive Object-Model Dynamic Factory**: in AOM-based architectures [25], [26], [24], the DYNAMIC FACTORY can be used to create the instances of the PROPERTIES, ENTITIES, ACCOUNTABILITIES, and RULE OBJECTS and their corresponding TYPE OBJECTS [10].

11. Known Uses

Microsoft ASP.NET uses this pattern to configure its extensibility features. 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 [14].

Adaptive Object-Models. An Adaptive Object-Model is a system that represents user-defined classes, attributes, relationships, and behavior as metadata [25], [26]. The system domain model is based on instances rather than classes. Users change these metadata descriptions to reflect changes in the domain model. These changes modify the system's behavior. AOM-based architectures extensively use dynamic creation of objects based on metadata.

Rule based systems. The rules are configured using a VISUAL LANGUAGE [17] 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, typically a general abstraction (e.g. interface or abstract class) is implemented and its type information is 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. It includes the Dynamic Factory Generator that lets you generate factory objects on the fly [20].

Eclipse Tools with Plugins provides a dynamic factory mechanism for instantiating and plugging in new types of tools [28].

12. Related Patterns

FACTORY METHOD [7] and ABSTRACT FACTORY can be evolved to DYNAMIC FACTORY. Since both establish an interface for creating products, they can be evolved to use metadata.

The DYNAMIC FACTORY can use the CACHING pattern [16] to hold the configuration data (XML metadata), a prototypical instance or the instance itself when the product is stateless. In this case an EVICTOR [16] may be used to remove cached instances of concrete products.

The DYNAMIC FACTORY can be a SINGLETON [7] and can also be a dynamic REGISTRY [2]

STRATEGY [7] may be used to change the configuration storage access strategy used by the *MetadataReader*. It may use several strategies aimed at fetching data from different types of repositories, e.g., XML, relational database, flat file, etc.

DEPENDENCY INJECTION [31] can use DYNAMIC FACTORY for abstracting and moving to metadata the information about the implementations to be injected in the system.

13. Summary

This paper presented the ABSTRACT FACTORY pattern, which allows for dynamic creation of product instances based on the interpretation of externally stored metadata. Variants of this pattern support the creation of a family of products, improve performance, or support Adaptive-Object Model implementations.

We used as an example the creation of rules for a workflow system. Since rules can be added frequently, applying this pattern provides the capability to dynamically create rule instances based on the interpretation of externally stored metadata descriptions.

14. ACKNOWLEDGEMENTS

We would like to thank our shepherd Scott Schneider for his great help and advice for improving the contents of this paper. We would also like to gratefully thank to the participants of the PLoP 2008 "Rock and Roll" Writers Workshop (Ralph Johnson, Ademar Aquiar, Alexander Ernst, Srinivas Rao, Hugo Ferreira, Filipe Correia, Nono Flores, Sachin Bammi, Peter Swinburne, and Atsuto Kubo) and to OOPSLA 2008 for supporting us on having PLoP in Nashville TN.

15. References

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

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

[3] 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.

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

[5] Fowler, Martin. *Agile Software Development, Principles, Patterns, and Practices.* Prentice Hall. 2002.

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

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

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

[9] 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

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

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

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

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

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

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

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

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

[18] 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.

[19] 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.

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

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

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

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

[24] 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.

[25] 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.

[26] 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

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

[28] Bolour, A. *Notes on the Eclipse Plug-in Architecture*. Eclipse.org. http://www.eclipse.org/articles/Article-Plug-in-architecture/plugin_architecture.html

[29] Microsoft Developers Network. *ObjectHandle Class*. .NET Framework Class Library. http://msdn.microsoft.com/en-us/library/system.runtime.remoting.objecthandle_members.aspx

[30] Microsoft Developers Network. *Activator Class*. .NET Framework Class Library. http://msdn.microsoft.com/en-us/library/system.activator.aspx

[31] Fowler, M. Inversion of Control Containers and the Dependency Injection pattern. http://martinfowler.com/articles/injection.html