GenericFactory

DRAFT

Oliver Vogel
SYSTOR AG
Peter-Merian-Strasse 84
CH-4002 Basel
Switzerland
E-mail: oliver.vogel@systor.com

Abstract

The AbstractFactory pattern is a solution often used for isolating clients from the construction of concrete objects. Normally a new concrete factory class is developed for each concrete product family. Therefore new specific construction code has to be written each time a new product family needs to be supported. This approach influences software productivity negatively as it does only cover the reuse of design ideas. Although design reuse is an important and valuable factor for successful software development, it isn't enough in the case of the AbstractFactory pattern. As described above, new construction code has to be implemented for each product family. In order to minimize the effort involved in supporting a new product family, the reuse of code should also be achieved. A configurable factory as described in the implementation section of the AbstractFactory pattern would be an appropriate solution. For every product family, a factory object could be instantiated and configured with the product family objects, so that the only code, that would have to be written, would be the one configuring the factory. This results in less implementation effort and thus in an increased software productivity. The GenericFactory pattern describes how to realize a configurable factory. Moreover, it shows how normal objects can become singletons at run time and how objects can be specified via an abstract or a concrete product identification.

Intent

To provide a flexible, extendable and configurable product construction mechanism that can deliver different objects specified by an abstract or concrete product identification, as well as allow a combination of products in a product family. Further, enlarge the reuse of code, and support the construction of singletons.

Classification

Object Creational

Motivation

Consider an application that has to deal with two problem domains: workflow and organisation. Within the workflow domain, abstractions like Process, ProcessDef and WfHandler can be found. A ProcessDef object describes a process and a Process object represents an actual workflow process that can be initiated on a workflow system via a workflow handler object (WfHandler). A WfHandler is a singleton. That means that there can be only one WfHandler per workflow system. Imagine that the application should be able to support different workflow systems. At first, the Livelink workflow system shall be used. In the future, it might be possible to migrate to a workflow system by another vendor, i.e. Staffware, or to support several simultaneously. In order to enable clients to work with different concrete workflow systems without knowing them directly, interfaces need to be introduced. This allows clients to com-

municate with concrete objects through their abstract interface. In the workflow example the vendor specific classes are derived from Process, ProcessDef and WfHandler. This is illustrated in Abbildung 1. The classes that belong to the Livelink product family are prefixed with “LL” and the ones belonging to the Staffware product family with “SW”.

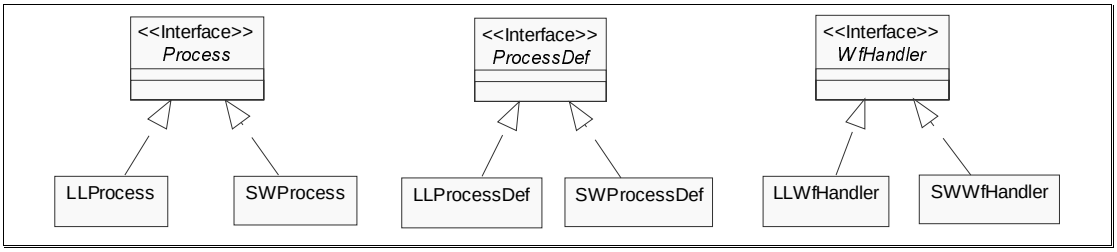

Abbildung 1: Workflow Domain Classes

The next step would normally be the usage of the typical AbstractFactory pattern by modeling an abstract factory that declares the needed interface for the construction of workflow products.

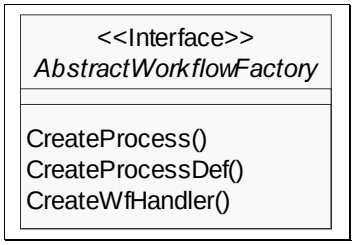

Abbildung 2: Abstract Workflow Factory

After that two concrete factories would be derived from AbstractWorkflowFactory: one that handles the Livelink and one that handles the Staffware product family. This is illustrated below.

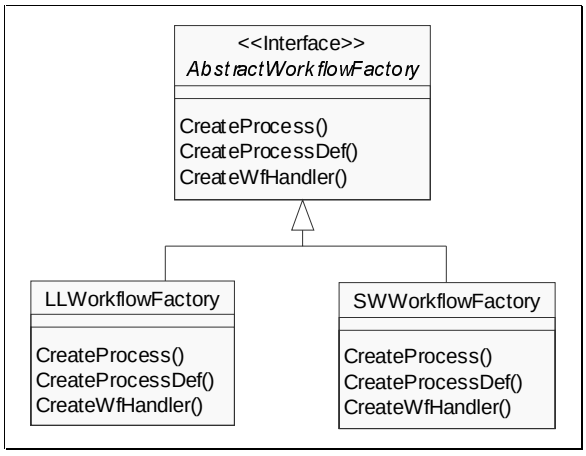

Abbildung 3: Workflow Factory Hierarchy

The concrete factories would override the construction methods and return either a Livelink or a Staffware specific product. LLWorkflowFactory would, for example, return a pointer to a LLProcess object if a client calls its CreateProcess() method. Normally there is only one concrete factory available per product family. The actual concrete factory is often requested from a static GetFactory() method, which is declared in the AbstractWorkflowFactory class. By instantiating a different concrete factory within this method, the whole product family can be exchanged at run time.[1]

[1] Refer to Gamma E. e.a. (1995), p. 87ff. and p. 107ff. for further information

The above solution can also be applied to the organisation domain, where abstractions like Department, Company and Division are used. As the mentioned application must be able to work in different banking environments, an abstract organisation factory would be introduced. Further, the concrete factories UBSOrgFactory and CSOrgFactory would be derived from AbstractOrgFactory. The former would construct specific products for the UBS, and the latter for the CreditSuisse environment.

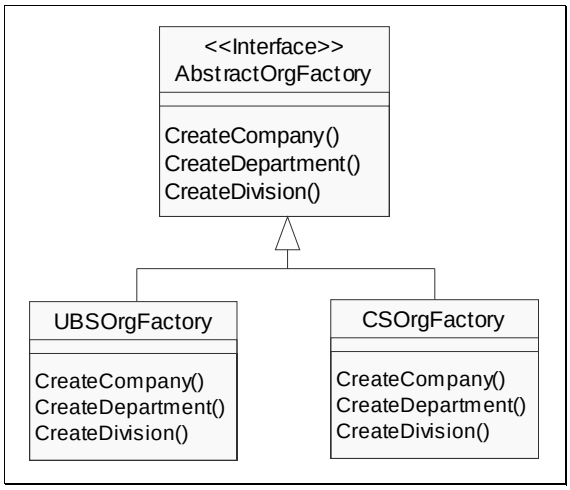

Abbildung 4: Organisation Factory Hierarchy

There are several disadvantages of the above approach. For each new product family, a new concrete factory must be derived and implemented. This is quite time consuming if there is no mechanism that automatically generates the relevant source code skeleton. If a forward engineering approach is used in the software development process, it would also be necessary to generate the relevant classes within the design model. Furthermore, the static GetFactory() method of the corresponding abstract factory needs to be changed. Although the reuse of design is achieved by using the AbstractFactory pattern, the overall implementation time of this pattern does negatively influence software productivity. Therefore, it would be advantageous to have a factory class that can be configured at run time and extended dynamically. This would result in a blackbox implementation of the AbstractFactory pattern and so combine the typically intended reuse of design ideas with the reuse of code. Moreover, for every new product family a blackbox factory could be instantiated and configured with the product family objects. Further, the only code that would have to be actually written would be the one configuring the factory. This would lead to less implementation effort and thus to increased software productivity.

Applicability

Use the GenericFactory pattern when[2]

- a system should be independent of how its products are created, composed and represented.
- a family of related product objects is designed to be used together and you need to enforce this constraint.
- reuse of code shall be enlarged.
- you develop a framework where black box reuse shall be achieved.
- the object construction and delivery mechanism shall be a generic base service.
- normal classes shall become singletons without changing their structure.

[2] s. Gamma, E. e.a. (1995), p. 88 for the first two points

Structure

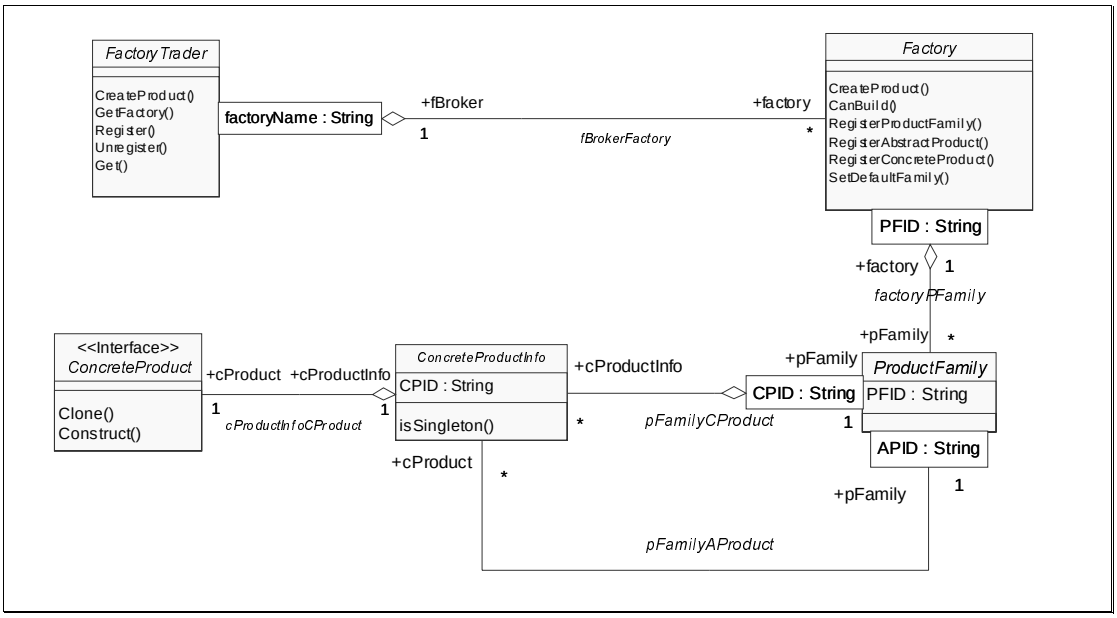

Figure 1: Generic Factory Structure

Participants

- **FactoryTrader**
  - is repsonsible for the delivery of products.
  - administrates factories.
  - shields clients from the used construction mechanism.
- **Factory**
  - administrates ConcreteProducts.
- **ConcreteProduct**
  - declares an interface for the needed Prototype operations.
- **ProductFamily**
  - acts as an container for ConcreteProducts belonging to a ProductFamily.
- **ConcreteProductInfo**
  - holds additional information, which is needed for the construction mechanism.

Collaborations

- Clients ask the FactoryTrader to create specified ConcreteProducts.
- The FactoryTrader asks all registered Factories, which one can build the demanded ConcreteProduct.
- Each Factory checks its ConcreteProduct pool and constructs the ConcreteProduct, if it is available in the pool.

Consequences

The GenericFactory pattern has the following benefits and liabilities:

1. *Clients are isolated from the construction process of concrete objects.* Thus GenericFactory also offers a solution to the design problem addressed by the AbstractFactory pattern, but in a generic way.
2. *GenericFactory eases black box reuse and still enables white box reuse.* The Factory can be subclassed to supply more specific services. If clients would like to work with a concrete Factory, they can request it from the FactoryTrader. Afterwards clients can downcast to the concrete interface and use the declared services.
3. *Application developers do not have to know how the construction mechanism works,* as they don't have to implement a new factory for a new product family. They can create a new instance of Factory instead and configure it with their ConcreteProducts.

4. *Especially appropriate for framework development.* The construction mechanism is a base service which can be integrated in the framework core and reused in several applications.
5. *Normal objects can become singletons.* There is no need to implement the Singleton pattern. Instead ConcreteProducts can just be registered as Singletons at a Factory.
6. *All ConcreteProduct classes must implement the Prototype interface.* This may sound as a disadvantage, but normally they are anyhow implemented to increase performance. If a prototype method is offered by an object a client only needs to request an instance from the corresponding factory once and can then use the prototype method to create clones.
7. *Clients must take care of type safety.* To actually work with the ConcreteProducts, clients have to downcast to the expected interface. They cannot be sure, that they really get what they wanted as a totally different ConcreteProduct could have been registered under the given CPID or APID. That's the typical trade off between a highly flexible concept and type safety.
8. *Clients do not know, if objects with which they are working are singletons.* There is no possibility to ask a ConcreteProduct, if it is a Singleton. Singleton characteristics of a ConcreteProduct have to be documented somewhere else, for example in the design-model.
9. *Clients may not violate against the construction process.* For example, it is possible for clients to create a clone of a Singleton. Thus every application developer needs to follow the construction rules to help avoiding unexpected behaviours.
10. *Who shall delete ConcreteProducts?* If there is no garbage collection, application developers must agree upon a policy on how to handle the deletion of ConcreteProducts. Imagine following example policy: Factories delete their registered ConcreteProducts and clients delete the ConcreteProducts they requested. If a ConcreteProduct is a Singleton, clients mustn't delete it.
11. *Construction mechanism may not be very performant.* Normally every time a client requests a ConcreteProduct the FactoryTrader loops through all Factories to find the one, which can deliver the ConcreteProduct. This could lead to a performance bottleneck. Therefore a smart FactoryTrader could be introduced. A smart FactoryTrader remembers, which Factories can deliver which ConcreteProducts. Thus if a client requests a ConcreteProduct, thas has already been delivered the FactoryTrader directly knows, which Factory to use. This increases performance a lot. Furthermore, a client can request a ConcreteProduct only once and then use the Prototype methods.

Implementation

The GenericFactory pattern describes a way to implement a black box factory. It also shows how normal objects can become singletons at run time and how objects can be specified via an abstract or a concrete product id. Moreover, GenericFactory enables the grouping of products to product families. GenericFactory is a composite pattern as it does combine several patterns to solve its addressed design problem.[3] GenericFactory is composed of the Singleton, the Prototype[4], the Broker/Trader[5], the FactoryMethod[6] and the PropertyList pattern. During the explanation of this pattern the different base patterns will be mentioned where appropriate. In the next paragraph, the concept of the GenericFactory pattern will be introduced. After that it will be explained by applying it to the Motivation example.

How to design the FactoryTrader?

Within the GenericFactory pattern the FactoryTrader is the central point of communication. It is responsible for administrating all Factories and for fulfilling requests of clients by delivering the demanded objects resp. ConcreteProducts. Normally, clients only communicate with the FactoryTrader. Therefore, the FactoryTrader acts as a trader or broker between clients and factories when it receives the clients' requests and then tries to find a factory that can satisfy them by returning the ConcreteProducts. This concept conceals the existence of factories from clients and so eases the interchange of the construction mechanism. For example, it

[3] s. Riehle, D. (1995) for composite patterns
[4] s. Gamma, E. e.a. (1995), p. 117ff.
[5] s. Buschmann, F. e.a. (1996), p. 99ff.
[6] s. Gamma, E. e.a. (1995), p. 107ff.

would also be possible that the FactoryTrader directly communicates with the requested ConcreteProducts.[7] After a Factory has returned the demanded ConcreteProduct, the FactoryTrader forwards it to the client. As stated above, the FactoryTrader needs to know all available Factories. For this reason, it must be able to register and unregister Factories. Further, after all Factories have been registered, the FactoryTrader has to be able to ask the Factories, which ConcreteProducts can be constructed by them. The Factories must contain a service which tells the FactoryTrader if a demanded ConcreteProduct can be built. The Factories must also offer the actual construction mechanism.

How to design a Factory?
A Factory has to be initialized with the ConcreteProducts that it should be able to construct. Therefore a Factory must also provide a registration service like the FactoryTrader. Nevertheless, this will differ from the FactoryTrader registry service because a black box factory should behave like a typical ConcreteFactory of the AbstractFactory pattern. Thus a Factory should be able to handle product families. This results in the necessary ability to group ConcreteProducts to a product family and to associate ConcreteProducts with their AbstractProducts[8]. So a Factory must supply appropriate operations which, for example, take the identification of an AbstractProduct as a parameter and execute the needed internal steps to administrate them. Afterwards, the ConcreteProducts can be registered. Thereby the identification of the ConcreteProduct, the ConcreteProduct itself, the identification of the corresponding AbstractProduct must be supplied to enable the association of the ConcreteProduct with the AbstractProduct.

How to handle Singletons?
Often a product family can contain one or more ConcreteProducts that must behave like Singletons[9]. Singletons have to be treated differently by Factories. Normally, ConcreteFactories would know which ConcreteProducts are Singletons and implement the corresponding create methods appropriately. A black box factory doesn't know what kind of objects it will have to handle. Thus, a different solution has to be found. One solution would be to integrate the singleton specific code in each ConcreteProduct that must be a Singleton in the given context. The construction process is integrated into each ConcreteProduct. Therefore ConcreteProducts act as Prototypes since they are able to clone themselves. In the simplest case, each ConcreteProduct must provide a method which creates a clone of the ConcreteProduct and returns it to the caller. In the case of a Singleton this method would not return a clone, rather a pointer to itself. Thus, there would be only one instance of the ConcreteProduct. This approach has one drawback: It couples the singleton characteristic with the ConreteProduct. In another context, it is possible, however, that the ConcreteProduct is not a Singleton. In such a situation the clone method would have to be modified. To avoid this and enable that any ConcreteProduct can become a Singleton at run time, the Factory needs to be designed in another way. During the registration of a ConcreteProduct at a Factory, there must also be a flag supplied indicating if the ConcreteProduct shall be treated as a Singleton. If the FactoryTrader then requests a ConcreteProduct, the Factory checks to see if the ConcreteProduct is a Singleton, and either returns a clone or a pointer to the registered ConcreteProduct.

How to request ConcreteProducts?
The communication between clients and the FactoryTrader hasn't been covered yet. Clients need to specify somehow the requested ConcreteProduct. However sometimes clients really don't and actually shouldn't know, which ConcreteProduct they need. In other cases clients might need a ConcreteProduct, which is derived from an AbstractProduct of a specific product family. Again imagine the Motivation example. A client could request the Process object of the Livelink product family in one case and in the other case the Process object of the Staffware product family. How can this considerations be realized in a flexible manner? The FactoryTrader could offer a method, which provides parameters for all of the explained cases. A client would then supply the needed parameters and initialize the ones which are not needed to a default value. For the above example a client would e.g. call the method of the FactoryTrader like this:

[7] reflects a special implementation of the Prototype pattern
[8] the reason for relating ConcreteProducts with their AbstractProducts will be discussed later
[9] s. Gamma, E. e.a. (1995), p. 127ff.

```
Signature: FactoryTrader::CreateProduct(String AbstractProductID,
                                        String ConcreteProductID,
                                        String ProductFamily)

Process rProcess=rFactoryTrader.CreateProduct("Process",
                                              "",
                                              "Livelink-Wf-Product-Family");
```

The above approach is not flexible as every time a new parameter is needed, the method signature and therefore the class interface must be changed. This can lead to major problems as clients maybe need to be recompiled and relinked.[10] Furthermore, clients need to adapt their method invocation. These problems can be avoided by applying the PropertyList pattern[11]. This pattern is a flexible solution to allow the evolution of classes without the modification of their interfaces. The PropertyList pattern describes how a list of name/value pairs can be used to hold the parameters of a method. Thus instead of supplying all parameters explicitely a client just supplies a ProperList object, which contains the parameters. A refactored solution of the above example is illustrated below:

```
PropertyList      rPropertyList = new PropertyList();
rPropertyList.Add("APID","Process");
rPropertyList.Add("PFID","Livelink-Wf-Product-Family");

Process rProcess=rFactoryTrader.CreateProduct(pPropertyList);
```

Further, parameters can now be added in a very flexible manner by simply using the Add() method of the PropertyList object.

How to apply GenericFactory?

After the general concepts of the GenericFactory pattern have been illustrated the application of the pattern in the case of the Motivation example will be discussed. Please note that the supplied code samples are only given to show the necessary steps.

In order to handle product families of different domains it is useful to use one Factory per domain. Let's take a look at the workflow domain. There, two product families are present and therefore need to be registered at the Factory object:

```
Factory           rWfFactory = new Factory();
// Register Product Families
rWfFactory.RegisterProductFamily("LLFamily");
rWfFactory.RegisterProductFamily("SWFamily");
```

After registering the product families their abstract members need to be attached :

```
// Register AbstractProduct Familiy Members
rWfFactory.RegisterAbstractProduct("Process","LLFamily");
rWfFactory.RegisterAbstractProduct("ProcessDef","LLFamily");
rWfFactory.RegisterAbstractProduct("WfHandler","LLFamily");

rWfFactory.RegisterAbstractProduct("Process","SWFamily");
rWfFactory.RegisterAbstractProduct("ProcessDef","SWFamily");
rWfFactory.RegisterAbstractProduct("WfHandler","SWFamily");
```

Then the ConcreteProducts must be registered at the Factory and associated to their ProductFamilies and AbstractProducts. For example, the LLProcess object has to be attached to the Livelink family and to the abstract Process object. Moreover, it has to be stated, if the ConcreteProduct shall be treated as a Singleton. Please recall that the registered ConcreteProducts are Prototypes and therefore must be instantiated.

[10] for example, if C++ is used

[11] s. Sommerlad P. & Rüedi M. (1998)

```
// Register ConcreteProducts

PropertyList      rPropertyList     = new PropertyList();

// Register LLProcess

Process           rProcess          = new LLProcess();
rPropertyList.Clear();
rPropertyList.Add("CPID","LLProcess");
rPropertyList.Add("APID","Process");
rPropertyList.Add("ProductFamily","LLFamily");
rPropertyList.Add("Singleton","false");

// Register LLProcessDef

ProcessDef        rProcessDef       = new LLProcessDef();
rPropertyList.Clear();
rPropertyList.Add("CPID","LLProcessDef");
rPropertyList.Add("APID","ProcessDef");
rPropertyList.Add("ProductFamily","LLFamily");
rPropertyList.Add("Singleton","false");

rWfFactory.RegisterConcreteProduct(rProcessDef,rPropertyList);

// Register LLWfHandler

WfHandler         rWfHandler        = new LLWfHandler();
rPropertyList.Clear();
rPropertyList.Add("CPID","LLWfHandler");
rPropertyList.Add("APID","WfHandler");
rPropertyList.Add("ProductFamily","LLFamily");
rPropertyList.Add("Singleton","true");

rWfFactory.RegisterConcreteProduct(rProcessDef,rPropertyList);

// Register the necessary ConcreteProducts of the SWFamily
…
```

Did you notice a possible problem, that would occur during the registration of ConcreteProducts? RegisterConcreteProduct() has been called several times and each time with a different type, although parameter type need to be unique. Therefore, all ConcreteProducts must implement the same interface. In this case, the top level interface is ConcreteProduct. ConcreteProduct declares the needed Prototype interface. The method Clone() and Create() must be overriden in every concrete ConcreteProduct.

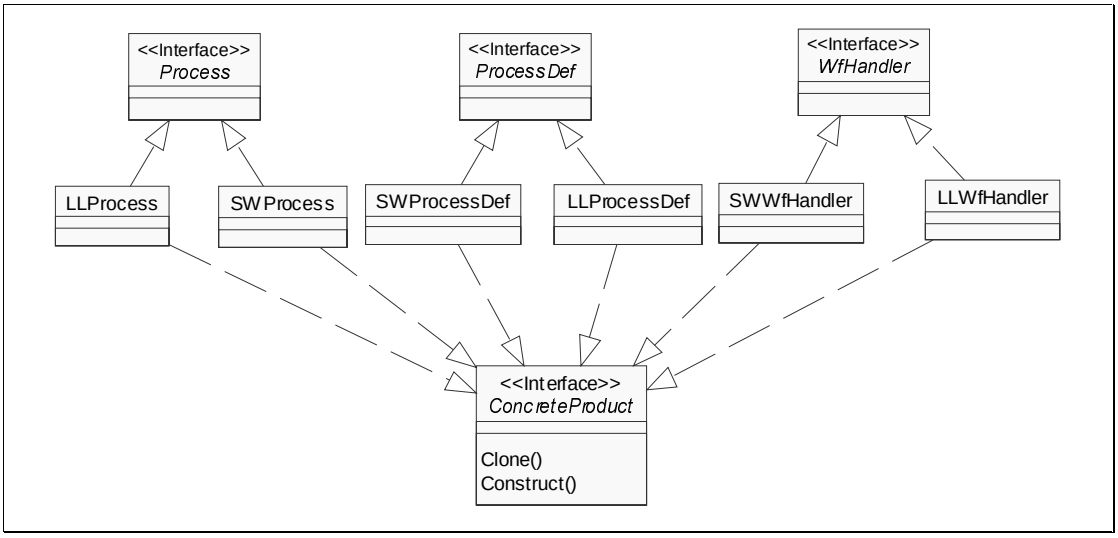

Figure 2: Interface Implementations

Thus during the registration process the pointers to the ConcreteProducts are upcasted to ConcreteProduct. This also forces clients to downcast to the interface of the requested type:

```
PropertyList      rPropertyList = new PropertyList();
rPropertyList.Add("APID","Process");
rPropertyList.Add("PFID","Livelink-Wf-Product-Family");

Process rProcess=(Process)pFactoryTrader.CreateProduct(pPropertyList);
```

After the ConreteProducts have been registered the default ProductFamily needs to be specified, as the Factory has to know, which ConcreteProduct to deliver, if only an APID is supplied as a parameter.

```
// Set default Product-Family
rFactory.SetDefaultFamily("LLFamily");
```

Now the Factory is operational and can be attached to the FactoryTrader. Thereby a unique ID has to be supplied.

```
// Attach WfFactory to the FactoryTrader
FactoryTrader rFactoryTrader=FactoryTrader::Get();
rFactoryTrader.Register(rWfFactory,"WorkflowFactory");
```

How to handle Product Families?
It has been pointed out that a GenericFactory must administrate product families and their related ConcreteProducts. This housekeeping can be realized with the following approach: A GenericFactory creates a ProductFamily object for each registered product family. A ProductFamily is identified by its unique identification. If a ConcreteProduct is registered a ConcreteProductInfo object (CPInfo) is constructed and initialized with the ID of the ConcreteProduct. Furthermore the ConcreteProduct is attached to the CPInfo object. Afterwards the CPInfo object is associated to its corresponding ProductFamily object. Thereby it is associated with its concrete and abstract product id. This allows that a ConcreteProduct can be requested either by a CPID or an APID.

Known Uses

WISE: WISE allows the integration of different workflow management systems (wms) in heterogenous and complex system landscapes. In order to deal with several wms in a flexible

manner, the GenericFactory pattern has been used to develop a generic infrastructure for the creation and the delivery of wms specific objects.

Related Patterns

- *AbstractFactory*
  GenericFactory is an extension of the AbstractFactory pattern as it is applyable in the same context but in a generic way. AbstractFactory uses inheritance where GenericFactory uses composition.
- *ProductTrader*
  GenericFactory is closely related to ProductTrader as it also allows the specification and configuration of ConcreteProducts. Further, both patterns decouple clients from the concrete construction mechanism. Nevertheless, GenericFactory focuses on product families where ProductTrader concentrates on single objects. Moreover, GenericFactory addresses the construction of Singletons.
- *FactoryChain*
  FactoryChain also uses composition to extend functionality, but on factory level. Concrete factories are linked together in a chain and if a factory cannot handle the request it forwards it to its successor.
- *FactoryMethod*
- *Prototype*
- *Singleton*

Bibliography

Bäumer, D. & Riehle D. (1996)	Late Creation (Product Trader), A Creational Pattern, Submitted to PloP'96
Buschmann, F. e.a. (1996):	A System of Patterns, Pattern-Oriented Software-Architecture, Wiley & Sons, West Sussex 1996
Gamma, E. e.a. (1995):	Design Patterns, Elements of Reusable Object-Oriented Software-Design, Addison-Wesley, Bonn 1995
Riehle, D. (1997):	Composite Design Patterns, OOPSLA 1997, ACM Press, p. 218-228, 1997
Sommerlad P. & Rüedi M. (1998):	Do-it-yourself Reflection, IFA Informatik, Zürich, Submitted to EuroPLoP´98