Product Construction Service

Oliver Vogel
SYSTOR AG
e-mail: oliver.vogel@systor.com

Abstract

The ABSTRACT FACTORY 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. 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 ABSTRACT FACTORY pattern would be an appropriate solution. This would result in less implementation effort and thus in an increased software productivity. The PRODUCT CONSTRUCTION SERVICE pattern describes how to realize a configurable factory.

Also known as

Configurable Factory
Generic Factory

Classification

Object Creational

Example

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 instantiated 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 communicate 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 Figure 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".

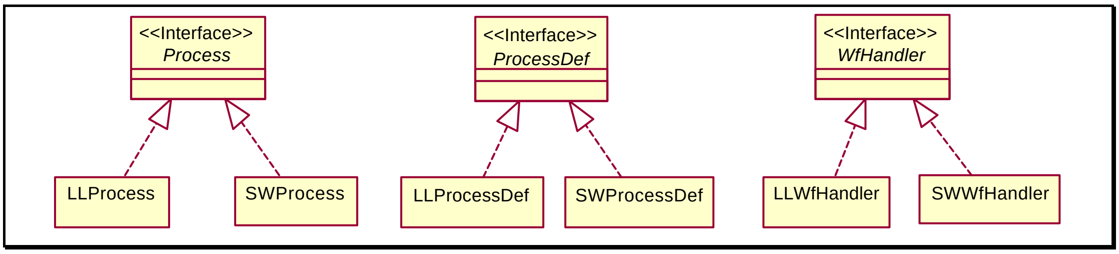
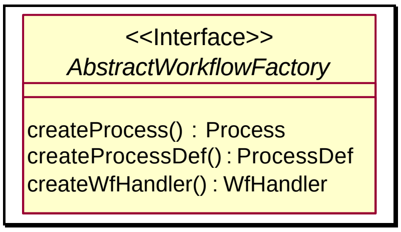


Figure 1: Workflow Domain Classes

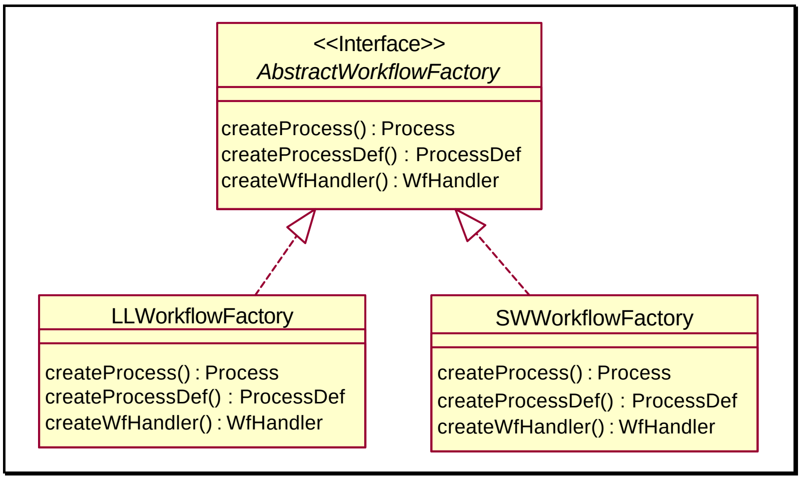
The next step would normally be the usage of the typical ABSTRACT FACTORY pattern by modeling an abstract factory that declares the needed interface for the construction of workflow products.



<<Interface>>
AbstractWorkflowFactory
createProcess() : Process
createProcessDef() : ProcessDef
createWfHandler() : WfHandler

Figure 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.



<<Interface>>
AbstractWorkflowFactory
createProcess() : Process
createProcessDef() : ProcessDef
createWfHandler() : WfHandler
LLWorkflowFactory
createProcess() : Process
createProcessDef() : ProcessDef
createWfHandler() : WfHandler
SWWorkflowFactory
createProcess() : Process
createProcessDef() : ProcessDef
createWfHandler() : WfHandler

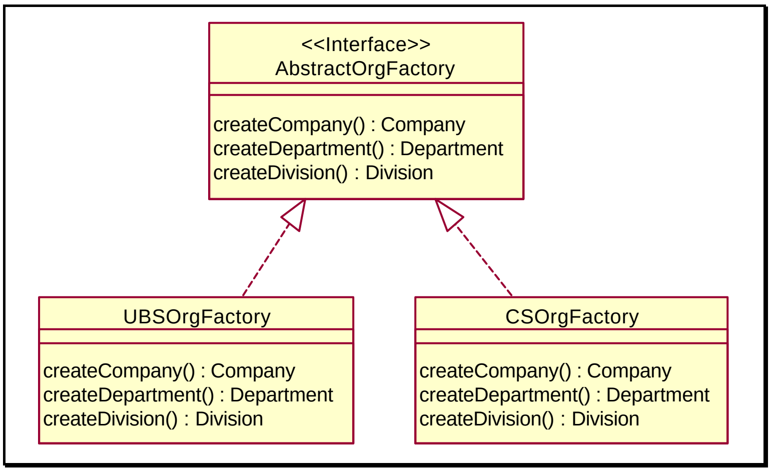
Figure 3: Workflow Factory Hierarchy

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. Usually 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]

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.

[1] refer to [Gamma+1995], p. 87ff. and p. 107ff. for further information

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.



<<Interface>>
AbstractOrgFactory
createCompany() : Company
createDepartment() : Department
createDivision() : Division
UBSOrgFactory
createCompany() : Company
createDepartment() : Department
createDivision() : Division
CSOrgFactory
createCompany() : Company
createDepartment() : Department
createDivision() : Division

Figure 4: Organisation Factory Hierarchy

Context

The development of an extendable system is a challenging task as it must be able to support objects, which are not known during development time. To avoid the coupling of such a system to concrete objects the ABSTRACT FACTORY pattern is usually used. However, the ABSTRACT FACTORY pattern has several disadvantages.

For example, 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 ABSTRACT FACTORY pattern, the overall implementation time of this pattern does negatively influence software productivity.

The PRODUCT CONSTRUCTION SERVICE pattern overcomes the discussed disadvantages and offers a generic way to handle products. It is especially suitable for the development of frameworks.

FORCES

Problem

How do you build a generic object construction and delivery mechanism, that overcomes the inherent disadvantages of the ABSTRACT FACTORY pattern and enables your system to handle new products in a generic manner?

Solution

To avoid the problem discussed above, it would be advantageous to have a factory class that can be configured at run time and extended dynamically. This would result in a black box implementation of the ABSTRACT FACTORY pattern and so combine the typically intended reuse of design ideas with the reuse of code. Moreover, for every new product family a black box 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.

The *PRODUCT CONSTRUCTION SERVICE* pattern describes a way to design an object construction and delivery mechanism that can be used as a generic base service. Thereby it overcomes the disadvantages associated with the *ABSTRACT FACTORY* pattern. 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, *PRODUCT CONSTRUCTION SERVICE* enables the grouping of products to product families.

PRODUCT CONSTRUCTION SERVICE is a compound pattern as it does combine several patterns to solve its addressed design problem.[2] *PRODUCT CONSTRUCTION SERVICE* is composed of the *SINGLETON*, the *PROTOTYPE*[3], the *PRODUCT TRADER*[4], the *FACTORY METHOD*[5] and the *PROPERTY LIST* pattern. During the explanation of this pattern the different base patterns will be mentioned where appropriate. In the next paragraph, the concept of the *PRODUCT CONSTRUCTION SERVICE* pattern will be introduced. After that it will be explained by applying it to the mentioned example.

A *FactoryTrader*, as the central component in the *PRODUCT CONSTRUCTION SERVICE* pattern, allows clients to communicate with *Factories* indirectly. It decouples clients from *Factories* and therefore allows the interchange of the construction mechanism.

Factories handle the construction of *ConcreteProducts* and offer a registry service for *ConcreteProducts*. Moreover, Factories enable the grouping of *ConcreteProducts* to *ProductFamilies*.

ConcreteProduct declares an interface for the prototype operations, which are needed to create products.

ProductFamily acts as a container for *ConcreteProducts* belonging to a *ProductFamily*. It is used by Factories to group *ConcreteProducts*.

Additional information about *ConcreteProducts* such as *SINGLETON* characteristics are stored in *ConcreteProductInfo* objects and used by *Factories* during the construction process.

[2] [Riehle1997] for compound patterns
[3] [Gamma+1995], p. 117ff.
[4] [MBR1998], Product Trader
[5] [Gamma+1995], p. 107ff.

Structure

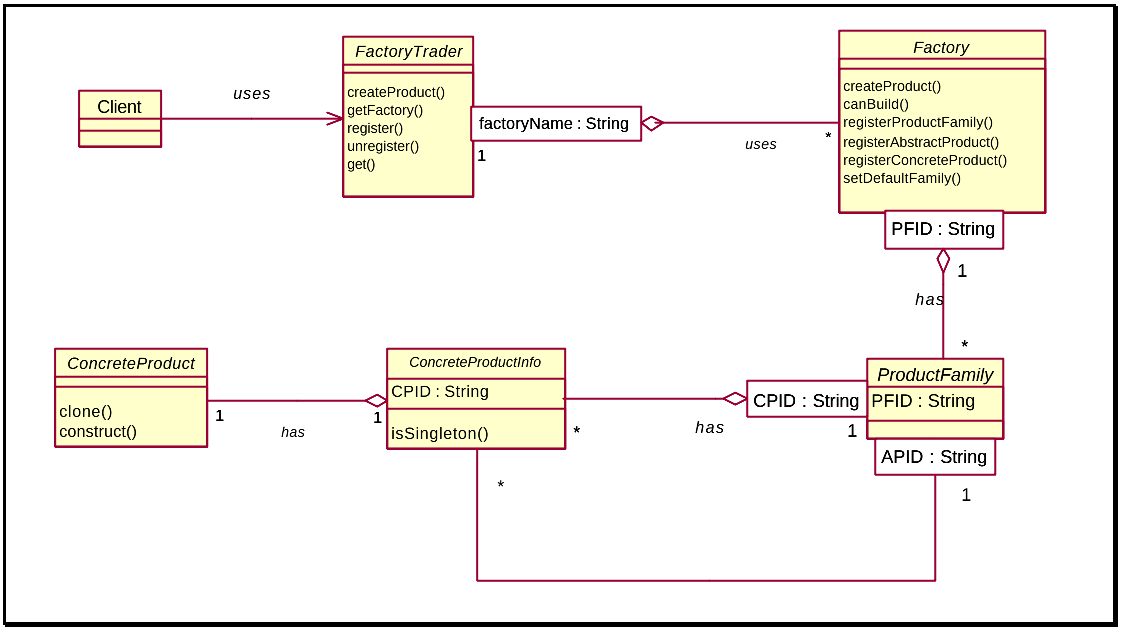


Figure 5: Product Construction Service Structure

How to design the FactoryTrader

Within the *PRODUCT CONSTRUCTION SERVICE* pattern the *FactoryTrader* is the central point of communication. It is responsible for administrating all *Factories* and for fulfilling requests of clients by delivering demanded objects resp. *ConcreteProducts*. Normally, clients only communicate with the *FactoryTrader*. Therefore, the *FactoryTrader* acts as a trader between clients and factories when it receives the clients' requests and 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 substitution of the construction mechanism. For example, it would also be possible that the *FactoryTrader* communicates with the requested *ConcreteProducts* directly.[6] 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 *Factories* which *ConcreteProducts* they can construct. 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.

Class *FactoryTrader*	***Collaborators*** *Client* *Factory*
Responsibility • is responsible for the delivery of products. • administrates factories. • shields clients from the used construction mechanism.	

[6] reflects a special implementation of the Prototype pattern

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 *ABSTRACT FACTORY* pattern. Thus a Factory should be able to handle product families. This results in the necessary ability to group *ConcreteProducts* to product families and to associate *ConcreteProducts* with their *AbstractProducts*[7]. 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 and the identification of the corresponding *AbstractProduct* must be supplied to enable the association of the *ConcreteProduct* with the *AbstractProduct*.

Class *Factory*	***Collaborators*** *FactoryTrader*
Responsibility • administrates ConcreteProducts. • creates ConcreteProducts • administrates ProductFamilies	*ProductFamily* *ConcreteProductInfo* *ConreteProduct*

How to design ConcreteProducts

Often a product family can contain one or more *ConcreteProducts* that must behave like *SINGLETONS*[8]. *SINGLETONS* have to be treated differently by Factories. Usually, *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. Therefore each *ConcreteProduct* must implement the *PROTOTYPE* pattern besides its problem oriented interface, i.e. *Process*, *ProcessDef* or *WfHandler*. This is illustrated in Figure 6. 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, it has to be stated 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*.

[7] the reason for relating *ConcreteProducts* with their *AbstractProducts* will be discussed later

[8] [Gamma+1995], p. 127ff.

Class *ConcreteProduct*	***Collaborators*** *Factory* *ConcreteProduct*
Responsibility • declares an interface for the needed Prototype operations.	

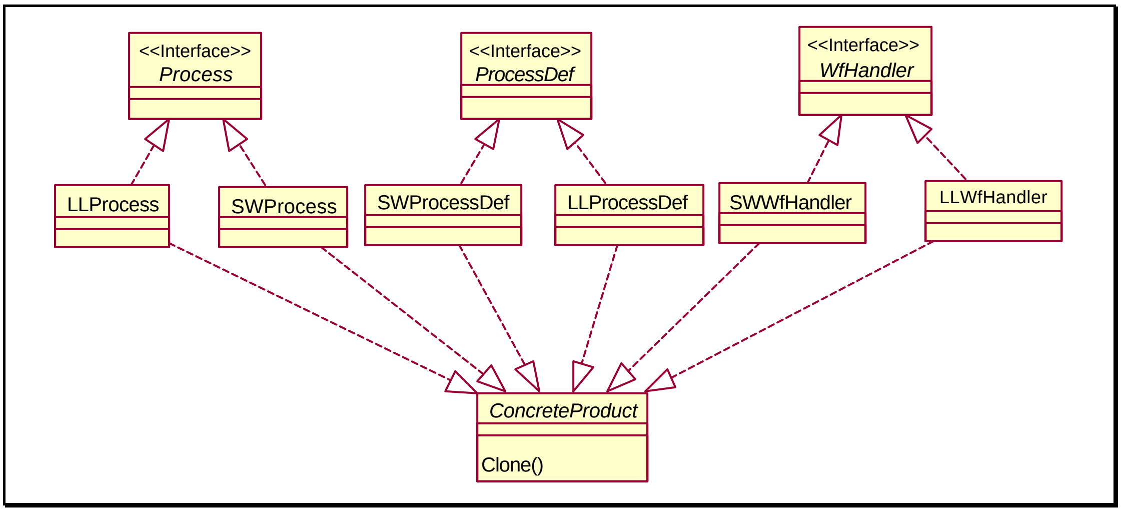


Figure 6: Interface Implementations

Pattern overview

Figure 7 shows the used design patterns in *PRODUCT CONSTRUCTION SERVICE* besides *FACTORY METHOD* and *PROPERTY LIST*. *FACTORY METHOD* is part of the *SINGLETON* and the *PROTOTYPE* pattern and *PROPERTY LIST* is used as a parameter object in several methods.[9]

[9] s. Figure 5 for the method signatures.

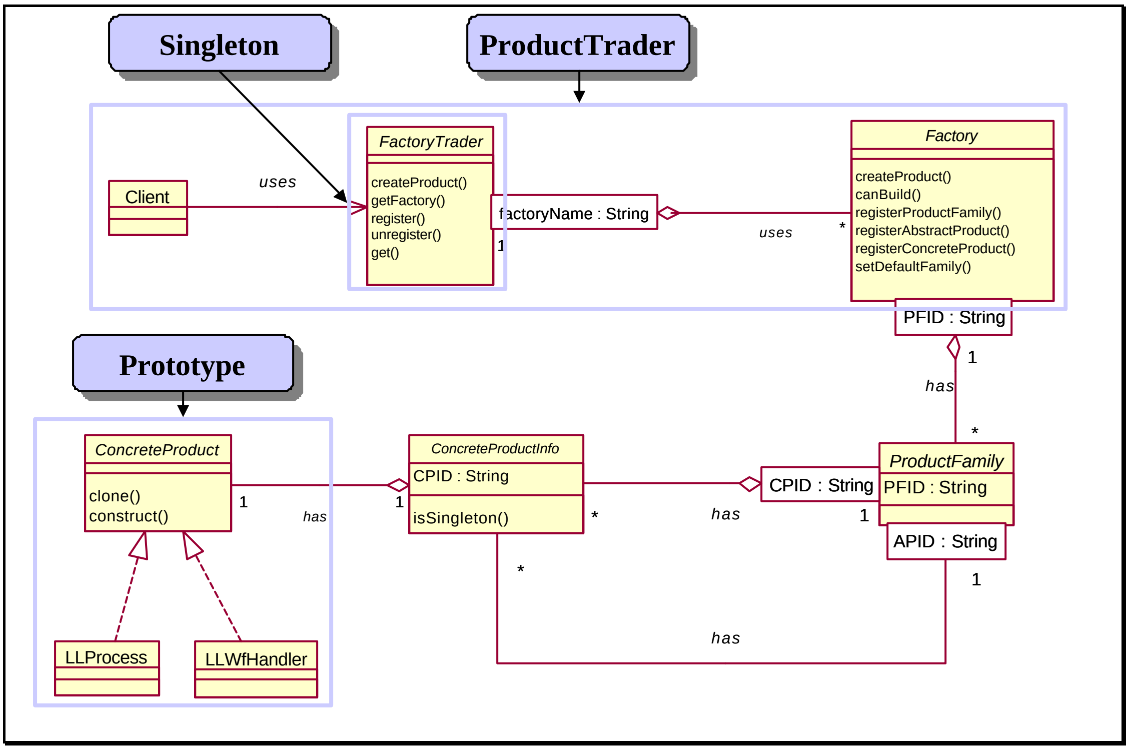


Figure 7: Design Patterns used in Product Construction Service

How to handle Product Families

It has been pointed out that a *PRODUCT CONSTRUCTION SERVICE* must administer product families and their related *ConcreteProducts*. To achieve this administration a *PRODUCT CONSTRUCTION SERVICE* 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* (CPID). 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 (APID). Thus, a *ConcreteProduct* can be requested either by a CPID or an APID.

Class *ProductFamily*	***Collaborators*** *Factory* *ConcreteProductInfo*
Responsibility • acts as an container for ConcreteProducts belonging to a ProductFamily	

Class ConcreteProductInfo	***Collaborators*** *Factory* *ProductFamily* *ConcreteProduct*
Responsibility • holds additional information, which is needed for the construction mechanism	

Dynamics

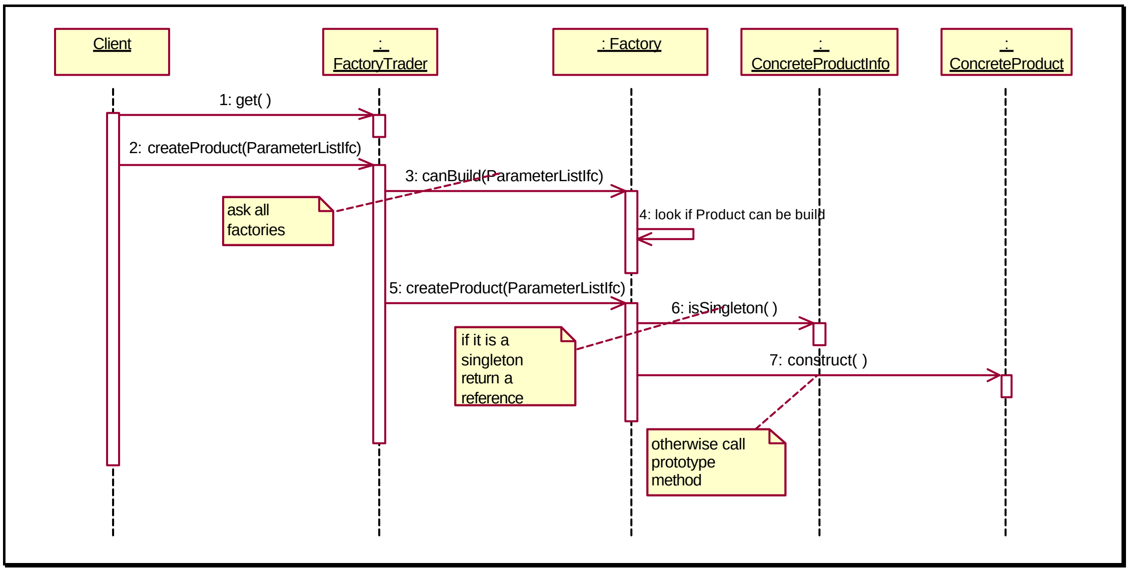

Client
: FactoryTrader
: Factory
: ConcreteProductInfo
: ConcreteProduct
1: get( )
2: createProduct(ParameterListIfc)
3: canBuild(ParameterListIfc)
ask all factories
4: look if Product can be build
5: createProduct(ParameterListIfc)
6: isSingleton( )
if it is a singleton return a reference
7: construct( )
otherwise call prototype method

Figure 8: Request for a Concrete Product

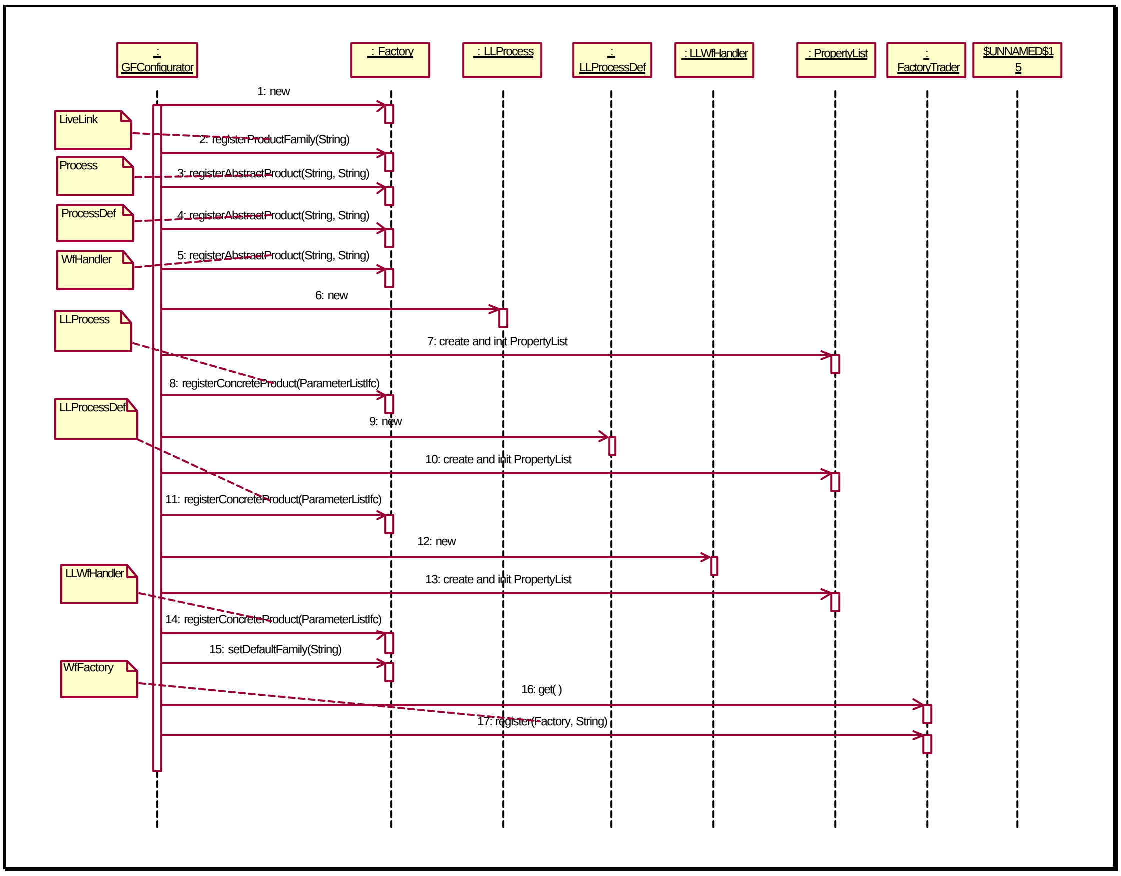

: GFConfigurator
: Factory
: LLProcess
: LLProcessDef
: LLWfHandler
: PropertyList
: FactoryTrader
$UNNAMED$15
1: new
LiveLink
2: registerProductFamily(String)
Process
3: registerAbstractProduct(String, String)
ProcessDef
4: registerAbstractProduct(String, String)
WfHandler
5: registerAbstractProduct(String, String)
6: new
LLProcess
7: create and init PropertyList
8: registerConcreteProduct(ParameterListIfc)
LLProcessDef
9: new
10: create and init PropertyList
11: registerConcreteProduct(ParameterListIfc)
12: new
LLWfHandler
13: create and init PropertyList
14: registerConcreteProduct(ParameterListIfc)
15: setDefaultFamily(String)
WfFactory
16: get( )
17: register(Factory, String)

Figure 9: Configuration and registration of a factory

Consequences

The *PRODUCT CONSTRUCTION SERVICE* pattern has the following benefits and liabilities:

Benefits

1. ***Clients are isolated from the construction process of concrete objects***. Thus *PRODUCT CONSTRUCTION SERVICE* also offers a solution to the design problem addressed by the *ABSTRACT FACTORY* pattern, but in a generic way.
2. ***PRODUCT CONSTRUCTION SERVICE eases black box reuse and still enables white box reuse***. The *Factory* can be sub classed 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. ***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. ***PRODUCT CONSTRUCTION SERVICE can be configured automatically.*** By using a data driven approach *PRODUCT CONSTRUCTION SERVICE* can be configured automatically. A generic configurator could, for example, read all needed information from a xml file and instantiate and initialize the factories.

Liabilities

1. ***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.
2. ***Clients must take care of type safety.*** To actually work with the *ConcreteProducts*, clients have to downcast to the expected interface. *PRODUCT CONSTRUCTION SERVICE* doesn't assure that the expected object is returned as a totally different *ConcreteProduct* could have been registered under the given CPID or APID.
3. ***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.
4. ***Clients must agree on 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 avoid unexpected behaviors.
5. ***Deletion of 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.
6. ***Construction mechanism may not be very efficient***. Usually 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*, that 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

How to request ConcreteProducts?
The communication between clients and the *FactoryTrader* hasn’t been covered yet. Clients need to specify 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 these 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 call the method of the *FactoryTrader* in the following manner.

```
Signature: FactoryTrader.createProduct(String iAbstractProductID,
                                       String iConcreteProductID,
                                       String iProductFamily)

Process lProcess = (Process) lFacTrader.createProduct(“Process”,
                                                      “”,
                                                      ”LLFamily”);
```

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 *PROPERTY LIST* pattern[11]. This pattern is a flexible solution to allow the evolution of classes without the modification of their interfaces. The *PROPERTY LIST* 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 explicitly a client just supplies a *PropertyList* object, which contains the parameters. A refactored solution of the above example is illustrated below:

```
PropertyList lPropertyList = new PropertyList();
lPropertyList.add(“APID”,”Process”);
lPropertyList.add(“PFID”,”LLWf -Family”);

Process lProcess = (Process) lFacTrader.createProduct(lPropertyList);
```

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

Example Resolved

In this section the configuration of the *PRODUCT CONSTRUCTION SERVICE* implementation for the mentioned example will be illustrated. Please note that the supplied code samples are only given to show the necessary steps.

[10] for example, if C++ is used
[11] [Sommerlad+1998]

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 lWfFactory = new Factory();
// Register Product Families
lWfFactory.registerProductFamily(“LLFamily”);
lWfFactory.registerProductFamily(“SWFamily”);
```

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

```
// Register AbstractProduct Familiy Members
lWfFactory.registerAbstractProduct(“Process”,“LLFamily”);
lWfFactory.registerAbstractProduct(“ProcessDef”,“LLFamily”);
lWfFactory.registerAbstractProduct(“WfHandler”,“LLFamily”);

lWfFactory.registerAbstractProduct(“Process”,“SWFamily”);
lWfFactory.registerAbstractProduct(“ProcessDef”,“SWFamily”);
lWfFactory.registerAbstractProduct(“WfHandler”,“SWFamily”);
```

Then the *ConcreteProducts* must be registered at the *Factory* and associated with 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.

```
// Register ConcreteProducts

PropertyList lPropertyList = new PropertyList();

// Register LLProcess

Process lProcess = new LLProcess();
lPropertyList.clear();
lPropertyList.add(“CPID”,”LLProcess”);
lPropertyList.add(“APID”,”Process”);
lPropertyList.add(“ProductFamily”,”LLFamily”);
lPropertyList.add(“Singleton”,”false”);

// Register LLProcessDef

ProcessDef lProcessDef = new LLProcessDef();
lPropertyList.clear();
lPropertyList.add(“CPID”,”LLProcessDef”);
lPropertyList.add(“APID”,”ProcessDef”);
lPropertyList.add(“ProductFamily”,”LLFamily”);
lPropertyList.add(“Singleton”,”false”);

lWfFactory.registerConcreteProduct(lProcessDef,lPropertyList);

// Register LLWfHandler

WfHandler   lWfHandler  = new LLWfHandler();
lPropertyList.clear();
lPropertyList.add(“CPID”,”LLWfHandler”);
lPropertyList.add(“APID”,”WfHandler”);
lPropertyList.add(“ProductFamily”,”LLFamily”);
lPropertyList.add(“Singleton”,”true”);

lWfFactory.registerConcreteProduct(lProcessDef,lPropertyList);

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

There might occur a problem during the registration of *ConcreteProducts*. `registerConcreteProduct()` has been called several times and each time with a different type, although parameter types 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 `construct()` must be overridden in every *ConcreteProduct*.

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 lPropertyList = new PropertyList();
lPropertyList.add(“APID”,”Process”);
lPropertyList.add(“PFID”,”Livelink-Wf-Product-Family”);

Process lProcess=(Process)lFacTrader.createProduct(lPropertyList);
```

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
lFactory.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 lFacTrader=FactoryTrader.get();
lFacTrader.register(lWfFactory,”WorkflowFactory”);
```

The above configuration concept lacks flexibility as the *PRODUCT CONSTRUCTION SERVICE* implementation has to be configured by writing source code. This might lead to a situation where the configuration task influences software productivity negatively. Therefore it would be much better to use a data driven approach. By specifying the different *ProductFamilies*, *AbstractProducts* and *ConcreteProducts* in a configuration file no additional source code has to be written, if new *ProductFamilies* shall be supported. This requires a generic configuration algorithm, which can parse the configuration file and perform the configuration task. The instantiation of the *ConcreteProduct* prototypes can raise a problem, if the implementation language does not support run time type information (RTTI). To achieve the desired flexibility the implementation language must offer a way to create instances of a class at run time as the configuration details aren’t available until the configuration file is read. In JAVA the problem can be solved by using the *Class* object. The *Class* object offers a static method `forName()` that takes the name of a class as a parameter and returns a reference to it, if it could be loaded. From the referenced *Class* object new instances can be requested by calling the method `newInstance()`. This feature can be used to create the required *PROTOTYPES*. The next example shows a configuration file that configures the *PRODUCT CONSTRUCTION SERVICE* implementation in order to support the LiveLink and Staffware *ProductFamilies*. It has been written using the eXtensible Markup Language (XML)[12]. The corresponding Document Type Definition (DTD) is illustrated in Figure 11.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE Factories SYSTEM "PDSConfig.dtd">
<Factories>
   <Factory id="WfFactory">
      <DefaultFamily familyRef="LLWfFamily"/>
```

[12] [W3C2000]

```
      <ProductFamily id="LLWfFamily">
         <AbstractProduct>ProcessDef</AbstractProduct>
         <AbstractProduct>Process</AbstractProduct>
         <AbstractProduct>WfHandler</AbstractProduct>
         <ConcreteProduct  id="LLProcessDef"
                        className="com.systor.cmt.workflow.livelink.LLProcessDef"
                        isSingleton="false"/>
         <ConcreteProduct  id="LLProcess"
                        className="com.systor.cmt.workflow.livelink.LLProcess"
                        isSingleton="false"/>
         <ConcreteProduct  id="LLWfHandler"
                        className="com.systor.cmt.workflow.livelink.LLWfHandler"
                        isSingleton="true"/>
      </ProductFamily>
      <ProductFamily id="SWWfFamily">
         <AbstractProduct>ProcessDef</AbstractProduct>
         <AbstractProduct>Process</AbstractProduct>
         <AbstractProduct>WfHandler</AbstractProduct>
         <ConcreteProduct  id="SWProcessDef"
                     className="com.systor.cmt.workflow.staffware.SWProcessDef"
                        isSingleton="false"/>
         <ConcreteProduct  id="SWProcess"
                        className="com.systor.cmt.workflow.staffware.SWProcess"
                        isSingleton="false"/>
         <ConcreteProduct  id="SWWfHandler"
                        className="com.systor.cmt.workflow.staffware.SWWfHandler"
                        isSingleton="true"/>
      </ProductFamily>
   </Factory>
</Factories>
```

Figure 10: XML configuration file

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT   Factories         (Factory*)>
<!ELEMENT   Factory           (DefaultFamily,ProductFamily+)>
<!ATTLIST   Factory id        ID        #REQUIRED>
<!ELEMENT   DefaultFamily     EMPTY>
<!ATTLIST   DefaultFamily     familyRef   IDREF    #REQUIRED>
<!ELEMENT   ProductFamily     (AbstractProduct+,ConcreteProduct+)>
<!ATTLIST   ProductFamily     id          ID       #REQUIRED>
<!ELEMENT   AbstractProduct   (#PCDATA)>
<!ELEMENT   ConcreteProduct   EMPTY>
<!ATTLIST   ConcreteProduct   id          ID       #REQUIRED
                              className   CDATA    #REQUIRED
                              isSingleton (true|false)   "false">
```

Figure 11: XML-DTD of configuration file

Known Uses

WISE: WISE allows the integration of different workflow management systems (wms) in heterogeneous and complex system environments. In order to deal with several wms in a flexible manner, the *Product Construction Service* pattern has been used to develop a generic infrastructure for the creation and the delivery of wms specific objects. WISE has been used to develop a credit risk management system for a leading Swiss bank.

RETO: RETO is a retail platform, that has also been developed for a Swiss bank. There, a similar approach as *Product Construction Service* has been used to administer concrete factories from which clients could request concrete products.

COM+: COM+ uses *Product Construction Service* for the creation of components.

eSpeak:: eSpeak[13] as part of the eServices package offered by Hewlett-Packard also uses the *Product Construction Service* pattern. E-speak is HP's open software platform for the creation of dynamic, intelligent e-services. The platform consists of the Services Framework Specification (SFS) and the Service Engine.

Related Patterns

- ***Abstract Factory***
  Product Construction Service is an extension of the *Abstract Factory* pattern as it is applicable in the same context but in a generic way. *Abstract Factory* uses inheritance where *Product Construction Service* uses composition.
- ***Product Trader***
  Product Construction Service is closely related to *Product Trader*[14] as it also allows the specification and configuration of ConcreteProducts. Further, both patterns decouple clients from the concrete construction mechanism. Nevertheless, *Product Construction Service* focuses on product families where *Product Trader* concentrates on single objects. Moreover, *Product Construction Service* addresses the construction of *Singletons*.
- ***Factory Chain***
  Factory Chain[15] also uses composition to extend functionality, but on the factory level. Concrete factories are linked together in a chain. If a factory cannot handle a request it forwards it to its successor. *Factory Chain* is a combination of *Abstract Factory* and *Chain of Responsibility*[16].
- ***Factory Method***
  Factory Method also keeps a client of an object independent of its creation. This pattern focuses on individual objects and not on product families. Nevertheless it is a base pattern of higher level patterns, like *Abstract Factory*, *Singleton* and *Prototype*. *Product Construction Service* uses the *Factory Method* pattern to retrieve the FactoryTrader object.
- ***Prototype***
  Prototype uses *Factory Method* to solve its design problem. There is no differentiation between the object and its creator. They are actually the same object. *Product Construction Service* uses the *Prototype* pattern to enable the creation of Concrete Products.
- ***Singleton***
  The *Singleton* pattern is used to make sure that there is only one instance of a class, which can be accessed globally. Access is typically realized by offering a static *Factory Method*. In the *Product Construction Service* pattern the FactoryTrader is a *Singleton*.

Acknowledgements

I'd like to thank my shepherd Wolfgang Berger for his useful remarks and my colleagues Arif Chughtai, Martin Fabini, Thomas Neumann and Ralf Steck at Systor AG for their time and help. Furthermore, I would like to thank everybody at EuroPLoP 2000 for their valuable technical comments and for the real EuroPLoP atmosphere.

[13] [eSpeak2000]

[14] [MRB1998), Product Trader

[15] [Kriha1997], p. 97ff (unfortunately not published yet)

[16] [Gamma+1995], p. 223ff.

Bibliography

[eSpeak2000] eSpeak software platform, Hewlett Packard, 2000, http://e-speak.hp.com/product/overview.shtm

[Gamma+1995]: Gamma, E. e.a., Design Patterns, Elements of Reusable Object-Oriented Software-Design, Addison-Wesley, Bonn 1995

[MRB1998]: Martin, Riehle, Buschmann (eds), Pattern Languages of Program Design 3, Addison-Wesley, 1998

[Riehle1997]: Riehle, D., Composite Design Patterns, OOPSLA 1997, ACM Press, p. 218-228, 1997

[Sommerlad+1998] Sommerlad, P. & Ruedi, M., Do-it-yourself Reflection, IFA Informatik, Zürich, Submitted to EuroPLoP´98

[Kriha1997]: Kriha, W., Frameworking, Strukturen und Verbindungen in einem Framework, SYSTOR AG, Basel 1997

[W3C2000]: eXtensible Markup Language, W3C Architecture Domain, http://www.w3.org/XML/