# Late Creation
# A Creational Pattern

**Dirk Bäumer**
RWG GmbH, Germany
Räpplenstraße 17, D-70191 Stuttgart
E-mail: dirk_baeumer@rwg.e-mail.com

**Dirk Riehle**
UBILAB, Union Bank of Switzerland
Bahnhofstrasse 45, CH-8021 Zürich
E-mail: riehle@ubilab.ubs.ch

## Intent

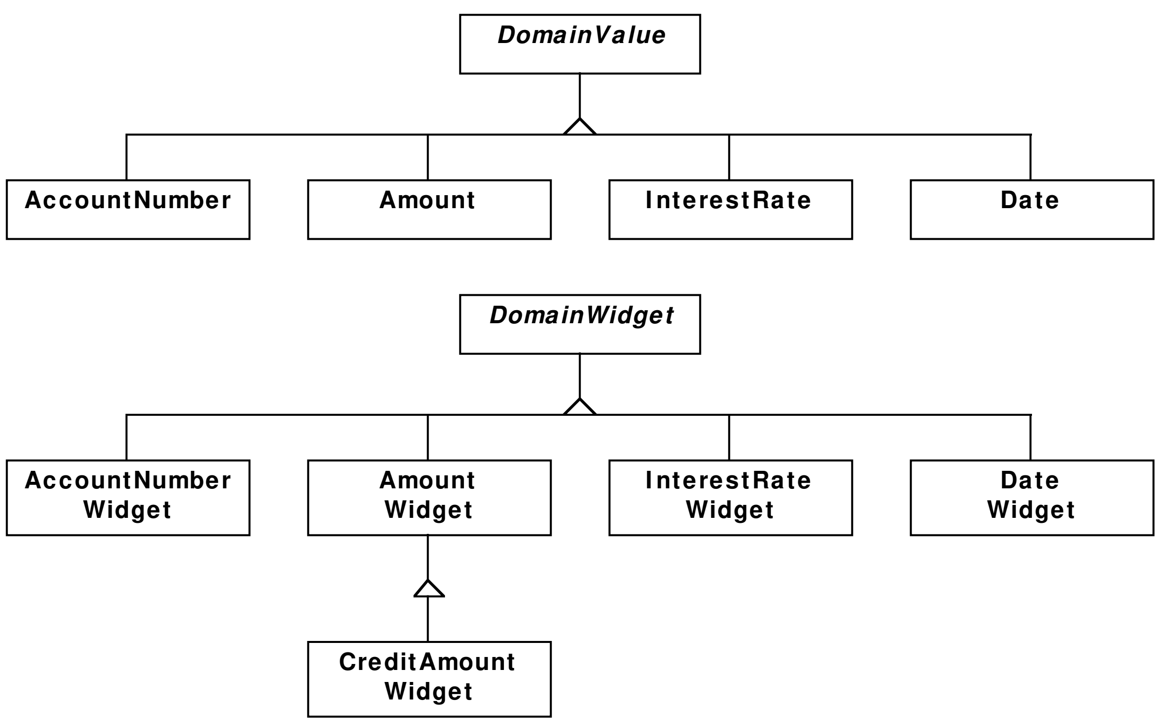
Let clients create objects by naming an abstract superclass and by providing a specification. Late creation decouples the client from the product easing the evolution and adaptation of class hierarchies as well as easing the configuration of applications.

## Also Known As

Virtual Constructor

## Motivation

Suppose you have designed a class hierarchy of domain value types like AccountNumber, Amount, InterestRate or SocialSecurityNumber. These value types will be used in several applications. For example, one such application will present a form on the screen and let users edit its fields. The value types of the fields correspond to the domain value types. Since each value type has its own semantics and editing constraints, you might want to provide specialized widgets for each of these value types as shown in the following figure.




Given some abstract form object with all the fields and associated value types within, how do you create the widgets corresponding to the value types?

**Dirk Bäumer and Dirk Riehle. “Late Creation.” Submitted to PLoP ‘96.**

You might be tempted to write a big case statement for all value types which creates a widget for a given value type, but this cumbersome and might require frequent changes and enhancements. You might use a Factory Method for the value types so that each value type can return a default widget for it. But then you will only be able to provide a single widget for a value type. Moreover, if the value types are used in a non-interactive environment, like nightly batch runs, you make the batch application link the window system even though it isn't needed. You might also consider using an Abstract Factory, but this isn't better than the naive approach, because the form object will have to do a case statement on the value type to call the appropriate widget creation operation of the Abstract Factory.

It is much better to go directly to the abstract Widget class and ask it to return an instance of a subclass which fits a given value type best. Late Creation is all about how to do this flexibly and efficiently.

Suppose, the Widget class offers a class operation that lets you do this. In C++, it might look like

```
static DomainWidget* DomainWidget::createFor(DomainValue*);
```

and in Smalltalk it might look like

```
DomainWidget class>>createFor: aDomainValue.
```

Internally, the Widget class might maintain a table of all its subclasses which it holds in a dictionary indexed by the class of the value types. `createFor` can be easily implemented then: The `DomainWidget` class simply takes the value type's class as a key for the dictionary, which will return the widget subclass specifically designed for the value type. It can then go ahead and create an instance of it which it returns to the client. This process is fast and requires only constant time.

We call this process Late Creation since it resembles late binding. The actual class which is to be instantiated is determined only at runtime, and it is accessed using abstract classes only.

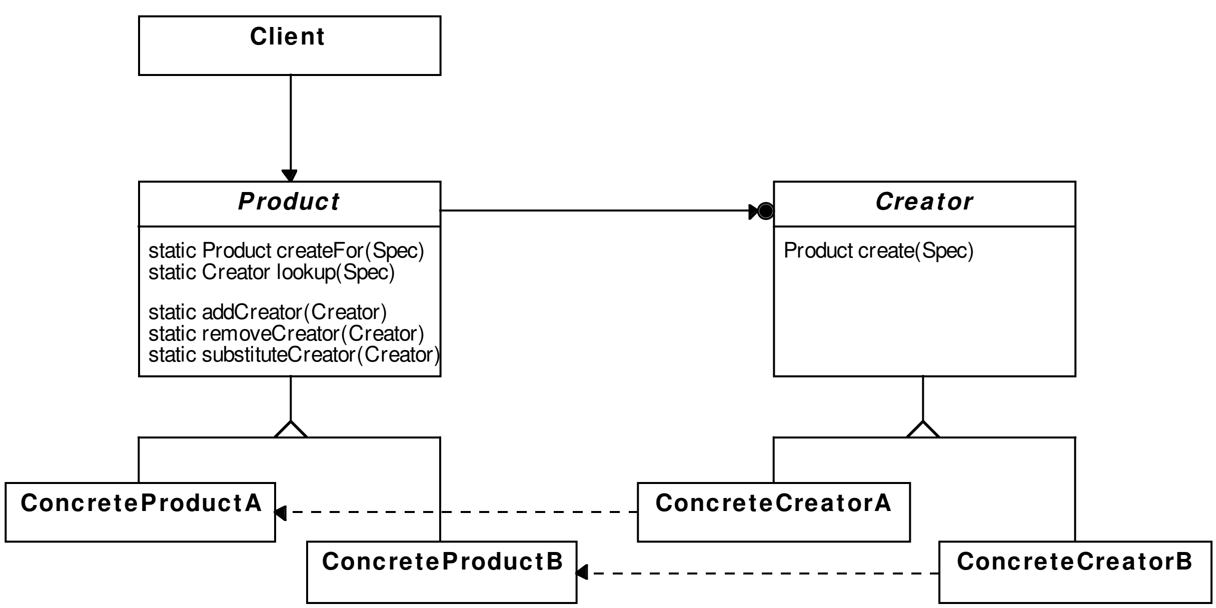
# Applicability

Use Late Creation, if

- you don't want to create an object directly but only want to name its abstract superclasses and provide a specification for it,
- you want to configure the kind of products created in the context of some other objects instead of hardcoding them by a Factory method in those other objects.

Do not use Late Creation as a replacement for Factory Methods or direct object creation.

# Structure



# Participants

**Client**

- initiates the creation process by providing the abstract Product class with a specification.

**Product**

- defines the interface to a class hierarchy of which objects are to be created;
- provides operations which let clients create instances of subclasses by providing specifications;
- implements a mapping from a specification to the Creator for instances of the class matching the specification;
- provides operations to configure the mapping by some configuration mechanism.

**ConcreteProduct**

- represents a concrete Product class hidden from the client.

**Creator**

- defines the interface to create instances of a the Product class.

**ConcreteCreator**

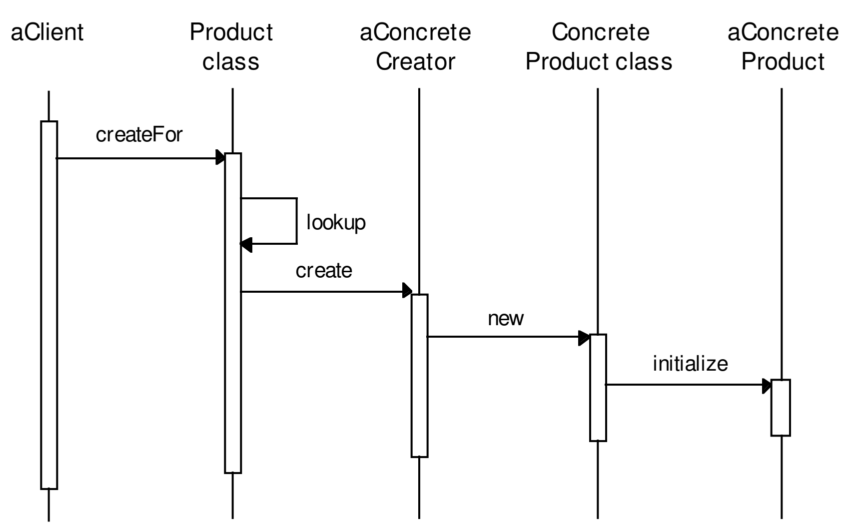
- represents a concrete Creator responsible for a specific ConcreteProduct class;
- knows how to create instances of the ConcreteProduct class.

**Specification**

- represents a specification for a ConcreteProduct class;
- is used as the lookup argument for ConcreteCreators;
- can be as lightweight as an integer and as heavyweight as a propositional calculus formula.

# Collaborations

- The Client asks the abstract Product class for an instance matching a Specification.
- The abstract Product class looks up a ConcreteCreator for a given Specification.
- The abstract Product class delegates the actual creation process to a ConcreteCreator.
- A ConcreteCreator creates a ConcreteProduct instance.
- A ConcreteProduct class provides its semantics so that a ConcreteCreator can get a Specification of it.



# Consequences

Late Creation has the following consequences and liabilities:

1. *Late Creation does not compete with Factory Method but rather serves as its companion.* Late Creation works well where Factory Method works poorly and vice versa. Factory Methods often introduce cyclic dependencies between the Product and the Creator, in particular if the Product is going to work on the Creator later on. This is the case with applying Factory Method in the introductory example where a value object would create a widget object. This is the case with a container creating its iterators. This is the case with a model object creating its views. If a different product is required, the creator has to be changed. The widget class, the container class and the model class would have to be changed. Furthermore, no two different products can coexist and be selected due to some dynamic selection criteria.
2. *Late Creation is no replacement for regular object creation.* Late Creation should only be used when the protocol of the abstract Product class is powerful enough and the gained creation in flexibility is really needed. Late Creation should not be used to undermine a static type system, for example by providing a general create operation which takes the class name as a string and returns an instance of the denoted class.
3. *Products can be configured for specific domains and even at runtime.* Late Creation lets you configure the type of products created via the abstract product class by adding, substituting and removing concrete creators for concrete product classes. This both leads to highly flexible adaptation and configuration of domain specific applications, as well as the need for such configuration mechanisms.
4. *Advanced configuration mechanisms are needed.* When applying Late Creation, objects of classes can and will be created which are not statically referenced starting with some root class. Thus, a linker might consider those classes as superfluous and decide not to link them to a final executable. Advanced configuration techniques have to be deployed to prevent runtime failures due to missing classes. See the Implementation section for techniques of dealing with this problem.
5. *Specifications can be ambiguous.* Depending on the type and implementation of a specification, not only a single class can match it but rather a set of equivalent classes (with respect to the specification). This does not pose problems, since any class matching the requirements will do the job. However, ambiguities might indicate that the specification itself was either imprecise or underdetermined.
6. *Development of new product classes is eased*. It is very easy to add new concrete products to an existing framework—it does not require any changes to the framework. Subclasses of product classes can be written without having to change existing code. Only the configuration code or makefiles have to be changed.
7. *Dynamic link libraries must be considered explicitly.* Classes from dynamically linked libraries are not directly accessible, simply because they are not contained in the runtime image. However, the creator objects are usually very lightweight. They are therefore a fine means of representing not yet loaded classes. The creation procedure of a creator for class in a dynamically linked library might have to load that class first, but this might be exactly what you want. Creators and the CreatorManager can be used effectively to hide the factoring of large applications into DLL's on the expense having to deal with complex configurations issues.
8. *Providing construction parameters in the early object building process requires overhead.* Since the creation procedure is the same for all product classes of a certain class hierarchy, it might be difficult to supply object constructors with parameter needed for the early object building process. One solution is to break up the creation process into a very basic part which only allocates the object memory, and another initialization part, which is client triggered. Another solution is to package all potentially relevant parameters into a constructor parameter object and to hand it over to specialized creator objects which in turn pass it over to freshly create product class instance.

# Implementation

The implementation of Late Creation requires four main dimensions to be considered: the realization of the mapping from Specification to Creator, the realization of the Creator itself, the realization of Specifications and the needed configuration mechanisms.

1. *Implementing the mapping from Specification to Creator.* The mapping can be realized by a simple dictionary in the abstract Product class, or by delegating the task to a product specific CreatorManager, or by delegating the task to a universal CreatorManager.

- *Using a simple dictionary* puts the management of the mapping into the abstract Product class. This is only advisable, if Late Creation is applied for the Product hierarchy only, and if the handling of specifications is very specialized for this particular class hierarchy. Otherwise consider using a CreatorManager.
- *Using a CreatorManager* takes the burden of managing specifications, creators, and the mapping from specifications to creators from the Product class. Each class hierarchy can have its own CreatorManager which deals with specifications for this class hierarchy. This solution is both flexible and offers good code reuse.
- *Using a single global CreatorManager* centralizes all specifications and all class semantics in one single place. This goes beyond the class hierarchy specific CreatorManager and should be applied if complex class retrieval and runtime configuration issues must be considered.

2. *Implementing Creators.* Creators can be implemented as prototypes, class objects, or specific dedicated Creator objects, for example by using C++ templates to generate them. Generally speaking, you will want to avoid introducing a Creator class for every Product class, so consider using one of these variants.

   - *Using prototypes or class objects* amounts to about the same: These objects come for free with any major system and usually provide a clone or a create operation which can carry out the actual creation of the Product instance. The clone operation must be robust with respect to copying.
   - *Using dedicated Creator objects.* In C++, you can use macros or templates to generate the Creator objects. We present a C++ template based version as a type-safe solution. An analysis of the Creator's responsibilities leads us the following definition of Creator and ConcreteCreator:

   ```
   template<class ProductType, class SpecType>
   class Creator
   {
   public:
      Creator(SpecType aSpec) : _aSpecification(aSpec) {}
      SpecType getSpecification() { return _aSpecification; }
      ProductType * create() =0;
   private:
      SpecType _aSpecification;
   };

   template<class ProductType, class ConcreteProductType, class SpecType>
   class ConcreteCreator : public Creator<ProductType, SpecType>
   {
   public:
      ConcreteCreator(SpecType aSpec) : Creator<ProductType, SpecType>(aSpec) {}
      ProductType * create() { return new ConcreteProductType; }
   }
   ```

   The Creator template requires two formal arguments, the ProductType and the SpecificationType. Usually, the Specification itself is not maintained by the Creator, rather it is the argument to the mapping which leads to the Creator in the first place. However, for management purposes in CreatorManagers it is convenient to store a specification with the Creator, given that there is only a single specification which leads to the specific Creator.

   The ConcreteCreator template adds the code for actually creating an instance of a ConcreteProduct class. A closer look at the templates reveals that they are fairly simple, essentially only defining interfaces and the create operation for concrete product classes.

   These templates can only be used for product classes where the constructor doesn't need any parameters. In case that parameters must be passed to the Product class constructor, additional templates are needed. A replication of the above two template classes is the result.

3. *Implementing Specifications.* Specifications can be realized either directly in the parameters of the creation operation by some native value types, or by encapsulating them as classes of their own. This question affects the implementation of the mapping:

   - *Using built-in value types* like integers, character strings, etc. often requires additional support like free floating hash-operations, etc. Essentially, you must define operations which make the value types usable as lookup arguments for the mapping.

- *Using explicit Specification objects* allows you to define the proper interface for specifications once and reuse it for every new specification. Moreover, CreatorManagers can be written based on this explicit interface and reused as well. Such a Specification interface might look like this:

```
class Specification
{
public:
   // returns Product class, that is the root class of classes
   // addressed by this type of specification
virtual Class* getProductClass() =0;
   // initialize specification to match semantics of a given Product class
virtual void initForProduct(Class* pc) { this->adaptTo(pc); }
   // initialize specification to match semantics of a given Product class
virtual void adaptTo(Class*);
   // match two specifications for equivalence
virtual bool matches(Specification*) =0;
};
```

4. *Configuring frameworks and applications.* Late Creation creates objects only indirectly by interpreting specifications. Thus, ConcreteProduct classes are not directly referenced by framework or application code, but only abstract interface classes like the Product class are. As a consequence, it must be specified explicitly which classes are to be linked to a specific application, because linkers might not link classes which are not directly referenced by code reachable from some root object (or main). Thus, systems must be configured explicitly, either through configuration code or on a meta-level, for example by explicit commands to the linker.
   - *Using makefiles* is a straightforward approach. For every application, the set of ConcreteProduct classes to be linked has to be specified explicitly. This can be done by issuing direct commands to the linker. Since dependencies between ConcreteProduct classes from different class hierarchies might exist (covariant redefinition), care must be exercised in order not to forget dependent classes.
   - *Using source code to configure systems* is straightforward as well: You might simply write a configuration operation in the system's root class, the main application class, or close to the main operation which references the classes required to be linked once. The linker will take care then that the right classes are linked to the final executable.
   - *Using register objects.* If you want to support Late Creation for all classes linked to the system you should consider using register objects. An object of a register class is responsible to add a ConcreteCreator object with a specification to the Product class.
   - *Using configuration scripts.* In large projects, where a lot of applications are developed, it is indispensable that the configuration of the system can be understood easily. When using Late Creation in C++, we recommend the use of configuration scripts. The goals of configuration scripts are:
     – to represent one place that describes the configuration of the system.
     – to reference all needed ConcreteProduct classes, so that the linker includes them into the application.

     An easy way to satisfy these goals are to create the ConcreteCreator and Specification objects inside a configuration script and add these objects to the Product class depending on the needed configuration of the system. The Sample Code section presents some examples.

Next to these four dimensions of implementing Late Creation, you might also consider the following issues:

1. *Providing convenience operations.* The abstract Product class can be used to hide the implementation of a specification as a Specification instance from the client. The client simply calls a createFor operation with parameters which are native to the specification, and the implementation of createFor in the Product class converts these parameters into a Specification object.
2. *Providing initialization parameters.* Sometimes objects require elaborate initialization parameters which must be available during the early object building process. These parameters can be packaged into the Specification objects, or they can be passed along a chain of operation calls which then must provide matching parameter lists (which asks for code generation as discussed above).

# Sample Code

As a first example, we will present a simple implementation using prototypes as Creators and a dictionary as the mapping from Specification to ConcreteCreator. The dictionary is directly maintained by the Product class.

Assume, you are reading objects from a stream. An initial string indicates the class of the object to be read. Thus, the class name represents the specification for the class to be instantiated, and we use Late Creation to instantiate it.

`Streamable` is the abstract Product class for all objects that can be read from a stream. `String` is the specification type, and we assume that every subclass of `Streamable` can provide a prototype.

The `Streamable` class might then implement `createFor` like this:

```
Streamable* Streamable::createFor(String& className) {
   Streamable* prototype = lookup(className);
   return prototype->clone();
}
```

Lookup simply retrieves the prototype for a given class name from a dictionary maintained by the Product class (a static member variable in C++ or a class variable in Smalltalk):

```
Streamable* Streamable::lookup(String& className) {
   return mapping[className];
}
```

A simple configuration scheme might be to make the prototypes register themselves at the `Streamable` class during system startup time (when the prototypes are created):

```
Customer::Customer(CustomerPrototypeConstructorIndicator* dummy) {
   Streamable::addCreator(this);
}
```

`Streamable` implements `addCreator` by using the class name to put the prototype into the dictionary:

```
Streamable::addCreator(Streamable* prototype) {
   mapping[prototype->getClassName()] = prototype;
}
```

This implementation uses prototypes as Creators, a simple built-in value type specification, and a static registering and configuration scheme. It is a rather lightweight approach which works well if Late Creation is to be applied to single class hierarchies with only simple specification requirements.

As a second example we will use C++ templates to illustrate an implementation of the introducing domain widget and value problem. In this example, specifications are realized by C++ typeid's denoting the class of the `DomainValue`. The configuration scheme of the system is realized by configurations scripts. The mapping from the Specification to the Creator is realized by a simple dictionary.

We use the templates from the Implementation section:

```
DomainWidget* DomainWidget::createFor(DomainValue* aValue) {
   Creator<DomainWidget, type_info&>* aCreator = mapping[typeid(aValue)];
   return aCreator->create();
}

DomainWidget::addCreator(Creator<DomainWidget, type_info&>* aCreator) {
   mapping[aCreator->getSpecification()] = aCreator;
}
```

A default configuration to map the typeid of the `DomainValues` to the ConcreteCreator object for the `DomainWidgets` is as follows:

```
StandardDomainWidgetConfiguration() {
   ...
      // map Amount on AmountWidget
   DomainWidget::addCreator(
      new ConcreteCreator<DomainWidget, AmountWidget, type_info&>(typeid(Amount)));
```

```
      // map Currency on CurrenyWidget
   DomainWidget::addCreator(
      new ConcreteCreator<DomainWidget, CurrencyWidget, type_info&>(typeid(Currency)));
   ...
}
```

If we want to offer a new configuration for an application we only have to write an new configuration script to make the required changes. As an example, consider a credit application which uses a special domain widget for the domain value Amount. To add the new widget to the application, you have to implement the new domain widget and then use a configuration script to register it. The new configuration script might look like this:

```
CreditConfiguration() {
   ...
      // use the specific CreditAmountWidget for the domain value Amount
   DomainWidget::substituteCreator(
      new ConcreteCreator<DomainWidget, CreditAmountWidget, type_info&>(typeid(Amount)));
   ...
}
```

The system's root object, or the main operation, must call the DomainWidgetConfiguration and the CreditConfiguration operation to install the necessary Creator objects. Only these calls are needed, no further code changes are required.

As a final example, we will use Smalltalk to implement a full CreatorManager that can be reused by every class hierarchy that wants to provide Late Creation at its root for any number of different specifications. This example is derived from (Riehle and Schnyder; 1995) where a more detailed discussion can be found. This implementation uses a general ClassManager which maintains a picture of the semantics of all classes in the system by means of specifications. The ClassManager organizes these specifications in such a way that it can easily lookup the classes matching a given specification.

Every Product class can provide convenience operations for Late Creation which are implemented in terms of the system-wide ClassManager, a Singleton. Classes take over the role of Creators, Specifications are implemented as subclasses of a general Specification class, and the CreatorManager is realized by the ClassManager just introduced.

The `ClassManager` provides a mapping from a specification to a class object for every specification type. Specification types are represented by the classes in a `Specification` class hierarchy. The mapping is implemented as a dictionary which takes a `Specification` instance as a key and returns a collection of all classes that match the specification. Since the specification type is represented as a class, it can be used in a further dictionary to retrieve the dictionary containing the specification to classes mapping.

The ClassManager has two main lookup methods:

```
ClassManager>>getSpecificationDictionary: aClass
   "returns dictionary representing a specification to classes mapping"
   ( self specTypeDict includesKey: aClass )
      ifTrue: [ ^self specTypeDict at: aClass ]
      ifFalse: [ ^nil ]

ClassManager>>getClassCollection: aSpecification
   "returns collection of classes matching aSpecification"
   | specDict |
   specDict := self getSpecificationDictionary: aSpecification class.
   ( specDict includesKey: aSpecification )
      ifTrue: [ ^self specDict at: aSpecification ]
      ifFalse: [ ^nil ]
```

The first method returns the mapping dictionary for a given specification type, and the second method returns a collection of all classes matching the given specification. The implementation of `createFor` becomes a simple two stage retrieval process now. The Product class, for example `DomainWidget`, simply does the following:

```
DomainWidget>>createFor: aDomainValue
   "create Specification instance and delegate to CreatorManager"
   | aSpecification |
   aSpecification := DomainWidgetSpecification new: aDomainValue class.
   ^self CreatorManager createFor: aSpecification
```

```
ClassManager>>createFor: aSpecification
   "create object for given specification"
   | classCol |
   classCol := getClassCollection: aSpecification.
   ( classCol notNil )
      ifTrue: [ ^classCol first new ]
      ifFalse: [ ^nil ]
```

The actual problem is how to set up the dictionaries so that the retrieval and creation process can be carried out so easily. This is a two step process. In the first step, the `ClassManager` collects all classes from the `Specification` class hierarchy. In the second step, it calculates the mapping for every `Specification` class; this is done by traversing the Product class hierarchy as indicated by the `Specification` class, and by creating a `Specification` instance for every Product class.

This process is based on information which has to be provided by the specification classes themselves. Thus, `Specification` declares the following class and instance methods:

```
Specification class>>getProductClass
   "returns Product class"
   self subclassResponsibility

Specification class>>newFromClient: anObject
   "used by clients which supply the specification information"
   self subclassResponsibility

Specification class>>newFromManager: aClass
   "used by ClassManager to instantiate spec for a ConcreteProduct class”
   ^super new adaptTo: aClass

Specification>>adaptTo: aClass
   "adapts spec instance to given product class"
   self subclassResponsibility

Specification>>matches: aSpecification
   "matches with another specification"
   ^( self class = aSpecification class )
```

These methods must be overwritten in subclasses in order to fit a certain specification’s semantics. For `DomainWidgetSpecification`, this leads to the following code:

```
DomainWidgetSpecification class>>getProductClass
   ^DomainWidget

DomainWidgetSpecification class>>newFromClient: aDomainValueClass
   self domainValueClass: aDomainValueClass

DomainWidgetSpecification>>adaptTo: aDomainWidgetClass
   self domainValueClass: aDomainWidgetClass getDomainValueClass

DomainWidgetSpecification>>matches: aSpecification
   ( super matches: aSpecification ) ifFalse: [ ^false ].
   ^( self domainValueClass = aSpecification domainValueClass )

DomainWidgetSpecification>>hash
   ^domainValueClass symbol
```

When created by a client or a convenience operation of `DomainWidget`, the `DomainWidgetSpecification` instance receives the `DomainValue` class the new `DomainWidget` instance will have to fit. When created by the `ClassManager`, the `DomainWidgetSpecification` instance is created for a specific `DomainWidget` class which it asks for the `DomainValue` class it has been written for. The last method is used to build the mapping for domain values to domain widgets. The `ClassManager` traverses the `DomainWidget` class hierarchy, creates a `DomainWidgetSpecification` instance for each class, and puts the class into the mapping dictionary with the specification instance as the key. The hash method then maps a client created `DomainWidgetSpecification` instance on a collection of `DomainWidget` classes.

This approach is based on the assumption that the specification instance is able to retrieve enough information from the Product class to serve as a key in the mapping. In the domain widget example, this required the `DomainWidget` class to provide a `getDomainValueClass` method which returns the domain value class, the domain widget class has been designed for. If this is not an option, the class semantics of Product classes must be specified external to that class, for example in a database.

This approach lets us introduce new specifications simply by introducing a new specification subclass. Implementing Late Creation for object streaming as shown in the first example can be done by simply creating a `ClassSymbolSpecification` which looks like this:

```
ClassSymbolSpecification class>>getProductClass
   ^Object

ClassSymbolSpecification class>>newFromClient: aSymbol
   self classSymbol: aSymbol

ClassSymbolSpecification>>adaptTo: aClass
   self classSymbol: aClass symbol

ClassSymbolSpecification>>matches: aSpecification
   ( super matches: aSpecification ) ifFalse: [ ^false ].
   ^( self classSymbol = aSpecification classSymbol )

ClassSymbolSpecification>>hash
   ^classSymbol
```

The general `ClassManager` is a mighty approach for avoiding the hardcoding of creation dependencies in the classes implementations themselves. It has served us well in making our frameworks easier adaptable and configurable.

# Known Uses

Coplien reports about the *generic exemplar idiom* in (Coplien, 1992). In this idiom, Creators are represented through prototypes, and the mapping is implemented as a list traversed by the Product class. As illustrated in the first example in the Sample Code section, a prototype registers itself with the Product class.

Berczuk reports about the use of a *hierarchy of factories* (Berczuk, 1996). In this pattern, Creators are represented as Abstract Factories, which are connected with each other in parallel to the Product class hierarchy. A Product is looked up by traversing the Abstract Factory hierarchy top down. This is practically equivalent to traversing a class hierarchy in search of a class matching a specification.

Lortz and Shin report about concepts for class hiding (Lortz and Shin, 1994). These concepts are similar to Coplien's generic exemplar idiom. They are used in the context of a real-time database system for creating specific implementations of container classes.

Stroustrup reports about the notion of *Virtual Constructor* (Stroustrup, 1992). This denotes the process of calling the constructor of an abstract superclass and receiving an instance of a subclass. This corresponds exactly to the idea behind Late Creation, even though Stroustrup does not elaborate as much as is done here.

Lange reports about the use of an *Abstract Constructor* which is again equivalent to Coplien's generic exemplar idiom (Lange, 1996). The Abstract Constructor was used in the design of a medical system by Hewlett-Packard.

Weinand and Gamma use a ClassManager in ET++ (Weinand and Gamma, 1994), which is similar to our first example in the Sample Code section.

An early C++ macro based version has been used in the Tools and Materials frameworks developed at the University of Hamburg (Riehle and Züllighoven, 1995).

The C++ template solution presented in the Sample Code section has been used in the GeBOS series of banking projects (Bäumer et al., 1996) with big success. It is an important asset in making the frameworks consisting of more than 2000 classes adaptable, configurable and, eventually, manageable.

The general ClassManager approach presented as the last example in the Sample Code section has been used in a Tools and Materials framework developed at UBILAB (Riehle and Schnyder, 1995).

# Related Patterns

Late Creation is similar to the generic exemplar idiom of Coplien (1992), the abstract constructor of Lange (1996), and the hierarchy of factories of Berczuk (1996). These patterns are based on implementations which are slower and less easily configurable than the one suggested by Late Creation. We therefore think of Late Creation as a sound consolidation of those patterns, in particular since those implementations have been part of our implementation experience over the last three years as well.

We think of Late Creation as a fourth fundamental creational pattern next to Factory Method, Abstract Factory and Prototype.

# Bibliography


**Bäumer et al. (1996)**
Dirk Bäumer, Rolf Knoll, Guido Gryczan and Heinz Züllighoven. "Large Scale Object-Oriented Software-Development in a Banking Environment—An Experience Report." LNCS 10??, ECOOP '96, *Conference Proceedings.*

**Berczuk (1996)**
Steve Berczuk. "Organizational Multiplexing: Patterns for Processing Satellite Telemetry with Distributed Teams." *Pattern Languages of Program Design, 2*. Edited by John M. Vlissides, James O. Coplien and Norm L. Kerth. Reading, Massachusetts: Addison-Wesley, 1996. 193-206.

**Coplien (1992)**
James O. Coplien. *Advanced C++: Programming Styles and Idioms.* Reading, Massachusetts: Addison-Wesley, 1992.

**Gang-of-four (1995)**
Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. (1995). *Design Patterns: Elements of Reusable Design*. Reading, Massachusetts: Addison-Wesley.

**Lange (1996)**
Manfred Lange. "Abstract Constructor." Submitted to EuroPLoP '96.

**Lortz and Shin (1994)**
Victor B. Lortz and Kang G. Shin. "Combining Contracts and Exemplar-Based Programming for Class Hiding and Customization." OOPSLA '94, *ACM SIGPLAN Notices* 29, 10 (October 1994): 453-467.

**Riehle and Züllighoven (1995)**
Dirk Riehle and Heinz Züllighoven. "A Pattern Language for Tool Construction and Integration Based on the Tools and Materials Metaphor." *Pattern Languages of Program Design*. Edited by James O. Coplien and Douglas C. Schmidt. Reading, Massachusetts: Addison-Wesley, 1995. 9-42.

**Stroustrup (1992)**
Bjarne Stroustrup. *The C++ Programming Language.* Reading, Massachusetts: Addison-Wesley, 1991.

**Weinand and Gamma (1994)**
André Weinand and Erich Gamma. "ET++ — a Portable, Homogenous Class Library and Application Framework." *Computer Science Research at UBILAB*. Edited by Walter R. Bischofberger and Hans-Peter Frei. Konstanz: Universitätsverlag Konstanz, 1994. 66-92.