# Patterns of Object Creation

Dirk Riehle, dirk@riehle.org, www.riehle.org

Brian Foote, foote@cs.uiuc.edu

James Noble, kjx@comp.vuw.ac.nz

## Abstract

This paper puts a set of well-known and some new patterns together to form a pattern language of object creation. The object creation patterns presented are *Creation Method*, *Factory Method*, *Conversion Method*, *Cloning Method*, *Trading Method*, *Object Factory*, *Abstract Factory*, *Builder* and *Prototype*. Language context is provided by the *Initialization Method*, *Finalization Method*, *Cascaded Delete*, *Default Implementation*, *Class Object*, *Exemplary Instance*, and *Specification* patterns. The purpose of this presentation is to separate out the different patterns rather than to provide an introduction to Object Creation to the novice reader. The pattern language is aimed at intermediate and expert developers.

## 1 Overview

This pattern language presents nine object creation patterns and a number of related patterns. The nine object creation patterns form the core of the language. This core has two parts: method patterns and object patterns. The method patterns talk about how to design and implement a method that creates an object, and the object patterns talk about how to have a dedicated factory object that creates other objects. Figure 1 shows these patterns and their relationships.

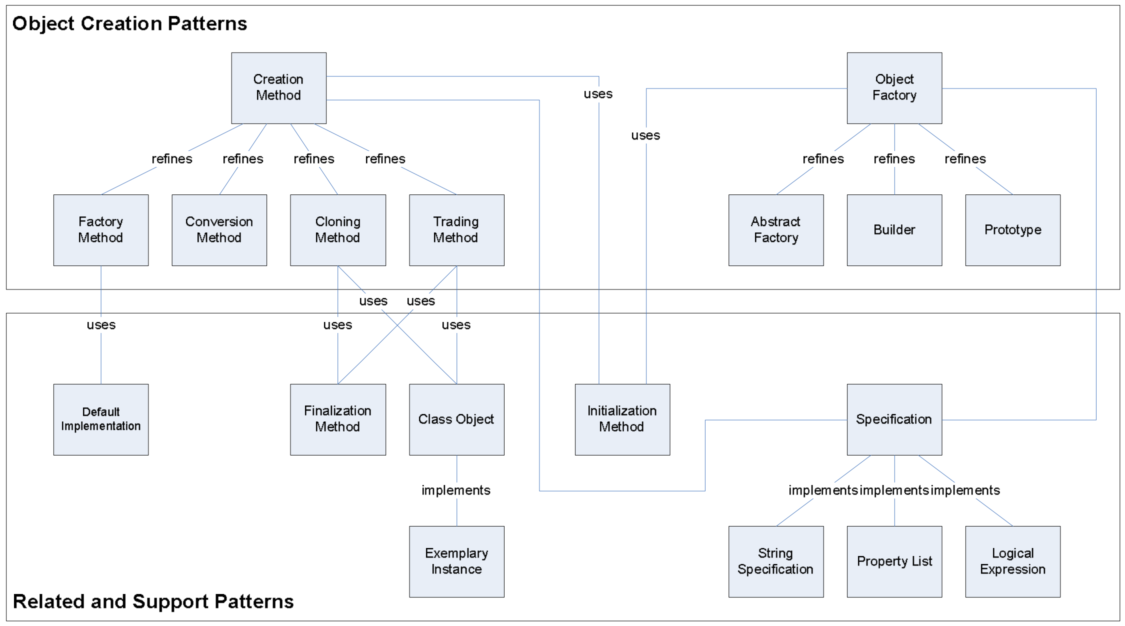


Figure 1: Outline of pattern language, showing pattern relationships

Needs to be updated, I don't have visio. Logical Expression -> full spec.

## 1.1 Audience

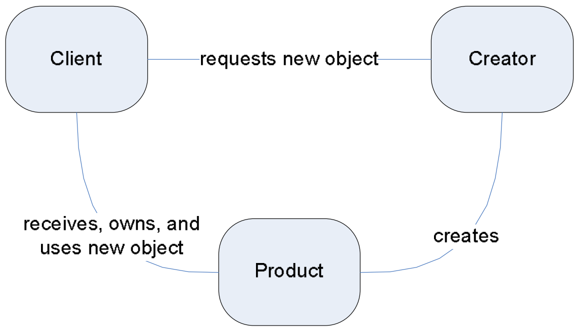
The language is aimed at moderately experienced software developers; it is helpful if you already know and are familiar with the common creational patterns, such as Factory Method and Abstract Factory [Gamma+95]. If you are a novice, you will find it difficult to use these patterns straight away. Where appropriate, we refer readers back to the original sources. This allows us to keep the details on each pattern quite brief, to focus on the vocabulary and structure of creational patterns, the relationships to other patterns and the differences between them.

Please be aware, however, that we provide a more fine-grained vocabulary of the object creation space than the Design Patterns book, distinguishing patterns more clearly from each other.

## 1.2 Object roles

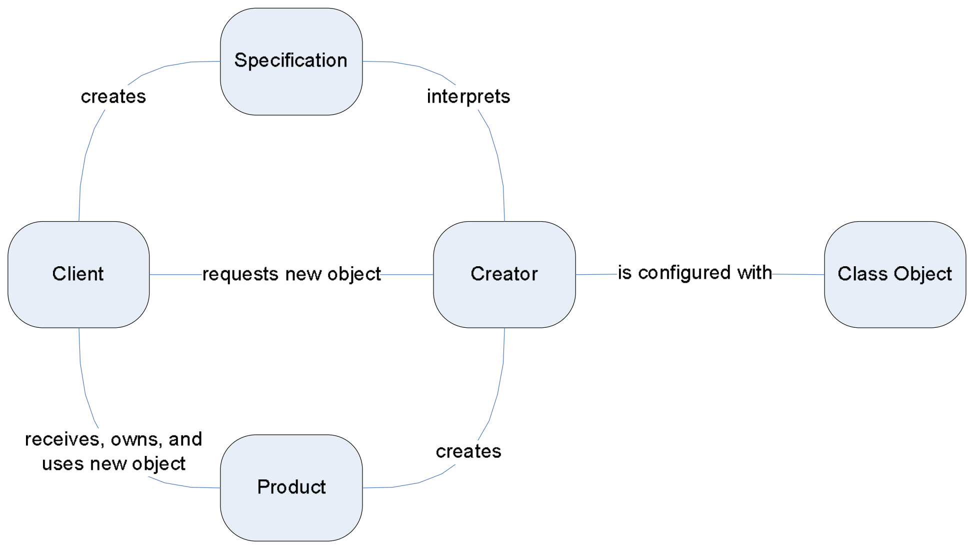
The best way to understand object creation is to understand the roles that objects play in the creation process. The three key roles are Client, Creator, and Product. The Client wants a new object; the Creator can provide new objects; and the Product is the new object the Creator returns to the Client.

Client, Creator and Product are *roles* that objects play in an object-oriented program, not classes or objects themselves. Classes only specify the roles that their instances can play. For example, in *Factory Method*, the Client and the Creator role may be played by the same object (but don't have to), or in *Prototype*, the Creator and Product role are played by different object of the same class.



Basic model of object roles and their relationships during the object creation process

These three roles are not the only participants in the creational patterns. Sometimes the client describes its requirements using a Specification, and sometimes the Creator has been configured with a Class Object or Exemplary Instance. But at the most basic level, all these patterns are about a Client asking Creator to make them a Product.



More elaborate role model of object roles and their relationships during the object creation process

## 1.3 Context and forces

The patterns of this language have to deal with a number of forces—constraints on why a pattern may be applicable, or benefits or liabilities from using the patterns.

- *Coupling*. Fundamentally, these patterns help you decouple clients from the products they create. There are a number of ways clients can be coupled to products, and different patterns tackle this in different ways:
  - What class should the product object be?
  - How should the product be initialized — with what values for variables?
  - To which other objects should the product be related?
  - Which other objects know the information that the product object will need?
- *Brevity*. These show how to create objects using code that is written once and only once.
- *Flexibility*. By reducing coupling, these patterns can make programs more flexible, particularly at runtime. That is, not only do the patterns allow you to decouple the decisions about the product from the client, but they also allow your program to revisit those decisions every time a new object is created.
- *Extensibility*. Also by reducing coupling, these patterns can be used to increase the extensibility of the system. A common form of extensibility — e.g. in the *Strategy* or *Template Method* patterns — is to extend the system as a whole by subclassing and then using one or more of the classes that it defines. But this requires some way of creating objects of the new subclasses, not the existing superclasses. These patterns can help programmers do this.
- *Safety*. Programming language mechanisms such as constructors or class methods cannot by themselves create objects configured the way programmers need them. Initializing objects to a known state (all null or all zero) is certainly preferable to handing out uninitialised memory from malloc(), but that just means programs might crash with null pointer exceptions or divide-by-zero errors rather than taking an operating system exception or crashing the whole machine: It doesn't mean your program will execute correctly. These patterns help programmers ensure that every object is created into a correct, consistent, and stable state, immediately ready to use by the rest of the program.
- *Complexity*. These patterns often make your program more complex, thus harder to read and understand (especially if you don't understand which patterns are being used and why). The simpler patterns, like *Factory Method*, are much less likely to gratuitously complicate things than e.g. a Builder, Trader, or External

Configuration. Especially with the more complex patterns, be sure the benefits (reduced coupling, flexibility, extensibility) outweigh the costs imposed by the patterns themselves.

## 1.4 Roadmap

The remainder of this paper presents the patterns shown in Figure 1 above. We begin with the most basic pattern, *Creation Method*, and then walk through its variants *Factory Method*, *Conversion Method*, *Cloning Method* and *Trading Method*. We then continue on to *Object Factory* and discuss its *Abstract Factory*, *Builder*, and *Prototype* variants.

# 2 Method Patterns

The *Creation Method* pattern tells you to create a method for creating product objects. *Conversion Method*, *Factory Method*, *Cloning Method*, and *Trading Method* then provide different ways of doing so. With each successive pattern variant, more power is given to the client requesting the new product. While *Factory Method* does not allow the Client to have any say in what kind of product it gets back, *Trading Method* lets the Client specify to a large degree the properties of the desired new object. Finally *Initialization Method* allows a Client to configure a Product separately from its creation.

## 2.1 Creation Method

**Problem** How to avoid creating redundant code for creating objects.

**Context** Different places within one or several clients need to create products.

The object creation code itself is relatively simple.

**Solution** Put the object creation code into one method, a *Creation Method*, and make everyone call this method.

**Roles** The client is the object that needs the product.

The creator should be the object in the system that knows the most about the object to be created. Sometimes this may even be the client itself.

**Comments** Creation methods can generally be implemented using the object creation features of the employed programming language, for example, by calling new(). Creation methods are useful even if the client could directly create the product because they localize creation code.

If the client code resides in more than one class, you may want to introduce an *Object Factory* first, for which you then define the *Creation Method*.

Products returned by creation methods (or indeed any of these patterns, since the more complex patterns typically use creation methods of one kind or another to create products) should be created and initialized to a usable state. Depending on the usage context, they may also have undergone more complex configuration. Fields should not be left uninitialised, but should at least receive default values.

**Examples** An email application needs to create a new Message object whenever the user hits the reply, reply to all, forward, or “New Message” buttons. e.g.:

```
Message msg = new Message(CurrentUser.getEmailAddress());
```

Rather than repeating this code wherever the new Message object is needed, move it into a Creation Method to say it once and only once:

```
Message newMessage(EmailAddress address) {
    return new Message(address);
}
```

The client can then follow up by providing additional information. For example, if the reply button was hit, the client will set the to: field accordingly. If the forward button was hit, nothing no further information is set.

**See Also** Noble, Natural Creation; Beck STBBP; Null Object

## 2.2 Conversion Method

**Problem** How to get a product object into the right form.

**Context** You have an object at hand that has the right information but presents it in a wrong form (typically the wrong class). You want a different product with the same information but more suited to your task.

**Solution** Implement a *Creation Method* on the object's class that returns a new product of a class that better meets the client's requirements. This method is a *Conversion Method*, because it presents a converted form of the object to the client.

**Roles** This pattern "converts" an existing product into a better suited form of product. Hence, the creator role is on the original product's class, and the product role is on another, simpler, class.

**Comments** The Creator should return a simpler type of object than itself to promote loose coupling. If the object is more complex, it is more likely to have code dependencies back to the Creator's class, and hence the two class implementations become mutually dependent.

In C# 3.0 (with attached methods), Smalltalk/ENVY or AspectJ, creation methods can be packaged with the Product class rather than the Creator, which can reduce coupling problems. Alternatively, you may be able to implement the conversion method as a static method on the Product class. This may improve coupling, but can contort your code and may not be possible if the Product needs information private to the Creator.

Also please note that you are not actually converting the original object; you are just returning a different form of (some of) its information. A better name might have been *Interpretation Method*, but so far, *Conversion Method* seems to have stuck.

**Examples** Common Java examples are the `toString()` or `asInteger()` methods on many classes, and static `asString(String)` methods on a few.

**See Also** Cope?, Beck SmalltalkBPP, Java standards?

## 2.3 Factory Method

**Problem** How to vary by creator the specific variant of a general product type to be instantiated.

**Context** Sometimes, from the client's perspective, the product must conform to a general class but can in reality be any of its subclasses. It typically depends on the creator which of the different subclasses it should be.

This scenario is called dual class hierarchies, where the creator has to work with the product later on, so you make the creator decide what product best matches its needs. In this scenario, the client is often the creator itself.

**Solution** You declare the *Creation Method* abstract or "as to be overridden", or in an interface, and make each subclass implement the method in such a way that they return a new instance of the proper type.

**Roles** The client calls the factory method on the interface, but interacts with a creator of a concrete class at runtime. Client and creator may be the same object where the client code is written to the interface, even if the separate factory method names a specific concrete class.

**Comments** An implementation of *Factory Method* may use the *Default Implementation* pattern. Here, you don't define the *Creation Method* as abstract but rather provide a default case of the product type to be instantiated. You then override the method only if special circumstances make the default case unsuitable.

**Examples** The Design Pattern book uses the example of a class hierarchy of Application classes for different types of editors and their matching Document classes. The Application and Document class hierarchies are dual hierarchies, mirroring each others structure. Hence, an Application object will only create a specific kind of Document product, one that it can handle.

**See Also** GoF covers this one pretty well

## 2.4 Cloning Method

**Problem** How to allow the system to configure dynamically which object gets created.

**Context** A *Conversion* or a *Factory Method* fixes the product class in code. Sometimes, this is too restricting, and you need to configure what type of object to create. Such configuration typically happens at system startup time.

**Solution** You configure the Creator with an *Exemplary Instance*, also known as a *Prototype*. This object is a stand-in for the type of object to be created. The *Cloning Method*, when called, creates the new object by cloning (copying) the exemplary instance.

**Roles** Like with a regular creation method, the definition of the client, creator, and product roles may or may not be on different classes. The client and creator role may be played the same object.

**Comments** The *Exemplary Instance* pattern is usually called the *Prototype* pattern, which is one of the standard creational patterns from the Design Patterns Catalog. The Design Patterns' book *Prototype* has two purposes: To create a new object, and to initialize that object in a particular way. We use the term Cloning Method to refer to the first part only, and leave Prototype to refer to a way of configuring a complex product.

The cloning process can become quite complex if a whole object structure is to be cloned, also see the discussion of the Prototype pattern. Here, we assume that only one object is being cloned, that is we make a shallow clone.

**Examples** Tbd.

**See Also** GoF, etc.

## 2.5 Trading Method

**Problem** The Client has information about the product to be created , but doesn't know the full details of the creation process

**Context** The client does not want to interfere with the process, but it has some extra information that determines the type of object that will be created.

**Solution** The Client creates a *Specification* that captures the needed information. The Creator then uses the *Specification* to look up a *Class Object* (or *Exemplary Instance*) that matches the specification. Once the *Class Object* has been determined the Creator instantiates and return the new object to the Client.

**Roles** The Creator (who may be the same as the Client) takes the specification and returns the product.

**Comments** *Trading Method* is called trading method because the original idea of using specifications to retrieve products goes back to the CORBA trading service (and earlier). *Specification* is a general term here, and there are at least three different forms of specifications: String Specification, Property List, and Logical Expressions.

.

**Example** An example is the editor application again, which wants to create a View for a Document. The type of view primarily depends on the type of document, so you make the Application object hold a dictionary that maps Document classes on View classes. For a given Document object then, you look up the View class, instantiate it, and return it.

**See Also** Dirk's Product Trader writeups

## 2.6 Initialisation Method

(**aka. configuration method)**

**Problem** A client needs to create a product now, but cannot completely initialize it in one step.

**Context** Sometimes creational relationships between objects are cyclic – e.g. a window needs a document but that document needs a window. Unfortunately, that means you can't create a fully-functioning window before you have created your document — but you can't create the document before you have the window.

Sometimes, it is not feasible to provide all values to the new product in the constructor or creation method call. The list of arguments may be unwieldy long. Or the arguments different clients supply vary so much that different constructors for each client would be overkill. d. Or, some systems, most notably C++, provide a functioning object only after the constructor call has finished. Or the product initialization process can be crazily complex,

**Solution** Make a clear distinction between basic product creation and subsequent product initialisation. An Initialization Method or Configuration Method is a method that sets up the object for use. If you use this pattern, the Creation Method does not have to do that; it could well just return an empty shell of the right type. The Initialization Method then makes sure the attribute values of the object are right and so are the links to other objects.

The client supplies basic values through the constructor or creation method call only, and recieves an unconfigured object from the creator in return. Then, the client itself sends a series of initialization or configuration methods to the product

**Roles** Creator returns a (partly or completely) unitialised product. The Client must call the inisitlisation method(s) on the product before the product is ready for use.

**Comments** This pattern is intrinsically dangerous, since it allows uninitialized objects loose in your system. This must compromise the integrity and safety of your design, the fidelity of your modeling, and leads to buggy programming. But, this danger is what provides the power in the pattern: when initialization methods are called, the product is already in functioning albeit bare-bones state. These initialization methods provide the more complex initialization values to the new product, and while doing so can already rely on at least some of the product's methods working properly.

There are a number of things you can do to mitigate this danger. The easiest is that the product should only be allowed to escape from the client once it is fully configured and ready for use. This hopefully confines the problems within the client and product, but the problems remain, especially if there is any ambiguity at all about the precise state of a partially-initialized object. Preventing objects escaping is also harder than it sounds, so this approach can give a false sense of security.

Second, and really required for good practice (but rather more work) is to ensure the initialization state of the product is clearly embedded in its design and code. Ensure that all incorrect uses of the product after creation fail safely and rapidly. This may mean guarding every method with if-statements that check the product's initialization state, using the *State* pattern, but in any case it typically leads to defensive programming that is buggy in itself, contributes to a false sense of security, and is usually too much effort for most programmers.

Probably the best option is to move initialized into the creator, from the client, typically by using a more advanced creational pattern such as *Builder*. A Builder can create a whole collection of interrelated, initialized, configured, and consistent objects in what appears to be one transactions from the client's point of view: as with initialization methods, a client can drip-feed information to a Builder piece by piece, but the product object is only allowed out once both construction and configuration is complete. The disadvantage is that most programmers seem to find Builder one of the most scary patterns around, although the code is usually simpler, less routine, and less fragile than implementing proper support for partially initialized objects.

But best of all is to just kid yourself you don't have a problem, which is also the simplest thing that could possible work…

**Example** Tbd.

**See Also** Kent BPP again? Cope? Where else? The Self book?

# 3 Factory Patterns

The *Object Factory* pattern tells you to move the *Creation Methods* to an object of their own. The main purpose of an Object Factory is to satisfy some constraints on the types of objects being created, and to do so in a central place. There are three main variants of the basic pattern: *Abstract Factory*, *Builder*, and *Prototype*.

## 3.1 Object Factory

**Problem** Your program needs to create products of a given type.

**Context** Sometimes, the client code that needs the new object is spread around multiple different classes. The Client classes shouldn’t redundantly implement the same *Creation Method*.

The product creation is an important part of the real world that you need to model. The object creation code itself is relatively complex.

**Solution** Create a dedicated *Object Factory* class whose purpose is to create the needed products. You then make the Factory class provide one or more *Creation Methods* for the products. Clients go to an instance of the Object Factory class, the Factory, and ask for a new product.

**Roles** This typically separates the client, creator & product roles, with the new Factory class playing the creator role

**Comments** Frequently, there is only one instance of the *Object Factory* class, though in multi-threaded environments you may want multiple instances, usually one for each thread. Sometimes, the Factory Object is stateless (except for some initial configuration state) in case of which multi-threading issues are less problematic.

An Object Factory is a good place to do more fancy stuff like reusing objects from an *Object Pool* rather than creating a new object for every request, or always returning a *Singleton* instance of the type of object to be instantiated. Also, a Factory Object frequently keeps track of the products it created, like a *Registry*.

An *Object Factory* can use any kind of *Creation Method*. Thus, you may need to configure the Factory before it can be used. You typically do so at system startup time.

The key liability of *Object Factory* pattern is that you’ve introduced another class into your design, a class which probably has little relationship to the world the program models or the internal technical architecture of the program: it’s just there because you need to create things.

This is related to the difference between the generic *Creator* role and a particular *Object Factory* class: an Object Factory reifies the creator rule, creating particular classes and objects to play that role, either standalone or in conjunction with other patterns. But where other objects in your design can naturally play the creator role, this often works better if it doesn’t overcomplicate the candidate creator class. So, for example, if a Document class can create a View object that displays it, that may much more direct than creating a special ViewFactory class that creates the views. On the other hand, if it takes 10 pages of code to create a View, you’re better off putting it into its own Object Factory, even if the client’s interface to that is via a Creation method on the Document class that delegates to an internal ViewFactory.

**Examples** Tbd

**See Also** Tbd. Probably the #1 pattern the GOF missed, according to Ralph I believe..

## 3.2 Abstract Factory

**Problem** You need to create sets of interrelated objects

**Context:** Clients who turn to an *Object Factory* expect that the products they ask for can work together, for example, because the Factory provides different products from a complex collaboration rather than a single class.

**Solution** You group the *Creation Methods* for the different products from the collaboration, or family, in one *Object Factory* class, and make sure that the implementations of the *Creation Methods* are aligned in such a way that they ensure the desired integrity between the products being created.

**Roles** A single Abstract Factory plays the Creator role for a series of different Creators and Products.

This is a good example of a “dense”, overlapping use of one pattern (in this case, many Object Factories) being sufficiently common that it has been named as a new pattern in its own right.

**Comments** *Abstract Factory*, like *Object Factory*, can use any of the *Creation Methods*, but typically it uses the same type of *Creation Method* for objects from one family of classes, because this makes ensuring consistency easier.

Abstract factories have all the benefits of Object factories, with the additional advantages of increasing the extensibility of the system

**Examples** Tbd

**See Also** I always thought GOF did this well, though I know others disagree — but that’s mainly cos they didn’t have Object Factory first.

## 3.3 Prototype

**Problem** Clients sometimes want not just one simple product, but a complex product consisting of many different parts. Moreover, the structure of this complex product is frequently not predetermined---it is either derived from configuration files or even more complicated, is configured at runtime.

**Context** There are multiple situation, in which you have such complex products. They might get configured in configuration files. They might be built using some algorithm that in turn draws on configuration files and system events. You might even have a user fully defining the complex product through a user interface. The effect is always the same: A complex objecy with a structure that might change every moment.

**Solution** You create the complex product in memory, before the first Client ever asks for it. This is the *Prototype*. When the first client comes along, you make a copy (clone) the *Prototype* and return it to the Client as the Product.

**Roles** In the Prototype pattern, the Creator and Product are instances of the same class. The prototype uses a *Cloning Method* to create a copy of itself as the product, which it then returns to the client

**Comments** Typically, the *Prototype* has one root object, by which it is being held. Where it is being held may vary. There may be a global variable holding this reference, or there may be an *Object Factory* that hides the *Prototype* behind a *Cloning Method*.

The cloning process can become quite complex. Usually, you have multiple phases. In each phase, different things are done. First, the basic object structure is created, following owning relationships. Next, object attributes are initialized. Then, non-owning references within the structure and to the outside are created. Finally, you may have to make the outside world link back to the prototype properly. Futhermore, the product cannot generally be an identical clone of the prototype, so the cloning method may have to make any necessary changes.

Sometimes, you allow the *Prototype* to change after clones have been created. You need to define then what to do with the existing clones. You may want to leave them alone, so that they get out of sync with the original *Prototype* object, or you may want to change them so that they maintain conformance with the original *Prototype*.

If you delete a *Prototype*, you are most likely going to use a *Cascaded Delete*. In a *Cascaded Delete*, the root object makes sure all its owned objects are deleted too, which in turn delete their owned objects, etc. Much like the initial cloning process, the *Cascaded Delete* is a recursive multiphase algorithm that traverses the full object.

The original *Prototype* pattern from the Design Patterns book serves many different purposes.

It is difficult to tear them apart, so we know what we are talking about. Alternatively, this pattern could be named *Template*, which together with *Exemplary Instance* would cover the two main uses of the original *Prototype* pattern.

The prototype pattern as two key liabilities. First, this pattern means that the product's structure is obscured or even completely invisible in the program code: everything depends on the way the prototypes are constructed at runtime. This makes programs using prototype harder to read, understand, and change, although increases flexibility, extensibility, and can make programs much shorter (especially if prototypes can .e.g be read out of serialized files or object dumps). The second liability of prototypes is the *prototype corruption problem* — rather than returning a new product, you accidentally return, directly or indirectly, the prototype or part of it. This kind of problems can be very difficult to detect and resolve.

**Examples** tbd

**See Also** Gof,s OK here, right? How about ATTACK of the CLONES? (Chinese translation)

## 3.4 Builder

**Problem** How can you create a really complex, dynamically configured, product?

**Context** Clients sometimes don't just need one product object, but several objects, set up properly to work together. Unlike the *Prototype*, though, the Client wants to determine this structure rather than receive a predetermined one.

Often these complex products — or sets of objects more likely — also need to be configured dynamically, so you find yourself using many *Initialisation Methods*. The resulting code gets difficult to read and maintain, is brittle and buggy, and pollutes the clients own code.

**Solution** You create a special kind of *Object Factory*, called the *Builder*, that provides a clean procedural API for creating the products. The *Builder* implements the building process of the complex object. Clients supply information to the *Builder* as it becomes available to them, helping the Builder build the complex object structure piece by piece.

**Roles** The Builder plays the part of the Creator. The Client can have a very extended interaction with the builder, before finally asking for the Product, which is eventually delivered fully configured.

**Comments** The *Builder* can have very different methods, some of which are *Creation Methods*, some of which are *Initialization Methods*. Still others are methods that provide some information helping control the build process. *Builders* are usually stateful, keeping the information needed to build the object. Their interface resembles that of a transaction protocol, with a dedicated beginning of the building process and a conclusion of it.

So why do programmers hate the Builder pattern? One reason could be that it is not so much complex as subtle — nonobvious. Many programmers seem happier dealing with lots of crufy detailed code than designing a (stateful) builder protocol and rewriting all their creation code to use it. Because Builders are aggressively about invariants and encapsulating progesses, its hard to see how they can be introduced by a number of small refactorings: like Façade, the point is to sweep lots of evil stuff under a rug, then protect it so you can't get at it outside: you need to impose Builder (and Façade) in one larger composite refactoring, or you'll end up supporting e.g. ad-hoc initialization and the builder forever (Wizards anyone).

(Hmm – that probably is the intermediate step to a full builder, an ad-hoc pile of crap helper methods) — James

**Examples** tbd

**See Also** Gof. Where else? Perhaps the gof's writeup is just to complex. Perhaps it makes more sense after reading all the rest of these. I like the story in initialization method about how you're better off with builder.

## 3.5 Dependency Injection

**Problem,** How can you move issues of object creation and interconnection OUT of your program

**Context** You're doing EJB or components. You don't understand how your system is designed. You are going to have lots of little components, all alike, that are going to be deployed in lots of different situations. You need to be able to reconfigure you whole system in to lots of different ways —- e.g. single machine client, test rig, business server version.

**Solution** Give each product class a wide selection of initialiseation methods. Have a god-like creator that builds and configures everything.

**Roles** Client does nothing (or doesn't exist). Creator reads config file, creates products, uses initialization methods to wire them together.

**Comments** OO people hate this. AOP & EJB people love it. First done in Xerox Parc Mesa or Cesar I think.

The practical implementation of "software architecture descriptions languages"

So in many ways I don't think this belongs here at all.

**Examples** tbd

**See Also** Marty. Spring! Rod!!

# 4 Related Patterns

A *Class Object* is an object that represents a class. In most object-oriented programming languages these days, the Class Object *is* the class. The *Class Object* knows how to create an instance of the class it represents. In C++, if type_info is not enough, you create an *Exemplary Instance* to represent the class. An *Exemplary Instance* is a singled-out instance of the class that plays the role of the class object.

The *Specification* pattern is a pattern language of how to describe properties of objects for the purposes of selecting these objects from some collection. Simple specifications are strings like a class name, more complex specifications are *Property Lists* that provide a number of named attributes that a specified class or object has to match, and really complex specifications are expressions in a full-fledged specification language.

A *Default Implementation* is a method implementation that provides the default case. You use a *Default Implementation* if most implementations of the method are going to be the same, with a few exceptions that then override the *Default Implementation.*

Complementary to object creation is object destruction. For objects that hold important resources, you typically want a *Finalization Method* that cleans up those resources before the object is destroyed. If the object is complex, you use a *Cascaded Delete* that traverses the complex object and makes sure all owned objects are finalized and deleted.

## 5 References

Still didn't get to do the references…

Me Neither, but here are some.

GOF. POSA? BeckBPP, James' NatCreat. Dirk's method roles papet

Brian's Powerpoint; Kevlin's stuff on factories — both of which we really should look at again.

What else?

## 6 Acknowledgements

To Joe Bergin, the most patient and kindly shepherd, many of whose best ideas we stole, and who must have argued really hard to the PC, given what a slack job we did during our "iterations"

To John Vlissides, without whom etc.

## 7 Miscellaneous left overs

*Modeling*. Objects don't just exist within programs, but generally also represent some phenomena of the "real world". These patterns (especially *Creation Method)* can make programs better models by letting object creation in the program follow the same patterns as in the world. By managing when and how product objects created, they can also help to ensure not only that products' invariants and internal states are correct, but that they faithfully model the parts of the world they are supposed to reprsent.