

# Factory and Disposal Methods

## *A Complementary and Symmetric Pair of Patterns*

Kevlin Henney  
kevin@curbralan.com  
kevin@acm.org

May 2004

---

### *Abstract*

**complementary** (of two or more different things) combining in such a way as to form a complete whole or to enhance or emphasize each other's qualities.

**symmetry** the quality of being made up of exactly similar parts facing each other or around an axis.

- correct or pleasing proportion of the parts of a thing.
- similarity of exact correspondence between different things.

The New Oxford Dictionary of English

Manual object creation may be in conflict with information hiding or instance-controlling requirements. The consequences of such separation and encapsulation can be addressed by the FACTORY METHOD pattern. Further control, economy, and symmetry may be found in the DISPOSAL METHOD pattern, in effect a mirror of FACTORY METHOD.

This paper revisits the classic FACTORY METHOD pattern, broadening the scope of this general pattern in line with the common usage of its name. Four specific variants are examined: PLAIN FACTORY METHOD, CLASS FACTORY METHOD, POLYMORPHIC FACTORY METHOD, and CLONING METHOD. FACTORY METHOD is accompanied by DISPOSAL METHOD, making the consideration of object lifecycle more clearly balanced. Two specific variants are examined: FACTORY DISPOSAL METHOD and SELF-DISPOSAL METHOD.

FACTORY METHOD and DISPOSAL METHOD are, in essence, quite high level whereas each of their variants is a more specific pattern. In the context of a specific pattern language or sequence it often makes more sense to zoom in on the specific variants rather than refer abstractly to the zoomed-out generalizations. This paper does not present a specific pattern language or a complete pattern sequence, more of a generative phrase or expression that can be incorporated and reified in a language or sequence.

---

## Introduction

There is an inherent tension between data hiding and object creation. For example, if you hide object use behind an interface, how do you know which concrete class to use for creation? With any luck, if you are an experienced OO developer, you will now be sitting back in your seat, confident in the knowledge of at least one good answer. There is a good chance that this answer is FACTORY METHOD [Gamma+1995]:

Define an interface for creating an object, but let subclasses decide which class to instantiate. FACTORY METHOD lets a class defer instantiation to subclasses.

Well, you can lean forward now: this pattern deserves a revisit and revision to free it from a purely inheritance-centric view; it also warrants a counterpart to make it part of a greater design whole.

Both before and since the Gang of Four published FACTORY METHOD, the term *factory* has been used by programmers in a slightly broader sense, one not necessarily restricted to class hierarchies. Programmers will happily name a non-polymorphic method a *factory method*, so long as the obvious creational role indicated by a literal reading of the pattern name is followed. A factory is therefore generally a defined location with responsibility for encapsulating object creation.

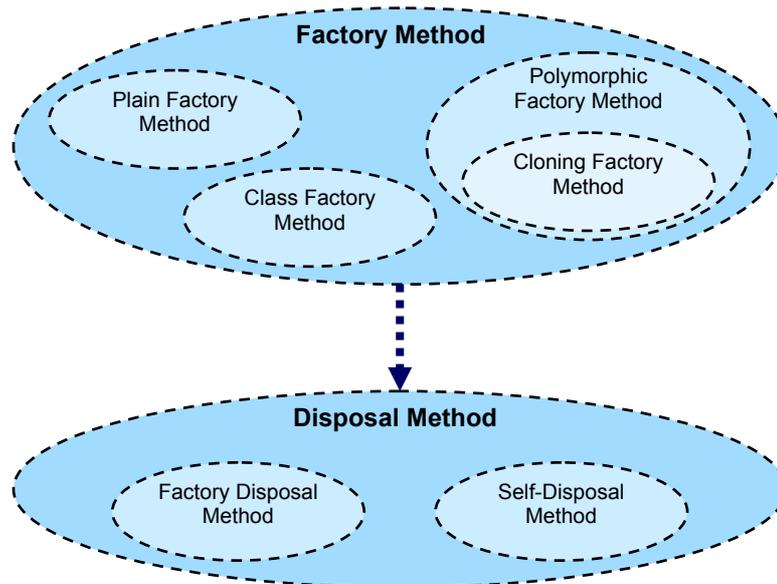
There is also something missing from the common discussion of object creation through factories: object disposal. Contemplating the sound of one hand clapping is a spiritual question not always well suited to the classically utilitarian materialism of objects. The absence of symmetry in the discussion of FACTORY METHOD suggests DISPOSAL METHOD. This relationship is not so much a tiny pattern language or short pattern sequence as a simple generative pattern phrase or subsequence, something that might be uttered in conversation in a language or included in a longer, domain-specific sequence. The symmetric pairing marries and mirrors FACTORY METHOD with DISPOSAL METHOD: one seeks closure in the other. As with any real mirror or marriage, the reflection is not perfect: in the detail of these patterns there is a great deal of independent variation that contrasts with the sketch-level symmetry.

Symmetry is a fundamental consideration [Alexander2002, Henney2003, Zhao+2003] that typically has the effect of simplifying a design, making it easier to comprehend and work with, not to mention more elegant and more whole. The simplification comes from the resulting regularity: something that is regular is easier to recall or second-guess than something that is not. Symmetry encourages consistency, becoming its own design map. This does not mean that designs should be globally and thoroughly symmetric but that, where transparency is not possible, a design should be predominantly and locally symmetric. A purely symmetric design is typically quite a dull one; a purely asymmetric one is unmemorable for different reasons.

A common question confronting both pattern authors and readers is that of specificity. Each occurrence of a pattern in a system is clearly highly specific, but at what level should the pattern itself be described? How specific should the problem be? What differentiates a variant of a pattern from the core pattern? Both the level and context of interest often dictate whether a pattern should be expressed at the most general level, e.g. a FACTORY METHOD is a method responsible for the creation of objects, or whether different flavors should be singled out and named, e.g. a POLYMORPHIC FACTORY METHOD defers the knowledge of exact creation type to be pushed down a class hierarchy. In the context of a specific pattern language it tends to make sense to focus on specific variants because the problems they address in the context of the language are similarly specific, e.g. while FACTORY METHOD offers a general heading for describing an approach to creational

encapsulation, a PLAIN FACTORY METHOD does not solve quite the same problem as a CLASS FACTORY METHOD, nor does it have quite the same consequences.

In this paper the focus is not a pattern language but on two patterns that, at a general level, form part of a vocabulary for object lifecycle design. PLAIN FACTORY METHOD, CLASS FACTORY METHOD, and POLYMORPHIC FACTORY METHOD are presented in the context of the more general FACTORY METHOD, with CLONING METHOD as a further flavor of POLYMORPHIC FACTORY METHOD. FACTORY DISPOSAL METHOD and SELF-DISPOSAL METHOD are presented in the context of DISPOSAL METHOD. The following diagram illustrates the relationships:



A low-ceremony pattern form is used. In general, the focus of the patterns is on the statically typed OO model of Java, C++, and C#. Specifically, code fragments are in Java.

---

## FACTORY METHOD

*Encapsulate the concrete details of object creation by providing a method for object creation instead of letting object users instantiate the concrete class themselves.*

**Problem:** Code that depends on instances of a class or from a class hierarchy may need to create the objects itself. This may not be as easy as simply using a new expression:

- What if the creational logic cannot be contained easily inside a constructor? What if external validation is needed or object relationships must be established that might be considered beyond the scope of the object's immediate responsibility? For example, the constructor of a bank account object should not be responsible for allocating its account number, running a credit check, or ensuring that the instance is persisted by its associated bank.
- What if the class must be instance controlled, so that unconstrained use of new would be inappropriate? For example, neither ENUMERATION VALUES [Henney2000a, Henney2000b] nor SINGLETON objects [Gamma+1995] should be created manually or directly by their users.
- What if the appropriate concrete class is unknown to the user because the user manipulates an object only via a declared interface and not via its concrete class? For example, an object whose actual type depends on the actual type of another object should not cause the user to copy and paste repetitious type-dependent code. Cascaded if else if statements that hardwire instanceof, dynamic\_cast, or is runtime type checks are a good way of obscuring a method's intent and reducing a class's openness and extensibility.
- A more specific example of needing object creation in the presence of hierarchical abstraction is the wish to take a proper copy of an object without knowing its concrete type.

These different scenarios are unified under a common pair of opposing forces:

- Objects are most simply and intuitively created using a new expression, specifying a concrete class and constructor arguments. This provides the user of a class with full control over instantiation.
- Direct object creation may inadvertently obfuscate and reduce the independence of the calling code if any of the necessary ingredients for correct object creation are not readily available. The concrete class, the full set of constructor arguments or the enforcement of other constraints may not be known at the point of call; to require them would increase the complexity of the calling code.

**Solution:** Provide a method for fully and correctly creating the appropriate object instead of relying on a new expression. The knowledge of creation is encapsulated within this *factory method*. The ability to create instances directly is hidden from the caller either by making constructors non-public or by pushing it down a class hierarchy.

However, unless created specifically for the purpose, including the role of *creator* in a class's repertoire can sometimes be considered an addition that dilutes its cohesiveness. The solution is certainly more encapsulated than the alternatives, but the cohesion can be considered a little lower than in a design where such creation was never needed.

There are three basic and one extended variant of FACTORY METHOD that determine how the different roles of *product* and *creator* (also known as the *factory*) are realized:

- PLAIN FACTORY METHOD: The *creator* is an object – not necessarily in a class hierarchy – and the type of the *product* either is fixed or varies only with environmental settings

or the arguments to the *factory method*. A PLAIN FACTORY METHOD implementation is normally just a case of providing an ordinary, possibly `final` or `sealed`, method that creates instances of another class, with no specific intent to be inherited or overridden.

- CLASS FACTORY METHOD: The *creator* is a class rather than an object, and so the *factory method* is static. The *creator* is often the same class as the *product* object type, which is not normally defined in a class hierarchy. Direct creation of *product* objects is often prevented by ensuring that instance constructors are non-public. CLASS FACTORY METHOD pattern is also known as STATIC FACTORY METHOD [Bloch2001, Haase2002].
- POLYMORPHIC FACTORY METHOD: The possible types of the *product* object are defined in a class hierarchy. Mirroring the hierarchy of what is created, an interface for *creator* objects is provided, offering the *factory method* abstractly, and the responsibility for creation is deferred to an implementing subclass. The knowledge of which type of *product* is required is contracted out to the *creator* hierarchy, removing the need for a closed and clumsy instanceof solution. This FACTORY METHOD variant is the classic Gang of Four version.
- CLONING METHOD: The *product* class is the same as the *creator* class. However, unlike a CLASS FACTORY METHOD the relationship is properly reflexive: the *creator* is an instance of the class, rather than the class, so that its result is another object of its own type. To be precise, the *product* is a proper copy of its *creator*. A CLONING METHOD is a specific kind of POLYMORPHIC FACTORY METHOD.

A PLAIN FACTORY METHOD is fairly straightforward in its common form. The *product* is normally concrete, and may have only non-public constructors:

```
public class ConcreteProduct
{
    ...
    private ConcreteProduct(...) ...
}
```

The *creator* is also normally concrete, and has sufficient access to the *product* type to create instances:

```
public class ConcreteCreator
{
    public ConcreteProduct create()
    {
        return new ConcreteProduct(...);
    }
    ...
}
```

For the bank account example, a bank object would adopt the role of *creator* and an account object would be a *product*. The bank would hide the details of creation and the account class would prevent general creation by users. The design is encapsulated between the two classes and need not involve any inheritance.

The form of a CLASS FACTORY METHOD is simple, with the class as *creator* and its instances as product:

```
public class ConcreteProduct
{
```

```

public static ConcreteProduct create()
{
    return new ConcreteProduct(...);
}
...
private ConcreteProduct(...) ...
}

```

For example, as an example of symmetry, a Java class that supports a meaningful `toString` override could consider providing a `fromString` or `valueOf` CLASS FACTORY METHOD in preference to a public `String` constructor. Using a CLASS FACTORY METHOD names the conversion concept explicitly. It sets string-based creation apart from other constructors to emphasize the inverse relationship with the common `toString` method.

The general POLYMORPHIC FACTORY METHOD has the most intricate detail, spanning two class hierarchies where the previous two variants typically address one or two classes on their own. There is the *product* hierarchy:

```

public interface Product
{
    ...
}
...
public class ConcreteProduct implements Product
{
    ...
}

```

And there is the *creator* hierarchy:

```

public interface Creator
{
    Product create();
    ...
}
...
public class ConcreteCreator implements Creator
{
    public Product create()
    {
        return new ConcreteProduct(...);
    }
    ...
}

```

Where the caller and the used class hierarchy are one and the same, TEMPLATE METHOD [Gamma+1995] is often used:

```

public abstract class ProductUser
{
    public void useNewProduct()
    {
        Product produce = create();
        ...
    }
}

```

```

    protected abstract Product create();
}
...
public class ConcreteProductUser implements ProductUser
{
    protected Product create()
    {
        return new ConcreteProduct(...);
    }
}

```

A degenerate arrangement of POLYMORPHIC FACTORY METHOD is CLONING METHOD (or VIRTUAL COPY CONSTRUCTOR or SELF-FACTORY METHOD), which is normally used in its own right to support polymorphic copying but can also be found in support of a PROTOTYPE approach to object creation [Gamma+1995, Coplien1992], with which it is often confused. In CLONING METHOD the types of the *product* and the *creator* are the same, and the *creator* instance provides itself as the model from which a new instance is built:

```

public class Product implements Cloneable
{
    public Object clone()
    {
        ... // cloning carried out and resulting object returned
    }
    ...
}

```

The cloning is instigated directly by the object user:

```

...
public void takeSnapshot(Product other)
{
    snapshot = (Product) other.clone();
}
...

```

In PROTOTYPE an object is held by a factory to be used as the prototypical instance from which new factory products are built. This may involve a CLONING METHOD:

```

public class ConcreteCreator implements Creator
{
    public Product create()
    {
        return (Product) prototype.clone();
    }
    ...
    private Product prototype;
}

```

Or not:

```

public class ConcreteCreator implements Creator
{

```

```
public Product create()
{
    return new Product(prototype.attributes());
}
...
private Product prototype;
}
```

In the second fragment the factory product is created using the attributes of the prototype object and explicitly constructing the object. In both cases the prototype is used as the instance on which factory products are based, but only in the first does the implementation mechanism qualify as a **FACTORY METHOD**.

---

## DISPOSAL METHOD

*Encapsulate the concrete details of object disposal by providing an explicit method for clean up instead of letting object users either abandon objects to the tender mercies of the garbage collector or terminate them with extreme prejudice and delete.*

**Problem:** How should objects with significant clean-up behavior be disposed of after use? For garden-variety objects, the usual mechanism of the language for disposing of objects is normally sufficient. However, for some kinds of objects, such as resources, this may not be enough. Just as a `FACTORY METHOD` may hide details of an object's creation that cannot be handled fully by a constructor, details of an object's destruction may go further than can be adequately expressed by conventional finalization, whether a `finalize` method or destructor.

A resource can be defined by its use and context rather than in terms of its abstraction. A resource is any object that could easily become scarce in a system and whose scarcity would cause problems. Therefore, a resource can be defined liberally as anything that should be returned after acquiring and using it. For example, memory in C and C++ is a resource, but in a well-endowed Java or C# program it is typically not. However, in a smaller environment memory again becomes a resource. In the common application of a `FACTORY METHOD`, instance creation is controlled but object disposal is not. Because resource usage may need to be conserved and resources recycled, the user of a resource should have a clear contract for how a resource's usage lifetime is bounded.

In C++ an explicit `delete` by a factory-product user is asymmetric with the hidden `new` in the factory. A `delete` expression deterministically triggers the end of an object's life, which may be a somewhat more severe disposal than is wanted: it is difficult to recycle an object if it no longer exists. There is also no guarantee that an object was created using a plain `new`, hence a `delete` may be precisely the wrong action even to end an object's life. Memory that is acquired independently of construction would rely on a placement `new` expression for construction and an explicit destructor call for destruction; there is no corresponding `delete` expression.

Java and C# programmers can discard objects for later automatic collection by the garbage collector. It is, however, naïve to assume that a GC system solves all memory and resource management issues out of the box [Bloch2001]:

When you switch from a language with manual memory management, such as C or C++, to a garbage-collected language, your job as a programmer is made much easier by the fact that your objects are automatically reclaimed when you're through with them. It seems almost like magic when you first experience it. It can easily lead to the impression that you don't have to think about memory management, but this isn't quite true.

It is possible to further dilute confidence in totally transparent GC: your objects *may* be reclaimed automatically. There is little guarantee that they will be claimed in a timely manner, or even at all — although such low (or non-existent) quality-of-service would find favor with few developers. Frequent creation of fine-grained objects, such as iterators or value objects, can potentially lead to inefficient use of resources or even resource exhaustion.

With respect to resources, a specific and timely clean-up action may be required, but in the absence of explicit control over the tail end of an object's life this cannot be made implicit — and whatever the problem, Java's `finalize` is rarely the answer. GC addresses the issue of object collection to make memory resourcing transparent, but this does not apply to other resources.

**Solution:** Provide a method for explicit clean up and disposal of an object. Mirroring `FACTORY METHOD`, `DISPOSAL METHOD` answers the question of who is responsible for the clean up and disposal of an object by making the clean up an explicit operation for the user. Just as the user requested an object for use, they must also mark the end of its use.

`DISPOSAL METHOD` may be expressed as one of two basic variants:

- `FACTORY DISPOSAL METHOD`: Provide a method on the factory that originally created the object. The knowledge of an object's lifecycle is isolated in a single place, which allows transparent instance control, such as an object pool that caches and recycles objects.
- `SELF-DISPOSAL METHOD`: Provide a method on the object to be disposed of. This method either performs the clean up itself or, if a factory was involved in the object's creation, it returns of the object to its maker.

An obvious and negative consequence of this pattern is that the user must remember to both make the call and make the call exception safe. This situation is tedious and error prone, and can be ameliorated through additional encapsulation, such as a `COMBINED METHOD` [Henney2000c], `EXECUTE-AROUND METHOD` [Henney2001a], or a `COUNTING HANDLE` [Henney2001b]. Where instance control is neither about resource management nor in the hands of an object user, no disposal is required, so `DISPOSAL METHOD` is not necessarily appropriate.

`FACTORY DISPOSAL METHOD` is the natural complement of `FACTORY METHOD`, and its truest reflection:

```
public interface Creator
{
    Product create();
    void dispose(Product toDisposeOf);
    ...
}
```

In the bank account example closing an account would be a good example of a `FACTORY DISPOSAL METHOD`. The code that decides to dispose of a factory-created object must have access to both the *creator* and the *product*. This not only means that the lifetime of the product must be contained within that of its creator, but that the caller of the `DISPOSAL METHOD` is expected to co-ordinate the disposal correctly, i.e. it should ensure that it matches the right product with the right factory. This is often not a significant issue because factories and products are normally well matched in terms of types and scope usage. However, this slight increase in coupling can present a potential liability for some programs.

`SELF-DISPOSAL METHOD` is sometimes known as an `EXPLICIT TERMINATION METHOD` [Bloch2001]:

```
public interface Product
{
    void dispose();
    ...
}
```

Where a `SELF-DISPOSAL METHOD` is simply a forwarder to a `FACTORY DISPOSAL METHOD`, it clearly has to retain some kind of reference to the factory of origin. In such a case it

successfully encapsulates the knowledge of its origin and therefore the correct coordination of *product* to *creator*. The product user – or, to be precise, disposer – is freed from maintaining and using this extra reference. Although this offers a better encapsulation of the constraints governing the factory-product pairing, it can be seen as slightly less cohesive because responsibility for disposal is represented in the product interface, which would otherwise be focused purely on matters of product usage.

In C++ a DISPOSAL METHOD displaces the use of a public `delete` for the product type in question. The lifetime of an object is no longer subject to the operators in the language but to the higher-level interfaces and object lifecycle choices of a specific application. To ensure no clash between the use of a DISPOSAL METHOD and the common use of a `delete`, the destructor of the target object should not be public at the level of the interface. This restriction prevents any attempt to mix the `delete` and DISPOSAL METHOD models at compile time.

---

## Acknowledgments

This paper is derived from a previous article [Henney2002].

I would like to thank Klaus Marquardt for his thorough and insightful shepherding of this paper for VikingPLoP 2003, Mark Radford for his additional comments both before and after the conference, and Neil Harrison for his comments following the conference. From the workshop at the conference I would like to thank Jacob Borella, Franco Guidi-Polanco, Alan O'Callaghan, and Titos Saridakis.

---

## References

- [Alexander2002] Christopher Alexander, *The Nature of Order, Book 1: The Phenomenon of Life*, Center for Environmental Structure, 2002.
- [Bloch2001] Joshua Bloch, *Effective Java*, Addison-Wesley, 2001.
- [Coplien1992] James O Coplien, *Advanced C++: Programming Styles and Idioms*, Addison-Wesley, 1992.
- [Gamma+1995] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.
- [Haase2002] Arno Haase, "Idiome in Java", *JavaSPEKTRUM* 36, February 2002.
- [Henney2000a] Kevlin Henney, "Patterns of Value", *Java Report* 5(2), February 2000, available from <http://www.curbal.com>.
- [Henney2000b] Kevlin Henney, "Value Added", *Java Report* 5(4), April 2000, available from <http://www.curbal.com>.
- [Henney2000c] Kevlin Henney, "A Tale of Two Patterns", *Java Report* 5(12), December 2000, available from <http://www.curbal.com>.
- [Henney2001a] Kevlin Henney, "Another Tale of Two Patterns", *Java Report* 6(3), March 2001, available from <http://www.curbal.com>.
- [Henney2001b] Kevlin Henney, "C++ Patterns: Reference Accounting", *EuroPLoP 2001*, July 2001, available from <http://www.curbal.com>.
- [Henney2002] Kevlin Henney, "Symmetrie in Java", *JavaSPEKTRUM*, July 2002, available in English as "The Importance of Symmetry" from <http://www.curbal.com>.
- [Henney2003] Kevlin Henney, "Five Possible Things after Breakfast", *The Road to Code* blog at *Artima*, 23<sup>rd</sup> June 2003, <http://www.artima.com>.
- [Zhao+2003] Liping Zhao and James Coplien, "Understanding Symmetry in Object-Oriented Languages", *Journal of Object Technology* 2(5), September–October 2003, [http://www.jot.fm/issues/issue\\_2003\\_09/article3](http://www.jot.fm/issues/issue_2003_09/article3).