Patterns in Java
The Importance of Symmetry

Kevlin Henney
kevlin@curbralan.com

Change and Stability

Sometimes change is the only constant. This is true of both software development and publishing. But as much as things change there is always something that stays the same. At Michael Stal's invitation this new column is a continuation of my previous patterns column in the late *Java Report* (may it rest in peace). The intent of that column was to explore patterns, going beyond the classic Gang-of-Four patterns [Gamma+1995], either revisiting them to see how they can be revised based on what further experience has taught us or looking at quite unrelated patterns from other sources.

Everything is much the same as before... except that it's now intended for German rather than English publication. This platform independence is possible thanks to Martina Buschmann's translation skills rather than any native interface of my own.

This mixture of change and similarity is also the essence of symmetry: some things remain apparently invariant under translation, rotation or reflection, and other things are radically transformed. When applied to design, symmetry can be considered a matter of balance.

Symmetry and Simplification

Symmetry typically has the effect of simplifying a design, making it easier to comprehend and work with. The simplification comes from the resulting regularity: something that is regular is easier to remember or second-guess than something that is not.

Where can we find symmetry in our Java programs? Consider first the contract for the `equals` method in `Object`: any implementation of `equals` should be symmetric, so that `a.equals(b)` should always have the same result as `b.equals(a)` for *any* non-null `a` and `b`. If `equals` were not commutative the effect would be surprising rather than effective. Symmetry has consequences: it both simplifies and constrains your code. A commonly overlooked consequence of the symmetric requirement on `equals` is that it is not possible to inherit from a concrete class *and* override `equals` to account for any new comparable state *and* preserve symmetry [Bloch2001]. This conclusion supports the general guideline that a concrete class should not in general subclass another concrete class.

The `equals` method can be said to demonstrates symmetry in rotation, where `a` and `b` are 'rotated'. Reflective symmetry in programming is found in matching opposites. If a class's `toString` method is overridden to return a meaningful presentation of an object's state — as opposed to précis information for use in logging or debugging — the class designer should consider providing a constructor taking a `String` argument. The constructor should allow the stored result of `toString` to be later used to create a new object with the same state. This only makes sense for value-based objects [Henney2000a, Henney2000b], such as dates and money, rather than entity-based objects, such as diaries and bank accounts. What are the benefits of the balancing symmetry in such a design? It eliminates the need for

programmers to have to parse the representation for themselves. It offers better encapsulation because the relationship between the use of the object and the interface to the object is one of completeness and self-containment.

Staying with the theme of object creation, 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 narrow inheritance-based view; it also warrants a corresponding shadow.

Pattern: 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 objects of a class or in 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 in a constructor? What if external validation 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 an account number, performing a credit check or ensuring that it is persisted by its associated bank.
- What if the appropriate concrete class is unknown to the user because the user manipulates objects only via their interfaces and not via their concrete classes? For example, an object whose actual type depends on the actual type of another object should not cause the user to write large amounts of type-dependent code. Cascaded `if else if` statements that hardwire `instanceof` checks are a good way of obscuring a method's intent and reducing a class's openness and extensibility.
- What if the class must be instance controlled, so that unconstrained use of `new` would be inappropriate? For example, Enumeration Values [Henney2000a, Henney2000b] or Singleton objects [Gamma+1995] should not be created manually and directly by their users.

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 the *factory method*. The ability to create instances is hidden from the caller either by making constructors non-`public` or by pushing it down a class hierarchy.

There are three basic variants of Factory Method that determine how the different roles of *product* and *creator* (also known as the *factory*) are realised:

- *Plain Factory Method*: The *creator* is an object — not necessarily in a class hierarchy — and the type of the *product* is either fixed or varies only with environmental settings or the arguments to the *factory method*. 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. This variant of the Factory Method pattern is normally just a case of providing an ordinary, possibly `final`, method that creates instances of another class, with no specific intent to inherit and override.
- *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 variant of the Factory Method pattern is the classic Gang-of-Four version.
- *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 the constructor is non-`public`. This variant of the Factory Method pattern is also known as Static Factory Method [Bloch2001, Haase2002]. A class that supports a meaningful `toString` override could consider providing a `fromString` or `valueOf` Class Factory Method instead of a `String` constructor.

The Polymorphic Factory Method has the most intricate detail, spanning two class hierarchies where the other 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 case of Polymorphic Factory Method is Cloning, 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]. The distinction with Cloning is that 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.

Pattern: Disposal Method

Encapsulate the concrete details of object disposal by providing an explicit method for clean up instead of letting object users leave the objects to the tender mercies of the garbage collector.

Problem: How are objects with significant clean-up behaviour disposed of after use? For instance, a scarce and encapsulated resource object created by a Factory Method. Instance creation is controlled but object disposal is not. Resource usage may need to be conserved, and resources recycled, but Java's object model implicitly does not provide control over the tail end of an object's life — and whatever the problem, `finalize` is rarely the answer.

Java programmers leave objects lying around to be collected automatically 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 this confidence in GC: your objects *may* be automatically reclaimed. There is no guarantee that they will be claimed in a timely manner, or even at all — although such low (in fact, non-existent) quality-of-service would find favour with few developers. Frequent creation of fine-grained objects, such as iterators, can potentially lead to inefficient use of resources. Performance and correctness of execution can suffer.

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.

Factory Disposal Method is the natural complement of Factory Method:

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

Self-Disposal Method is sometimes known as an Explicit Termination Method [Bloch2001]:

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

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, error prone and can be ameliorated through additional encapsulation, such as a Combined Method [Henney2000c] or an Execute-Around Method [Henney2001]. Where instance control is neither dynamic nor fully transparent — e.g. Singleton or Enumeration Values — no disposal is required, and this pattern is not appropriate.

Conclusion

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 and economy can be realised in the Disposal Method pattern, which provides a mirror image of Factory Method.

Symmetry encourages consistency and yields predictability. This does not mean that designs should be one hundred percent symmetric, but that they should be predominantly symmetric. A purely symmetric design is probably quite a dull one; a purely asymmetric one is unmemorable for different reasons.

References

[Bloch2001] Joshua Bloch, *Effective Java*, Addison-Wesley, 2001.

[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, Februar/März 2002.

[Henney2000a] Kevlin Henney, "Patterns of Value", *Java Report* 5(2), February 2000, available from `http://www.curbralan.com`.

[Henney2000b] Kevlin Henney, "Value Added", *Java Report* 5(4), April 2000, available from `http://www.curbralan.com`.

[Henney2000c] Kevlin Henney, "A Tale of Two Patterns", *Java Report* 5(12), December 2000, available from `http://www.curbralan.com`.

[Henney2001] Kevlin Henney, "Another Tale of Two Patterns", *Java Report* 6(3), March 2001, available from `http://www.curbralan.com`.