Patterns in Java

The Temptations of Symmetry

Kevlin Henney
kevlin@curbralan.com

Balancing Symmetry and Asymmetry

Quoted in Jon Bentley's *Bumper Sticker Computer Science* [Bentley1988], Andy Huber cautioned: "Avoid asymmetry". On the other hand, Victor Hugo proclaimed: "Symmetry is boredom, and boredom is the very source of death. Despair yawns." ("La symétrie, c'est l'ennui, et l'ennui est le fond même du deuil. Le désespoir baîlle."). So what is the best guidance for software design? The bias should be towards symmetry, with enough asymmetry to make a difference, but not so much as to cause indifference.

A rule is easier to recall than its exceptions, and a rule that appears only to be exceptions is harder still. In other words, there is less information in a symmetric design than in an asymmetric one. For this reason, symmetry tends to have a simplifying effect, whether in the detail of code or in a system's gross architecture. Gratuitous asymmetry creates the semblance of disorder in a design: everything is different, nothing is balanced; to describe and understand it requires protracted effort; to recall and apply it is tedious and error prone.

Symmetry breaking should be the exception rather than the norm, but it should certainly not be ruled against. Asymmetry does have a positive role to play in design — it is not all chaos. Inside some apparent symmetries hide asymmetry. For instance, I/O gives the impression of symmetry: input is mirrored by output, as are the classes and methods that support them in the library. However, when you zoom in, you realise that the symmetry does not run all the way through. Experience teaches us that dealing with input is always much harder than dealing with output. This is true in all programming languages, but in Java is accentuated — and made more tedious — by the various checked exceptions that adorn reading and parsing methods. At the high level the domain is inherently symmetric: asymmetry appears in the detail. So what does a symmetric view of I/O buy us? It is a convenient and unifying umbrella beneath which conceptual differences and physical detail can shelter.

Returning What You Borrow

Object memory is like beer: you never truly own it, only borrow it. The `new` expression initiates a chain of events that, when successful, results in the creation of a new object from raw memory. Once forgotten, the object, and its borrowed memory, may — or may not — be collected at some later time by the garbage collector. The memory can be recycled for use in another object. There is clearly asymmetry in this model: the point of creation is precise, deterministic and under the control of the programmer; the point of destruction is vague, undetermined and the responsibility of the virtual machine.

For common objects, such as values, asymmetry in creation and destruction has a simplifying effect for the programmer: a rule of two parts — *create* then *destroy* — becomes a rule of one part — *create* — or a rule of one part with a silent partner — *create* then *forget*. So how do we judge when it is symmetry that simplifies or when it is asymmetry that simplifies? In this particular example, asymmetry has been a simplifier when it has removed from rather than added something to the design. This is not a bad rule of thumb to follow: a symmetric model is easy to understand; an asymmetric model may be easier to understand if it removes something from a symmetric model; an asymmetric model may be harder to understand if it adds to a symmetric model.

The need to recover the design symmetry emerges when the assumptions change. What if the creation of the object is encapsulated, so that the `new` is hidden from view behind a Factory Method [Gamma+1995]? Is it safe to assume that the default forget-and-collect policy is also satisfactory? What if the object encapsulates a facility that cannot simply be left to the whims of the garbage collector to recover? The previous column [Henney2002] revisited the classic Factory Method pattern, broadening its remit and listing its variations, and balanced it with a Disposal Method. A Disposal Method acts as the logical opposite to a Factory Method: a method for reclamation to counterbalance the method for manufacture. It may be realised as an empty operation, it may cache the object for later reuse, or it may indeed be a physical opposite, disassembling the object as carefully as it was originally assembled. The following code sketches an interface for a Polymorphic Factory Method with a Factory Disposal Method:

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

And here is a sketch of the Self-Disposal Method variation:

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

The rule for using either approach is a simple one: create an object, with the understanding that you are only borrowing it; use it; when you are done, say so.

Not Returning What You Didn't Borrow

It is easy to become enchanted with the idea of symmetry and the simplicity it can bring to design. Symmetry is many things, but it is not everything: it creates the canvas on which you can paint, but it is not the whole picture. Getting carried away with the idea can lead to some common programming errors:

```
public void example(Creator factory)
{
    Product product = null;
    try
    {
        product = factory.create();
        ... // use product
    }
    finally
    {
        product.dispose();
    }
}
```

The code is clear and well intentioned. It is also wrong. The programmer has tried to ensure that no matter what happens when the object is used it will always be disposed of explicitly. However, the block structure of the `try finally` syntax has misled the programmer. The release in the `finally` part is mirrored by acquisition in the `try` part. This is seemingly symmetric, and superficially pleasing, but means that should `create` fail `dispose` will be called on a `null` reference, replacing the factory's own exception with a `NullPointerException`.

Unfortunately there is a lot of published and production code that makes this mistake. The fault is often not picked up because testing is not thorough enough. Object creation by a factory may fail for a number of reasons, but it is not likely to be a common occurrence so the code will slip through the net. It is possible that it may never fail in the release version, so the bug will remain hidden. It is also possible that its occasional failure may perplex the software's developers: yes, an exception would be handled and expected, but no, the exception type expected was not `NullPointerException`.

Sun's `javac` compiler in JDK 1.0 did not perform correct path analysis for `try` blocks, allowing the following code to compile:

```
public void example(Creator factory)
{
    Product product;
    try
    {
        product = factory.create();
        ... // use product
    }
    finally
    {
        product.dispose();
    }
}
```

You may notice that the `product` variable is not initialised at declaration. If you examine the possible paths of execution, it is possible to reach the `finally` block and use the `product` variable for the `dispose` method call without it ever being initialised: `factory.create()` could fail, so that product would never receive its initial assignment. The Java language specification expressly forbids this. In practice, JVMs typically null the memory for local variables before use, so that the code would behave as if `product` had been `null` initialised. This bug was fixed in later compilers.

So there are effectively two problems to address here: (1) practice and process for trapping such coding errors and (2) the correct code for acquisition and release.

Fault Injection

Why is programming through interfaces considered to be good practice? And why, increasingly, is it considered to be poor practice to use Singleton objects [Gamma+1995]? These two questions have the same answer: flexibility. This is a loose word, so here is a more precise answer: accommodation of alternative implementations.

It is common to implement factories as Singletons. Why have many factories when a single, system-wide one will do? Where are the alternative implementations?

```java
public class Singleton implements Creator
{
    public static Creator instance()
    {
        return instance;
    }
    ...
    private Singleton() {...}
    private static Singleton instance = new Singleton();
}
```

It would be used as follows:

```java
public void example()
{
    Product product = Singleton.instance().create();
    ...
}
```

This apparently simple act has far reaching consequences. Consider, how would you test that code using a factory interface responded to failed creation correctly? What you want to do is inject faults into the system, so that the factory fails to create an object. If this were C, C++ or C# you might consider introducing a pre-processor hack: conditionally compile a failing test version and a normal release version. The test version of the factory would throw exceptions instead of creating an object. Java does not have a pre-processor, so you are forced to create some kind of parameterisation for the factory to behave one way or another, either external configuration picked up at runtime or a runtime flag that can be set, or to perform clever tricks with dynamic class loading. These options are particularly unattractive.

The solution is to pass in a Mock Object [Mackinnon+2000] that is a failing factory. To make this work you need to adopt a more decoupled approach to coding: use interfaces. Here is the usage code:

```java
public void example(Creator factory)
{
    Product product = factory.create();
    ...
}
```

And here is a fault injection test that takes advantage of this flexibility:

```
class TestException extends RuntimeException
{
}
class FailingCreator implements Creator
{
    public Product create()
    {
        throw new TestException();
    }
}
try
{
    testee.example(new FailingCreator());
}
catch(TestException caught)
{
    System.err.println("Exception propagated correctly");
}
catch(NullPointerException caught)
{
    System.err.println("Exception propagated incorrectly");
}
```

In the production code, the proper factory will be passed in. Adding a single argument and working through interfaces is all it takes. The principle here is that a system's configuration should be parameterised from above rather than from below: factories, configuration info, logging objects, etc. should be passed in from the root of the program downwards, and should not be picked up as facilities hardwired into the lower layers of a system. This means that, in general, interfaces should be used and Singletons avoided.

Encapsulating What You Borrow

We can now look at the correct code schema for ensuring that acquisition is always matched by release, no matter what the circumstances. The problem with the original code was that it was mis-scoped: the exception safety is required for object usage after the object has been created, not for the creation itself. This leads to the following code:

```
public void example(Creator factory)
{
    Product product = factory.create();
    try
    {
        ... // use product
    }
    finally
    {
        product.dispose();
    }
}
```

This syntactic asymmetry is part of the solution structure for the Finally for Each Release idiom [Henney2001]. This code fragment can recur somewhat tediously in a large body of code, and it has scope for error. It is possible to eliminate — or rather, hide — the asymmetry of the solution using an Execute-Around Method [Beck1997, Henney2001]. There are many places that the Execute-Around Method could be located: in the factory, in the product or in a separate intermediate object. The following shows the design with intermediate *user* and *assistant* objects:

```
interface ProductUser
{
    void use(Product product);
}
class ProductAssistant
{
    public void apply(ProductUser user)
    {
        Product product = factory.create();
        try
        {
            user.use(product);
        }
        finally
        {
            product.dispose();
        }
    }
    ...
    private Creator factory;
}
```

In use, the passed `ProductUser` object is a Command [Gamma+1995], and most often a Block [Henney2001]:

```
assistant.apply(
    new ProductUser()
    {
        public void use(Product product)
        {
            ... // use product
        }
    });
```

Java's somewhat verbose rendition of closures means that this solution may not be an attractive one for small amounts of usage code or for systems where product usage does not often follow the scoped acquisition and release style shown. The choice is a matter of design; we can now see more clearly the interaction and trade-offs between symmetry and asymmetry that inform such a design.

References

[Beck1997] Kent Beck, *Smalltalk Best Practice Patterns*, Prentice Hall, 1997.

[Bentley1988] Jon Bentley, *More Programming Pearls*, Addison-Wesley, 1988.

[Gamma+1995] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

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

[Henney2002] Kevlin Henney, "Symmetrie in Java", *JavaSPEKTRUM*, Juli–August 2002, available in English as "The Importance of Symmetry" from `http://www.curbralan.com`.

[Mackinnon+2000] Tim Mackinnon, Steve Freeman and Philip Craig, "Endo-Testing: Unit Testing with Mock Objects", *XP2000*, `http://www.mockobjects.com/misc/mockobjects.pdf`.