

# Java theory and practice: Decorating with dynamic proxies

Dynamic proxies are a convenient tool for building Decorators and Adapters

Level: Intermediate

Brian Goetz ([brian@quiotix.com](mailto:brian@quiotix.com)), Principal Consultant, Quiotix

30 Aug 2005

The *dynamic proxy facility*, part of the `java.lang.reflect` package and added to the JDK in version 1.3, allows programs to create *proxy objects*, which can implement one or more known interfaces and dispatch calls to interface methods programmatically using reflection instead of using the built-in virtual method dispatch. This process allows implementations to "intercept" method calls and reroute them or add functionality dynamically. This month, Brian Goetz explores several applications for dynamic proxies. Share your thoughts on this article with the author and other readers in the accompanying [discussion forum](#). (You can also click **Discuss** at the top or bottom of the article to access the forum.)

➔ [More dW content related to: RMI dynamic reflection](#)

Dynamic proxies provide an alternate, dynamic mechanism for implementing many common design patterns, including the Facade, Bridge, Interceptor, Decorator, Proxy (including remote and virtual proxies), and Adapter patterns. While all of these patterns can be easily implemented using ordinary classes instead of dynamic proxies, in many cases the dynamic proxy approach is more convenient and compact and can eliminate a lot of handwritten or generated classes.

## The Proxy pattern

The Proxy pattern involves the creation of a "stub" or "surrogate" object, whose purpose is to accept requests and forward them to another object that actually does the work. The Proxy pattern is used by Remote Method Invocation (RMI) to make an object executing in another JVM appear like a local object; by Enterprise JavaBeans (EJB) to add remote invocation, security, and transaction demarcation; and by JAX-RPC Web services to make remote services appear as local objects. In each case, the behavior of a potentially remote object is defined by an interface, which by its nature admits multiple implementations. The caller cannot (for the most part) tell that they only hold a reference to a stub and not the real object because they both implement the same interface; the stub takes care of the work of finding the real object, marshalling the arguments, sending them to the real object, unmarshalling the return value, and returning it to the caller. Proxies can be used to provide remotng (as in RMI, EJB, and JAX-RPC), wrap objects with security policies (EJB), provide lazy loading for expensive objects (EJB Entity Beans), or add instrumentation such as logging.

In JDKs prior to 5.0, RMI stubs (and their counterpart, skeletons) were classes generated at compile time by the RMI compiler (`rmic`), which is part of the JDK tool set. For each remote interface, a stub (proxy) class is generated, which impersonates the remote object, and a skeleton object is also generated, which does the opposite job of the stub in the remote JVM -- it unmarshals the arguments and invokes the real object. Similarly, the JAX-RPC tools for Web services generate proxy classes for remote Web services that make them appear like local objects.

Whether the generated stub classes are generated as source code or bytecode, code generation still adds extra steps to the compilation process and introduces the potential for confusion because of a proliferation of similarly named classes. On the other hand, the dynamic proxy mechanism allows for the creation of a proxy object at run time without generating stub classes at compile time. In JDK 5.0 and later, the RMI facility uses dynamic proxies instead of generated stubs, with the result being that RMI became easier to use. Many J2EE containers also use dynamic proxies to implement EJBs. EJB technology relies heavily on the use of

interception to implement security and transaction demarcation; dynamic proxies simplify the implementation of interception by providing a central control flow path for all methods invoked on an interface.

---

## The dynamic proxy mechanism

At the heart of the dynamic proxy mechanism is the `InvocationHandler` interface, shown in Listing 1. The job of an invocation handler is to actually perform the requested method invocation on behalf of a dynamic proxy. The invocation handler is passed a `Method` object (from the `java.lang.reflect` package) and the list of arguments to be passed to the method; in the simplest case, it could simply call the reflective method `Method.invoke()` and return the result.

### Listing 1. InvocationHandler interface

```
public interface InvocationHandler {
    Object invoke(Object proxy, Method method, Object[] args)
        throws Throwable;
}
```

Every proxy has an associated invocation handler that is called whenever one of the proxy's methods is called. In keeping with the general design principle that interfaces are for defining types and classes are for defining implementations, proxy objects can implement one or more interfaces, but not classes. Because proxy classes do not have accessible names, they cannot have constructors, so they must instead be created by factories. Listing 2 shows the simplest possible implementation of a dynamic proxy that implements the `Set` interface, and dispatches all `Set` methods (as well as all `Object` methods) to the encapsulated `Set` instance.

### Listing 2. Simple dynamic proxy that wraps a Set

```
public class SetProxyFactory {
    public static Set getSetProxy(final Set s) {
        return (Set) Proxy.newProxyInstance
            (s.getClass().getClassLoader(),
             new Class[] { Set.class },
             new InvocationHandler() {
                 public Object invoke(Object proxy, Method method,
                                     Object[] args) throws Throwable {
                     return method.invoke(s, args);
                 }
             });
    }
}
```

The `SetProxyFactory` class contains one static factory method, `getSetProxy()`, which returns a dynamic proxy implementing `Set`. The proxy object really does implement `Set` -- the caller cannot tell (except by reflection) that the object returned is a dynamic proxy. The proxy returned by `SetProxyFactory` doesn't do anything other than dispatch the method to the `Set` instance passed into the factory method. While reflection code is often hard to read, there's so little going on here that it's not hard to follow to control flow -- whenever a method gets invoked on the `Set` proxy, it gets dispatched to the invocation handler, which simply reflectively invokes the desired method on the underlying wrapped object. Of course, a proxy that did absolutely nothing would be silly -- or would it?

## Do-nothing adapters

There actually is a good use for a do-nothing wrapper such as `SetProxyFactory` -- it can be used to safely

narrow an object reference to a specific interface (or set of interfaces) in such a way that the caller cannot upcast the reference, making it safer to pass object references to untrusted code such as plug-ins or callbacks. Listing 3 contains a set of class definitions that implement a typical callback scenario; you'll see how dynamic proxies can more conveniently replace an Adapter pattern that is commonly implemented by hand (or by code generation wizards provided by IDEs).

### Listing 3. Typical callback scenario

```
public interface ServiceCallback {
    public void doCallback();
}

public interface Service {
    public void serviceMethod(ServiceCallback callback);
}

public class ServiceConsumer implements ServiceCallback {
    private Service service;

    ...

    public void someMethod() {
        ...
        service.serviceMethod(this);
    }
}
```

The `ServiceConsumer` class implements `ServiceCallback` (which is often a convenient way to support callbacks) and passes the `this` reference to `serviceMethod()` as the callback reference. The problem with this approach is there's nothing stopping the `Service` implementation from upcasting the `ServiceCallback` to `ServiceConsumer` and calling methods that the `ServiceConsumer` doesn't intend the `Service` to call. Sometimes you don't care about this risk -- but sometimes you do. If you do, you could make the callback object an inner class, or write a do-nothing adapter class (see `ServiceCallbackAdapter` in Listing 4) and wrap the `ServiceConsumer` with the `ServiceCallbackAdapter`. The `ServiceCallbackAdapter` prevents the `Service` from upcasting the `ServiceCallback` to a `ServiceConsumer`.

### Listing 4. Adapter class that safely narrows an object to an interface so it cannot be upcast by malicious code

```
public class ServiceCallbackAdapter implements ServiceCallback {
    private final ServiceCallback cb;

    public ServiceCallbackAdapter(ServiceCallback cb) {
        this.cb = cb;
    }

    public void doCallback() {
        cb.doCallback();
    }
}
```

Writing adapter classes such as `ServiceCallbackAdapter` is simple but tedious. You have to write a forwarding method for each method in the wrapped interface. In the case of `ServiceCallback`, there was only one method to implement, but some interfaces, such as the `Collections` or `JDBC` interfaces, contain dozens of methods. Modern IDEs reduce the amount of work involved in writing an adapter class by providing a "Delegate Methods" wizard, but you still have to write one adapter class for each interface you want to wrap, and there is something unsatisfying about classes that contain only generated code. It seems like there should be a way to express the "do-nothing narrowing adapter pattern" more compactly.

### A generic adapter class

The `SetProxyFactory` class in [Listing 2](#) is certainly more compact than the equivalent adapter class for the `Set` interface, but it still only works for one interface: `Set`. But by using generics, you can easily create a generic proxy factory that can do the same for any interface, as shown in [Listing 5](#). It is almost identical to `SetProxyFactory`, but it can work for any interface. Now you never have to write a narrowing adapter class again! If you want to create a proxy object that safely narrows an object to interface `T`, simply invoke `getProxy(T.class, object)`, and you've got one, without the extra baggage of a pile of adapter classes.

#### Listing 5. Generic narrowing adapter factory class

```
public class GenericProxyFactory {
    public static<T> T getProxy(Class<T> intf,
        final T obj) {
        return (T)
            Proxy.newProxyInstance(obj.getClass().getClassLoader(),
                new Class[] { intf },
                new InvocationHandler() {
                    public Object invoke(Object proxy, Method method,
                        Object[] args) throws Throwable {
                        return method.invoke(obj, args);
                    }
                });
    }
}
```

## Dynamic proxies as Decorators

Of course, the dynamic proxy facility can do a lot more than simply narrow the type of an object to a specific interface. It is only a short leap from the simple narrowing adapter in [Listing 2](#) and [Listing 5](#) to the Decorator pattern, where the proxy wraps invocations with additional functionality, such as security checks or logging. [Listing 6](#) shows a logging `InvocationHandler`, which writes a log message showing the method invoked, the arguments passed, and the return value, in addition to simply invoking the method on the desired target object. With the exception of the reflective `invoke()` call, all the code here is simply part of generating the debugging message -- and there's still not that much of it. The code for the proxy factory method is almost identical to `GenericProxyFactory`, except that it uses a `LoggingInvocationHandler` instead of the anonymous invocation handler.

#### Listing 6. Proxy-based Decorator that generates debug logging for each method call

```
private static class LoggingInvocationHandler<T>
    implements InvocationHandler {
    final T underlying;

    public LoggingHandler(T underlying) {
        this.underlying = underlying;
    }

    public Object invoke(Object proxy, Method method,
        Object[] args) throws Throwable {
        StringBuffer sb = new StringBuffer();
        sb.append(method.getName()); sb.append("(");
        for (int i=0; args != null && i<args.length; i++) {
            if (i != 0)
                sb.append(", ");
            sb.append(args[i]);
        }
        sb.append(")");
        Object ret = method.invoke(underlying, args);
        if (ret != null) {
            sb.append(" -> "); sb.append(ret);
        }
    }
}
```

```

        }
        System.out.println(sb);
        return ret;
    }
}

```

If you wrap a `HashSet` with a logging proxy and execute the following simple test program:

```

Set s = newLoggingProxy(Set.class, new HashSet());
s.add("three");
if (!s.contains("four"))
    s.add("four");
System.out.println(s);

```

You get the following output:

```

add(three) -> true
contains(four) -> false
add(four) -> true
toString() -> [four, three]
[four, three]

```

This approach is a nice, easy way to add a debugging wrapper around an object. It is certainly a lot easier (and more generic) than generating a delegating class and manually creating a lot of `println()` statements. I could take this approach a lot further; instead of generating the debugging output unconditionally, the proxy could instead consult a dynamic configuration store (initialized from a configuration file, and that could be dynamically modified by a JMX MBean) to determine whether to actually generate the debugging statements, perhaps even on a class-by-class or instance-by-instance basis.

At this point, I expect the AOP fans in the audience to be practically bursting with "But that's what AOP is good for!" And it is, but there is more than one way to solve any given problem -- and just because a technology can solve a problem doesn't mean it is the best solution. In any case, the dynamic proxy approach has the advantage of working entirely within the bounds of "Pure Java," and not every shop uses (or should use) AOP.

## Dynamic proxies as adapters

Proxies can also be used as true adapters, providing a view of an object that exports a different interface than the underlying object implements. The invocation handler need not dispatch every method call to the same underlying object; it could examine the name and dispatch different methods to different objects. As an example, suppose you have a set of JavaBeans interfaces for representing persistent entities (`Person`, `Company`, and `PurchaseOrder`) that specify getters and setters for properties, and you are writing a persistence layer that maps database records to objects implementing these interfaces. Rather than writing or generating a class for each interface, you might instead have one generic JavaBeans-style proxy class, which stores properties in a `Map` instead.

Listing 7 shows a dynamic proxy that examines the name of the called method and implements getter and setter methods directly by consulting or modifying the property map. This one proxy class can now implement objects of multiple JavaBeans-style interfaces.

### Listing 7. Dynamic proxy class that dispatches getters and setters to a Map

```

public class JavaBeanProxyFactory {
    private static class JavaBeanProxy implements InvocationHandler {
        Map<String, Object> properties = new HashMap<String,
            Object>();
    }
}

```

```

    public JavaBeanProxy(Map<String, Object> properties) {
        this.properties.putAll(properties);
    }

    public Object invoke(Object proxy, Method method,
        Object[] args)
        throws Throwable {
        String meth = method.getName();
        if (meth.startsWith("get")) {
            String prop = meth.substring(3);
            Object o = properties.get(prop);
            if (o != null && !method.getReturnType().isInstance(o))
                throw new ClassCastException(o.getClass().getName() +
                    " is not a " + method.getReturnType().getName());
            return o;
        }
        else if (meth.startsWith("set")) {
            // Dispatch setters similarly
        }
        else if (meth.startsWith("is")) {
            // Alternate version of get for boolean properties
        }
        else {
            // Can dispatch non get/set/is methods as desired
        }
    }
}

public static<T> T getProxy(Class<T> intf,
    Map<String, Object> values) {
    return (T) Proxy.newProxyInstance
        (JavaBeanProxyFactory.Class.getClassLoader(),
            new Class[] { intf }, new JavaBeanProxy(values));
}
}

```

While there is some potential loss of type-safety because reflection works in terms of `Object`, the getter handling in `JavaBeanProxyFactory` "bakes in" some of the extra type-checking needed, as I do here with the `isInstance()` check for getters.

## Performance costs

As you've seen, dynamic proxies have the potential to simplify a lot of code -- not only can they replace a lot of generated code, but one proxy class can replace multiple classes of handwritten or generated code. What is the cost? There is probably some performance cost because of dispatching methods reflectively instead of using the built-in virtual method dispatch. In the very early JDKs, reflection performance was poor (as was the performance of nearly everything else in early JDKs), but reflection has gotten a lot faster in the last 10 years.

Without getting into the subject of benchmark construction, I wrote a simple, unscientific test program that loops, stuffing data into a `Set`, randomly inserting, looking up, and removing elements from the `Set`. I ran it with three `Set` implementations: an unadorned `HashSet`, a handwritten `Set` adapter that simply forwards all methods to an underlying `HashSet`, and a proxy-based `Set` adapter that also simply forwards all methods to an underlying `HashSet`. Each loop iteration generated several random numbers and performed one or more `Set` operations. The handwritten adapter generated only a few percent of performance overhead compared to the raw `HashSet` (presumably because of effective inline caching at the JVM level and branch prediction at the hardware level); the proxy adapter was measurably slower than the raw `HashSet`, but the overhead was less than a factor of two.

My conclusion from this experiment is that for the vast majority of cases, the proxy approach performs well enough even for lightweight methods, and as the operations being proxied become more heavyweight (such as remote method calls or methods that use serialization, perform IO, or fetch data from a database), the proxy overhead will effectively approach zero. While there certainly will be cases where the proxy approach

introduces unacceptable performance overhead, these are likely to constitute the minority of situations.

---

## Conclusion

Dynamic proxies are a powerful and underutilized tool in implementing many design patterns, including Proxy, Decorator, and Adapter. Proxy-based implementations of these patterns are easy to write, harder to get wrong, and lend themselves to greater genericness; in many cases, one dynamic proxy class can serve as a Decorator or Proxy for all interfaces, rather than having to write a static class for each interface. For all but the most performance-critical applications, the dynamic proxy approach may be preferable to the handwritten or machine-generated stub approach.

---

## Resources

### Learn

- ["Object Adapter based on Dynamic Proxy"](#) (Heinz Kabutz, Artima Developer, May 2005): Explore the use of dynamic proxies to implement common design patterns.
- ["EJB best practices: The limits of delegation"](#) (Brett McLaughlin, developerWorks, December 2002): Learn how dynamic proxies can simplify the use of the business delegate pattern.
- [Build interoperable Web services with JSR-109](#) (Jeffrey Liu and Yen Lu, developerWorks, August 2003): Find out how JSR-109 supports the use of dynamic proxies for Web Services endpoints, simplifying Web Services development.
- [Java theory and practice](#): The complete series by Brian Goetz.

### Get products and technologies

- [JProxy Javadoc](#): Details and explanation for the JProxy class.

### Discuss

- [Participate in the discussion forum](#).
  - [developerWorks blogs](#): Get involved in the developerWorks community.
- 

## About the author

[Brian Goetz](#) has been a professional software developer for over 18 years. He is a Principal Consultant at [Quiotix](#), a software development and consulting firm located in Los Altos, California, and he serves on several JCP Expert Groups. See Brian's [published and upcoming articles](#) in popular industry publications.

---

