The Proxy Pattern

Dynamic Proxy

Proxies for Remote Access

Proxies in Enterprise Java

Proxy Versus Decorator

Conclusion

Also in This Issue

DESIGN PATTERNS

The Proxy Pattern

A good solution when you need to enable or mediate access to objects, either local or remote

by Ian Darwin

A *proxy* is a stand-in for something else. Corporate shareholders appoint proxies to vote for them at business meetings. Climate scientists use averaged temperature as a proxy for having a thermometer in every square meter of the world. Developers use proxies to substitute for objects that are remote, require protection, or otherwise need mediated access.

The basic approach is that a client object requests a *service object* of a given type from a third party such as a factory, and what it gets is a proxy object that can stand in for, and usually pass control to, the service object. This arrangement requires that the proxy implement the same interface or extend the same class as the one that was requested, so the proxy can be assigned to a variable of the correct type. For this set of examples, I'll use a simple "inspirational quote of the day" `QuoteService` interface with just two methods, which are used as follows:

```
// Normal use
System.out.println("The quote of the day is: " + quo

// Admin use
quoteServer.addQuote("Only the educated are free --
```

All the code samples for this article are in my GitHub repository.

In an application, I might use a factory method to obtain the instance of the server, instead of calling the constructor directly. This step allows more flexibility, and it removes tight coupling or dependence of the client code on a particular class implementing the interface.

```
QuoteService x = getQuoteService(); // Not = new Quo
```

The factory method might simply instantiate a fixed class. But more likely, it will use some configuration to determine which implementation class to create (see the Factory patterns), or it will wrap the known implementation class in a proxy object. I say "wrap" advisedly, because the proxy's main job is to mediate access to the target, so it must maintain a reference to the target.

For these demos, I will use a simple logging proxy, because it's easy to see what the code is doing. The implementation of the `getQuoteService` method might create and return a subclass of the existing `QuoteServerImpl` implementation class, overriding its methods and adding some functionality to the original. This example is short enough that I just use an anonymous class.

```
public static QuoteServer getQuoteServer() {
    final QuoteServer target = new QuoteServerImpl(
    QuoteServer proxy = new QuoteServer() {
        public String getQuote() {
            System.out.println("Calling getQuote()"
            return target.getQuote();
        }
        public void addQuote(String newQuote) {
            System.out.println("Calling addQuote()"
            target.addQuote(newQuote);
        }
    };
    return proxy;
}
```

This example shows a logging proxy where I know the class is being proxied. But it is, in fact, tightly coupled to the target class. What if you want to apply proxying to a variety of classes?

Dynamic Proxy

Java SE provides a mechanism called dynamic proxy, which allows you to synthetically create a proxy for a list of arbitrary interfaces—that is, you can set up a proxy at runtime instead of at compile time. This capability has been around practically forever, since the days of Java 1.3. It does require you to create an object that subclasses `InvocationHandler`. This object will act as the go-between from the caller to the objects being proxied. You can think of the `InvocationHandler` as basically being the proxy. In fact, if you print out the call stack in the target, using either a debugger or `new RuntimeException().printStackTrace()`, you will see that other than some reflection classes, the overall structure is basically the same as in **Figure 1**.

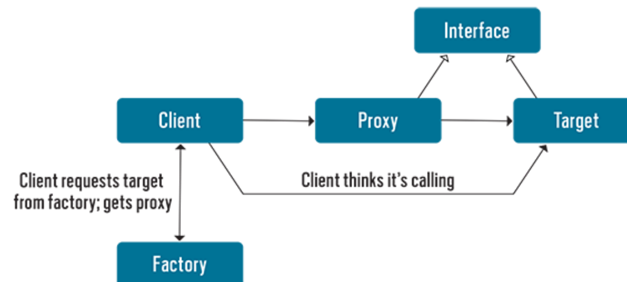

Figure 1. Proxy pattern

The `InvocationHandler` class contains a convenience method, `newProxyInstance(ClassLoader, Class<?>[], InvocationHandler)`
, which, as the name says, gets you a proxy instance for the interfaces given as class descriptors and the given `InvocationHandler`. The `InvocationHandler` interface that you must implement has only one method in it, `invoke`:

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

The arguments passed to your `InvocationHandler`'s `invoke()` method are the proxy object (which the method often doesn't need, but it's there for the times you do), a `java.lang.reflect.Method` descriptor for the method being called, and the list of arguments being passed to that method. Because the API has no way of knowing ahead of time what kinds of objects you will be using, the parameters, the return, and the `throws` clause are written to be as general as possible, using `Object` for the first two and `Throwable` for the third.

Note especially that unless there is a reason not to, the `invoke()` method of the `Invocation- Handler` *must* do the actual invocation of the real target. This is the call to `method.invoke()` in the middle of my demo handler's `invoke()` method—the same name and the same arguments, minus the method descriptor itself, which is the object on which you call `invoke()`.

Here is a version of the logging proxy done as an `InvocationHandler`:

```
class MyInvocationHandler implements InvocationHandl

    private Object target;

    public MyInvocationHandler(Object target) {
        super();
        this.target = target;
    }

    /**
     * Method that is called for every call into the
     * this must invoke the method on the real objec
     * This method demonstrates both logging and sec
     */
    public Object invoke(Object proxy, Method methoc
      throws Throwable {
        String name = method.getName() + "()";
        System.out.println("Proxy got request for "
        // Could put security checking here
        Object ret = method.invoke(target, argList)
        System.out.println("Proxy returned from " +
        return ret;
    }
}
```

Here is the `getQuoteServer()` method for the client program:

```
// from DynamicProxyDemo.java

public static QuoteServer getQuoteServer() {
    QuoteServer target = new QuoteServerImpl();
    InvocationHandler handler = new MyInvocationHanc
    return (QuoteServer) Proxy.newProxyInstance(
        QuoteServer.class.getClassLoader(),
        new Class[] { QuoteServer.class }, handler)
}
```

If you examine the object returned from this method by calling `getClass().getName()` on it, you can see that it is a synthetic class, as indicated by its generated name:

```
QuoteServer object is com.sun.proxy.$Proxy0
```

In my online example of dynamic proxy, in my version of the `InvocationHandler`, I added a few lines as a proxy for real security protection. In place of the code comment "could put security checking here," I wrote this:

```
final String userName = System.getProperty("user.nar
if (name.startsWith("add") && !userName.equals("ian'
    throw new SecurityException(
        "User " + userName + " not allowed to add qu
```

The net result of all this coding is that we have a proxy made up of the dynamic proxy (class `$Proxy0` in this example) and the `InvocationHandler`. The dynamic proxy is generated for us, and the `InvocationHandler` doesn't need to know anything about the actual target, although in more complicated cases it might.

Proxies for Remote Access

Here's one last example from standard APIs: remote access. There's a general term, *remote procedure call* (RPC), for which there are dozens of examples throughout the history of networked computing. The basic idea is that after some setup (such as getting an object from a factory), you invoke an object by using what looks like a local method call, but the object is actually a network proxy that communicates over the network to a peer proxy on the server side, which in turn calls the real service; and the return value is passed back over the same channel. Older examples of RPC include Sun RPC, DCE RPC, and Microsoft Windows RPC. Standard Java APIs that use the RPC paradigm include RMI, CORBA (which was removed from Java 11), JAX-RS, and JAX-WS.

There is not room here to give a full working example, but see my RMI tutorial online for an example.

Proxies in Enterprise Java

The dynamic proxy mechanism works nicely for situations where you know the class or classes to be proxied; however, the `InvocationHandler` itself does not need to be written in a target-specific way. There are cases where you might not know the target class in advance, but you still want to provide services to it. A common example from enterprise Java is the provision of transactional control to business-tier objects controlling data access objects. Both Java EE (now Jakarta) and the Spring Framework provide annotations that are normally placed on business-tier classes and cause a proxy object to begin or join a transaction when a given method begins executing. The proxy will either commit the transaction when the method returns normally or roll it back if the method returns abnormally (for example, by throwing an exception). Here is some pseudocode for a persistent shopping cart using this approach:

```
public class ShoppingService {
    private ShoppingCart cart;
    private Dao dao;

    @Transactional(TransactionType.REQUIRED)
    public void addToCart(Product p) {
        // do validation/calculation work here
        dao.saveCart(cart);
    }
    ...
}
```

The important thing to note is that, in this scheme, *you don't need to write the proxy* or even know its class name for common operations such as transactional control, because these common proxies are provided by the enterprise framework (CDI/EJB or Spring) in response to the annotations. Nor do you need to modify your code to use the proxy (other than annotating it), which means you don't have any runtime dependencies. This design makes the services and data layers easier to unit test (unit testing, after all, means testing each unit in isolation).

However, CDI and Spring give you the option to provide additional proxies of your own. For example, the CDI mechanism supports a form of proxying that uses `Interceptors` that can be applied to enterprise components via annotations (usually) or XML configuration.

Here is how a CDI implementation of the logging interceptor might be used in a business method (the curly braces around the class descriptor remind you that it's an array, in case you want to apply multiple interceptors to the same method). This annotation also can be applied at the class level.

```
@Interceptors({CdiLoggingInterceptor.class})
public void validateCredit() {
```

```
        // do some work here
    }
```

Here is the code for the logging interceptor or proxy:

```
import javax.interceptor.AroundInvoke;
import javax.interceptor.Interceptor;
import javax.interceptor.InvocationContext;

/**
 * A logging interceptor for CDI.
 */
@Interceptor
public class CdiLoggingInterceptor {

    // @AroundInvoke applies to business method; the
    // also annotations for constructors, timeouts,
    @AroundInvoke
    public Object log(InvocationContext ctx) throws
        Object[] parameters = ctx.getParameters();
        String firstArg = (parameters.length > 0) ?
            "First is: " + formatArg(parameters[0])
        String methodName = ctx.getMethod().getName
        log(String.format("About to call %s with %d
                methodName, parameters.length, first
        Object o = ctx.proceed();   // The actual ca
        log("Returned " + format(o) + " from method
        return o;
    }
    ...
}
```

Unlike the dynamic proxy API, in this code a single parameter, an `InvocationContext`, is passed. It contains the method descriptor, the arguments, and so on. The `InvocationContext` has a `getMethod()` call that returns the standard `Method` descriptor and a `getParameters()` call that provides the argument list if you want to examine or modify it. The `format()` and `log()` methods aren't shown here but are in the online source code. The context `proceed()` method takes the place of the `invoke()` method.

You might think this approach is a Decorator rather than a Proxy because you are naming the implementation class. However, as an advanced topic, CDI does allow you to use an interface in the `@Interceptors` and resolve the implementation class at runtime by using other annotations. See the official documentation for more details on the `javax.interceptor` package.

Proxy Versus Decorator

As I mentioned in a previous article on the Decorator pattern, Proxy and Decorator both allow you to wrap extra functionality around an object, so the implementation code can look similar. Although there is often overlap, the primary differences are:

- Proxy is primarily about mediating access, whereas Decorator is about adding functionality.
- Proxy is normally hidden from the client (by some kind of creational method), but the client is aware that it is using a Decorator because it must do so explicitly.

Conclusion

Proxy is a good pattern when you need to control access to objects for any purpose, and it can be used for a wide variety of purposes, including enforcing security restrictions, auditing method calls and parameters, hiding the complexity of access (such as with remote objects), or transparently adding behavior (such as logging).

Also in This Issue

- Javalin: A Simple, Modern Web Server Framework
- Building Microservices with Micronaut
- Helidon: A Simple Cloud Native Framework
- Loop Unrolling
- Quiz Yourself
- Size Still Matters
- Book Review: Modern Java in Action

Ian Darwin

Ian Darwin (@Ian_Darwin) is a Java Champion who has done all kinds of development, from mainframe applications and desktop publishing applications for UNIX and Windows, to a desktop database application in Java, to healthcare apps in Java for Android. He's the author of *Java Cookbook* and *Android Cookbook* (both from O'Reilly). He has also written a few courses and taught many at Learning Tree International.

Share this Page

Contact

- US Sales: +1.800.633.0738
- Global Contacts
- Support Directory
- Subscribe to Emails

About Us

- Careers
- Communities
- Company Information
- Social Responsibility Emails

Downloads and Trials

- Java for Developers
- Java Runtime Download
- Software Downloads
- Try Oracle Cloud

News and Events

- Acquisitions
- Blogs
- Events
- Newsroom

ORACLE | Integrated Cloud
Applications & Platform Services

© Oracle | Site Map | Terms of Use & Privacy | Cookie Preferences | Ad Choices