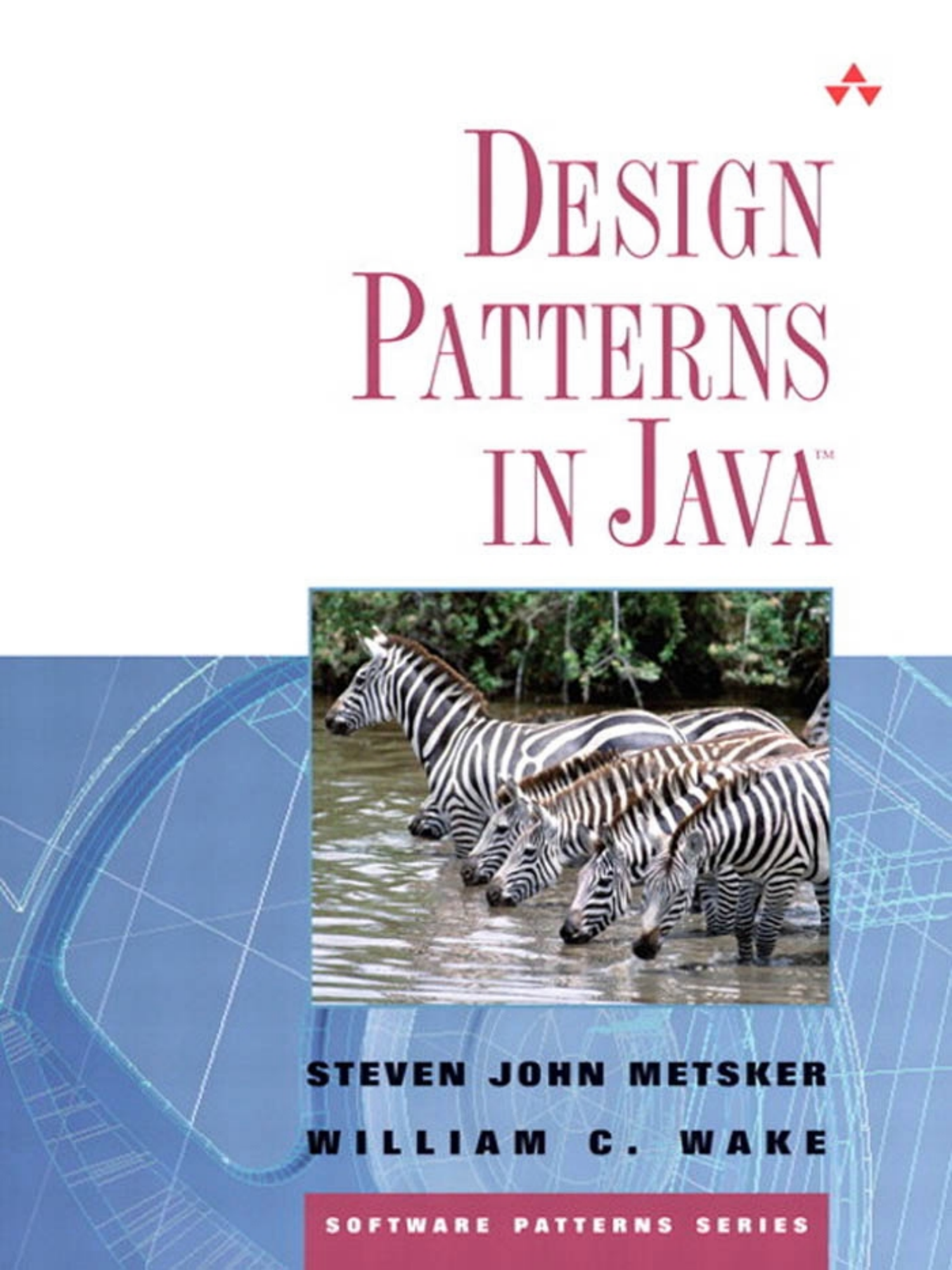
DESIGN PATTERNS IN JAVA™
STEVEN JOHN METSKER
WILLIAM C. WAKE
SOFTWARE PATTERNS SERIES

# DESIGN PATTERNS IN JAVA™

If you compile and run this code, the program displays a confirmation that the rocket is registered:

```
Registered biggie
```

You need to replace the `//Challenge!` line in the `RegisterRocket` class with code that creates a `biggie` object that models a rocket. The remaining code in the `main()` method registers this object. A description of the mechanics of the `Naming` class is outside the scope of this discussion. However, you should have enough information to create the `biggie` object that this code registers.

### CHALLENGE 11.4

Replace the `//Challenge!` line with a declaration and instantiation of the `biggie` object. Define `biggie` to model a rocket with a price of $29.95 and an apogee of 820 meters.

*A solution appears on page 377.*

Running the `RegisterRocket` program makes a `RocketImpl` object—specifically, `biggie`—available on a server. A client that runs on another machine can access `biggie` if the client has access to the `Rocket` interface and the `RocketImpl_Stub` class. If you are working on a single machine, you can still test out RMI, accessing the server on `localhost` rather than on another host.

```
package com.oozinoz.remote;

import java.rmi.*;
public class ShowRocketClient {
    public static void main(String[] args) {
        try {
            Object obj = Naming.lookup(
                "rmi://localhost:5000/Biggie");
```

```
            Rocket biggie = (Rocket) obj;
            System.out.println(
                "Apogee is " + biggie.getApogee());
        } catch (Exception e) {
            System.out.println(
                "Exception while looking up a rocket:");
            e.printStackTrace();
        }
    }
}
```

When this program runs, it looks up an object with the registered name of "Biggie." The class that is serving this name is `RocketImpl`, and the object `obj` that `lookup()` returns will be an instance of `RocketImpl_Stub` class. The `RocketImpl_Stub` class implements the `Rocket` interface, so it is legal to cast the object `obj` as an instance of the `Rocket` interface. The `RocketImpl_Stub` class actually subclasses a `RemoteStub` class that lets the object communicate with a server.

When you run the `ShowRocketClient` program, it prints out the apogee of a "Biggie" rocket.

```
Apogee is 820.0
```

Through a proxy, the `getApogee()` call is forwarded to an implementation of the `Rocket` interface that is active on a server.

### CHALLENGE 11.5

Figure 11.6 shows the `getApogee()` call being forwarded. The rightmost object appears in a bold outline, indicating that it is active outside the `ShowRocketClient` program. Fill in the class names of the unlabeled objects in this figure.

*A solution appears on page 377.*

The benefit of RMI is that it lets client programs interact with a local object that is a proxy for a remote object. You define the interface for the object that you want clients and servers to share. RMI supplies the communication mechanics and isolates both server and client from

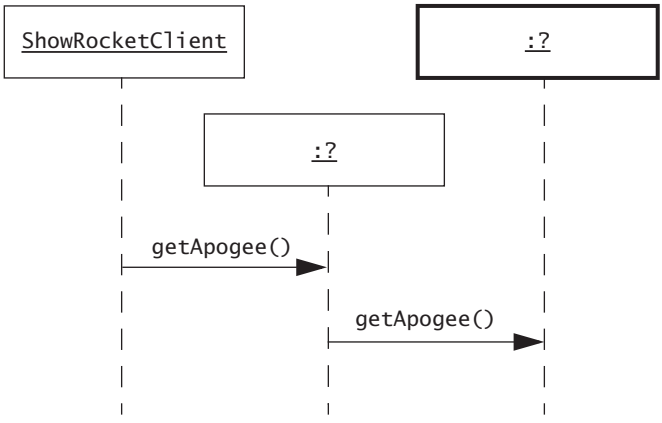


**FIGURE 11.6** This diagram, when completed, will show the flow of messages in an RMI-based distributed application.

the knowledge that two implementations of `Rocket` are collaborating to provide nearly seamless interprocess communication.

## Dynamic Proxies

The engineers at Oozinoz occasionally face performance problems. They'd like a way to instrument code without making major changes to the design.

Java has a feature that can help with this: *Dynamic proxies* let you wrap a proxy object around another object. You can arrange for the outer object—the proxy—to intercept all the calls intended for the wrapped object. The proxy will usually pass these calls on to the wrapped object, but you can add code that executes before or after the intercepted calls. Limitations to dynamic proxies prevent you from wrapping any arbitrary object. Under the right conditions, though, dynamic proxies give you complete control over the operation of an object that you want to wrap with a proxy.

Dynamic proxies work with the *interfaces* that an object's class implements. The calls that the proxy can intercept are calls that one of these interfaces defines. If you have a class that implements an interface

with methods you want to intercept, you can use dynamic proxies to wrap an instance of that class.

To create a dynamic proxy, you must have a list of the interfaces that you want to intercept. Fortunately, you can usually obtain this list by interrogating the object that you want to wrap, using a line such as:

```
Class[] classes = obj.getClass().getInterfaces();
```

This code establishes that the methods you want to intercept are those that belong to interfaces that an object's class implements. To build a dynamic proxy, you need two other ingredients: a class loader and a class that contains the behavior that you want to execute when your proxy intercepts a call. As with the list of interfaces, you can obtain an appropriate class loader by using the one associated with the object that you want to wrap:

```
ClassLoader loader = obj.getClass().getClassLoader();
```

The last ingredient that you need is the proxy object itself. This object must be an instance of a class that implements the `InvocationHandler` interface in the `java.lang.reflect` package. That interface declares the following operation:

```
public Object invoke(Object proxy, Method m, Object[] args)
    throws Throwable;
```

When you wrap an object in a dynamic proxy, calls intended for the wrapped object are diverted to this `invoke()` operation, in a class that you supply. Your code for the `invoke()` method will probably need to pass each method call on to the wrapped object. You can pass on the invocation with a line such as:

```
result = m.invoke(obj, args);
```

This line uses reflection to pass along the desired call to the wrapped object. The beauty of dynamic proxies is that you can add any behavior you like before or after executing this line.

Suppose that you want to log a warning if a method takes a long time to execute. You might create an `ImpatientProxy` class with the following code:

```
package app.proxy.dynamic;

import java.lang.reflect.*;

public class ImpatientProxy implements InvocationHandler {
    private Object obj;

    private ImpatientProxy(Object obj) {
        this.obj = obj;
    }

    public Object invoke(
            Object proxy, Method m, Object[] args)
            throws Throwable {
        Object result;
        long t1 = System.currentTimeMillis();
        result = m.invoke(obj, args);
        long t2 = System.currentTimeMillis();
        if (t2 - t1 > 10) {
            System.out.println(
                "> It takes " + (t2 - t1)
                + " millis to invoke " + m.getName()
                + "() with");
            for (int i = 0; i < args.length; i++)
                System.out.println(
                    ">     arg[" + i + "]: " + args[i]);
        }
        return result;
    }
}
```

This class implements the `invoke()` method so that it checks the time it takes for the wrapped object to complete an invoked operation. If that execution time is too long, the `ImpatientProxy` class prints a warning.

To put an `ImpatientProxy` object to use, you need to use the `Proxy` class in the `java.lang.reflect` package. The `Proxy` class will need a list of interfaces and a class loader, as well as an instance of

ImpatientProxy. To simplify the creation of a dynamic proxy, we might add the following method to the ImpatientProxy class:

```
public static Object newInstance(Object obj) {
    ClassLoader loader = obj.getClass().getClassLoader();
    Class[] classes = obj.getClass().getInterfaces();
    return Proxy.newProxyInstance(
        loader, classes, new ImpatientProxy(obj));
}
```

This static method creates the dynamic proxy for us. Given an object to wrap, the newInstance() method extracts the object's list of interfaces and class loader. The method instantiates the ImpatientProxy class, passing it the object to wrap. All these ingredients are then passed to the Proxy class's newProxyInstance() method.

The returned object will implement all the interfaces that the wrapped object's class implements. We can cast the returned object to any of these interfaces.

Suppose that you are working with a set of objects, and some operations seem to run slowly for some objects. To find which objects are behaving sluggishly, you can wrap the set in an ImpatientProxy object. The following code shows this example:

```
package app.proxy.dynamic;

import java.util.HashSet;
import java.util.Set;
import com.oozinoz.firework.Firecracker;
import com.oozinoz.firework.Sparkler;
import com.oozinoz.utility.Dollars;

public class ShowDynamicProxy {
    public static void main(String[] args) {
        Set s = new HashSet();
        s = (Set)ImpatientProxy.newInstance(s);
        s.add(new Sparkler(
            "Mr. Twinkle", new Dollars(0.05)));
        s.add(new BadApple("Lemon"));
        s.add(new Firecracker(
            "Mr. Boomy", new Dollars(0.25)));
        System.out.println(
            "The set contains " + s.size() + " things.");
    }
}
```