10.03

Javamagazin

Deutschland € 5,50 Österreich € 6,50
Schweiz SFr 10,70 / Luxemburg € 6,50

S&S

Javamagazin™

Internet & Enterprise Technology

XML extra included

Jakarta Struts

Das Framework ■ Neu in Struts 1.1 ■ Visuelle Struts-Tools ■ Dynamische Formulare

Code-Generierung mit MDA

Die wichtigsten Techniken und Generatoren

Rich Clients für J2EE
Alternative Canoo ULC

Web Services mit Apache WSIF
Unabhängig vom Transportprotokoll

Collaborative Development
Herausforderung weltweite Entwicklung

Skripte in Java
BeanShell anwenden

Tom C@ – Die Kolumne
Administration und Management des Tomcat-Servers

mit CD!

www.javamagazin.de

D 45867

4 194586 705506 10

Praktische Anwendungsszenarien des Dynamic Proxy API

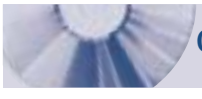
von Stephan Anft und Peter Friese

Der Prokurist

Welcher Entwickler hat sich nicht schon darüber geärgert, alle Klassen einer Anwendung noch mal anfassen zu müssen, um beispielsweise ein Logging einzubauen? Wer die aktuelle Entwicklung im Bereich Java mitverfolgt, kommt vielleicht auf die Idee, für diese Zwecke AspectJ einzusetzen. Doch auch mit Java-Bordmitteln lassen sich Probleme dieser Art elegant lösen. Das Dynamic Proxy API bietet hier eine interessante Alternative.

Das Proxy-Pattern

Im Allgemeinen agiert ein Proxy als Stellvertreter eines nachgelagerten Objekts (im Folgenden „Subjekt" genannt). Die so eingeführte Indirektion kann der Proxy dazu nutzen, den Zugriff auf das Subjekt zu kontrollieren oder den Aufruf um bestimmte Dienste (z.B. Logging) anzureichern. Für den Aufrufer ist dieses Verhalten in der Regel transparent, sodass er nicht merkt, ob er mit einem Proxy oder dem Subjekt direkt arbeitet [1]. Die Transparenz wird in der Objektorientierung erzeugt, indem der Proxy das Interface des Subjekts implementiert und so als Subjekt auftreten kann.



Das Proxy-Pattern der Gang of Four [2] beschreibt den eben geschilderten Sachverhalt im Detail und nennt als Hauptmotivation die Kontrolle über das Subjekt. In Abbildung 1 ist das Proxy-Pattern dargestellt. Je nach Einsatzzweck unterscheidet man die in Tabelle 1 zusammengefassten Arten von Proxies.

Das Proxy-Pattern besteht aus folgenden Teilnehmern:

- *Subject*: Die gemeinsame Schnittstelle vom Proxy und dem eigentlichen Subjekt (*RealSubject*).
- *RealSubject*: Hierbei handelt es sich um das eigentliche Objekt.
- *Proxy*: Der Proxy hält eine Referenz auf das *RealSubject* und implementiert die identische Schnittstelle.

Das Dynamic Proxy API

Das Dynamic Proxy API ist seit dem JDK 1.3 verfügbar. Von der Funktionsweise her dem Proxy-Pattern ähnlich, ist die Implementierung doch sehr unterschiedlich. Während beim Proxy-Pattern ein Proxy in der Regel nur ein Subjekt kapseln kann, kann der Dynamic Proxy beliebig viele Subjekte kapseln.

Bei der Implementierung einer Remoting-Architektur macht sich dies sehr schnell bezahlt. Selbst eine große Anzahl von Remote-Schnittstellen kann durch einen einzigen Dynamic Proxy dargestellt werden, da der Dynamic Proxy zur Laufzeit alle benötigten Remote-Objekte (d.h. Subjekte) repräsentieren kann. Ein Dynamic Proxy besteht aus folgenden Bestandteilen [3]:

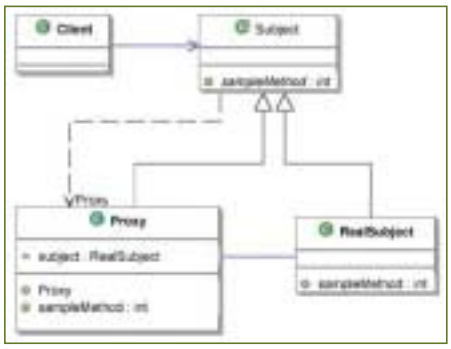


Abb. 1: Das Proxy-Pattern der Gang of Four

- Einer Dynamic Proxy Class,
- einem oder mehreren Proxy-Interfaces sowie
- Proxy Instance, eine Instanz der Proxy Class.

Um die *Proxy*-Klasse erzeugen zu können, verwendet man die statische Methode *getProxyClass()* der Klasse *java.lang.reflect.Proxy*. Als Parameter werden der Methode ein Classloader sowie ein Array mit den Schnittstellen übergeben. Die Proxy-Instanz wird im Kontext des übergebenen Classloaders erzeugt, die Reihenfolge der Interfaces ist relevant (siehe Kasten „Eigenschaften der Proxy Class"). Ein Aufruf der Methoden mit identischen Interfaces, allerdings in anderer Reihenfolge, erzeugt eine neue Proxy-Instanz. Die Proxy-Instanz wird dann durch den Aufruf des Konstruktors mit Hilfe des Java Reflection APIs und durch Übergabe eines Invocation Handlers erzeugt (Listing 1).

Um diese umständliche Erzeugung zu umgehen, verwendet man in der Regel die Methode *newProxyInstance()* der Klasse *Proxy*. So werden sowohl die *Proxy*-Klasse als auch der Invocation Handler mit einem einzigen Aufruf erzeugt:

```
Subject subjectProxy = (Subject)Proxy.newProxyInstance(
 Subject.class.getClassLoader(),
 new Class[] {Subject.class},
 new SubjectInvocationHandler());
```

Remote Proxy	Der Proxy kapselt den Zugriff auf ein entferntes Subjekt.
Virtueller Proxy	Der virtuelle Proxy bewirkt ein lazy loading bei teuren Subjekten, z.B. wenn eine Stücklistenhierarchie nicht vollständig aus der Datenbank geladen wird, sondern immer nur wie vom Benutzer benötigt.
Schutz-Proxy	Dieser Proxy kontrolliert den Zugriff auf ein Subjekt, indem beispielsweise nur autorisierte Zugriffe vom Proxy weitergeleitet werden.

Tab. 1: Die verschiedenen Proxies

Einfaches Beispiel

Als einfaches Beispiel soll die allseits bekannte „HelloWorld"-Anwendung genutzt werden. Wie man am Quellcode der *Proxy*-Klasse erkennen kann, lässt sich die Erstellung der Proxy-Klasse und die Erzeugung der Proxy-Instanz in die Klasse einbinden, wodurch der Umgang mit dem Proxy erheblich erleichtert wird (Listings 2 bis 4).

Der Dynamic Proxy in diesem Beispiel ist nicht auf die HelloWorld-Anwendung festgelegt. Im Gegenteil: Jedes beliebige Subjekt kann durch ihn repräsentiert werden. Nachdem der Methode *newInstance()* das entsprechende reale Subjekt übergeben wurde, muss der Rückgabewert durch einen Typecast in das entsprechende Subjekt umgewandelt werden. Durch die Klasse *DynamicProxy* haben wir bereits jetzt einen Proxy geschaffen, der beliebige Klassen repräsentieren kann.

Besonderheiten

Es gibt einige Besonderheiten, die man bei der Benutzung von Dynamic Proxies beachten muss [4]. Besondere Beachtung verdienen die Methoden *equals()*, *hashCode()* und *toString()* der Klasse *java.lang.Object*. Auch wenn diese Methoden von einer der Schnittstellen deklariert werden, die der Proxy repräsentiert, wird beim Aufruf der Methode *invoke()* im Proxy die Klasse *java.lang.Object* statt der Schnittstelle als deklarierende Klasse des *Method*-Objekts angegeben.

Ein weiterer Sonderfall ist die mehrfache Definition einer identischen Methode. Diese mehrdeutigen Methodensignaturen können auftreten, wenn der Proxy mehr als ein Interface repräsentiert. In diesem Fall spielt die Reihenfolge der Schnittstellen im übergebenen Array eine wichtige Rolle. Wird vom Client eine dieser mehrfach deklarierten Methoden aufgerufen, verwendet der Proxy immer die Methode der Schnittstelle, die an vorderster Stelle im Array hinterlegt ist. Hat der Client also ursprünglich die Methode in einer Schnittstelle aufgerufen, die weiter hinten im Ar-

Eigenschaften der Proxy Class

Die *Proxy*-Klasse hat die folgenden Eigenschaften [4]:

- Die Klasse ist *public, final* und nicht abstrakt.
- Der Name der *Proxy*-Klasse ist nicht vorgegeben. Namen allerdings, die mit *$Proxy* beginnen, sollten für *Proxy*-Klassen reserviert sein.
- Die Klasse ist abgeleitet von *java.lang.reflect.Proxy*.
- Die *Proxy*-Klasse implementiert exakt die Interfaces, die bei der Erzeugung spezifiziert wurden, in der gleichen Reihenfolge.
- Implementiert die *Proxy*-Klasse ein Interface, das nicht *public* ist, so muss sich die *Proxy*-Klasse im gleichen Paket befinden. Dies ist alleine schon durch die Java-Compilereigenschaften vorgegeben.
- Da die *Proxy*-Klasse alle Interfaces implementiert, die bei der Erzeugung angegeben wurden, liefert auch der Aufruf von *getInterfaces()* ein Array aller implementierten Interfaces, in der angegebenen Reihenfolge. Die Methode *getMethods()* liefert demzufolge eine Liste mit allen Methoden aller Interfaces.
- Mit der Methode *isProxyClass()* kann überprüft werden, ob es sich beim übergebenen Objekt um einen Proxy handelt.
- Die Sicherheitsdomäne der *Proxy*-Klasse entspricht der von Systemklassen, also beispielsweise *java.lang.Object*. Diese Klassen erhalten standardmäßig alle Berechtigungen.
- Die *Proxy*-Klasse hat einen öffentlichen Konstruktor, der ein Argument übergeben bekommt. Bei diesem Argument handelt es sich um eine Implementierung des Invocation Handler für die Proxy-Instanz.

ray liegt, so führt dies dazu, dass der Invocation Handler ein falsches *Method*-Objekt übergeben bekommt. Dieses gilt übrigens nicht nur für die Methoden der Schnittstellen selbst, sondern auch für die Methoden, die in den Superklassen der jeweiligen Schnittstellen definiert sind. Dieses Verhalten rührt daher, dass die *Proxy*-Klasse nicht zur Laufzeit herausfinden kann, von welchem Interface die *invoke()*-Methode aufgerufen wurde. Listings 5 bis 7 verdeutlicht diesen Effekt.

Das Beispiel wird durch den folgenden Code aufgerufen:

```
InvocationHandler handler = new StdOutInvocationHandler();
SubjectB b = (SubjectB)Proxy.newProxyInstance(
                SubjectB.class.getClassLoader(), new Class[]
                { SubjectA.class, SubjectB.class }, handler);

b.doSomething();
b.doSomethingElse();
```

Der Invocation Handler gibt auf der Konsole den Namen der deklarierenden Klasse des *Method*-Objekts aus. Folgende Ausgabe wird bei dem Beispiel auf der Konsole erzeugt:

```
SubjectA
SubjectB
```

Obwohl beide Methoden in der Schnittstelle *SubjectB* aufgerufen werden, wird für das *Method*-Objekt der Methode *doSomething()* „*SubjectA*“ als deklarierende Klasse ausgegeben, da diese im übergebenen Schnittstellen-Array an erster Stelle hinterlegt ist.

Haben diese Besonderheiten der *Object*-Methoden und der mehrdeutigen Methodendeklarationen aber nun eine praktische Relevanz für die Benutzung der Dynamic Proxies? Eher weniger. Wie schon am Beispiel des HelloWorld-Proxys gezeigt, wird in der Praxis gerne die Implementierung des Invocation Handlers und die Erzeugung des Proxy-Objekts zur einfacheren Handhabung in einer Klasse vereinigt. Der Proxy wird durch Übergabe des realen Subjekts erzeugt und repräsentiert alle Schnittstellen des realen Subjekts. Ein Methodenaufruf auf dem Proxy-Objekt wird von diesem an das reale Subjekt weitergeleitet. Durch diese Art der Implementierung spielt die deklarierende Klasse der Methode keine Rolle.

Listing 1

```
java.lang.reflect.InvocationHandler handler =
                        new SubjectInvocationHandler();
Class proxyClass = java.lang.reflect.Proxy.getProxyClass
            (Subject.class.getClassLoader(), new Class[] {
                                                Subject.class });
Subject subjectProxy = (Subject)proxyClass.
            getConstructor(new Class[] {java.lang.reflect.
                InvocationHandler.class }).newInstance
                                    (new Object[] { handler });
```

Listing 2

```
package definition.dynamic;

public interface HelloWorld {
 public void sayHelloWorld();
}
```

Listing 3

```
package definition.dynamic;

public class HelloWorldImpl implements HelloWorld {

 public void sayHelloWorld() {
  System.out.println("Hello world.");
 }

}
```

Listing 4

```
package definition.dynamic;

import java.lang.reflect.*;

public class DynamicProxy implements InvocationHandler
{
 /** Invocation Handler Objekt */
 private Object obj;

 public DynamicProxy(Object obj) {
  this.obj = obj;
 }

 public static Object newInstance(Object obj) {
  return java.lang.reflect.Proxy.newProxyInstance(
   obj.getClass().getClassLoader(),
   obj.getClass().getInterfaces(),
   new DynamicProxy(obj));
 }

 public Object invoke(Object proxy, Method method,
                                Object[] args) throws Throwable {
  Object result;
  try {
   System.out.println("Proxy invokes implementation...");
   result = method.invoke(obj, args);
  }
  catch (InvocationTargetException e) {
   throw e.getTargetException();
  }
  catch (Exception e) {
   throw new RuntimeException(
    "unexpected invocation exception: " + e.getMessage());
  }
  return result;
 }
}
```

Begriffsdefinitionen

- Proxy Class: Eine Klasse, die eine zur Laufzeit festgelegte Menge von Interfaces implementiert. Diese Klasse hat bestimmt Eigenschaften (siehe Kasten „Eigenschaften der Proxy Class“).
- Proxy Interface: Ein Interface, das von einer Proxy Class implementiert wird.
- Proxy Instance: Ist eine Instanz einer Proxy Class. Jeder Proxy Instance ist ein Invocation Handler zugeordnet.
- Invocation Handler: Implementiert das Interface *InvocationHandler* und stellt die Methode *invoke()* zur Verfügung, die bei jedem Aufruf einer Methode der Proxy Class aufgerufen wird. Der Invocation Handler muss diesen Aufruf verarbeiten und ein geeignetes Ergebnis zurückliefern.

Listing 5

```
public interface SubjectA {
 public void doSomething();
}
```

Listing 6

```
public interface SubjectB {
 public void doSomething();
 public void doSomethingElse();
}
```

Listing 7

```
public class StdOutInvocationHandler implements
                                        InvocationHandler {

 public StdOutInvocationHandler() {
  super();
 }

 public Object invoke(Object proxy, Method method,
  Object[] args) throws Throwable {
  System.out.println(method.getDeclaringClass().
                                                getName());
  return null;
 }
}
```

Dennoch sollten diese Besonderheiten im Hinterkopf bleiben, da es auch Proxy-Implementierungen gibt, die die deklarierende Klasse der Methode benötigen und deshalb diese Besonderheiten beachten müssen.

Dynamic Proxy vs. GoF-Proxy

Vergleicht man den Dynamic Proxy mit dem Proxy-Pattern der GoF so wird deutlich, dass der Zweck identisch ist. Allerdings unterscheiden sich Proxy-Pattern und Dynamic Proxy wesentlich in der Implementierung. Während der GoF-Proxy in der Regel nur ein Subjekt repräsentiert, kann der Dynamic Proxy eine beliebige Menge von Subjekten widerspiegeln, und das mit sehr geringem Implementierungsaufwand. Im HelloWorld-Beispiel haben wir das anschaulich gesehen. Während beim Proxy-Pattern jede Methode des Subjekts implementiert werden muss (allein schon der Compiler erzwingt dies), existiert beim Dynamic Proxy typischerweise nur eine einzige Methode, die beliebige Methoden des Subjekts bedienen kann. Möglich ist dies durch ausgiebige Nutzung der Reflection-Möglichkeiten von Java. Dynamic Proxies - der Name sagt es bereits - haben ein dynamisches Verhalten, während GoF-Proxies statischer Natur sind. Beides kann Vor- und Nachteile haben. Der Einsatz von Reflection hat generell negative Auswirkungen auf die Performance einer Applikation. Eventuell ist man aber bereit, diesen Nachteil zu tragen, um den Vorteil des dynamischen Verhaltens und somit der Rekonfiguration der Software zur Laufzeit zu nutzen. Positiv zu bewerten ist außerdem die Zeitersparnis bei der Entwicklung, da in der Regel nur ein Proxy für eine beliebige Anzahl von Klassen implementiert werden muss.

Einsatzzwecke

Nachdem nun die Grundlagen erläutert wurden, soll anhand einiger Beispiele gezeigt werden, wofür Dynamic Proxies eingesetzt werden können. Denkbare Einsatzmöglichkeiten sind:

- Logging
- Profiling
- Call History
- Remoting

Logging

Da jeder Aufruf auf einer Proxy Instance vom Invocation Handler abgefangen wird, kann der Invocation Handler dafür sorgen, dass alle Aufrufe mitgeloggt werden. Wie das geschieht und welche Informationen dabei aufgezeichnet werden sollen, bleibt der Phantasie des Einzelnen bzw. den Anforderungen des Projekts überlassen. In unserem Beispiel wird einfach der Methodenname auf die Konsole ausgegeben. Möglich ist z.B. auch, die Werte und Typen der tatsächlichen Parameter auszulesen und anzuzeigen (Listing 8).

Profiling

Konsequent weiterverfolgt ergibt sich aus der Idee alle Aufrufe aufzuzeichnen die Möglichkeit, die Laufzeiten aller Aufrufe zu berechnen (Listing 9).

Call History

Eine weiterer Verwendungszweck für Dynamic Proxies wird in [5] aufgezeigt: Hier wird nach jedem erfolgreichen Methodenaufruf die Signatur des Aufrufs in einer Liste gespeichert. Schlägt ein Aufruf fehl, kann als Reaktion auf diesen Fehler die Liste der zuletzt aufgerufenen Methoden auf dem entsprechenden Objekt ausgegeben werden.

Remoting

Nachdem wir uns nun einige einfache Einsatzzwecke für Dynamic Proxies angesehen haben, werden wir uns nachfolgend mit der Frage beschäftigen, wie man mit Dynamic Proxies eine Remoting-Architektur aufbauen kann. Die Anforderungen an die zu entwickelnde Remoting-Architektur sind überschaubar:

- Einfach, d.h. es sollen keine zusätzlichen Dateien generiert werden.

Listing 8

```
package com.lhsystems.proxies.echo;

import java.lang.reflect.*;

public class EchoProxy implements InvocationHandler {

 ...

 public Object invoke(
  Object proxy,
  Method method,
  Object[] args)
  throws Throwable {

  try {
   System.out.println("Methode: " + method.getName());
   return method.invoke(delegate, args);
  }
  catch (IllegalAccessException e) {
   throw e;
  }
  catch (InvocationTargetException e) {
   throw e.getTargetException();
  }
 }

...

}
```

Listing 9

```
package com.lhsystems.proxies.performance;

import java.lang.reflect.*;

public class PerformanceProxy implements
                                    InvocationHandler {

 ...

 public Object invoke(
  Object proxy,
  Method method,
  Object[] args)
  throws Throwable {

  Object result = null;

  try {
   long start = System.currentTimeMillis();
   result = method.invoke(delegate, args);
   long stop = System.currentTimeMillis();
   System.out.println(
    "Methode: "
     + method.getName()
     + ", Aufruf dauerte "
     + (stop - start)
     + " ms.");
   return result;
  }
  catch (IllegalAccessException e) {
   throw e;
  }
  catch (InvocationTargetException e) {
   throw e.getTargetException();
  }
 }

 ...

}
```

- Möglichst transparent, d.h. der Client soll möglichst nicht merken, dass er es mit einer Remote-Instanz eines Objekts zu tun hat.
- Firewall-tauglich, d.h. als Protokoll muss HTTP verwendet werden (natürlich beherrschen Firewalls neben HTTP auch andere Protokolle, allerdings ist HTTP das einzige Protokoll, das auf nahezu allen Firewalls freigeschaltet ist).

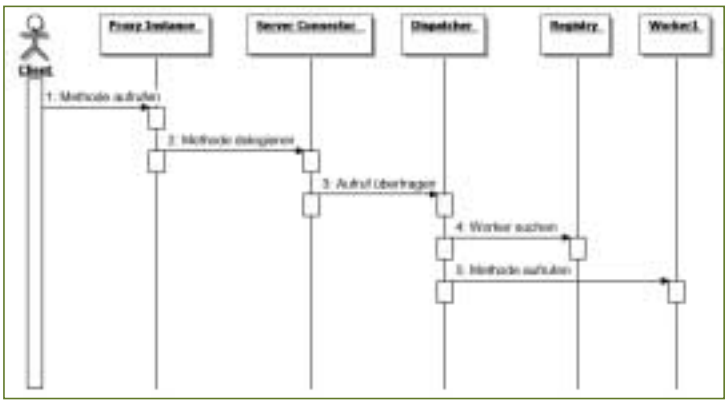
Es müssen also folgende Probleme gelöst werden:

- Verwalten der entfernten Objekt-Instanzen mit einem Object Broker.
- Übertragen der Methodenaufrufe durch Serialisierung.
- Umgehung der Generierung von Stubs und Skeletons mit Hilfe von Dynamic Proxying.

Der Ablauf eines Methodenaufrufs ist in Abbildung 2 dargestellt und kann folgendermaßen beschrieben werden:

- Der Client ruft eine Methode auf einem Objekt auf.
- Der Invocation Handler fängt den Aufruf ab und delegiert den Aufruf an einen Server Connector.
- Der Server Connector überträgt den Namen der Methode und die Werte der Parameter an einen Dispatcher.
- Der Dispatcher (der in einem getrennten Prozess, evtl. sogar auf einem entfernten System läuft) nimmt den Aufruf entgegen

Abb. 2: Entfernter Methodenaufruf mit Dynamic Proxy



- und sucht mit Hilfe einer Registry einen passenden Worker.
- Wird ein Worker gefunden, so wird die gewünschte Methode aufgerufen und die übermittelten Parameter übergeben.
- Das Ergebnis des Methodenaufrufs wird auf dem Rückwege übertragen.

Der Object Broker hat die Aufgabe, die von Clients gesendeten Methodenaufrufe an die entfernten Objekt-Instanzen zu verteilen (dispatching). Darüber hinaus muss er die entfernten Objekt-Instanzen verwalten und ihren Lebenszyklus steuern.

Um diese beiden Aufgaben wahrnehmen zu können, teilt sich der Object Broker in zwei Teile auf: Das *HTTPConnectionServlet* nimmt die per HTTP gesendeten Methodenaufrufe entgegen und leitet sie an den *RemoteInvocationProcessor* weiter (Listing 10).

Die Verwaltung der Objekt-Instanzen wird von der *ObjectRegistry* wahrgenommen, die den gesamten Lebenszyklus der Instanzen verwaltet.

Zur Vereinfachung wird in unserem Beispiel nur ein Objekt pro Klasse erzeugt. Über eine Hashmap wird jedem Proxy-Interface das entsprechende Objekt zugeordnet, sodass der Client nur den Namen des deklarierenden Interfaces mitliefern muss, um dem *RemoteInvocationProcessor* mitzuteilen, welche Objekt-Instanz den Aufruf schlussendlich durchführen soll. Der Object Broker lässt sich natürlich beliebig verfeinern: So wäre es beispielsweise denkbar, pro Interface einen Pool von implementierenden Klassen anzulegen.

Der auf dem Client laufende Invocation Handler muss Informationen über den auszuführenden Methodenaufruf zum Object Broker übertragen. Hier sind viele verschiedene Techniken denkbar, z.B. SOAP oder serialisierte Objekte. In der vorliegenden Implementierung haben wir uns für serialisierte Objekte entschieden, da diese Technik keine zusätzlichen Bibliotheken benötigt und eine relativ kompakte Form der Kommunikation zulässt.

Wie schon in den vorausgegangenen Beispielen könnten wir für das Abfangen der Methodenaufrufe einen Dynamic Proxy einsetzen, der das aufzurufende Objekt umhüllt und die Methodenaufrufe an die entfernte Instanz weiterleitet, die den Aufruf dann tatsächlich durchführt. Bei diesem Vorgehen hätten wir jedoch nicht viel gewonnen, da wir zu jedem entfernte Objekt eine gleiche lokale Instanz benötigen würden.

Einige kleine Experimente ergaben, dass ein Invocation Handler erstens einen Methodenaufruf „wegwerfen" kann und zweitens nicht zwangsläufig ein konkretes Objekt umhüllen muss. Was hier in einem Nebensatz daherkommt, muss man sich wirklich auf der Zunge zergehen lassen: Ein Invocation Handler kann ein beliebiges Objekt umhüllen und dabei eine nahezu beliebige Liste von Proxy-Interfaces „implementieren", ohne dass irgendeines dieser Interfaces von der Proxy Class tatsächlich implementiert wird.

Listing 10

```
protected void doGet(
 HttpServletRequest request,
 HttpServletResponse response)
 throws ServletException, IOException {

 ObjectInputStream ois =
  new ObjectInputStream(request.getInputStream());

 try {
  // Den eingepackten Methodendeskriptor aus dem
  // Stream holen
  ProxiedMethod method =
    (ProxiedMethod) ois.readObject();

  // Den Aufruf durchführen
  RemoteInvocationProcessor
   .getInstance()
   .processInvocation(
   method);

  // Das Ergebnis zurückschicken
  ObjectOutputStream oos =
   new ObjectOutputStream(
    response.getOutputStream());
  oos.writeObject(method);
  oos.flush();
  oos.close();
 }
 catch (Exception e) {
 }
     }
```

Und das ist verblüffend einfach: In der hier vorgestellten Architektur umhüllt der Invocation Handler *RemotingProxy* einfach eine Instanz der Klasse *Object* und bietet ein *Factory*-Methode [2] an, mit der Clients sich Proxy-Instanzen mit beliebigen Interfaces erzeugen lassen können. Der *RemotingProxy* nimmt alle Methodenaufrufe auf diesen Pseudo-Stubs an, verpackt die Methodensignatur in ein serialisierbares Objekt und sendet dieses an den Object-Server. Dieser führt wie oben geschildert die eigentliche Verarbeitung durch und liefert das Ergebnis wiederum in einem serialisierten Objekt zurück. Der *RemotingProxy* packt dieses Objekt aus und liefert seinem Client das gewünschte Ergebnis als Rückgabewert.

Best practices

Aus den obigen Ausführungen lassen sich zwei Ratschläge ableiten:

- Wir unterstützen den Vorschlag von Joshua Bloch [6] und empfehlen Ihnen, beim Design Ihrer Anwendung Interfaces einzusetzen, wo immer dies möglich ist. So stellen Sie sicher, dass Sie alle Objekte Ihrer Anwendung mit einem Proxy kapseln können. Basiert Ihr Design nicht auf Interfaces, so müssen Sie im Bedarfsfall für jede zu kapselnde Klasse ein entsprechendes Interface deklarieren.

- Setzen Sie Factories [2] ein, um die Erzeugung von umhüllten Objekten zentral steuern zu können. Beispielsweise könnten Sie Ihre Factories so programmieren, dass Sie über eine Konfigurationsdatei steuern können, welche Dienste durch den umhüllenden Proxy angeboten werden sollen.

Fazit

Es soll an dieser Stelle nicht verschwiegen werden, dass alle hier behandelten Probleme auch mit anderen Mitteln gelöst werden können: So könnte man das Logging von Methodenaufrufen und das Messen von Methodenlaufzeiten ohne Weiteres per Hand programmieren (mit entsprechendem Aufwand) oder AspectJ [7] einsetzen. Das Remoting von Objekten könnte ebenso gut mit RMI oder SOAP durchgeführt werden.

Warum also sollte man Dynamic Proxies einsetzen, wenn doch alle Probleme auch anders gelöst werden können?

Im Gegensatz zu Dynamic Proxies haben alle eben genannten Verfahren einen Nachteil: Sie erfordern einen zusätzlichen Schritt bei der Erzeugung des Programms. Bei AspectJ hält sich das noch in Grenzen, hier muss nur das Build-Script angepasst werden. Bei RMI oder SOAP jedoch müssen jeweils die entsprechenden Stubs und Definitionsdateien erzeugt werden – und zwar bei jeder Änderung an der Schnittstelle der Subjekte.

Hier kann der Dynamic Proxy punkten, da keine zusätzlichen Stubs erzeugt werden müssen. Auch der Build-Prozess muss an keiner Stelle modifiziert werden. Das Verhalten des Systems kann sogar zur Laufzeit modifiziert werden, dies ist bei AspectJ, RMI und SOAP aufgrund der statischen Natur der Stubs nicht möglich.

Die oben skizzierte Remoting-Architektur wird in einem von uns entwickelten J2EE-Framework eingesetzt und vereinfacht hier die Kommunikation zwischen den einzelnen Komponenten des Frameworks. Eine weitere denkbare Einsatzmöglichkeit für Dynamic Proxies ist die Implementierung eines Persistenz-Frameworks, das herkömmliche Java-Klassen dynamisch um die Fähigkeit erweitert, sich in einer Datenbank zu persistieren. ■

Stephan Anft (stephan.anft@LHsystems.com) und Peter Friese (peter.friese@LHsystems.com) sind bei Lufthansa Systems in Hamburg für die Entwicklung eines J2EE-konformen Frameworks (JEF) verantwortlich.

Links & Literatur

[1] Arnold, Ken; Gosling, James; Holmes, David: „Die Programmiersprache Java", Addison-Wesley, 2001

[2] Blosser, Jeremy: „Explore the Dynamic Proxy API", developer.java.sun.com/developer/technicalArticles/DataTypes/proxy/

[3] Bloch, Joshua: „Effective Java", Addison-Wesley, 2001

[4] Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John: „Design Patterns: Elements of Reusable Object-Oriented Software" Addison-Wesley, 1995

[5] Sun: Dynamic Proxy Classes. java.sun.com/j2se/1.3/docs/guide/reflection/proxy.html

[6] Sun: API Documentation for the Java 2 SDK. java.sun.com/j2se/1.3docs/api/. 2001/

[7] AspectJ Homepage: aspectj.eclipse.org/aspectj/