entwickler.press

Dynamic Proxies

Effizient programmieren

Dr. Heinz Kabutz und Sven Ruppert

schnell + kompakt

Dr. Heinz Kabutz und Sven Ruppert

Dynamic Proxies

Effizient programmieren

schnell+kompakt

entwickler.press

Dr. Heinz Kabutz und Sven Ruppert
Dynamic Proxies. Effizient programmieren
schnell+kompakt
ISBN 978-3-86802-340-4

Ein Imprint der Software & Support Media GmbH

http://www.entwickler-press.de
http://www.software-support.biz

Ihr Kontakt zum Verlag und Lektorat: lektorat@entwickler-press.de

Bibliografische Information Der Deutschen Bibliothek
Die Deutsche Bibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.ddb.de abrufbar.

Lektorat: Corinna Neu
Korrektorat: Frauke Pesch, Corinna Neu
Copy-Editor: Nicole Bechtel, Jennifer Diener
Satz: Dominique Kalbassi
Umschlaggestaltung: Maria Rudi
Belichtung, Druck und Bindung: Media-Print Informationstechnologie GmbH, Paderborn

Inhaltsverzeichnis

Vorwort

Dynamic Proxies wurden mit der Version 1.3, das war im Mai 2000, dem JDK hinzugefügt. Einige der Java-Entwickler, die wir heute antreffen, waren zu diesem Zeitpunkt noch in der Schule, andere sogar noch im Sandkasten. Das, worüber wir hier schreiben, ist demnach nichts Neues.

In diesem Buch setzen wir einiges an Wissen voraus. Wir gehen davon aus, dass das Wissen über Entwurfsmuster – und im Speziellen über die strukturellen Muster Proxy, Object Adapter, Composite und Decorator – präsent ist. Sollten diese Entwurfsmuster nicht bekannt sein, ist es besser, zu diesem Thema vorher ein wenig zu lesen. Andernfalls kann es sein, dass dieses Buch an einigen Stellen schwer verständlich sein wird. Entwurfsmuster begleiten uns schon recht lange. Das erste Buch darüber wurde 1995 auf den Markt gebracht und sollte für jeden professionellen Entwickler ein Standardwerk sein.

Dynamic Proxies sind manchmal bestens geeignet, um komplizierte Probleme zu lösen. Der absolute Extremfall, der uns bisher begegnet ist, war ein Kundensystem, das aus 600 000 Zeichen einmalig generiertem Quelltext bestand. Änderungen mussten nachträglich durch Entwickler manuell eingefügt werden. Es ist leicht, sich vorzustellen, dass dieses System nicht mehr wartbar gewesen ist. Allerdings sahen alle Klassen sehr ähnlich aus. Wir waren in der Lage, diese 600 000 Zeichen Quelltext mit einem einzigen Dynamic Proxy zu ersetzen. Der Gewinn war nicht nur die wesentlich kleinere Menge Quelltext, die noch gewartet werden musste.

Allerdings sind sie nicht immer ein dem Problem adäquates Werkzeug. Bei der Verwendung von Dynamic Proxies haben Me-

thodenaufrufe einen größeren Overhead als normale Methoden. In diesem Buch werden wir einen Weg vorstellen, wie man In-Memory-Codeerzeugung realisieren kann. Es ist keine grundsätzlich schlechte Idee, Quelltext zu genieren, aber es ist eine schlechte Idee, diesen Quelltext einen Entwickler sehen und modifizieren zu lassen.

Wir werden zeigen, wie man durch das In-Memory-Kompilieren von dynamisch generiertem Quelltext die Vorteile eines Dynamic Proxys mit den Vorteilen von statisch kompilierten Proxies kombinieren kann, um performanten, wartbaren Code zu erzeugen.

Wir wünschen Ihnen viel Vergnügen mit diesem Buch und hoffen, Sie finden es nützlich. Es ist keine leichte Lektüre. Setzen Sie also am besten frischen Kaffee auf, schnappen Sie sich den Code von GitHub und versuchen Sie, die Codebeispiele durchzuarbeiten: *https://github.com/svenruppert/shortcut-proxies.*

Dr. Heinz Kabutz und Sven Ruppert

Einführung

1.1 Vergleich von Proxy und Decorator

Was ist eigentlich der Unterschied zwischen einem Proxy und einem Decorator? Diese Frage sollte man sich stellen, sobald man sich mit delegierenden Patterns auseinandersetzt. Nehmen wir das Buch der GoF als Grundlage, so ist ein Proxy direkt assoziiert mit dem Element, auf das der Proxy weiterleitet. Die Darstellung in Java zeigt Listing 1.1.

```
interface Subject {
    public void request();
}
class RealSubject implements Subject {
    public void request() { /* do something */ }
}
class Proxy implements Subject {
    private Subject realSubject;
    Proxy(Subject realSubject) {
        this.realSubject = realSubject;
    }
    public void request() {
        /* do something, then */
        realSubject.request();
    }
}
```

Listing 1.1: Proxy in Java

Gut zu erkennen ist die Assoziation an dem Attribut *realSubject* und der Delegation des Methodenaufrufs in der Methode *request()* der Klasse *Proxy*. Der Decorator hingegen reicht zum einen per Delegation die Methodenaufrufe weiter, und zum an-

deren erweitert er auch die Schnittstelle um individuelle Eigenschaften. Der Decorator hat auf jeden Fall dieselbe Schnittstelle wie das zu dekorierende Element (Listing 1.2).

```
interface Component {
    public void operation();
}
class ConcreteComponent implements Component {
    public void operation() {/* do something */}
}
class Decorator implements Subject {
    private Component component;
    Decorator(Component component) {
        this.component = component;
    }

    public void operation() {
    /* do something, then */
        component.operation();
    }
}
class ConcreteDecorator extends Decorator {
    ConcreteDecorator(Component component) {
        super(component);
    }
    public void anotherOperation() {
        /* decorate the other operation in
            some way, then call the
        other operation() */
        operation();
    }
}
```

Listing 1.2: Decorator in Java

Wenn man nun die beiden Beispiele ein wenig genauer betrachtet, fällt auf, dass es kaum Unterschiede zwischen der Implementierung eines Proxys und einem Decorator gibt. Um das noch ein