PEARSON Education

ADDISON-WESLEY

Joshua Kerievsky

Refactoring to Patterns

ADDISON-WESLEY

An imprint of Pearson Education

München • Boston • San Francisco • Harlow, England
Don Mills, Ontario • Sydney • Mexico City
Madrid • Amsterdam

3 Patterns

In diesem Kapitel erfahren Sie, was ein Pattern ist, was es bedeutet, ein Patternoholic zu sein, und wie wichtig es ist zu verstehen, dass Patterns auf viele verschiedene Arten implementiert werden können. Wir betrachten Refactorings zu, in Richtung auf und weg von Patterns und beschäftigen uns mit der Frage, ob Patterns den Code verkomplizieren. Außerdem behandeln wir die Themen Pattern-Wissen und Design mit Patterns.

3.1 Was ist ein Pattern?

Der Architekt, Professor und Gesellschaftstheoretiker Christopher Alexander inspirierte die Entwicklung von Software Patterns mit seinen Büchern *A Timeless Way of Building* [Alexander, TWB] und *A Pattern Language* [Alexander, PL]. Ende der 1980er-Jahre beschäftigte sich eine Gruppe erfahrener Softwareentwickler mit Alexanders Studien zu den Themen Pattern und komplexe Patternnetzwerke und begründete die so genannten Pattern Languages. Es folgten zahlreiche bedeutende Artikel und Bücher über Patterns und Pattern Languages in den Bereichen Objektorientierung, Analyse und Domain-Design, Prozess- und Organisationsdesign sowie Benutzerschnittstellenentwicklung.

Ein häufiger Diskussionspunkt unter den verschiedenen Autoren besteht darin, wie der Begriff Pattern definiert werden sollte, denn je nachdem, wie nah die Beteiligten der ursprünglichen Theorie Alexanders stehen, fällt die Definition anders aus. Ich persönlich tendiere zur Begriffsbestimmung Alexanders, die ich im Folgenden zitiere:

> *Jedes Pattern ist eine dreiteilige Regel, die eine Beziehung zwischen einem bestimmten Kontext, einem Problem und seiner Lösung ausdrückt.*
>
> *Als Bestandteil der Welt beschreibt jedes Pattern ein Verhältnis zwischen einem bestimmten Kontext, einem bestimmten Kräftesystem, das in diesem Kontext wiederholt auftritt, und einer bestimmten räumlichen Konstellation, die es diesen Kräften gestattet, ein Gleichgewicht herzustellen.*

Als Element einer Sprache ist ein Pattern eine Anweisung, die angibt, wie diese räumliche Konstellation wiederholt verwendet werden kann, um das entsprechende Kräftesystem immer wieder ins Gleichgewicht zu bringen, falls der Kontext dies erfordert.

Kurz gesagt ist ein Pattern gleichzeitig ein Bestandteil der Welt sowie die Regel, die angibt, wie dieser Bestandteil erzeugt wird und wann es erzeugt werden muss. Es ist sowohl ein Prozess als auch ein Ding; sowohl eine Beschreibung eines existierenden Dings und eine Beschreibung des Prozesses, der dieses Ding erzeugt. [Alexander, TWB].

In unserer Branche ist die Sichtweise von Patterns weitgehend durch Kataloge mit einzelnen Patterns wie *Entwurfsmuster* [DP] oder Martin Fowlers *Patterns of Enterprise Application Architechtures* [Fowler, PEAA] geprägt. Solche Kataloge enthalten eigentlich keine eigenständigen Patterns, da ihre Autoren vorrangig der Frage nachgehen, welches Pattern als Alternative eingesetzt werden kann, falls das gewünschte nicht geeignet ist. In den letzten Jahren sind außerdem einige Veröffentlichungen zu Sprachen erschienen, die der von Alexander sehr ähneln, zum Beispiel *Extreme Programming Explained* [Beck, XP], *Domain-Driven Design* [Evens] oder *Checks: A Pattern Language of Information Integrity* [Cunningham].

3.2 Patternoholics

Im Klappentext von *Contributing to Eclipse* [Gamma und Beck] heißt es in einer kurzen Biografie über Erich Gamma: »Seine Liebe zur Ordnung und Schönheit von Softwaredesigns ließ Erich Gamma als Mitverfasser in das klassische Buch *Entwurfsmuster* einfließen.« Wer jemals ein ausgezeichnetes Design mit Patterns erstellt oder verwendet hat, weiß, wie sehr ihr Einsatz zufriedenstellen kann.

Dasselbe gilt jedoch auch im umgekehrten Sinn: Wenn Sie jemals schlecht entworfenen Code erstellt oder verwendet haben, der auf Patterns basierte, obgleich er die Flexibilität und Komplexität von Patterns gar nicht erforderte, wissen Sie, wie tückisch Patterns sein können.

Die übermäßige Verwendung von Patterns wird von Patternoholics praktiziert. Wir werden zu Patternoholics, wenn uns die Liebe zu Patterns blind für die eigentlichen Erfordernisse des Codes macht. Patternoholics unternehmen große Anstrengungen, Patterns in den Code einzubauen, nur um Erfahrung mit ihrer Implementierung zu sammeln oder möglicherweise auch nur, um Anerkennung für wirklich komplexen Code zu ernten.

Die folgende Version von »Hello World« des Programmierers Jason Tiscioni auf SlashDot (*http://developers.slashdot.org/comments.pl?sid=33602&cid=3636102*) ist eine treffende Parodie der Programmierweise von Patternoholics:

```
interface MessageStrategy {
   public void sendMessage();
}
abstract class AbstractStrategyFactory {
   public abstract MessageStrategy createStrategy(MessageBody mb);
}

class MessageBody {
   Object payload;
   public Object getPayload() {
      return payload;
   }
   public void configure(Object obj) {
      payload = obj;
   }
   public void send(MessageStrategy ms) {
      ms.sendMessage();
   }
}
class DefaultFactory extends AbstractStrategyFactory {
   private DefaultFactory() {
   }
   static DefaultFactory instance;
   public static AbstractStrategyFactory getInstance() {
      if (instance == null)
         instance = new DefaultFactory();
      return instance;
   }
   public MessageStrategy createStrategy(final MessageBody mb) {
      return new MessageStrategy() {
         MessageBody body = mb;
         public void sendMessage() {
            Object obj = body.getPayload();
            System.out.println(obj);
         }
      };
   }
}
public class HelloWorld {
   public static void main(String[] args) {
      MessageBody mb = new MessageBody();
      mb.configure("Hello World!");
      AbstractStrategyFactory asf = DefaultFactory.getInstance();
      MessageStrategy strategy = asf.createStrategy(mb);
      mb.send(strategy);
   }
}
```

Haben Sie schon einmal Code wie Jasons »Hello World«-Programm gesehen? Ich leider ja, und viel zu häufig.

Die Patternsucht ist nicht nur auf Programmieranfänger beschränkt, auch fortgeschrittene Entwickler fallen ihr zum Opfer, insbesondere nach der Lektüre vermeintlich schlauer Bücher oder Artikel über Patterns. In einem System, an dessen Entwicklung ich beteiligt war, fand ich zum Beispiel eine Implementierung des Patterns Closure. Der verantwortliche Programmierer hatte sich seine Verwendung erst kürzlich auf der Wiki Web-Website (*http://c2.com/cgi/wiki?UseClosureNot-Enumerations*) angeeignet.

Ich sah mir seine Implementierung von Closure genauer an und konnte beim besten Willen keinen Grund für die Verwendung des Patterns erkennen. Es war schlichtweg nicht erforderlich. Also führte ich ein Refactoring durch, um es aus dem Code zu entfernen, und fragte anschließend die Programmierer in meinem Team, ob sie den neuen Code einfacher fänden. Die Antwort lautete »ja«. Letztendlich gab sogar der ursprüngliche Verfasser zu, dass der Code nach der Umstrukturierung benutzerfreundlicher war.

Vermutlich lässt es sich nicht verhindern, dass Programmierer, die sich mit Patterns beschäftigen, irgendwann einmal in die Patternoholics-Falle tappen. Schließlich lernen wir ja auch aus unseren Fehlern. Ich persönlich war bereits mehr als einmal »Pattern-happy«.

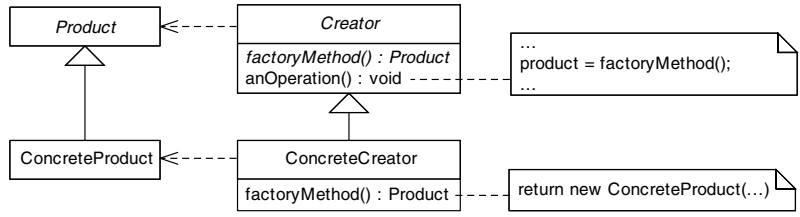
Die wirkliche Freude an Patterns erfahren Sie, wenn Sie sie intelligent einsetzen. Dazu wenden Sie Refactorings an, mit denen Duplikate entfernt, der Code vereinfacht und sein Zweck deutlicher herausgearbeitet wird. Wenn Sie Patterns mithilfe von Refactoring zu einem System weiterentwickeln, verringert sich die Gefahr des Over-Engineering. Je besser Sie beim Refactoring werden, desto mehr Freude werden Sie an Patterns haben.

3.3 Viele Wege führen zum Pattern

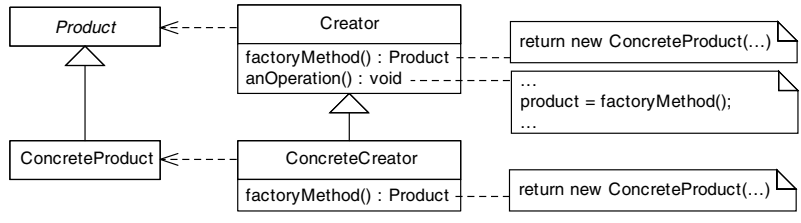
> *Jedes Pattern beschreibt ein in unserem Umfeld immer wieder auftretendes Problem und dann den Kern der Lösung dieses Problems auf eine solche Weise, dass diese Lösung ohne Wiederholungen millionenfach angewendet werden kann. [Alexander, PL]*

Im Klassiker *Entwurfsmuster* [DP] ist für jedes Pattern ein Strukturdiagramm abgebildet, wie z.B. das für Fabrik-Methoden.

In diesem Diagramm werden die Klassen `Creator` und `Product` als abstrakt und die Klassen `ConcreteCreator` und `ConcreteProduct` als konkret definiert. Aber dies ist bei Weitem nicht die einzige Möglichkeit, das Pattern Fabrik-Methode zu implementieren.

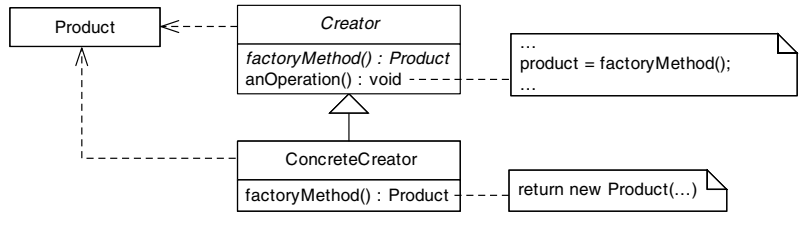


Tatsächlich haben sich die Autoren von *Entwurfsmuster* große Mühe gegeben, in den Implementierungsanmerkungen für jedes Pattern mehrere verschiedene Implementierungen aufzuzeigen, so auch für Fabrik-Methode, zum Beispiel wie im folgenden Strukturdiagramm:



In diesem Fall ist die Klasse `Product` abstrakt, während alle anderen Klassen konkret sind. Außerdem implementiert die Klasse `Creator` ihre eigene Methode `factoryMethod()`, die von `ConcreteCreator` überschrieben wird.

Dies sind jedoch noch lange nicht alle Möglichkeiten der Implementierung von Fabrik-Methode, eine weitere wird nachfolgend gezeigt:



In diesem Beispiel ist die Klasse `Product` konkret und hat keine Unterklasse. `Creator` ist abstrakt und definiert eine abstrakte Version der Factory-Methode, die `ConcreteCreator` implementiert, um eine Instanz von `Product` zurückzugeben.

Sehen Sie, worauf ich hinauswill? Viele Wege führen zum selben Pattern!

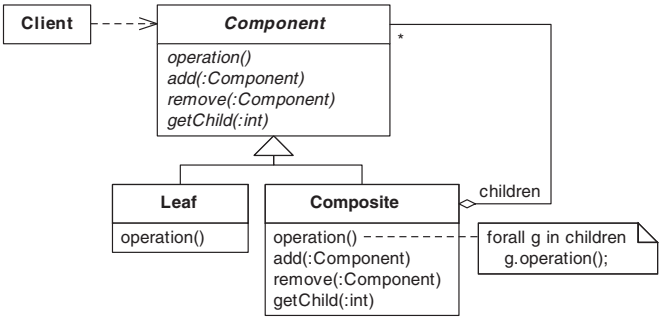
Leider nehmen viele Programmierer an, dass es sich bei den Strukturdiagrammen, die als Beispiele im Buch *Entwurfsmuster* angegeben sind, jeweils um die *einzige* Möglich-

keit handelt, das entsprechende Pattern zu implementieren. Dabei würde sie schon ein kurzer Blick auf die Implementierungsanmerkungen eines Besseren belehren. Doch leider beginnen zahlreiche Entwickler sofort nach dem flüchtigen Betrachten eines Diagramms mit dem Programmieren. Das Ergebnis ist Code, der exakt dem dargestellten Strukturdiagramm entspricht, das Pattern jedoch nicht so implementiert, dass es den eigentlichen Erfordernissen gerecht wird.

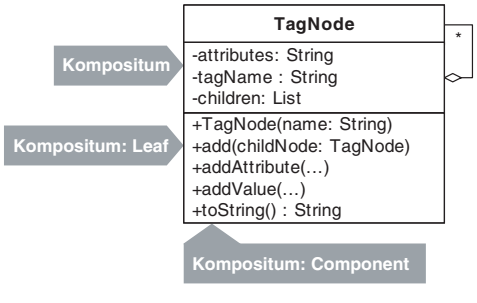
Einige Jahre nach der Veröffentlichung des Buchs *Entwurfsmuster* schrieb einer der Autoren, John Vlissides:

> *Es kann gar nicht genug betont werden, dass es sich bei dem Strukturdiagramm eines Patterns nur um ein Beispiel handelt, nicht um eine Spezifikation. Es stellt die häufigste Implementierung dar. Das heißt, das Strukturdiagramm gleicht Ihren eigenen Implementierungen wahrscheinlich in vielen Punkten, doch Unterschiede sind unvermeidbar und sogar erwünscht. Zumindest sollten Sie die Elemente umbenennen, um sie an Ihre Domäne anzupassen. Je mehr Anpassungen Sie vornehmen, desto weniger wird Ihre Implementierung dem Strukturdiagramm ähneln. [Vlissides]*

In diesem Buch finden Sie Pattern-Implementierungen, die sich von ihren Strukturdiagrammen in *Entwurfsmuster* erheblich unterscheiden. Im Folgenden wird zum Beispiel das Diagramm für das Pattern Kompositum gezeigt:



Im Folgenden sehen Sie zum Vergleich eine mögliche Implementierung von Kompositum:



Wie Sie sehen, weist die Implementierung von Kompositum nur wenig Ähnlichkeit mit dem Strukturdiagramm auf. In dieser minimalistischen Kompositum-Implementierung wird nur Code verwendet, der wirklich erforderlich ist.

Die minimalistische Programmierung mit Patterns ist ein Grundsatz des evolutionären Designs. In vielen Fällen kann es erforderlich sein, eine Implementierung, die nicht auf Patterns basiert, zu einer einfachen Pattern-Implementierung weiterzuentwickeln. In diesem Buch zeige ich zahlreiche Beispiele für diese Vorgehensweise.

3.4 Refactoring in verschiedene Richtungen

Gute Entwickler betreiben Refactoring in mehrere Richtungen, um das bestmögliche Design zu erzielen. Bei vielen meiner Umstrukturierungen spielen Patterns gar keine Rolle, sondern es handelt sich um kleine, einfache Änderungen wie z. B. *Methode extrahieren* [F], *Methode umbenennen* [F], *Methode verschieben* [F], *Methode nach oben verschieben* [F] usw. Wenn Patterns am Refactoring beteiligt sind, kann das bedeuten, dass ich eine Umstrukturierung zu, in Richtung auf oder weg von Patterns durchführe.

Welche Richtung das Refactoring nimmt, hängt meist von der Art des jeweiligen Patterns ab. Für *Methode komponieren (s.S. 148)* habe ich zum Beispiel Kent Becks Pattern *Komponierte Methode* [Beck, SBPP] umgestaltet. Wenn ich *Methode komponieren (s.S. 148)* verwende, ändere ich die Codestruktur *in* eine komponierte Methode, nicht *in Richtung* darauf. Das Refactoring in Richtung auf eine komponierte Methode reicht nicht aus, um eine Methode zu verbessern. Die Umstrukturierung zu diesem Pattern muss vollständig ausgeführt werden, um eine tatsächliche Verbesserung zu erzielen.

Dasselbe gilt für das Pattern Template-Methode [DP]. Die meisten Entwickler wenden das Refactoring *Template-Methode bilden (s.S. 232)* an, um duplizierten Code in Unterklassen loszuwerden. Ein Refactoring in Richtung der Template-Methode reicht nicht aus, um Duplikate zu entfernen, denn entweder implementieren Sie Template-Methode oder nicht. Falls nicht, können Sie auch keinen redundanten Code aus Ihren Unterklassen löschen.

Andererseits finden sich in diesem Buch auch zahlreiche Refactorings, mit denen akzeptable Verbesserungen am Design erzielt werden können, selbst wenn Sie keine vollständige Umstrukturierung zu einer bestimmten Pattern-Implementierung vornehmen. Ein gutes Beispiel hierfür ist *Ausschmückungen einem Dekorierer überlassen (s.S. 169)*. Für dieses Refactoring sollten Sie möglichst früh *Bedingte Logik durch Polymorphismus ersetzen* [F] einsetzen und dann überprüfen, ob dadurch bereits eine ausreichende Verbesserung erreicht wird. Falls Sie meinen, mit dem nächsten Schritt ein noch besseres Ergebnis zu erzielen, sieht das Verfahren die Verwendung des Refactorings *Vererbung durch Delegation ersetzen* [F] vor. Sind Sie mit dem Ergebnis zufrieden,

hören Sie an diesem Punkt auf. Falls Sie Ihr Design weiter verbessern können, indem Sie alle Schritte des Verfahrens ausführen, nehmen Sie weitere Strukturänderungen bis hin zu Dekorierer vor.

Ähnlich verhält es sich mit *Bedingten Verteiler durch Befehl ersetzen (s.S. 220)*. Die Refactoring-Schritte, die Sie in Richtung einer Implementierung von Befehl [DP] unternehmen, können Ihr Design verbessern, selbst wenn Sie nicht die gesamte Umstrukturierung zum Pattern Befehl durchführen.

Nach einem Refactoring Ihres Codes zu bzw. in Richtung auf ein Pattern müssen Sie überprüfen, ob das Design tatsächlich davon profitiert hat. Ist das nicht der Fall, sollten Sie die Schritte wieder rückgängig machen oder es mit einer Umstrukturierung in eine andere Richtung versuchen, zum Beispiel vom Pattern weg oder hin zu einem anderen Pattern. Mit *Singleton inline stellen (s.S. 139)* wird zum Beispiel das Pattern *Singleton* [DP] aus dem Design entfernt. Mit *Kompositum durch Erbauer kapseln (s.S. 120)* wird abhängiger Code so abgewandelt, dass er statt mit *Kompositum* mit *Erbauer* [DB] interagiert. *Akkumulation einem Besucher überlassen (s.S. 345)* ersetzt Code von *Iterator* [DP], der unklar geschrieben ist oder viele Duplikate enthält, durch eine Lösung mit *Besucher* [DP].

Die Refactorings in diesem Buch ermöglichen Ihnen die Umstrukturierung zu, in Richtung auf oder weg von Patterns. Denken Sie aber immer daran, dass Ihr Ziel die Verbesserung des Designs ist und nicht die zwanghafte Implementierung von Patterns. Die folgende Tabelle zeigt die verschiedenen Richtungen, die ich bei den Patternsbasierten Refactorings in diesem Buch häufig einschlage.

Pattern	Zum Pattern	In Richtung auf das Pattern	Weg vom Pattern
Adapter	*Adapter extrahieren (s.S. 284)*, *Schnittstellen durch Adapter vereinheitlichen (s.S. 273)*	*Schnittstellen durch Adapter vereinheitlichen (s.S. 273)*	
Befehl	*Bedingte Verteiler durch Befehl ersetzen (s.S. 218)*	*Bedingte Verteiler durch Befehl ersetzen (s.S. 218)*	
Beobachter	*Hartcodierte Benachrichtigungen durch Beobachter ersetzen (s.S. 262)*	*Hartcodierte Benachrichtigungen durch Beobachter ersetzen (s.S. 262)*	
Besucher	*Akkumulation einem Besucher überlassen (s.S. 345)*	*Akkumulation einem Besucher überlassen (s.S. 345)*	
Dekorierer	*Ausschmückungen einem Dekorierer überlassen (s.S. 169)*	*Ausschmückungen einem Dekorierer überlassen (s.S. 169)*	

Pattern	Zum Pattern	In Richtung auf das Pattern	Weg vom Pattern
Erbauer	*Kompositum durch Erbauer kapseln (s.S. 120)*		
Erzeugungsmethode	*Konstruktoren durch Erzeugungsmethoden ersetzen (s.S. 80)*		
Fabrik	*Die Erzeugung der Fabrik überlassen (s.S. 91)*		
	Klassen durch Fabrik kapseln (s.S. 104)		
Fabrik-Methode	*Polymorphe Erzeugung mit Fabrik-Methoden einführen (s.S. 112)*		
Interpreter	*Implizite Sprache durch Interpreter ersetzen (s.S. 294)*		
Iterator			*Akkumulation einem Besucher überlassen (s.S. 345)*
Komponierte Methode	*Methode komponieren (s.S. 148)*		
Kompositum	*1/n-Unterscheidungen durch Kompositum ersetzen (s.S. 250)*		*Kompositum durch Erbauer kapseln (s.S. 120)*
	Kompositum extrahieren (s.S. 240)		
	Implizite Bäume durch Kompositum ersetzen (s.S. 205)		
NULL-Objekt	*Das NULL-Objekt einführen (s.S. 327)*		
Sammelparameter	*Akkumulation einem Sammelparameter überlassen (s.S. 338)*		
Singleton	*Instanziierung durch Singleton begrenzen (s.S. 322)*		*Singleton inline stellen (s.S. 139)*
Strategie	*Bedingte Logik durch Strategie ersetzen (s.S. 153)*	*Bedingte Logik durch Strategie ersetzen (s.S. 153)*	
Template-Methode	*Template-Methode bilden (s.S. 232)*		

Pattern	Zum Pattern	In Richtung auf das Pattern	Weg vom Pattern
Zustand	*Zustandsverändernde Bedingungen durch den Zustand ersetzen (s.S. 193)*	*Zustandsverändernde Bedingungen durch den Zustand ersetzen (s.S. 193)*	

3.5 Führen Patterns zu kompliziertem Code?

Ich habe einmal in einem Team gearbeitet, in dem einige der Programmierer mit Patterns umgehen konnten und andere nicht. Bobby kannte sich gut mit Patterns aus und arbeitete bereits seit zehn Jahren als Programmierer. John hatte noch nicht viel mit Patterns zu tun gehabt und verfügte nur über vier Jahre Programmiererfahrung.

Als sich John eines Tages ein wichtiges Stück Refactoring-Code anschaute, das Bobby erstellt hatte, beklagte er: »Vorher hat mir der Code besser gefallen! Jetzt ist er viel komplizierter.«

Bobby hatte einen Großteil der mit der Validierung von Bildschirmeingaben verknüpften bedingten Logik umstrukturiert. Dabei war er schließlich beim Pattern Kompositum angekommen. Er konvertierte mehrere Teile der Validierungslogik in einzelne Validierungsobjekte, die alle dasselbe Interface verwendeten. Die Validierungsobjekte der einzelnen Eingabebildschirme fasste er jeweils in einem einzelnen Kompositum-Validierungsobjekt zusammen. Zur Laufzeit wurde eine Abfrage an das Composite-Validierungsobjekt des entsprechenden Bildschirms gesendet, um festzustellen, ob die Validierungsregeln eingehalten wurden.

Ich programmierte damals gerade mit John im Zweierteam, als er seiner Unzufriedenheit mit Bobbys Code Ausdruck verlieh. Da ich herausfinden wollte, was ihn genau störte, stellte ich ihm einige Fragen. Ich erfuhr schnell, dass John überhaupt nicht mit Patterns vertraut war, was natürlich einiges erklärte.

Ich bot John an, ihm das Pattern Kompositum zu erläutern, und er wollte gern mehr darüber lernen. Anschließend schauten wir uns Bobbys Refactoring-Code noch einmal an. Auf meine Frage, ob er mittlerweile anders über den Code dächte, musste John schließlich zugeben, dass dieser doch nicht so kompliziert war, wie er zunächst angenommen hatte. Allerdings war er immer noch nicht der Meinung, dass der neue Code besser als der alte war.

Meiner Ansicht nach stellte Bobbys Refactoring eine erhebliche Verbesserung des ursprünglichen Codes dar. Er hatte lange Abschnitte mit bedingter Logik entfernt und Duplikate in Klassen beseitigt, die mit den Eingabebildschirmen verknüpft waren. Außerdem war es jetzt weitaus einfacher, die Einhaltung der Validierungsregeln zu überprüfen.

Da ich mit dem Pattern Kompositum vertraut war und es selbst gern einsetzte, war meine Meinung über Bobbys Refactoring-Code vorhersehbar. Im Gegensatz zu John fand ich den neu strukturierten Code einfacher und sauberer als den alten.

Im Allgemeinen sollten Pattern-Implementierungen Duplikate beseitigen, die Logik vereinfachen, die Absicht des Codes vermitteln und mehr Flexibilität ermöglichen. Wie diese Geschichte zeigt, ist es für den Erfolg des Codes jedoch entscheidend, dass der Benutzer mit Patterns vertraut ist. Ich bevorzuge Teams, die sich über Patterns weiterbilden und nicht versuchen, sie zu vermeiden, weil sie ihnen zu kompliziert erscheinen. Andererseits gibt es aber auch einige Implementierungen von Patterns, die den Code umständlicher als notwendig machen. Ist das der Fall, muss die Umstrukturierung entweder rückgängig gemacht oder weitere Refactoring-Schritte müssen ausgeführt werden.

3.6 Pattern-Wissen ist Macht

Wenn Ihnen Patterns unbekannt sind, ist es unwahrscheinlich, dass Sie es zu großartigen Designs bringen werden. In Patterns steckt Wissen, und dieses Wissens sollten Sie nutzen.

Der Mozart-Biograf Maynard Solomon hat festgestellt, dass Mozart keine neuen Musikformen erfand, sondern vorhandene Elemente zu neuen, verblüffenden Kompositionen kombinierte [Solomon]. Mit Patterns verhält es sich analog: Vorhandene Formen können zu neuen, besseren Softwaredesigns zusammengesetzt werden.

Bei der Entwicklung eines Systems, an dem ich beteiligt war, spielte zum Beispiel die Kenntnis des Patterns Erbauer eine wichtige Rolle. Das System musste in zwei völlig unterschiedlichen Umgebungen ausgeführt werden können. Hätten wir bei der Weiterentwicklung des Designs nicht relativ früh das Pattern Erbauer eingesetzt, wären wir später sicherlich auf einige Probleme gestoßen.

Im hervorragenden Test-Framework JUnit, das testgesteuerte Entwicklung und Unit-Tests ermöglicht, kommen viele Patterns zum Einsatz. Seine Autoren Kent Beck und Erich Gamma haben jedoch nicht alle möglichen Patterns verwendet, sondern das Framework durch die Wiederverwendung von Pattern-Wissen weiterentwickelt. Da es sich bei JUnit um ein Open-Source-Tool handelt, konnte ich seine Evolution von Version 1.0 bis zur aktuellen nachverfolgen. Ich habe mir alle Versionen angesehen und mit Kent und Erich gesprochen und kann bestätigen, dass sie tatsächlich Refactorings zu und in Richtung auf Patterns durchgeführt haben. Ihr Pattern-Wissen hat ihnen dabei sicherlich gute Dienste geleistet.

Wie ich bereits zu Anfang dieses Buchs betont habe, reicht das Wissen über Patterns allein nicht aus, um großartige Software zu entwickeln. Sie müssen die Patterns auch

intelligent einsetzen, und das vorliegende Buch zeigt Ihnen wie. Dennoch sollten Sie nicht darauf verzichten, Patterns selbst zu studieren, da Sie daraus interessante und oftmals sehr elegante Ideen für Ihr Design entwickeln können.

Meine bevorzugte Methode zum Studium von Patterns sieht so aus, dass ich als Erstes einige gute Pattern-Bücher auswähle und dann in der Lerngruppe jede Woche eines der Patterns näher betrachte. In meinem Artikel »Pools of Insight« [Kerievsky, PI s.o.] beschreibe ich, wie Sie erfolgreich langfristige Pattern-Lerngruppen aufbauen können.

Die Lerngruppe »Design Patterns Study Group of New York City«, die ich 1995 gründete, trifft sich noch heute. Eine Gruppe namens »The Silicon Valley Patterns Group« besteht ebenfalls schon seit einigen Jahren. Einige ihrer Mitglieder haben sogar so viel Spaß an den Gruppensitzungen, dass sie ihren Wohnsitz näher an den Ort verlegt haben, an dem die Treffen stattfinden.

Bei den Mitgliedern dieser Gruppen handelt es sich um Personen, die bessere Softwareentwickler werden möchten; eine gute Lernmethode ist die Teilnahme an solchen wöchentlichen Treffen und Diskussionen.

Falls Sie befürchten, dass es bereits zu viele Bücher mit Patterns gibt, als dass Sie jemals alle kennen lernen könnten, befolgen Sie den Rat von Jerry Weinberg. Als ich ihn einmal fragte, wie er die Vielzahl neuer Veröffentlichungen bewältigte, antwortete er: »Ganz einfach! Ich lese nur die wirklich guten!«

3.7 Direktes Design mit Patterns

Im Frühjahr 1996 machte sich ein bekanntes Musik- und Fernsehunternehmen daran, eine Java-Version seiner Website zu erstellen. Die Manager des Unternehmens wussten, wie die Website aussehen sollte und welche Funktionen enthalten sein mussten, doch sie hatten keine Erfahrung mit Java. Also suchte man nach einem Partner für die Entwicklung.

Die Verantwortlichen des Unternehmens wandten sich an meine Firma Industrial Logic. Bei unserem ersten Treffen erläuterte man uns das Design der Benutzeroberfläche und erklärte, dass man später nicht für jede kleine Änderung am Verhalten der Website das Entwicklerteam erneut rufen wollte. Die Firmenleitung wollte von uns wissen, wie wir die Website programmieren würden, um dieser Anforderung gerecht zu werden.

In den nächsten Wochen entwickelte ich mit meinen Kollegen verschiedene Designs. Wir waren alle Mitglieder der Lerngruppe »Design Pattern Study Group«, die ich sechs Monate zuvor ins Leben gerufen hatte, und unser neu erworbenes Pattern-Wissen war für den Designprozess äußerst hilfreich. Für jede Anforderung, der die Website gerecht werden musste, suchten wir ein geeignetes Pattern.

Bald wurde deutlich, dass das Pattern Befehl [DP] in unserem Design eine wichtige Rolle spielen würde. Wir planten die Website so, dass das gesamte Verhalten über Befehl-Objekte gesteuert wurde. Bei jedem Klick auf den Bildschirm sollten ein oder mehrere Befehl-Objekte ausgeführt werden. Wir entwickelten außerdem ein einfaches Pattern namens Interpreter [DB], mit dem unser Kunde die Website für die Ausführung mit beliebigen Befehl-Objekten konfigurieren konnte, um das Verhalten ohne unsere Hilfe zu ändern.

Vor dem zweiten wichtigen Treffen mit den Verantwortlichen des Musiksenders verbrachten wir mehrere Tage mit der Dokumentation unseres Designs. Die Präsentation verlief erfolgreich und wir vereinbarten eine dritte Sitzung, auf der die technischen Einzelheiten des Website-Designs besprochen werden sollten.

Die Wochen verstrichen, dem dritten Treffen folgten weitere, doch im Sommer hatten wir noch immer keine einzige Zeile Code für die Website geschrieben. Alles, was wir hatten, war eine umfangreiche Sammlung von Designdokumenten. Endlich erfuhren wir Mitte August, dass wir den Zuschlag für die Entwicklung der Website erhalten hatten.

In den darauf folgenden Monaten programmierten wir die Website und hielten uns dabei genau an unser Design. Bei der Arbeit fanden wir einige Bereiche, in denen der Code durch Refactorings zu den Patterns Kompositum [DP], Iterator [DP] und NULL-Objekt [Woolf] verbessert werden konnte. Ein Beispiel für unsere Vorgehensweise beim Refactoring zu NULL-Objekt finden Sie im Abschnitt *Das NULL-Objekt einführen (s.S. 327)*.

Mitte Dezember, knapp einen Monat nach dem geplanten Veröffentlichungstermin, waren wir endlich fertig. Die Website ging online, und wir hatten Grund zum Feiern.

Wenn ich über Refactorings und die Rolle von Patterns nachdenke, kommt mir oft dieser Auftrag wieder in den Sinn. Ich frage mich, ob es möglicherweise besser gewesen wäre, evolutionär Code zu entwickeln, anstatt ein komplexes Design (BDUF, Big Design Up-Front) zu erstellen. Wären wir vielleicht erfolgreicher gewesen, wenn wir ausschließlich Refactoring zu oder in Richtung auf Patterns durchgeführt hätten, anstatt uns bereits frühzeitig auf ein paar wenige Patterns festzulegen?

Die Antwort lautet »nein«. Wir hätten niemals die Ausschreibung für das Projekt gewonnen, wenn wir kein überzeugendes Design mit allen Details präsentiert hätten. Das große Design war in diesem Fall notwendig und die direkte Entwicklung mit den Patterns Befehl und Interpreter ausschlaggebend für unseren Erfolg.

Ein häufiges Problem mit großen Designs besteht darin, dass sie sehr viel Zeit in Anspruch nehmen. Man entwickelt ein Design, das den Ansprüchen gerecht wird, doch dann ändern sich die Anforderungen plötzlich, Vorgaben verlieren an Bedeutung

oder die Prioritäten verschieben sich. In anderen Fällen bleiben die Anforderungen zwar gleich, aber man vergeudet viel Zeit damit, den Code eleganter oder intelligenter zu gestalten als tatsächlich erforderlich.

In unserem Fall hatten wir nicht mit den Problemen von großen Designs zu kämpfen, da es sich um ein relativ kleines Projekt handelte und die Anforderungen unveränderbar waren. Wir konnten daher alle Elemente unseres Designs bei der Programmierung nutzen, und der Code funktionierte genau so, wie unser Kunde es erwartete, ohne übermäßig kompliziert oder zu anspruchsvoll zu sein.

Allerdings lieferten wir die Website mit einem Monat Verspätung ab, da bei der Ausführung in den verschiedenen Webbrowsern zahlreiche Fehler auftraten. Etwa zur Mitte der Projektlaufzeit testeten wir die Mac-Version der Website und stellten fest, dass Netscape für Mac erhebliche Fehler aufwies, die wir bei der Programmierung umschiffen mussten. Außerdem gab es zahlreiche Unterschiede im Verhalten von Internet Explorer und Netscape, die uns zu weiteren zeitaufwändigen Änderungen des Codes zwangen.

Die verspätete Fertigstellung der Website war im Grunde kein ernsthaftes Problem, selbst wenn es uns damals viel Stress bereitete. Hätten wir bereits mit der Programmierung begonnen, bevor wir den Zuschlag zu dem Projekt erhalten hatten, hätten wir die Browserprobleme früher entdeckt. Damals waren wir allerdings nicht gern bereit, mit dem Schreiben von Code zu beginnen, bevor wir nicht auch einen zahlenden Kunden hatten.

Seit dieser Zeit habe ich nur sehr wenige andere Designs gleich mit Patterns begonnen. Stattdessen bin bei fast jedem unserer Projekte dem Ansatz gefolgt, ein bestehendes System weiterzuentwickeln und je nach Bedarf Refactorings zu, in Richtung auf oder weg von Patterns durchzuführen. Das Pattern Befehl stellt allerdings immer noch eine wichtige Ausnahme dar. Seit 1996 habe ich es in zwei oder drei Systemen bereits sehr früh im Design verwendet, da es so einfach zu implementieren ist und das Verhalten des Patterns für den Code eindeutig erforderlich war.

Ich hoffe, diese kleine Geschichte verdeutlicht, dass es Fälle gibt, in denen die Vorschläge in diesem Buch getrost ignoriert werden können. Generell bin ich zwar kein Befürworter von Designs mit Patterns, doch ich weiß, dass dieses Verfahren einen Platz im Programmierer-Werkzeugkasten hat. Ich wende es nur selten und immer mit viel Bedacht an, und empfehle Ihnen, ebenso vorzugehen.