

Kent Beck

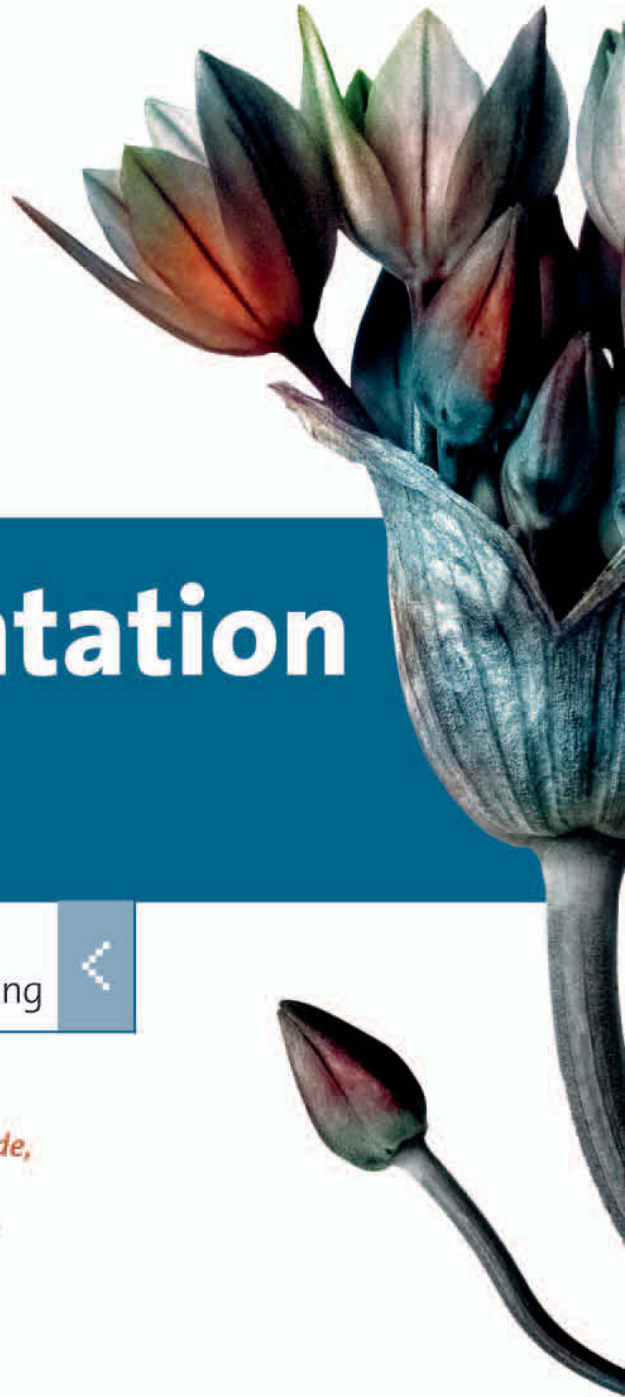
# Implementation Patterns

Der Weg zu einfacherer und  
kostengünstigerer Programmierung



*»Kent ist ein Meister im Erstellen von Code,  
der sich selbst gut dokumentiert, leicht  
verständlich ist und sich mit Vergnügen  
lesen lässt.«*

Erich Gamma, IBM Distinguished Engineer





### 3 Eine Theorie der Programmierung

Keine Liste von Patterns – egal wie erschöpfend sie ist – kann jede Situation abdecken, die bei der Programmierung auftaucht. Irgendwann (wenn nicht gar häufig) geraten Sie in eine Situation, die in keine Schablone passt. Dieses Bedürfnis nach allgemeinen Ansätzen für einzigartige Probleme ist ein Grund, um die Theorie der Programmierung zu studieren. Ein anderer ist das Gefühl der Überlegenheit, das sich einstellt, wenn man sowohl weiß, was zu tun ist, als auch warum. Gespräche über Programmierung sind ebenfalls interessanter, wenn es sowohl um die Theorie als auch um die Praxis geht.

Jedes Pattern bringt ein wenig Theorie mit sich. Allerdings wirken in der Programmierung größere und tiefer greifende Kräfte als sie in einzelnen Patterns abgedeckt werden. Um diese Querschnittsthemen geht es in diesem Abschnitt, und zwar gegliedert in zwei Typen: Werte und Prinzipien. Die Werte sind die universellen, überspannenden Themen der Programmierung. Gute Arbeit zeichnet sich bei mir dadurch aus, dass ich der Kommunikation mit anderen Personen einen hohen Stellenwert einräume, wobei ich übermäßige Komplexität aus meinem Code entferne und mir dennoch alle Optionen offenhalte. Diese Werte – Kommunikation, Einfachheit und Flexibilität – färben jede Entscheidung, die ich während der Programmierung treffe.

Die hier beschriebenen Prinzipien sind nicht so weitreichend oder durchdringend wie diese Werte, doch wird

jedes Prinzip durch viele der Patterns ausgedrückt. Die Prinzipien schlagen eine Brücke zwischen den Werten, die universell aber oftmals nicht ohne Weiteres direkt anzuwenden sind, und den Patterns, die klar anzuwenden aber spezifisch sind. Mir scheint es nützlich zu sein, die Prinzipien explizit für diese Situationen zu schaffen, in denen kein Pattern anwendbar ist, oder wenn zwei sich gegenseitig ausschließende Patterns gleichermaßen zutreffen. Wenn ich angesichts dieser Doppeldeutigkeit die Prinzipien verstanden habe, kann ich mir »etwas ausdenken«, das mit meiner sonstigen Praxis vereinbar ist und sich wahrscheinlich als gut erweist.

Diese drei Elemente – Werte, Prinzipien und Patterns – verkörpern einen ausgewogenen Entwicklungsstil. Die Patterns beschreiben, was zu tun ist. Die Werte bieten die Motivation. Die Prinzipien helfen, Motivation in Aktion zu übersetzen.

Die hier angegebenen Werte, Prinzipien und Patterns stammen aus meinen praktischen Erfahrungen, Überlegungen und Gesprächen mit anderen Programmierern. Wir zehren alle von den Erfahrungen vorheriger Programmierergenerationen. Das Ergebnis ist *ein* Entwicklungsstil, nicht *der* Entwicklungsstil. Unterschiedliche Werte und unterschiedliche Prinzipien führen zu unterschiedlichen Stilen. Stellt man einen Programmierstil in Form von Werten, Prinzipien und Praktiken dar, ist es leichter, einen produktiven Disput über die Verfahrensweise bei der Programmierung zu führen. Wenn Sie etwas auf die eine Weise tun möchten und ich auf eine andere, können wir den Grad unserer Abweichungen ermitteln und Zeitverschwendung vermeiden. Wenn wir uns über Prinzipien nicht einigen können und darüber streiten, wo die geschweiften Klammern hingehören, löst das nicht die zugrunde liegende Uneinigkeit.

### 3.1 Werte

Kompetenter Programmierung entsprechen drei Werte: Kommunikation, Einfachheit und Flexibilität. Auch wenn diese drei Werte nicht immer harmonieren, ergänzen sie sich doch in der Regel. Die besten Programme bieten viele Optionen für zukünftige Erweiterungen, enthalten keine irrelevanten Elemente und sind leicht zu erfassen und zu verstehen.

#### 3.1.1 Kommunikation

Code lässt sich am besten vermitteln, wenn ein Leser ihn verstehen, ändern und verwenden kann. Während der Programmierung ist es verlockend, nur an den Computer zu denken. Manchmal ist es aber vorteilhaft, wenn ich an andere Dinge denke, während ich programmiere. Ich komme zu klarerem Code, der leichter zu lesen und kosteneffizienter ist, mein Denken ist klarer, ich gebe mir selbst eine neue Perspektive, mein Stressfaktor sinkt und ich komme einigen meiner sozialen Verpflichtungen nach.

Zur Programmierung hat mich ursprünglich hingezogen, dass ich mit etwas außerhalb von mir selbst kommunizieren konnte. Allerdings wollte ich mich nicht mit unnachgiebigen, unerklärlichen, nervenden Wesen auseinandersetzen. Doch die Programmierung, die den Menschen vollkommen außen vor gelassen hatte, verblasste nach nur einigen Jahrzehnten. Und auch meine gedanklich immer raffinierter werdenden Konstruktionen wurden farblos und abgestanden.

Zu den frühen Erfahrungen, die mich dazu gebracht haben, mich auf die Kommunikation zu konzentrieren, gehört die Entdeckung des von Knuth geprägten Paradigmas *Literate Programming* (literarisches Programmieren): Ein Programm sollte sich wie ein Buch lesen lassen. Es sollte eine Handlung, einen Rhythmus und reizvolle kleine Formulierungen enthalten.

Als Ward Cunningham und ich über literarische Programme gelesen hatten, wollten wir das sogleich ausprobieren. Wir haben uns mit einem der saubersten Smalltalk-Codestücke überhaupt – dem `ScrollController` – hingesezt und versucht, es in eine Geschichte zu verwandeln. Stunden später hatten wir den Code auf unsere Weise zu einem verständlichen Dokument umgeschrieben. Immer wenn sich ein Stück Logik schwierig erklären ließ, war es einfacher, den Code umzuschreiben, als zu erläutern, warum der Code schwer zu verstehen ist. Die Forderung nach Kommunikation änderte unsere Perspektive der Codierung.

Es gibt eine solide wirtschaftliche Grundlage dafür, sich während der Programmierung auf die Kommunikation zu konzentrieren. Der Großteil der Softwarekosten entsteht, nachdem die Software erstmalig bereitgestellt wurde. Wenn ich über meine Erfahrungen zum Modifizieren von Code nachdenke, stelle ich fest, dass ich wesentlich mehr Zeit damit verbringe, den vorhandenen Code zu lesen, als neuen Code zu schreiben. Möchte ich also meinen Code billiger machen, sollte ich ihn verständlicher formulieren.

Konzentriert man sich darauf, die Ziele des Codes zu vermitteln, verbessert dies den Denkprozess – er wird realistischer. Das hängt zum Teil damit zusammen, dass das Gehirn stärker mobilisiert wird. Wenn ich denke: »Wie sieht dies jemand anderes?« werden andere Neuronen aktiviert, als wenn ich lediglich mit mir und meinem Computer beschäftigt bin. Ich trete von meiner isolierten Perspektive einen Schritt zurück und sehe mein Problem und die Lösung in einem neuen Licht. Die Verbesserung ergibt sich auch aus dem geringeren Druck in kommerzieller Hinsicht, wenn ich weiß, dass ich auf dem richtigen Weg bin. Und da ich schließlich einer sozial orientierten Spezies angehöre, ist es realistischer, soziale Fragen explizit zu berücksichtigen, als sie wider besseres Wissen zu ignorieren.

### 3.1.2 Einfachheit

In *The Visual Display of Quantitative Information* bringt Edward Tufte eine Übung, in der er in einem Graphen nacheinander alle Markierungen löscht, die keine Informationen beisteuern. Der resultierende Graph ist vollkommen neu und wesentlich leichter zu verstehen als das Original.

Beseitigt man übermäßige Komplexität, können diejenigen, die Programme lesen, verwenden und modifizieren, die Programme auch schneller verstehen. Bestimmte komplexe Elemente sind natürlich unabdingbar, um die Komplexität des zu lösenden Problems genau widerzuspiegeln. Ein Teil der Komplexität stammt jedoch aus den Phasen, als wir darum gekämpft haben, das Programm überhaupt zum Laufen zu bekommen. Gerade diese überflüssige Komplexität mindert den Wert der Software, weil es dadurch zum einen weniger wahrscheinlich ist, dass sie korrekt läuft, und es zum anderen schwierig sein wird, die Software später erfolgreich ändern zu können. Zur Programmierung gehört auch, dass man zurückblickt, sich ansieht, was man getan hat, und die Spreu vom Weizen trennt.

Einfachheit ist Ansichtssache. Was für einen erfahrenen Programmierer, der den Umgang mit Powertools beherrscht, einfach scheint, sieht für einen Einsteiger möglicherweise unüberschaubar komplex aus. So wie der Autor guter Romane seine Leser vor Augen hat, sollte ein Programmierer an sein Zielpublikum denken, wenn gute Programme entstehen sollen. Es ist zwar in Ordnung, die Leser etwas herauszufordern, doch zu viel Komplexität macht sie abspenstig.

Die Entwicklung der Rechentechnik vollzieht sich in Wellen von Komplexitätssteigerung und Vereinfachung. Mainframe-Architekturen wurden immer verschnörkelter, bis Minicomputer auf der Bildfläche erschienen. Auch wenn Minicomputer nicht alle Probleme der Mainframes lösen konnten, zeigte sich, dass diese Probleme bei vielen Anwendungen gar nicht ins Gewicht fielen. Programmiersprachen durchlaufen ebenfalls Wellen – sie werden komplexer und dann wieder einfacher. Aus C ist C++ und daraus wiederum Java hervorgegangen. Und Java wird jetzt für sich immer komplizierter.

Das Streben nach Einfachheit macht Innovation möglich. JUnit war wesentlich einfacher als die Testtools, die es größtenteils ersetzt hat. JUnit hat ein breites Spektrum ähnlich aussehender Instrumente, Add-ons und neuer Techniken für das Programmieren und Testen hervorgebracht. Die neueste Version, JUnit 4, fühlt sich nicht mehr so »reduziert« an, obwohl ich bei jeder Entscheidung, die zu einer höheren Komplexität geführt hat, mitgewirkt oder ihr zugestimmt habe. Eines Tages wird jemand ein Instrument vorstellen, mit dem Programmierer auf wesentlich einfachere Weise als mit JUnit Tests schreiben können. Die neue Idee wird eine weitere Welle der Innovation auslösen.

Wenden Sie Einfachheit auf allen Ebenen an. Bereiten Sie den Code so auf, dass sich kein Code löschen lässt, ohne Informationen zu verlieren. Verzichten Sie in Ihrem Entwurf auf überflüssige Elemente. Hinterfragen Sie die Anforderungen, um diejenigen herauszukristallisieren, die wirklich entscheidend sind. Wenn Sie übermäßige Komplexität beseitigen, erscheint der verbleibende Code in einem helleren Licht und Sie haben die Möglichkeit, sich ihm von neuem zu nähern.

Kommunikation und Einfachheit arbeiten oftmals Hand in Hand. Je geringer die Komplexität ist, desto leichter lässt sich ein System verstehen. Je mehr Sie sich auf Kommunikation konzentrieren, desto einfacher ist es zu sehen, welche Komplexität verworfen werden kann. Manchmal finde ich aber eine Vereinfachung, die es erschwert, ein Programm zu verstehen. In diesen Fällen ziehe ich die Kommunikation der Vereinfachung vor. Derartige Situationen sind selten, weisen aber gewöhnlich auf bestimmte Vereinfachungen in größerem Maßstab hin, die ich bislang noch nicht erkannt habe.

### **3.1.3 Flexibilität**

Von den drei hier aufgeführten Werten muss Flexibilität immer als Rechtfertigung für ineffiziente Codierung und Entwurfspraktiken erhalten. Ich habe schon Programme gesehen, die eine Konstante abrufen, indem sie nach einer Umgebungsvariablen suchen, die den Namen eines Verzeichnisses angibt, das eine Datei enthält, in der der konstante Wert zu finden ist. Weshalb diese ganze Komplexität? – Flexibilität! – Programme sollten flexibel sein, doch nur an den Stellen, die Änderungen unterliegen. Sofern sich die Konstante nie ändert, bringt die Komplexität nur Kosten ohne Nutzen.

Da die meisten Kosten eines Programms nach seiner ersten Bereitstellung entstehen, sollten sich Programme leicht ändern lassen. Vielleicht aber ist die Flexibilität, die mir für morgen vorschwebt, nicht das, was ich brauche, falls ich den Code ändern muss. Deshalb ist die Flexibilität, die sich durch Einfachheit und umfangreiche Tests erreichen lässt, effektiver als die Flexibilität, die mit spekulativem Design angestrebt wird.

Wählen Sie Patterns, die Flexibilität fördern und unmittelbaren Nutzen bringen. Bei Patterns mit unmittelbaren Kosten und nur verzögertem Nutzen ist Geduld oftmals die beste Strategie. Legen Sie sie in die Schublade zurück, bis sie benötigt werden. Dann können Sie sie in genau der Weise anwenden, wie es angebracht ist.

Flexibilität kann auf Kosten erhöhter Komplexität gehen. Zum Beispiel ist ein Programm mit benutzerkonfigurierbaren Optionen flexibler, erfordert aber eine Konfigurationsdatei und verlangt, die Optionen bei der Programmierung zu berücksichtigen. Einfachheit kann Flexibilität fördern. Wenn Sie im obigen Beispiel einen Weg finden, die konfigurierbaren Optionen ohne Werteinbuße zu eliminieren, erhalten Sie ein Programm, das sich später leichter ändern lässt.

Software wird auch flexibler, wenn man ihre Kommunikationsfähigkeit erweitert. Je mehr Programmierer den Code schnell lesen, verstehen und ändern können, desto mehr Optionen besitzt Ihre Organisation für zukünftige Änderungen.

Die in den folgenden Abschnitten aufgeführten Patterns fördern Flexibilität, indem sie den Programmierern helfen, einfache, verständliche Anwendungen zu erstellen, die sich ändern lassen.

## 3.2 Prinzipien

Die Implementation Patterns weisen nicht »einfach so« den Weg. Jedes einzelne Pattern drückt einen oder mehrere Werte von Kommunikation, Einfachheit und Flexibilität aus. Prinzipien verkörpern eine andere Ebene allgemeiner Ideen, die ebenfalls die Grundlage der Patterns bilden, aber für die Programmierung spezifischer sind als die Werte.

Die Untersuchung von Prinzipien ist aus mehreren Gründen wertvoll. Klare Prinzipien können zu neuen Patterns führen, genau wie das Periodensystem der Elemente zur Entdeckung neuer Elemente geführt hat. Prinzipien können eine Erklärung liefern für die Motivation hinter einem Pattern, die eher mit allgemeinen statt mit spezifischen Ideen verbunden ist. Entscheidungen bei der Auswahl widersprüchlicher Patterns werden oftmals am besten in Form von Prinzipien und nicht von Besonderheiten der beteiligten Patterns diskutiert. Denn wenn Sie die Prinzipien verstanden haben, können Sie sich besser orientieren, falls Sie mit einer neuen Situationen zu tun haben.

Wenn ich zum Beispiel mit einer neuen Programmiersprache zu tun habe, nutze ich meine Kenntnisse über Prinzipien, um einen effizienten Programmierstil zu entwickeln. Dabei muss ich weder vorhandene Stile nachahmen noch (was sogar schlechter wäre) an meinem Stil in irgendeiner anderen Programmiersprache festhalten. (FORTRAN-Code können Sie in jeder Sprache schreiben, doch sei davon abgeraten.) Habe ich die Prinzipien verstanden, kann ich mich in neue Situationen schnell einarbeiten und souverän agieren. Es folgt nun die Liste der Prinzipien hinter den Implementation Patterns.

### 3.2.1 Lokale Konsequenzen

Strukturieren Sie den Code so, dass sich Änderungen nur lokal auswirken. Kann eine Änderung an der einen Stelle ein Problem an einer anderen Stelle hervorrufen, so steigen die Kosten der Änderung drastisch an. Code mit überwiegend lokalen Konsequenzen teilt sich dem Leser effektiver mit. Er lässt sich schrittweise erfassen, ohne dass man sich zuerst das Gesamtkonzept erarbeiten muss.

Weil die Hauptmotivation der Implementation Patterns darin besteht, die Kosten für erforderliche Änderungen niedrig zu halten, liegt vielen Patterns das Prinzip der lokalen Konsequenzen zugrunde.

### 3.2.2 Minimale Wiederholung

Das Prinzip der minimalen Wiederholung trägt dazu bei, die Konsequenzen lokal zu halten. Wenn Sie den gleichen Code an mehreren Stellen implementieren und eine Instanz des Codes ändern, müssen Sie entscheiden, ob Sie auch alle anderen Vorkommen des Codes ändern oder nicht. Die Änderungen sind nicht mehr lokal. Je mehr Kopien des Codes vorkommen, desto höher sind die Änderungskosten.

Kopierter Code ist nur eine Form der Wiederholung. Parallele Klassenhierarchien sind ebenfalls wiederholend und verletzen das Prinzip der lokalen Konsequenzen. Wenn ich eine konzeptionelle Änderung vornehme, muss ich zwei oder mehr Klassenhierarchien ändern, und die Änderungen wirken sich in der Breite aus. Der Code lässt sich verbessern, wenn die Struktur neu aufgebaut wird, sodass die Änderungen wieder lokal bleiben.

Duplizierung tritt immer erst zu Tage, wenn sie entstanden ist, und bleibt manchmal selbst dann eine ganze Zeit lang unerkannt. Wenn ich sie bemerke, kann ich nicht immer einen guten Weg finden, sie zu eliminieren. Duplizierung an sich ist nicht schlimm, sie erhöht nur die Kosten, sofern Änderungen notwendig sind.

Zum Beispiel lässt sich Duplizierung beseitigen, indem man Programme in kleinere Einheiten aufteilt – kleine Anweisungen, kleine Methoden, kleine Objekte, kleine Pakete. In großen Logikabschnitten ist es eher wahrscheinlich, dass Teile anderer großer Logikabschnitte dupliziert werden. Gerade diese Gemeinsamkeiten machen Patterns möglich – während es Unterschiede zwischen verschiedenen Codeteilen gibt, sind auch viele Ähnlichkeiten vorhanden. Programme lassen sich leichter lesen und kostengünstiger modifizieren, wenn man deutlich ausdrückt, welche Teile des Programms identisch, welche Teile lediglich ähnlich und welche Teile vollkommen unterschiedlich sind.

### 3.2.3 Logik und Daten zusammenhalten

Ein anderes Prinzip, das sich aus dem Prinzip der lokalen Konsequenzen ableitet, ist es, Logik und Daten zusammenzuhalten. Bringen Sie die Logik und die Daten, auf denen sie operiert, nahe zueinander, möglichst in derselben Methode oder im selben Objekt, zumindest im selben Paket. Um eine Änderung vorzunehmen, müssen Logik und Daten höchstwahrscheinlich zur gleichen Zeit geändert werden. Wenn sie nebeneinander stehen, bleiben die Konsequenzen ihrer Änderung lokal.

Es ist nicht immer gleich offensichtlich, wohin Logik oder Daten gehören, um dieses Prinzip zu erfüllen. Vielleicht schreibe ich Code in A und erkenne, dass ich Daten von B brauche. Aber erst, nachdem ich den Code zum Laufen gebracht habe, stelle ich fest, dass er zu weit von den Daten entfernt ist. Dann muss ich entscheiden, was zu tun ist: Den Code zu den Daten verschieben, die Daten zum Code verschieben, den Code und die Daten zusammen in einem Hilfsobjekt unterbringen – oder sogar realisieren, dass ich mir momentan gar keine Gedanken darüber machen kann, wie ich beide Komponenten in einer Weise zusammenbringe, die sich dem Leser eines Programms effektiv vermitteln lässt.

### 3.2.4 Symmetrie

Symmetrie ist ein anderes Prinzip, das ich ständig anwende. Symmetrien sind in Programmen reichlich vorhanden. Eine `add()`-Methode wird von einer `remove()`-Methode begleitet. Eine Gruppe von Methoden übernimmt die gleichen Parameter. Alle Felder in einem Objekt besitzen die gleiche Lebensdauer. Symmetrien zu erkennen und deutlich auszudrücken, macht den Code verständlicher. Nachdem die Leser eine Hälfte der Symmetrie verstanden haben, erschließt sich ihnen die andere Hälfte im Handumdrehen.

Symmetrie wird häufig in räumlichen Begriffen diskutiert: achsensymmetrisch, rotationsymmetrisch usw. In Programmen hat Symmetrie nur selten eine grafische Bedeutung, sie ist konzeptionell. Symmetrie im Code zeigt sich, wo derselbe Gedanke überall dort, wo er im Code erscheint, in der gleichen Weise ausgedrückt wird. Das folgende Codefragment zeigt ein Beispiel für fehlende Symmetrie:

```
void process() {
    input();
    count++;
    output();
}
```

Die zweite Anweisung ist konkreter als die beiden Meldungen. Diesen Code würde ich auf der Basis der Symmetrie wie folgt umformulieren:

```
void process() {
    input();
    incrementCount();
    output();
}
```

Diese Methode verletzt immer noch die Symmetrie. Die Operationen `input()` und `output()` sind nach ihrem Zweck benannt, `incrementCount()` nach einer Implementierung. Auf der Suche nach Symmetrien denke ich darüber nach, warum ich den Zähler (`count`) inkrementiere, was vielleicht in folgendem Code resultiert:

```

void process() {
    input();
    tally();
    output();
}

```

Symmetrien aufzufinden und auszudrücken, ist oftmals ein vorbereitender Schritt, um doppelten Code zu entfernen. Sofern ein ähnlicher Gedanke an mehreren Stellen im Code existiert, ist schon viel gewonnen, wenn man diese Elemente einander symmetrisch macht, um sie dann zu vereinheitlichen.

### 3.2.5 Deklarative Ausdrucksform

Den Implementation Patterns liegt auch das Prinzip zugrunde, soviel wie möglich von meinen Absichten deklarativ auszudrücken. Imperative Programmierung ist leistungsfähig und flexibel, doch um das Programm zu lesen, müssen Sie dem Ausführungsfaden folgen. In meinem Kopf muss ich ein Modell vom Zustand des Programms sowie vom Kontrollfluss und von den Daten aufbauen. In denjenigen Teilen eines Programms, die eher einfache Fakten – ohne Abfolge oder Bedingungen – darstellen, lässt sich Code leichter lesen, wenn er lediglich deklarativ ist.

Zum Beispiel konnten Klassen in älteren Versionen von JUnit eine statische `suite()`-Methode besitzen, die eine Menge von auszuführenden Tests zurückgegeben hat:

```

public static junit.framework.Test suite() {
    Test result= new TestSuite();
    ...komplizierte Programmteile...
    return result;
}

```

Jetzt kommt die einfache und häufig gestellte Frage: »Welche Tests sollen ausgeführt werden?« In den meisten Fällen aggregiert die `suite()`-Methode lediglich die Tests in einem Bündel von Klassen. Da allerdings die `suite()`-Methode allgemein gehalten ist, muss ich die Methode lesen und verstehen, wenn ich sicher sein möchte.

Auf der anderen Seite verwendet JUnit 4 das Prinzip der deklarativen Ausdrucksform, um dasselbe Problem zu lösen. Anstelle einer Methode, die eine Suite von Tests zurückgibt, gibt es einen speziellen Testrunner, der die Tests in einer Menge von Klassen ausführt (der allgemeine Fall):

```

@RunWith(Suite.class)
@TestClasses({
    SimpleTest.class,
    ComplicatedTest.class
})
class AllTests {
}

```

Wenn ich weiß, dass Tests mit dieser Methode aggregiert werden, muss ich mir nur die `TestClasses`-Annotation ansehen, um zu wissen, welche Tests ausgeführt werden. Weil die Festlegung der Suite deklarativ ist, muss ich keine verzwickten Ausnahmen befürchten. Diese Lösung gibt zwar die Leistung und Allgemeingültigkeit der ursprünglichen `suite()`-Methode auf, doch lässt sich der Code durch den deklarativen Stil leichter lesen. (Die `RunWith`-Annotation bietet sogar noch mehr Flexibilität für das Ausführen von Tests als die `suite()`-Methode, doch ist das eine Geschichte für ein anderes Buch.)

### 3.2.6 Änderungsgeschwindigkeit

Ein letztes Prinzip ist es, Logik und Daten, die sich mit der gleichen Geschwindigkeit ändern, zusammenzubringen, sowie Logik und Daten, die sich mit unterschiedlichen Geschwindigkeiten ändern, zu trennen. Diese Änderungsgeschwindigkeiten sind eine Form von zeitlicher Symmetrie. Manchmal gilt das Prinzip Änderungsgeschwindigkeit für Änderungen, die ein Programmierer vornimmt. Schreibe ich zum Beispiel Steuer-Software, trenne ich Code, der allgemeine Steuerberechnungen durchführt, von dem Code, der speziell für ein bestimmtes Jahr vorgesehen ist. Der Code ändert sich mit unterschiedlichen Geschwindigkeiten. Wenn ich im nächsten Jahr Änderungen vornehme, möchte ich sicher sein, dass der Code aus dem vorhergehenden Jahr immer noch funktioniert. Durch die Trennung dieser Codebestandteile erhalte ich mehr Vertrauen in die lokalen Konsequenzen meiner Änderungen.

Für die Änderungsgeschwindigkeit von Daten gilt: Alle Felder in einem einzelnen Objekt sollten sich ungefähr mit derselben Geschwindigkeit ändern. Zum Beispiel sollten Felder, die nur während der Aktivierung einer einzelnen Methode geändert werden, lokale Variablen sein. Wenn sich zwei Felder zusammen, aber asynchron zu ihren benachbarten Feldern ändern, gehören sie wahrscheinlich in ein Hilfsobjekt. Können sich beispielsweise bei einem Finanzinstrument der Wert (`value`) sowie die Währung (`currency`) gemeinsam ändern, sollten diese beiden Felder besser als Hilfsobjekt (im folgenden Code `Money` genannt) ausgedrückt werden. So wird der Code

```
setAmount(int value, String currency) {
    this.value= value;
    this.currency= currency;
}
```

erst zu

```
setAmount(int value, String currency) {
    this.value= new Money(value, currency);
}
```

und dann später zu:

```
setAmount(Money value) {  
    this.value= value;  
}
```

Das Prinzip der Änderungsgeschwindigkeit ist eine Anwendung der Symmetrie, jedoch der zeitlichen Symmetrie. Im obigen Beispiel sind die beiden ursprünglichen Felder `value` und `currency` symmetrisch. Sie ändern sich zur selben Zeit. Allerdings sind sie mit den anderen Feldern im Objekt nicht symmetrisch. Drückt man die Symmetrie aus, indem man die Felder in ihr eigenes Objekt stellt, vermittelt das dem Leser ihre Beziehung und schafft wahrscheinlich weitere Möglichkeiten, um später Code-duplizierung zu verringern und Konsequenzen stärker zu lokalisieren.

### 3.2.7 Zum Schluss

Dieses Kapitel hat die theoretischen Grundlagen der Implementation Patterns eingeführt. Die Werte Kommunikation, Einfachheit und Flexibilität bieten weitreichende Motivation für die Patterns. Die Prinzipien der lokalen Konsequenzen, der minimalen Wiederholung, der gemeinsamen Unterbringung von Logik und Daten, der Symmetrie, des deklarativen Ausdrucks und der Änderungsgeschwindigkeit helfen, die Werte in Aktionen zu übersetzen. Jetzt kommen wir zu den Patterns, die spezifische Lösungen für sich wiederholende taktische Probleme in der Programmierung verkörpern.

Das nächste Kapitel beschreibt die wirtschaftlichen Faktoren, die die Konzentration auf Kommunikation per Code zu einer wertvollen Tätigkeit machen.