

Code-Smells: Schlechte Gerüche im Code

von Kent Beck und Martin Fowler

»Wenn es stinkt, muss es gewechselt werden.«

—Oma Beck zur Philosophie der Kindererziehung

Inzwischen dürften Sie eine ziemlich klare Vorstellung davon haben, wie Refactoring funktioniert. Aber nur weil Sie jetzt wissen, wie es anzuwenden ist, heißt das noch lange nicht, dass Sie auch wissen, wann Sie es einsetzen sollten. Die Entscheidung, wann man mit einem Refactoring anfängt – und wann man wieder aufhört –, ist von ebenso großer Bedeutung wie die versierte technische Durchführung.

Und das ist ein Dilemma: Es ist ganz leicht zu erklären, wie eine Instanzvariable gelöscht oder eine Hierarchie erzeugt wird. Das sind simple Angelegenheiten. Die Antwort auf die Frage, *wann* diese Aufgaben erledigt werden sollten, fällt hingegen nicht so klar und deutlich aus. Ich möchte mich dabei nicht auf irgendeine vage Vorstellung der Ästhetik des Programmierens berufen (was wir Berater zugegebenermaßen ja meistens tun), sondern stattdessen eine handfestere Antwort geben.

Während ich die erste Ausgabe dieses Buchs schrieb und bei Kent Beck in Zürich zu Besuch war, grübelte ich über genau diese Fragestellung nach. Vielleicht stand er unter dem Einfluss des olfaktorischen Eindrucks, den seine damals gerade geborene Tochter bei ihm hinterließ, jedenfalls hatte er sich ausgedacht, das »Wann« eines Refactorings in Form von Gerüchen zu beschreiben.

Nun denken Sie sich vielleicht: »Gerüche? Und das soll besser sein als eine vage Vorstellung der Ästhetik des Programmierens?« Ja, tatsächlich. Wir hatten bis dato wirklich eine Menge Code gesehen, der für Projekte geschrieben wurde, die auf der gesamten Skala von äußerst erfolgreich bis praktisch tot zu finden sind. Dabei haben wir gelernt, auf bestimmte Strukturen im Code zu achten, die es nahelegen – oder uns manchmal auch mit der Nase darauf stoßen –, dass ein Refactoring angebracht ist. (Wir wechseln in diesem Kapitel zur 1. Person Plural, um die Tatsache widerzuspiegeln, dass Kent und ich dieses Kapitel gemeinsam verfasst haben. Sie können den jeweiligen Autor daran erkennen, dass die guten Witze von mir stammen und die anderen von ihm.)

Wir werden nicht den Versuch unternommen, Ihnen genaue Kriterien dafür zu vermitteln, wann ein Refactoring überfällig ist. Nach unserer Erfahrung gibt es schlicht und einfach keine Kriterien, die mit dem Urteilsvermögen eines sachkundigen Menschen mithalten können. Wir werden jedoch auf Indikatoren zurückgreifen, die Hinweise darauf geben, dass es möglicherweise Probleme gibt, die durch ein Refactoring zu lösen sind. Sie werden Ihr ganz eigenes Gespür dafür entwickeln müssen, wie viele Instanzvariablen oder wie viele Codezeilen in einer Methode angebracht sind.

Kapitel 3

Code-Smells: Schlechte Gerüche im Code

Verwenden Sie dieses Kapitel und die Tabelle am Ende des Buchs, wenn Sie sich nicht sicher sind, welche Refactorings angebracht sind. Lesen Sie hier nach (oder überfliegen Sie die Tabelle), um festzustellen, welchen Code-Smell Sie wahrgenommen haben. Schlagen Sie anschließend die von uns empfohlenen Refactorings nach, um festzustellen, ob Sie damit weiterkommen. Sie werden womöglich nicht exakt das wiederfinden, was Sie wahrgenommen haben, kommen aber hoffentlich einen Schritt in der richtigen Richtung voran.

3.1 Rätselhafte Bezeichnung

Beim Lesen eines Krimis kann es durchaus Spaß machen, über den Text nachzugrübeln, um zu verstehen, was passiert – aber nicht beim Lesen von Code. In unserer Fantasie können wir Autoren faszinierender Mitratekrimis sein, aber unser Code muss nüchtern und klar gestaltet sein. Aussagekräftige Namen gehören zu den wichtigsten Bestandteilen gut verständlichen Codes, deshalb haben wir uns eine Menge Gedanken über die Benennung von Funktionen, Modulen, Variablen und Klassen gemacht, damit sie unmissverständlich aussagen, welchen Zweck sie erfüllen und wie sie zu verwenden sind.

Leider gehört die Auswahl geeigneter Bezeichnungen bei der Programmierung zu den beiden schwierigsten Aufgaben (<https://martinfowler.com/bliki/TwoHardThings.html>). Aus diesem Grund sind Umbenennungen vermutlich die am häufigsten vorgenommenen Refactorings: *Funktionsdeklaration ändern* (Abschnitt 6.5, hier zum Umbenennen einer Funktion), *Variable umbenennen* (Abschnitt 6.7) und *Feld umbenennen* (Abschnitt 9.2). Oft zögern Entwickler, Dinge umzubenennen, weil sie denken, dass sich der Aufwand nicht lohnt. Aber ein aussagekräftiger Name kann Ihnen zukünftiges, stundenlanges Herumrätselfen ersparen.

Bei Umbenennungen geht es nicht nur darum, die Bezeichnungen zu ändern. Wenn Ihnen nicht unverzüglich bessere Namen einfallen, ist das oft ein Hinweis auf einen tiefer liegenden Designfehler. Das Rätselraten über irreführende Namen hat schon häufig zu beträchtlichen Vereinfachungen unseres Codes geführt.

3.2 Redundanter Code

Wenn es mehrere Stellen mit gleichartiger Codestruktur gibt (sog. Redundanzen), können Sie sicher sein, dass sich Ihr Programm verbessern lässt, wenn Sie eine Möglichkeit finden, diese Stellen zu vereinen. Redundanter Code bedeutet, dass Sie bei jedem Zugriff darauf sorgfältig überprüfen müssen, ob es irgendeinen Unterschied zwischen den verschiedenen Duplikaten gibt. Wenn Sie Änderungen an redundantem Code vornehmen möchten, müssen Sie sämtliche Duplikate finden und modifizieren.

Das einfachste Problem mit mehrfach vorhandenem Code tritt auf, wenn zwei Methoden derselben Klasse den gleichen Ausdruck enthalten. Sie müssen dann lediglich *Funktion extrahieren* (Abschnitt 6.1) anwenden und den Code an beiden Stellen aufrufen. Wenn zwar ähnlicher, aber nicht vollständig identischer Code vorliegt, sollten Sie sich an dem Refactoring *Anweisungen verschieben* (Abschnitt 8.6) versuchen. Es sieht vor, den Code so anzutragen, dass ähnliche Teile zusammenhängen, um diese leicht extrahieren zu können. Sollten die mehrfach vorhandenen Codeabschnitte zu Unterklassen derselben Basisklasse gehören, können Sie *Methode nach oben verschieben* (Abschnitt 12.1) verwenden, um so ein gegenseitiges Aufrufen zu verhindern.

3.3 Lange Funktion

Unserer Erfahrung nach sind Programme mit kurzen Funktionen die nachhaltigsten und langlebigsten. Programmierer, die sich zum ersten Mal mit einer solchen Codebasis beschäftigen, haben oft den Eindruck, dass überhaupt keine Berechnungen stattfinden, sondern dass das Programm aus einer endlosen Verkettung von Funktionsaufrufen besteht. Wenn Sie einige Jahre mit solch einem Programm gearbeitet haben, wird Ihnen allerdings klar, wie nützlich diese kurzen Funktionen tatsächlich sind. Kurze Funktionen bieten alle Vorteile der Indirektion: Sie sind oft selbsterklärend, wiederverwendbar und leicht selektierbar.

Seit den Anfangstagen der Programmierung war klar: Je länger eine Funktion ist, desto schwieriger ist es, sie zu verstehen. In älteren Programmiersprachen war der Aufruf von Subroutinen mit einem gewissen Aufwand verbunden, und das hielt die Programmierer davon ab, kurze Funktionen einzusetzen. Bei modernen Sprachen entfällt dieser Aufwand bei prozessinternen Aufrufen weitgehend. Für den Betrachter des Codes ist es zwar immer noch etwas aufwendiger, weil man den Kontext wechseln muss, um zu erkennen, welche Aufgabe eine Funktion erledigt. Es gibt jedoch Entwicklungsumgebungen, die es ermöglichen, schnell zwischen dem Aufruf einer Funktion und ihrer Deklaration hin- und herzuwechseln oder beide gleichzeitig anzuzeigen und diesen Schritt somit zu erleichtern. Wirklich entscheidend für das bessere Verständnis kurzer Funktionen sind aber aussagekräftige Bezeichnungen. Wenn eine Funktion einen aussagekräftigen Namen hat, müssen Sie sich ihre Deklaration meistens gar nicht ansehen.

Daraus ergibt sich letztendlich, dass Sie bei der Aufteilung von Funktionen aggressiver vorgehen sollten. Wir folgen einer Heuristik, nämlich, dass wir, wenn wir das Bedürfnis verspüren, etwas zu kommentieren, stattdessen eine neue Funktion schreiben. Die Funktion enthält zwar den Code, den wir kommentieren wollten, sie wird aber *entsprechend ihrer Intention* benannt anstatt nach ihrer tatsächlichen Funktionsweise. Wir können diese Vorgehensweise auf mehrere Codezeilen oder auch nur auf eine einzelne Zeile anwenden. Wir gehen sogar auf diese Weise vor, wenn der Aufruf der Methode länger ist als der Code, der dadurch ersetzt wird – vorausgesetzt, die Bezeichnung der Methode zielt auf den Zweck des Codes ab. Hier ist nicht die Länge der Funktion entscheidend, sondern der *semantische Abstand*, also der inhaltliche Unterschied zwischen der Aufgabe, die der Code erledigt, und der Art und Weise, wie er sie erledigt.

In 99% der Fälle müssen Sie lediglich *Funktion extrahieren* (Abschnitt 6.1) anwenden, um eine Funktion zu verkürzen. Suchen Sie nach Teilen der Funktion, die gut zusammenpassen, und erstellen Sie eine neue Funktion.

Wenn eine Funktion viele Parameter und temporäre Variablen verwendet, sind diese beim Extrahieren hinderlich. Wenn Sie versuchen, *Funktion extrahieren* (Abschnitt 6.1) anzuwenden, müssen Sie so viele Parameter an die extrahierte Methode übergeben, dass das Ergebnis kaum besser verständlich ist als der ursprüngliche Code. Häufig ist *Temporäre Variable durch Abfrage ersetzen* (Abschnitt 7.4) anwendbar, um einige der temporären Variablen zu eliminieren. Lange Parameterlisten lassen sich durch *Parameterobjekt einführen* (Abschnitt 6.8) und *Vollständiges Objekt erhalten* (Abschnitt 11.4) verkürzen.

Wenn Sie das ausprobiert haben und noch immer zu viele temporäre Variablen und Parameter vorhanden sind, ist es an der Zeit, schwereres Geschütz aufzufahren: *Funktion durch Befehl ersetzen* (Abschnitt 11.9).

Kapitel 3

Code-Smells: Schlechte Gerüche im Code

Woran genau erkennt man den zu extrahierenden Code? Halten Sie Ausschau nach Kommentaren, denn sie weisen oft auf diese Art des semantischen Abstands hin. Ein Codeblock mit einem Kommentar, der die Anweisungen erläutert, kann durch eine Methode ersetzt werden, deren Bezeichnung auf dem Kommentar beruht. Es kann sich schon lohnen, eine einzige Zeile zu extrahieren, wenn sie einer Erklärung bedarf.

Bedingte Anweisungen, Fallunterscheidungen und Schleifen sind weitere Hinweise darauf, dass eine Extraktion angebracht sein kann. Verwenden Sie für bedingte Ausdrücke *Bedingung zerlegen* (Abschnitt 10.1). Eine umfangreiche *switch*-Anweisung können Sie mit *Funktion extrahieren* (Abschnitt 6.1) in eine einzige Funktion umwandeln. Wenn mehrere *switch*-Anweisungen die gleiche Bedingung verwenden, sollten Sie *Bedingung durch Polymorphie ersetzen* (Abschnitt 10.4) verwenden.

Extrahieren Sie eine Schleife und den Code innerhalb der Schleife und erstellen Sie eine eigene Methode dafür. Wenn es Ihnen schwerfällt, die Methode der extrahierten Schleife zu benennen, könnte das daran liegen, dass sie mehrere unterschiedliche Aufgaben erledigt. Zögern Sie in diesem Fall nicht, *Schleife aufteilen* (Abschnitt 8.7) zu verwenden, um die verschiedenen Aufgaben auszulagern.

3.4 Lange Parameterliste

In den Anfangstagen der Programmierung wurde gelehrt, einer Funktion alle benötigten Werte als Parameter zu übergeben. Das ist auch durchaus verständlich, denn die Alternative war die Verwendung globaler Daten – diese sind allerdings schwer beherrschbar und können schnell Unheil mit sich bringen. Zudem sorgen lange Parameterlisten oft für Verwirrung.

Wenn Sie einen Parameter erhalten können, indem Sie einen anderen Parameter danach fragen, können Sie durch *Parameter durch Abfrage ersetzen* (Abschnitt 11.5) den zweiten Parameter entfernen. Anstatt lauter Daten aus einer vorhandenen Datenstruktur abzurufen, können Sie *Vollständiges Objekt erhalten* (Abschnitt 11.4) verwenden, um die Datenstruktur selbst zu übergeben. Wenn bestimmte Parameter stets zusammen auftreten, sollten Sie diese mit *Parameterobjekt einführen* (Abschnitt 6.8) zu einer Datenstruktur kombinieren. Wird ein Parameter als Flag (Schalter) verwendet, der unterschiedliches Verhalten auslöst, wenden Sie *Steuerungs-Flag entfernen* (Abschnitt 11.3) an.

Klassen sind bestens dafür geeignet, die Länge von Parameterlisten zu verringern. Sie erweisen sich als besonders nützlich, wenn mehrere Funktionen die gleichen Parameter verwenden. In diesem Fall können Sie *Funktionen zu einer Klasse vereinen* (Abschnitt 6.9) verwenden, um die gemeinsamen Werte als Felder zu erfassen. Aus der Perspektive der funktionalen Programmierung handelt es sich hier um eine partiell angewandte Funktion (engl. *partially applied function*).

3.5 Globale Daten

Seitdem Software entwickelt wird, werden wir vor den Gefahren gewarnt, die mit globalen Daten einhergehen. Sie wurden von bösen Dämonen erfunden, die eine Ebene der Hölle bevölkern, die Ruhestätte derjenigen Programmierer ist, die es wagen, globale Daten zu verwenden. Hinsichtlich Feuer und Schwefel sind wir zwar einigermaßen skeptisch, aber globale Daten verbreiten sozusagen einen stechenden Geruch, dem wir regelmäßig begeg-

nen. Globale Daten sind so problematisch, weil sie überall in der Codebasis modifiziert werden können, und es gibt keine Möglichkeit, festzustellen, welcher Codeabschnitt Änderungen an ihnen vorgenommen hat. Das führt immer wieder zu Bugs, die einer spukhaften Erscheinung aus dem Jenseits gleichen – es ist nämlich sehr schwierig, herauszufinden, welcher Teil des Programms fehlerhaft ist. Dass globale Variablen zu den globalen Daten gehören, liegt auf der Hand, wir zählen jedoch auch Klassenvariablen und Singletons dazu.

Unser wichtigstes Verteidigungsmittel ist hier *Variable kapseln* (Abschnitt 6.6). Das ist immer der erste Schritt, wenn wir auf Daten stoßen, die dafür anfällig sind, von einem beliebigen Programmteil verändert zu werden. Wenn die Daten von einer Funktion gekapselt werden, können Sie zumindest ungefähr verfolgen, wo Änderungen vorgenommen werden, und den Zugriff auf die Daten allmählich kontrollieren. Dann ist es sinnvoll, den Gültigkeitsbereich so weit wie möglich zu begrenzen, indem die Funktion in eine Klasse oder ein Modul verlagert wird, sodass nur der Code der Klasse bzw. des Moduls Zugriffsrechte besitzt.

Globale Daten sind besonders tückisch, wenn sie veränderlich sind. Wenn Sie garantieren können, dass sich die globalen Daten nach dem Start des Programms niemals ändern, sind Sie auf der sicheren Seite – sofern Sie eine Programmiersprache verwenden, die eine solche Garantie ermöglicht.

Globale Daten sind ein schönes Beispiel für Paracelsus' Weisheit: »Alle Dinge sind Gift, und nichts ist ohne Gift; allein die Dosis macht, dass ein Ding kein Gift sei.« Mit einer kleinen Dosis globaler Daten werden Sie vielleicht ungeschoren davonkommen, es wird jedoch exponentiell schwieriger, sie zu handhaben, wenn ihr Umfang zunimmt. Auch wenn es nur sehr wenige Daten sind, ziehen wir es vor, sie zu kapseln – denn das ist der Schlüssel zur Realisierung von Änderungen, wenn die Software weiterentwickelt wird.

3.6 Veränderliche Daten

Änderungen an den Daten haben häufig unerwartete Folgen und führen zu kniffligen Fehlern. So kann ich etwa an einer Stelle Daten verändern, ohne zu bemerken, dass ein weiterer Teil der Software etwas ganz anderes erwartet und nicht mehr funktioniert. Ein solches Fehlschlagen ist besonders schwierig zu entdecken, wenn es nur unter selten vorkommenden Umständen auftritt. Aus diesem Grund beruht eine ganze Denkrichtung der Softwareentwicklung, nämlich die funktionale Programmierung, auf der Vorstellung, dass Daten sich niemals ändern sollten und dass beim Aktualisieren einer Datenstruktur stets eine Kopie der Datenstruktur mit den Änderungen zurückgegeben werden sollte, damit die ursprüngliche Datenstruktur unangetastet bleibt.

Programmiersprachen dieser Art kommen allerdings noch immer vergleichsweise selten zum Einsatz. Viele von uns verwenden Sprachen, in denen Variablen sich ändern dürfen. Das heißt aber nicht, dass wir die Vorteile der Unveränderlichkeit ignorieren sollten – es gibt noch immer viele Möglichkeiten, die mit uneingeschränkten Aktualisierungen der Daten einhergehenden Risiken zu begrenzen.

Sie können *Variable kapseln* (Abschnitt 6.6) verwenden, um sicherzustellen, dass Aktualisierungen durch schlanke Funktionen erfolgen, die sich einfacher überwachen und weiterentwickeln lassen. Wenn eine Variable aktualisiert wird, die unterschiedliche Dinge speichert, sollten Sie *Variable aufteilen* (Abschnitt 9.1) verwenden. Auf diese Weise trennen Sie die unterschiedlichen Dinge voneinander und vermeiden eine risikoreiche Aktualisierung.

Kapitel 3

Code-Smells: Schlechte Gerüche im Code

Versuchen Sie, so viel Programmlogik wie möglich aus dem Code, der die Aktualisierung vornimmt, zu verlagern, indem Sie *Anweisungen verschieben* (Abschnitt 8.6) und *Funktion extrahieren* (Abschnitt 6.1) verwenden, um den Code, der frei von Seiteneffekten ist, von allem zu trennen, was an der Aktualisierung beteiligt ist. Verwenden Sie für APIs *Abfrage von Veränderung trennen* (Abschnitt 11.1), um zu gewährleisten, dass Aufrufer keinen Code mit Seiteneffekten aufrufen müssen, wenn es nicht unbedingt erforderlich ist. Wir legen Wert darauf, baldmöglichst *Setter entfernen* (Abschnitt 11.7) anzuwenden – manchmal genügt es schon, nach Aufrufen eines Setters zu suchen, um Möglichkeiten zu finden, den Gültigkeitsbereich einer Variable zu verkleinern.

Veränderliche Daten, die an völlig anderen Stellen berechnet werden können, sind besonders töckisch und verbreiten diesen bereits beschriebenen stechenden Geruch. Sie sorgen nicht nur ständig für Verwirrung, Bugs und verpasste heimische Abendessen – sie sind auch gar nicht notwendig. Wir besprühen sie also mit einer hochkonzentrierten Essiglösung und verwenden *Abgeleitete Variable durch Abfrage ersetzen* (Abschnitt 9.3).

Veränderliche Daten stellen kein großes Problem dar, wenn es sich um eine Variable handelt, deren Gültigkeitsbereich nur einige wenige Zeilen umfasst – das Risiko nimmt jedoch zu, wenn der Gültigkeitsbereich größer wird. Verwenden Sie eines der beiden Refactorings *Funktionen zu einer Klasse vereinen* (Abschnitt 6.9) oder *Funktionen zu einer Transformation vereinen* (Abschnitt 6.10), um den Code für die Aktualisierung einer Variable einzuschränken. Wenn eine Variable Daten mit interner Struktur enthält, ist es für gewöhnlich besser, die gesamte Struktur zu ersetzen, indem Sie *Referenz durch Wert ersetzen* (Abschnitt 9.4) anwenden, anstatt ihre einzelnen Werte zu modifizieren.

3.7 Divergierende Änderungen

Wir strukturieren unsere Software, um Änderungen zu erleichtern. Der Begriff enthält den Wortbestandteil »Soft«, also »weich« oder »nachgebend«, und so sollte Software auch sein: leicht zu ändern. Wenn wir eine Änderung vornehmen, möchten wir, dass es möglich ist, diese Änderung an einer einzelnen, klar erkennbaren Stelle im System vorzunehmen. Wenn das nicht möglich ist, nehmen Sie einen von zwei eng verwandten Code-Smells wahr.

Divergierende Änderungen treten auf, wenn ein Modul häufig auf unterschiedliche Weise und aus verschiedenen Gründen geändert wird. Nehmen wir an, Sie untersuchen ein Modul und denken: »Ich werde diese drei Funktionen jedes Mal ändern müssen, wenn ich eine neue Datenbank bekomme. Und diese vier Funktionen werde ich ändern müssen, wenn es ein neues Finanzinstrument gibt.« Hierbei handelt es sich um einen Hinweis auf divergierende Änderungen. Fragen der Datenbankzugriffe und der Finanzverwaltung gehören zu verschiedenen Kontexten, und wir können uns das Programmiererleben sehr erleichtern, wenn wir dafür separate Module verwenden. Wenn wir eine Änderung in einem dieser Bereiche vornehmen, müssen wir auch nur diesen Bereich verstehen und können den anderen außer Acht lassen. Wir hielten das schon immer für wichtig, aber jetzt, da unsere Hirne mit zunehmendem Alter schrumpfen, ist es absolut unverzichtbar. Natürlich bemerkt man die Schwierigkeiten manchmal erst, nachdem schon einige Datenbanken oder Finanzinstrumente hinzugefügt wurden. Die Kontextgrenzen eines Programms sind in der Regel zu Beginn undeutlich und verschieben sich meist, wenn die Fähigkeiten eines Softwaresystems ausgebaut werden.

Wenn die beiden Aspekte auf natürliche Weise eine Sequenz bilden – Sie erhalten beispielsweise Daten aus der Datenbank und wenden anschließend die Finanzverfahren darauf an –, dann können Sie die beiden Bereiche mit *Phase aufteilen* (Abschnitt 6.11) durch eine eindeutige Datenstruktur voneinander trennen. Falls die Aufrufe hingegen eher hin- und herspringen, sollten Sie entsprechende Module erstellen und *Funktion verschieben* (Abschnitt 8.1) verwenden, um die Verarbeitungsschritte voneinander zu trennen. Wenn Funktionen beide Arten der Verarbeitungsschritte in sich vereinen, können Sie *Funktion extrahieren* (Abschnitt 6.1) verwenden, um sie vor dem Verschieben voneinander zu trennen. Falls die Module Klassen sind, kann *Klasse extrahieren* (Abschnitt 7.5) bei der formellen Unterteilung hilfreich sein.

3.8 Chirurgie mit der Schrotflinte

»Chirurgie mit der Schrotflinte« (engl. *Shotgun Surgery*) ist eng verwandt mit divergierenden Änderungen, ist aber das genaue Gegenteil davon. Sie nehmen diesen Code-Smell wahr, wenn Sie bei jeder Änderung, die Sie vornehmen, viele kleine Modifikationen an vielen verschiedenen Klassen durchführen müssen. Wenn die Stellen für notwendige Änderungen weit verstreut sind, ist es schwierig, erstere zu finden, und es passiert nur allzu leicht, dass eine wichtige Änderung übersehen wird.

In diesem Fall sollten Sie die Refactorings *Funktion verschieben* (Abschnitt 8.1) und *Feld verschieben* (Abschnitt 8.2) verwenden und alle Änderungen in einem einzigen Modul zusammenfassen. Falls eine ganze Reihe von Funktionen ähnliche Daten verarbeitet, sollten Sie *Funktionen zu einer Klasse vereinen* (Abschnitt 6.9) anwenden. Wenn Funktionen vorhanden sind, die eine Datenstruktur transformieren oder erweitern, sollten Sie *Funktionen zu einer Transformation vereinen* (Abschnitt 6.10) verwenden. Hier erweist sich *Phase aufteilen* (Abschnitt 6.11) oft als nützlich, wenn sich die Ergebnisse der betreffenden Funktionen für eine anschließende Programmphase kombinieren lassen.

Als nützliche Taktik gegen die »Chirurgie mit der Schrotflinte« erweist sich die Verwendung von Inline-Refactorings wie z.B. *Funktion inline platzieren* (Abschnitt 6.2) oder *Klasse inline platzieren* (Abschnitt 7.6), um schlecht voneinander getrennte Programmlogik zu bündeln. Das führt zwar zu langen Methoden oder großen Klassen, die Sie aber anschließend durch Extraktionen in sinnvollere Teile aufgliedern können. Auch wenn uns kleine Funktionen und Klassen in unserem Code unheimlich lieb sind, scheuen wir nicht davor zurück, bei der Umstrukturierung als Zwischenschritt auch mal größere zu erstellen.

3.9 Feature-Neid

Bei der Modularisierung eines Programms versuchen wir, den Code derart in verschiedene Bereiche aufzuteilen, dass die Interaktion innerhalb eines Bereichs maximiert und die Interaktion zwischen den Bereichen minimiert wird. Ein klassischer Fall von Feature-Neid (engl. *Feature Envy*) liegt vor, wenn eine Funktion in einem Modul den Funktionen oder Daten aus einem anderen Modul mehr Zeit widmet als denen im eigenen Modul. Wir können schon gar nicht mehr zählen, wie oft wir darauf gestoßen sind, dass eine Funktion ein halbes Dutzend Getter-Methoden eines anderen Objekts aufruft, um irgendeinen Wert zu berechnen. Erfreulicherweise gibt es hierfür eine offensichtliche Lösung: Die Funktion will eindeutig bei den Daten sein, verwenden Sie also *Funktion verschieben* (Abschnitt 8.1), um

Kapitel 3

Code-Smells: Schlechte Gerüche im Code

sie dorthin zu bewegen. Manchmal leidet nur ein Teil einer Funktion unter dem Phänomen des Neids. Verwenden Sie in diesem Fall zunächst *Funktion extrahieren* (Abschnitt 6.1) und anschließend *Funktion verschieben* (Abschnitt 8.1), um ihr ein passendes Zuhause zu bieten.

Natürlich sind nicht alle Fälle so eindeutig. Häufig verwendet eine Funktion Funktionalität mehrerer Module – wohin also soll man sie verschieben? Die Heuristik, die wir verwenden, beruht darauf, zu ermitteln, welches Modul die meisten der betreffenden Daten enthält, und die Funktion mit diesen Daten zu vereinen. Dieser Schritt wird oft vereinfacht, wenn Sie zunächst *Funktion extrahieren* (Abschnitt 6.1) verwenden, um die Funktion in mehrere Teile aufzugliedern, die dann an verschiedenen Stellen platziert werden.

Es gibt natürlich auch einige ausgeklügelte Muster, die gegen diese Regel verstossen. Von den Entwurfsmustern der »Gang of Four« (Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides: *Design Patterns: Entwurfsmuster als Elemente wiederverwendbarer objekt-orientierter Software*, mitp-Verlag 2015, ISBN 9783826697005) kommen mir spontan *Strategie* (Strategy) und *Besucher* (Visitor) in den Sinn. Kent Becks *Selbstdelegation* (Kent Beck: *SmallTalk Best Practice Patterns*, Prentice Hall 1996, ISBN 013476904X) ist ein weiteres. Verwenden Sie diese Patterns, um divergierende Änderungen zu verhindern. Die Faustregel lautet: Platzieren Sie Dinge, die sich gleichzeitig ändern, am selben Ort. Daten und das auf die Daten bezogene Verhalten ändern sich typischerweise gemeinsam – es gibt jedoch Ausnahmen. Wenn solche Ausnahmen auftreten, verschieben wir das Verhalten, um Änderungen nur auf eine Stelle zu beschränken. Mithilfe der Muster Strategie und Besucher lässt sich das Verhalten leicht ändern, weil sie das zu ändernde Verhalten isolieren – auf Kosten weiterer Indirektion.

3.10 Datenklumpen

Datenelemente sind tendenziell wie Kinder: Es macht ihnen Spaß, gemeinsam miteinander rumzuhängen. Ihnen werden an vielen Stellen die gleichen drei oder vier Datenelemente begegnen: als Felder in einigen Klassen oder als Parameter vieler Methoden. Datenbündel, die miteinander herumhängen, sollten wirklich ein gemeinsames Zuhause finden. Zunächst einmal müssen wir feststellen, wo diese Datenklumpen als Felder auftreten. Wenden Sie *Klasse extrahieren* (Abschnitt 7.5) auf die Felder an, um aus den Datenklumpen ein Objekt zu machen. Richten Sie Ihre Aufmerksamkeit anschließend auf Methodensignaturen und verwenden Sie *Parameterobjekt einführen* (Abschnitt 6.8) oder *Vollständiges Objekt erhalten* (Abschnitt 11.4), um sie zu verschlanken. Der unmittelbare Vorteil der gerade beschriebenen Schritte besteht darin, dass Sie viele Parameterlisten verkürzen und die Aufrufe der Methoden vereinfachen können. Machen Sie sich keine Gedanken über Datenklumpen, die nur einige der Felder des neuen Objekts verwenden. Wenn Sie mindestens zwei Felder durch das neue Objekt ersetzen können, sind Sie schon einen Schritt vorwärtsgekommen.

Ein guter Test ist es, darüber nachzudenken, einen der Datenwerte zu löschen. Wären die anderen dann noch sinnvoll? Wenn nicht, ist das ein sicheres Zeichen dafür, dass Sie ein Objekt vor sich haben, das es kaum erwarten kann, das Licht der Welt zu erblicken.

Sie werden bemerkt haben, dass wir hier für das Erstellen einer Klasse plädieren, nicht einer einfachen Datensatzstruktur. Das hat den Grund, dass eine Klasse Ihnen die Möglichkeit bietet, einen angenehmen Duft zu verbreiten (um im Bild zu bleiben). Sie können jetzt nach weiteren Fällen *Feature-Neid* (Abschnitt 3.9) suchen, bei denen es angebracht wäre,

das Verhalten in neue Klassen auszulagern. Wir haben schon häufig festgestellt, dass durch diese Vorgehensweise nützliche Klassen entstehen, die eine Menge Redundanzen überflüssig machen und die weitere Entwicklung beschleunigen, denn den Daten wird sozusagen ermöglicht, produktive Mitglieder der Gesellschaft zu werden.

3.11 Obsession für elementare Datentypen

Die meisten Programmierumgebungen beruhen auf einer Reihe gebräuchlicher elementarer Datentypen: Integer, Fließkommazahlen und Strings. Manche Bibliotheken fügen weitere kleine Objekte hinzu, wie etwa Kalenderdaten. Wir haben festgestellt, dass seltsamerweise viele Programmierer zögern, eigene grundlegende Datentypen zu erstellen, die für ihre Aufgabenbereiche nützlich wären – beispielsweise für Geldbeträge, für Koordinaten oder für Datumsbereiche. Uns begegnen also Berechnungen, die Geldbeträge wie einfache Zahlen handhaben, oder Berechnungen physikalischer Größen, die Einheiten ignorieren (Addieren von Millimetern zu Zentimetern), oder jede Menge Code, in dem `if (a < upper && a > lower)` vorkommt.

Strings sind eine besonders häufige Brutstätte für Code-Smells dieser Art: Eine Telefonnummer ist nicht nur eine schlichte Verkettung von Zeichen. Ein angemessener Typ kann zumindest für eine einheitliche Darstellung sorgen, wenn in der Benutzeroberfläche Werte angezeigt werden. Solche Werte in Form von Strings zu repräsentieren, ist ein Code-Smell, der so häufig auftritt, dass hier scherhaft von »string-typisierten« Variablen gesprochen wird.

Sie können die steinzeitliche Höhle der elementaren Typen verlassen und es sich in der zentralbeheizten Welt aussagekräftiger Typen gemütlich machen, indem Sie *Elementaren Wert durch Objekt ersetzen* (Abschnitt 7.3) verwenden. Falls es sich bei dem elementaren Wert um einen Typenschlüssel handelt, der bedingtes Verhalten steuert, können Sie *Typenschlüssel durch Unterklassen ersetzen* (Abschnitt 12.6) mit nachfolgendem *Bedingung durch Polymorphie ersetzen* (Abschnitt 10.4) verwenden.

Ansammlungen elementarer Werte, die häufig zusammen vorkommen, sind Datenkluppen und sollten mit *Klasse extrahieren* (Abschnitt 7.5) sowie *Parameterobjekt einführen* (Abschnitt 6.8) zivilisiert werden.

3.12 Wiederholte switch-Anweisungen

Wenn Sie sich mit überzeugten Verfechtern der objektorientierten Programmierung unterhalten, werden Ihre Gesprächspartner schon bald auf die mit der `switch`-Anweisung verbundenen Übel zu sprechen kommen. Sie argumentieren, dass eine `switch`-Anweisung geradezu nach *Bedingung durch Polymorphie ersetzen* (Abschnitt 10.4) schreit. Einige Leute sind sogar der Ansicht, dass sämtliche bedingte Programmlogik durch Polymorphie ersetzt werden sollte. Ein Großteil aller `if`-Anweisungen wäre dann Geschichte.

Selbst in unseren wildesten Zeiten haben wir Fallunterscheidungen oder bedingte Anweisungen nie uneingeschränkt abgelehnt. Allerdings enthielt die erste Ausgabe dieses Buchs einen Code-Smell namens »switch-Anweisungen«, weil wir Ende der 1990er-Jahre der Meinung waren, dass Polymorphie leider stark unterschätzt wurde, und es für richtig hielten, ihre Verwendung zu fördern.

Kapitel 3

Code-Smells: Schlechte Gerüche im Code

Heutzutage kommt Polymorphie häufiger zum Einsatz, und `switch`-Anweisungen sind nicht mehr das einfache Warnsignal, das sie vor fünfzehn Jahren oft waren. Darüber hinaus unterstützen viele Programmiersprachen ausgeklügelte `switch`-Anweisungen, die nicht nur elementare Typen zur Grundlage haben. Wir konzentrieren uns im Folgenden deshalb auf die wiederholte Verwendung der gleichen Steuerungslogik (entweder in einer `switch/case`-Anweisung oder in einer Abfolge von `if/else`-Anweisungen), die an mehreren Stellen auftaucht. Die redundante Verwendung solcher Fallunterscheidungen ist problematisch, weil Sie beim Hinzufügen eines weiteren Falls nach sämtlichen `switch`- bzw. `if`-Anweisungen suchen und sie aktualisieren müssen. Polymorphie bietet der Codebasis einen eleganten Schutz vor den dunklen Mächten solcher Redundanzen.

3.13 Schleifen

Seitdem es die ersten Programmiersprachen gab, sind Schleifen Kernbestandteil der Programmierung. Wir sind jedoch der Ansicht, dass sie heutzutage nicht bedeutsamer sind als Schlaghosen oder Samttapeten. Wir haben sie bei der ersten Ausgabe geringgeschätzt – allerdings bot Java damals, wie auch die meisten anderen Sprachen, keine bessere Alternative. Heute jedoch werden First-Class-Funktionen umfassend unterstützt, sodass wir *Schleife durch Pipeline ersetzen* (Abschnitt 8.8) verwenden können, um diese Anachronismen in den Ruhestand zu versetzen. Wir haben festgestellt, dass Pipeline-Operationen, wie etwa Filtern (`filter`) oder Zuordnen (`map`), dabei helfen können, schnell zu erkennen, welche Elemente verarbeitet werden und wie mit ihnen zu verfahren ist.

3.14 Träges Element

Wir verwenden zum Ergänzen von Struktur gern Programmelemente – sie bieten Möglichkeiten, Varianten zu erstellen, sie wiederzuverwenden oder einfach nur aussagekräftigere Bezeichnungen zu vergeben. In manchen Fällen ist diese Struktur jedoch überflüssig. Denken Sie beispielsweise an eine Funktion, deren Bezeichnung dem enthaltenen Code entspricht, oder an eine Klasse, die im Wesentlichen nur aus einer einfachen Funktion besteht. Manchmal handelt es sich dabei um eine Funktion, von der erwartet wurde, dass sie später wachsen und häufig verwendet werden würde, aber dazu kam es nicht. Es könnte sich auch um eine Klasse handeln, die ihre Aufgabe eigentlich gut erledigte, aber durch ein Refactoring verkleinert wurde. Wie dem auch sei, solche Programmelemente sollten in Würde sterben. Meistens bedeutet das, *Funktion inline platzieren* (Abschnitt 6.2) oder *Klasse inline platzieren* (Abschnitt 7.6) einzusetzen. Wenn Vererbung im Spiel ist, können Sie *Hierarchie abbauen* (Abschnitt 12.9) verwenden.

3.15 Spekulative Generalisierung

Die Bezeichnung »Spekulative Generalisierung« (engl. *Speculative Generality*) für einen Code-Smell, bei dem wir besonders hellhörig werden, wurde von Brian Foote vorgeschlagen. Sie tritt beispielsweise auf, wenn jemand sagt: »Früher oder später werden wir das wohl benötigen« und daraufhin alle möglichen Erweiterungen hinzugefügt und Spezialfälle berücksichtigt werden, die momentan tatsächlich überflüssig sind. Das führt häufig zu Code, der schwerer zu verstehen ist und sich schlechter warten lässt. Das Vorgehen könnte sinnvoll sein, wenn all diese Erweiterungen tatsächlich zum Einsatz kämen. Ist das aber

nicht der Fall, war die Arbeit umsonst. Sie werden durch all die ungenutzte Funktionalität nur behindert, also weg damit.

Falls Sie abstrakte Klassen verwenden, die nur wenige Aufgaben erledigen, sollten Sie *Hierarchie abbauen* (Abschnitt 12.9) verwenden. Unnötige Delegationen lassen sich durch *Funktion inline platzieren* (Abschnitt 6.2) oder *Klasse inline platzieren* (Abschnitt 7.6) entfernen. Funktionen mit ungenutzten Parametern können Sie mithilfe von *Funktionsdeklaration ändern* (Abschnitt 6.5) von den Parametern befreien. Dieses Refactoring sollten Sie ebenfalls anwenden, um nicht benötigte Parameter zu entfernen, die häufig aufgrund geplanter, aber später nie zustande kommender Erweiterungen angelegt wurden.

Spekulative Generalisierung können Sie daran erkennen, dass eine Funktion oder eine Klasse (wenn überhaupt) ausschließlich innerhalb von Testfällen verwendet wird. Falls Ihnen ein solches Getier über den Weg läuft, sollten Sie den Testfall löschen und *Toten Code entfernen* (Abschnitt 8.9) anwenden.

3.16 Temporäres Feld

Gelegentlich begegnen Sie einer Klasse, in der einem Feld nur unter bestimmten Umständen ein Wert zugewiesen wird. Code dieser Art ist schwer verständlich, weil Sie eigentlich erwarten, dass ein Objekt alle seine Felder benötigt. Es kann Sie in den Wahnsinn treiben, wenn Sie versuchen zu verstehen, warum ein Feld vorhanden ist, das offenbar nicht verwendet wird.

Verwenden Sie *Klasse extrahieren* (Abschnitt 7.5), um den verwaisten Variablen ein neues Zuhause zu geben, und *Funktion verschieben* (Abschnitt 8.1), um sämtlichen Code, der das Feld betrifft, in die neue Klasse zu verlagern. Möglicherweise können Sie auch bedingten Code entfernen, indem Sie *Sonderfall einführen* (Abschnitt 10.5) verwenden, um für den Fall, dass die Variablen ungültig sind, eine alternative Klasse zu erstellen.

3.17 Mitteilungsketten

Mitteilungsketten treten auf, wenn ein Client ein Objekt nach einem anderen Objekt fragt, das der Client wiederum nach einem weiteren Objekt fragt, das der Client um noch ein weiteres Objekt fragt und so weiter. Dieser Vorgang könnte als eine Aneinanderreihung von *getThis*-Methoden oder als Sequenz temporärer Variablen in Erscheinung treten. Diese Art der Navigation bedeutet, dass der Client an die Struktur der Navigation gekoppelt ist. Jede Änderung an den Beziehungen der beteiligten Objekte untereinander erfordert es, den Client entsprechend anzupassen.

Hier sollte *Delegation verborgen* (Abschnitt 7.7) zum Einsatz kommen. Sie können das an mehreren Stellen der Mitteilungskette durchführen, prinzipiell sogar für alle zur Kette gehörigen Objekte, allerdings können die beteiligten Objekte dadurch zu Vermittlern werden. Zu überprüfen, wofür das resultierende Objekt verwendet wird, ist vielfach eine bessere Alternative. Prüfen Sie, ob Sie *Funktion extrahieren* (Abschnitt 6.1) anwenden können, um einen Teil des Codes, der das resultierende Objekt verwendet, zu extrahieren, und anschließend *Funktion verschieben* (Abschnitt 8.1), um ihn in der Kette nach unten zu verschieben.

Manche halten die Verkettung von Methoden für Teufelswerk. Wir hingegen sind für unsere differenzierte Betrachtung der Dinge bekannt. Zumindest in diesem Fall.

Kapitel 3

Code-Smells: Schlechte Gerüche im Code

3.18 Vermittler

Zu den wichtigsten Merkmalen eines Objekts gehört die Kapselung – das Verbergen interner Details vor dem Rest der Welt. Kapselung und Delegation gehen oft Hand in Hand. Sie fragen Ihre Chefin, ob sie Zeit für ein Treffen hat; sie leitet Ihre Mitteilung an ihren Kalender weiter (»sie delegiert sie«) und gibt Ihnen eine Antwort. Alles schön und gut. Es ist nicht notwendig, zu wissen, ob sie einen Kalender einsetzt, einen digitalen Assistenten verwendet oder ob ihre Sekretärin die Termine im Auge behält.

Delegation kann aber auch zu weit gehen. Sie betrachten die Schnittstelle einer Klasse und stellen fest, dass die Hälfte der Methodenaufrufe an eine andere Klasse delegiert wird. Früher oder später ist es an der Zeit, *Vermittler entfernen* (Abschnitt 7.8) anzuwenden und sich direkt an das Objekt zu wenden, dem wirklich bekannt ist, was ausgeführt werden soll. Wenn nur ein paar Methoden vorhanden sind, die zudem wenige Aufgaben erledigen, sollten Sie *Funktion inline platzieren* (Abschnitt 6.2) anwenden, um sie in den Aufrufer zu verschieben. Wenn zusätzliches Verhalten vorhanden ist, können Sie *Basisklasse durch Delegation ersetzen* (Abschnitt 12.11) oder *Unterklassse durch Delegation ersetzen* (Abschnitt 12.10) verwenden, um den Vermittler zum Bestandteil des wirklich wichtigen Objekts zu machen. Auf diese Weise können Sie das Verhalten erweitern, ohne sämtliche Delegationen nachzuverfolgen zu müssen.

3.19 Insiderhandel

Softwareentwickler errichten gern hohe Mauern, um ihre Module voneinander zu trennen, und beklagen sich bitterlich, dass der übertriebene Austausch von Daten die Kopplung zu sehr verstärkt. Nun ist ein gewisser Austausch der Daten zwar unverzichtbar, wir sollten ihn jedoch auf ein Minimum beschränken und dabei sorgfältig vorgehen.

Module, die hinter unserem Rücken miteinander tuscheln, sollten durch *Funktion verschieben* (Abschnitt 8.1) und *Feld verschieben* (Abschnitt 8.2) voneinander getrennt werden, um die notwendige Kommunikation zu verringern. Wenn Module gemeinsame Interessen haben, sollten Sie versuchen, ein drittes Modul zu erstellen, um diese Gemeinsamkeiten zu vereinen, oder *Delegation verbergen* (Abschnitt 7.7) verwenden, um ein weiteres Modul zu erstellen, das als Vermittler fungiert.

Die Vererbung führt oft zu heimlichen Absprachen, und die Unterklassen wissen oft mehr über ihre Basisklassen, als Ihnen lieb ist. Wenn die Zeit gekommen ist, sie voneinander zu trennen, können Sie *Unterklassse durch Delegation ersetzen* (Abschnitt 12.7) oder *Basisklasse durch Delegation ersetzen* (Abschnitt 12.11) verwenden.

3.20 Umfangreiche Klasse

Wenn eine Klasse versucht, zu viele Aufgaben selbst zu erledigen, ist das oft daran zu erkennen, dass sie zu viele Felder besitzt. Und wenn das der Fall ist, geht damit meist auch redundanter Code (Abschnitt 3.2) einher.

Verwenden Sie *Klasse extrahieren* (Abschnitt 7.5), um mehrere Variablen zu bündeln. Identifizieren Sie Variablen, die sinnvollerweise zusammengehören. So werden beispielsweise die beiden Variablen `depositAmount` und `depositCurrency` für einen Geldbetrag und dessen Währung höchstwahrscheinlich in eine gemeinsame Komponente gehören. Im All-

gemeinen weisen gemeinsame Präfixe oder Suffixe der Bezeichnungen bei mehreren Variablen einer Klasse auf die Möglichkeit für eine eigene Komponente hin. Wenn für diese Komponente Vererbung angebracht ist, ist es oft einfacher, *Basisklasse extrahieren* (Abschnitt 12.8) oder *Typenschlüssel durch Unterklassen ersetzen* (Abschnitt 12.6) zu verwenden. (Letzteres ist im Wesentlichen die Extraktion einer Unterklasse.)

In manchen Fällen verwendet eine Klasse gar nicht durchgängig alle ihre Felder. In diesem Fall können Sie womöglich mehrere Extraktionen vornehmen.

Ebenso wie eine Klasse mit zu vielen Instanzvariablen ist auch eine Klasse, die zu viel Code enthält, eine Brutstätte für redundante Code, Chaos und Leid. Die einfachste Lösung (hatten wir schon erwähnt, dass wir einfache Lösungen mögen?) besteht darin, Redundanzen in der Klasse selbst zu beseitigen. Wenn es fünf hundertzeilige Methoden mit jeweils einigen redundanten Code-Abschnitten gibt, können Sie aus dem ursprünglichen Code vielleicht fünf zehnzeilige Methoden und zehn weitere zweizeilige Methoden machen.

Häufig bieten die Clients einer solchen Klasse die besten Hinweise für das Aufteilen der Klasse. Prüfen Sie, ob die Clients nur eine Teilmenge der Funktionalität der Klasse nutzen. Jede dieser Teilmengen ist potenziell für eine separate Klasse geeignet. Wenn Sie eine solche Teilmenge identifiziert haben, können Sie *Klasse extrahieren* (Abschnitt 7.5), *Basisklasse extrahieren* (Abschnitt 12.8) oder *Typenschlüssel durch Unterklassen ersetzen* (Abschnitt 12.6) für die Zerlegung verwenden.

3.21 Alternative Klassen mit unterschiedlichen Schnittstellen

Die Verwendung von Klassen bietet den großen Vorteil der Substituierbarkeit, die es bei Bedarf ermöglicht, eine Klasse durch eine andere auszutauschen. Das funktioniert allerdings nur, wenn die Schnittstellen übereinstimmen. Verwenden Sie *Funktionsdeklaration ändern* (Abschnitt 6.5), um Funktionen einander anzulegen. In vielen Fällen ist das nicht ausreichend; wenden Sie zum Verschieben des Verhaltens in Klassen gegebenenfalls so lange *Funktion verschieben* (Abschnitt 8.1) an, bis die Schnittstellen übereinstimmen. Sollte dies zu Redundanzen führen, können Sie zu deren Beseitigung das Refactoring *Basisklasse extrahieren* (Abschnitt 12.8) einsetzen.

3.22 Datenklasse

Hierbei handelt es sich um Klassen mit Feldern, die außerdem Getter- und Setter-Methoden für die Felder besitzen – und sonst nichts. Solche Klassen sind schlichte Datencontainer und werden oft von anderen Klassen in viel zu großem Ausmaß verändert. Unter Umständen besitzen diese Klassen öffentlich zugängliche Felder. Wenn das der Fall ist, sollten Sie umgehend *Datensatz kapseln* (Abschnitt 7.1) anwenden, bevor das irgendjemand bemerkt. Wenden Sie *Setter entfernen* (Abschnitt 11.7) auf jedes Feld an, das nicht geändert werden soll.

Halten Sie danach Ausschau, an welchen Stellen diese Getter- und Setter-Methoden von anderen Klassen verwendet werden. Versuchen Sie, das Verhalten mit *Funktion verschieben* (Abschnitt 8.1) in die Datenklasse zu verlagern. Wenn es nicht möglich ist, die gesamte Funktion zu verschieben, können Sie mit *Funktion extrahieren* (Abschnitt 6.1) eine Funktion erstellen, die sich verschieben lässt.

Kapitel 3

Code-Smells: Schlechte Gerüche im Code

Datenklassen sind oft ein Hinweis darauf, dass sich Verhalten an einer falschen Stelle befindet. Sie können eine große Verbesserung erzielen, indem Sie das Verhalten vom Client in die Datenklasse selbst verlagern. Hier gibt es jedoch Ausnahmen. Das beste Beispiel ist ein Datensatz, der von einem gesonderten Funktionsaufruf als Ergebnisdatensatz verwendet wird, wie etwa die temporäre Datenstruktur nach der Anwendung von *Phase aufteilen* (Abschnitt 6.11). Für einen solchen Ergebnisdatensatz ist charakteristisch, dass er unveränderlich ist (zumindest in der Praxis). Unveränderliche Felder müssen nicht gekapselt werden, und die aus unveränderlichen Daten abgeleiteten Informationen können als Felder anstatt als Getter-Methoden repräsentiert werden.

3.23 Ausgeschlagenes Erbe

Unterklassen erben die Methoden und Daten ihrer Eltern. Aber was ist, wenn sie dieses Erbe überhaupt nicht wollen oder benötigen? Sie erhalten ein großes Vermächtnis und wählen doch nur einige wenige Geschenke aus.

Das bedeutet klassischerweise, dass die Hierarchie falsch aufgebaut ist. Sie müssen eine neue Geschwisterklasse erstellen und *Methode nach unten verschieben* (Abschnitt 12.4) und *Feld nach unten verschieben* (Abschnitt 12.5) verwenden, um den überflüssigen Code in die Geschwisterklasse zu verlagern. Auf diese Weise enthält die Basisklasse nur noch die Gemeinsamkeiten. Hier wird oft der Rat gegeben, dass alle Basisklassen abstrakt sein sollten.

Aufgrund des despektierlichen Gebrauchs von »klassischerweise« ahnen Sie vielleicht schon, dass wir dem nicht zustimmen – zumindest nicht immer. Wir verwenden ständig Unterklassen, um ein bestimmtes Verhalten wiederzuverwenden, und betrachten das als völlig unproblematisch. Es liegt etwas in der Luft – das können wir nicht abstreiten – aber dieser Code-Smell ist für gewöhnlich erträglich. Wenn das ausgeschlagene Erbe Verwirrung oder Probleme verursacht, können Sie dem klassischen Rat folgen. Sie sollten jedoch nicht den Eindruck bekommen, dass dies in jedem Fall erforderlich ist. In neun von zehn Fällen ist dieser Code-Smell so harmlos, dass es nicht lohnt, ihn zu beseitigen.

Allerdings ist der Code-Smell des ausgeschlagenen Erbes nicht mehr so harmlos, wenn die Unterklasse Verhalten wiederverwendet, aber die Schnittstelle der Basisklasse nicht mehr unterstützen möchte. Die Implementierung ist uns egal, aber wenn die Schnittstelle nicht unterstützt wird, dann sitzen wir auf dem sprichwörtlichen hohen Ross fest im Sattel. In diesem Fall sollten Sie nicht an der Hierarchie herumtüfteln; verwenden Sie stattdessen *Unterklasse durch Delegation ersetzen* (Abschnitt 12.10) oder *Basisklasse durch Delegation ersetzen* (Abschnitt 12.11).

3.24 Kommentare

Keine Sorge, wir behaupten nicht, dass keine Kommentare verwendet werden sollten. Gemäß unserer olfaktorischen Analogie sind Kommentare kein unangenehmer Geruch, sondern vielmehr ein süßer Duft. Dass wir hier Kommentare erwähnen, hat folgenden Grund: Sie werden oft als Deodorant missbraucht. Es ist wirklich erstaunlich, wie oft man auf ausgiebig kommentierten Code stößt und feststellen muss, dass die Kommentare vorhanden sind, weil der Code zu wünschen übrig lässt.

Kommentare führen uns zu mangelhaftem Code, der sämtliche üblichen Geruchsnuancen in sich trägt, die wir in diesem Kapitel erläutert haben. Als Erstes entfernen wir die Code-Smells durch ein Refactoring. Wenn das erledigt ist, stellen wir häufig fest, dass die Kommentare überflüssig sind.

Wenn Sie einen Kommentar für erforderlich halten, um die Aufgabe eines Codeblocks zu erläutern, sollten Sie *Funktion extrahieren* (Abschnitt 6.1) ausprobieren. Wenn Sie die Methode bereits extrahiert haben und immer noch denken, dass Sie einen Kommentar benötigen, sollten Sie *Funktionsdeklaration ändern* (Abschnitt 6.5) verwenden, um sie umzubenennen. Wenn Sie einige Hinweise darlegen möchten, die den erforderlichen Zustand des Systems betreffen, können Sie dafür *Assertion einführen* (Abschnitt 10.6) nutzen.

Tipp

Wenn Sie das Bedürfnis haben, einen Kommentar zu verfassen, versuchen Sie zunächst, ein Refactoring vorzunehmen, sodass jeglicher Kommentar überflüssig wird.

Ein guter Moment, einen Kommentar zu verwenden, ist, wenn Sie nicht mehr wissen, was Sie tun sollen. Kommentare beschreiben nicht nur, was vor sich geht, sie können auch Hinweise auf Bereiche geben, in denen Sie noch unsicher sind. Ein Kommentar kann zudem darüber Aufschluss geben, warum Sie etwas getan haben. Diese Art von Information ist für zukünftige Bearbeiter hilfreich, vor allem, wenn sie vergesslich sind.