Kapitel 1
Refactorings für Refactoring

In diesem Kapitel

- Die Elemente des Refactorings verstehen
- Refactoring in den Arbeitsalltag aufnehmen
- Die Wichtigkeit von Sicherheit für Refactoring
- Vorstellung des durchgehenden Beispiels für Teil I

Es ist allgemein bekannt, dass höhere Codequalität zu niedrigeren Wartungskosten führt, zu weniger Fehlern und zu glücklicheren Programmierenden. Der einfachste Weg zu hoher Codequalität führt über Refactoring. Leider baut Refactoring so, wie es üblicherweise gelehrt wird – mit Code Smells und Unit Tests –, eine hohe Einstiegshürde auf. Doch ich glaube, dass jede und jeder mit etwas Übung einfache Refactorings ausführen kann.

In der Softwareentwicklung verorten wir Probleme, wie sie im Diagramm aus Abbildung 1.1 dargestellt sind. Sie entstammen einem Mangel an Fähigkeiten, an Kultur, an Werkzeugen oder aus einer Kombination daraus. Refactoring ist ein anspruchsvolles Verfahren und liegt deshalb in der Mitte des Diagramms. Es bedarf aller drei Komponenten.

- *Fähigkeiten* – Wir brauchen ein hohes Maß an Kompetenz, um zu erkennen, wann Code schlecht ist und ein Refactoring braucht. Versierte Programmierende erkennen dies dank ihrer Erfahrung mit Code Smells – übel riechendem Code. Aber Code Smells sind nicht scharf umrissen, sie zu erkennen braucht Urteilsvermögen und Erfahrung, sie sind Ermessenssache und schon dadurch nicht einfach zu erlernen. Auf unerfahrene Programmierende kann die Fähigkeit, Code Smells zu erkennen, wie ein siebter Sinn wirken.
- *Kultur* – Wir brauchen eine Arbeitskultur und einen Arbeitsablauf, die uns die Zeit und den Freiraum für Refactorings geben. Häufig wird diese Kultur durch die berühmte Rot-Grün-Refactoring-Schleife der testgetriebenen Entwicklung verwirklicht: Für eine neue Funktionalität wird zunächst ein Test geschrieben, der zu diesem Zeitpunkt natürlich noch fehlschlägt (Rot). Dann wird die Funktionalität implementiert, bis der Testfall erfolgreich durchläuft (Grün). Zuletzt werden Code Smells beseitigt, die man mit der Implementierung eingeführt hat (Refactoring). Allerdings ist die testgetriebene Entwicklung eine sehr fortgeschrittene Technik. Von Rot-Grün-Refactoring findet man außerdem nicht leicht den Übergang zum Refactoring an einem Legacy-System, zu dem meist keine Tests existieren.

- *Werkzeuge* – Wir brauchen etwas, das sicherstellt, dass unsere Änderungen keine Fehler verursachen. Automatisierte Tests sind hierfür das verbreitetste Werkzeug. Allerdings ist, wie oben bereits angesprochen, das Schreiben effektiver Testfälle auch nicht einfach.

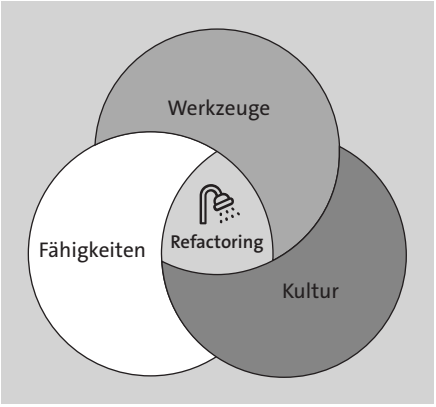

Abbildung 1.1 Fähigkeiten, Kultur und Werkzeuge

Die folgenden Abschnitte gehen ins Detail dieser drei Gebiete und beschreiben, wie wir das Abenteuer Refactoring von einfachem Grundwissen ausgehend starten können – ohne Tests und Code Smells.

1.1 Was ist Refactoring?

Die Frage, was Refactoring ist, werde ich im nächsten Kapitel tiefergehend beantworten. Zunächst aber brauchen wir ein grundlegendes Verständnis dafür, was Refactoring ist, bevor wir in die verschiedenen Arten eintauchen können, *wie* man sie einsetzt. In seiner einfachsten Form bedeutet Refactoring »den Code verändern, ohne zu verändern, was er tut«. Fangen wir mit einem Beispiel an, um zu erklären, wovon ich rede. Hier ersetzen wir einen Ausdruck durch eine lokale Variable.

Listing 1.1 Vorher

```
01  return pow(base, exp / 2)
02         * pow(base, exp / 2);
```

Listing 1.2 Nachher

```
01  let result = pow(base, exp / 2);
02  return result * result;
```

Es gibt viele Gründe für ein Refactoring:

- Den Code schneller machen (wie im Beispiel)
- Den Code kürzer machen
- Den Code verallgemeinern und wiederverwendbarer machen
- Den Code lesbarer oder wartbarer machen

Gerade der letzte Grund ist so zentral, dass wir ihn mit »gutem Code« gleichsetzen.

Definition: Guter Code

Guter Code ist für Menschen leicht zu lesen und zu warten, und er löst das Problem korrekt, zu dessen Lösung er geschrieben wurde.

Da Refactoring niemals verändern darf, was der Code tut, konzentrieren wir uns in diesem Buch auf die Eigenschaft, von Menschen lesbar zu sein und die Wartbarkeit. Wir werden in Kapitel 2, »Ein Blick unter die Haube«, mehr über die Gründe für Refactoring sprechen. Hier befassen wir uns nur mit Refactoring, das zu *gutem Code* führt, deshalb ist unsere Definition wie folgt:

Definition: Refactoring

Refactoring bedeutet, Code zu ändern, um ihn menschenlesbarer und wartbarer zu machen, ohne zu verändern, was der Code tut.

Ich sollte auch erwähnen, dass die Arten von Refactoring, mit denen wir uns befassen werden, explizit für objektorientierte Programmiersprachen gedacht sind.

Im Allgemeinen denkt man, Programmieren sei das Schreiben von Code. Doch das ist so nicht korrekt. Die meisten Programmierenden verbringen mehr Zeit mit dem Lesen von Code und dem Versuch, ihn zu verstehen, als mit dem Schreiben. Wir arbeiten in einem komplexen Umfeld, und etwas zu verändern, ohne es zu verstehen, kann zu katastrophalen Fehlern führen.

Das erste Argument für Refactoring ist deshalb ein rein wirtschaftliches: Die Zeit der Programmierenden ist mit Kosten verbunden. Je lesbarer die Codebasis, desto mehr Zeit hat die Programmiererin oder der Programmierer dafür, neue Features zu entwickeln. Zweitens: Wartbarer Code enthält weniger Fehler, die darüber hinaus einfacher zu beheben sind. Und drittens: Es macht mehr Spaß, mit einer guten Codebasis zu arbeiten. Wenn wir Code lesen, erstellen wir ein mentales Modell davon, was der Code bewirkt. Je mehr es dabei zu bedenken gibt, desto anstrengender wird die Sache. Das ist der Grund, warum es mehr Spaß macht, ein neues Projekt zu beginnen – und warum Fehlersuche grausam sein kann.

1.2 Fähigkeiten: Was sollte ich refactorn?

Wissen, wo ein Refactoring angebracht ist, ist die erste Eintrittshürde. Üblicherweise wird Refactoring zusammen mit dem Konzept der *Code Smells* gelehrt. Diese »Gerüche« sind Beschreibungen von Symptomen, die darauf hindeuten, dass unser Code schlecht ist. Zweifellos sind sie ein mächtiges Konzept, aber auch sehr abstrakt und wenig einsteigerfreundlich. Es braucht Zeit, ein Gefühl für sie zu entwickeln.

Dieses Buch geht einen anderen Weg. Es zeigt einfach zu erkennende und anzuwendende Regeln, wo ein Refactoring ratsam ist. Diese Regeln sind leicht und schnell zu erlernen. Manchmal sind sie jedoch auch zu streng und zwingen dich, Code zu ändern, der gar nicht stinkt. In seltenen Fällen befolgen wir die Regeln, und der Code stinkt noch immer.

Wie Abbildung 1.2 zeigt, sind Gerüche und Regeln nicht deckungsgleich. Meine Regeln sind nicht der Weisheit letzter Schluss, was guten Code angeht. Aber sie sind eine Starthilfe auf dem Weg zum untrüglichen Gefühl für guten Code. Schauen wir uns ein Beispiel an, das den Unterschied zwischen Code Smells und den Regeln dieses Buches illustriert.

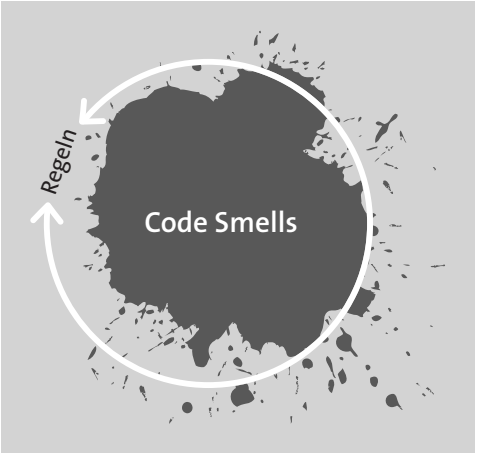

Abbildung 1.2 Regeln und Code Smells

1.2.1 Ein Beispiel für Code Smell

Ein bekanntes Beispiel für einen Code Smell lautet, dass eine Funktion genau eine Sache tun soll. Das ist eine großartige Richtlinie, aber es ist nicht offensichtlich, was diese eine Sache ist. Betrachte noch einmal den Code aus dem vorherigen Beispiel: Riecht er schlecht? Immerhin tut er drei Dinge: Er dividiert, multipliziert und potenziert. Sind das drei Dinge? Andererseits gibt er nur eine Zahl zurück und hat keine Seiteneffekte. Ist es also doch nur ein Ding?

```
let result = pow(base, exp / 2);
return result * result;
```

1.2.2 Ein Beispiel für eine Regel

Vergleiche den Code mit der folgenden Regel (auf die wir in Kapitel 3, »Lange Funktionen zerschlagen«, im Detail eingehen werden): Eine Methode sollte nie mehr als *fünf Zeilen Code* haben. Das können wir auf einen Blick prüfen, ohne offene Fragen. Die Regel ist einfach, eingängig, und leicht zu merken – erst recht, wenn ihr den Titel dieses Buches kennt.

Die Regeln, die ich in diesem Buch vorstelle, sind eine Art Stützräder. Sie können nicht in jeder Situation guten Code garantieren, und in manchen Situationen kann es sogar falsch sein, sie zu befolgen. Dennoch sind sie sehr nützlich, wenn man nicht weiß, wo man beginnen soll, und führen meistens zu gutem Refactoring.

Die Namen aller Regeln sind absolute Aussagen, sie enthalten Worte wie *immer* oder *nie*. Dadurch sind sie leicht zu merken. Die ausführliche Beschreibung einer Regel enthält aber häufig Ausnahmen, die besagen, wann diese Regel *nicht* angewendet werden sollte. Und sie umfasst die Absicht hinter einer Regel. Wenn du anfängst, Refactoring zu erlernen, folgst du den Namen der Regeln. Hast du diese verinnerlicht, lernst du die Ausnahmen und danach die Absicht hinter der Regel – bis du auf diese Weise selbst zum Code-Guru wirst.

1.3 Kultur: Wann sollte ich refactorn?

Refactorn ist wie duschen.

– Kent Beck

Refactoring funktioniert am besten – und kostet am wenigsten –, wenn es regelmäßig getan wird. Ich empfehle dir, es wenn möglich in deine tägliche Arbeit aufzunehmen. Die meiste Fachliteratur rät zur Rot-Grün-Refactoring-Schleife, wie oben beschrieben. Wie erwähnt, bindet dieser Ablauf aber das Refactoring an testgetriebene Entwicklung – in diesem Buch wollen wir diese Dinge voneinander trennen und uns allein auf das Refactoring konzentrieren. Dazu empfehle ich einen sechsteiligen Workflow, gezeigt in Abbildung 1.3, um jede Programmieraufgabe zu lösen:

1. *Erkunden*. Oft sind wir zu Anfang noch gar nicht sicher, was wir genau entwickeln müssen. Manchmal weiß der Kunde noch nicht, was genau er will. Manchmal sind die Anforderungen mehrdeutig. Manchmal wissen wir nicht einmal, ob die Aufgabe lösbar ist. Fang deshalb immer mit Ausprobieren an. Implementiere schnell etwas, und klär dann mit dem Kunden, ob ihr dasselbe Verständnis von der Aufgabe habt.
2. *Spezifizieren*. Wenn du weißt, was du entwickeln sollst, halte es fest. Idealerweise ist die Spezifikation ein automatisierter Test.
3. *Implementieren*. Schreibe den Code.
4. *Testen*. Stelle sicher, dass der Code die Spezifikation aus Schritt 2 erfüllt.
5. *Refactoring*. Bevor du den Code ablieferst, stelle sicher, dass er für den Nächsten leicht zu bearbeiten ist (der Nächste könntest du selbst einige Zeit später sein – mach dir das Leben leichter).
6. *Liefern*. Es gibt viele Arten, den fertigen Code zu liefern, zum Beispiel durch einen Pull Request oder durch Pushen in einen bestimmten Branch. Das Wichtige ist, dass der Code zu seinen Nutzenden gelangt. Warum sonst der ganze Aufwand?

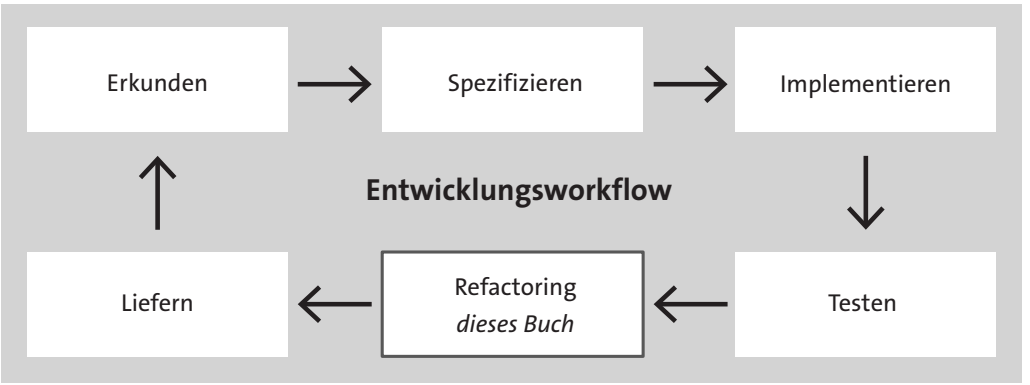

Abbildung 1.3 Workflow

Weil wir *regelbasiertes* Refactoring betreiben, ist es einfach, diesem Workflow zu folgen. Abbildung 1.4 zeigt Schritt 5, Refactoring, im Detail.

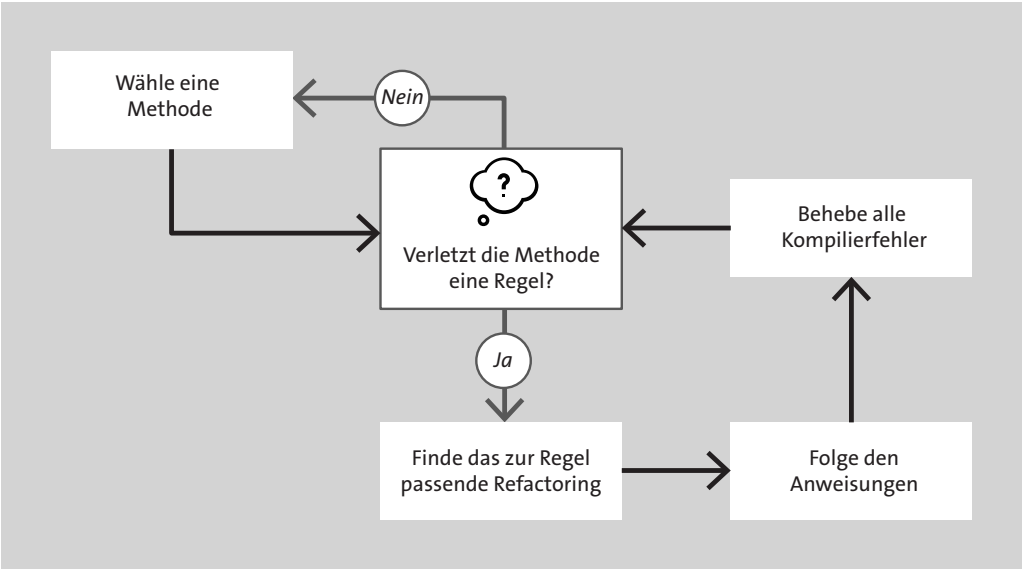

Abbildung 1.4 Detailansicht des Schritts »Refactoring«

Ich habe die Regeln so formuliert, dass sie hoffentlich leicht zu merken sind und die Gelegenheiten, sie anzuwenden, hoffentlich leicht und ohne Hilfe auffindbar: Eine Methode zu finden, die gegen die Regeln verstößt, ist meist trivial. Jeder Regel sind einige Refactorings zugeordnet, sodass leicht sichtbar ist, wie ein Problem gelöst werden kann. Die Refactorings erklären Schritt für Schritt, was zu tun ist, um sicherzustellen, dass du nichts kaputt machst. Viele der Refactorings in diesem Buch verwenden Kompilierfehler, um dir zu helfen, keine

Fehler in die Codebasis einzubauen. Mit ein wenig Übung wirst du Regeln und Refactorings ganz automatisch anwenden.

1.3.1 Refactoring in einem Legacy-System

Sogar wenn wir an einem großen Legacy-System arbeiten, gibt es eine clevere Methode, Refactoring in den Arbeitsalltag zu integrieren, ohne erst alles zu unterbrechen und die ganze Codebasis zu refactorn. Folge einfach diesem großartigen Zitat:

> *Mache zunächst Veränderung einfach, dann mache die einfachen Veränderungen.*
>
> *– Kent Beck*

Wir gehen vor wie beim Backen, wo wir auch zunächst alle Zutaten bereitstellen: Bevor wir anfangen, etwas Neues zu implementieren, führen wir immer zunächst Refactorings durch, bis unser neuer Code leicht zu schreiben ist.

1.3.2 Wann sollte ich nicht refactorn?

Im Großen und Ganzen ist Refactoring toll, aber es hat auch einige Nachteile. Es kann viel Zeit kosten, vor allem, wenn man es nicht regelmäßig macht. Und wie oben schon erwähnt, ist Zeit beim Programmieren grundsätzlich kostbar.

Es gibt drei Arten von Codebasis, in denen sich Refactoring wahrscheinlich nicht lohnt:

- Code, den du schreibst, um ihn genau einmal auszuführen und dann wieder zu löschen. Im Extreme Programming nennt man das eine *Spitze (Spike)*.
- Code, der nur noch gewartet werden muss, bis er außer Betrieb genommen wird.
- Code mit hohen Performanz-Anforderungen, beispielsweise Code für ein eingebettetes System oder die Physik-Engine eines Spiels.

In allen anderen Fällen ist meiner Meinung nach Refactoring eine schlaue Investition.

1.4 Werkzeuge: Wie sollte ich (sicher) refactorn?

Wie alle anderen Programmierenden auch bin ich ein großer Fan von automatisierten Tests. Aber gute Tests zu schreiben will gelernt sein. Wenn du schon weißt, wie man automatisierte Tests schreibt, dann kannst du dieses Buch benutzen, um nebenbei auch das zu trainieren. Falls nicht, auch kein Problem.

Automatisierte Tests sind für die Softwareentwicklung wie die Bremsen fürs Auto. Autos haben keine Bremsen, weil wir unbedingt langsam fahren wollen. Sondern sie haben Bremsen, weil es sonst gefährlich wäre, schnell zu fahren. Das gilt auch für Tests: Mit automatisierten Tests können wir uns sicher fühlen, wenn wir schnell arbeiten. In diesem Buch lernen wir neue Fähigkeiten, mit denen wir aber gar nicht schnell sein müssen.

Ich schlage vor, sich mehr auf andere Werkzeuge zu verlassen als auf Tests:

- Schritt-für-Schritt-Beschreibungen für Refactorings, ähnlich wie Kochrezepte
- die Versionskontrolle
- den Compiler

Ich glaube, dass gewissenhaft konstruierte Refactorings, in kleinen Schritten ausgeführt, es möglich machen, Refactorings auszuführen, ohne dabei Fehler einzubauen. Das gilt besonders, wenn die Entwicklungsumgebung ein Refactoring automatisch durchführen kann.

Da wir in diesem Buch nicht testen wollen, werden wir uns mehr auf den Compiler und das Typensystem der Programmiersprache unserer Wahl verlassen, um häufig gemachte Fehler gleich zu finden. Trotzdem empfehle ich, das Programm, an dem wir arbeiten, regelmäßig auszuführen. So können wir sicherstellen, dass es nicht komplett kaputt ist. Sobald das gesichert ist, machen wir einen Commit in unserer Versionskontrolle. Bauen wir später Fehler ein, die wir nicht sofort lösen können, können wir zu diesem Punkt zurückspringen.

Wenn wir an einem realen System ohne automatisierte Tests arbeiten, können wir trotzdem Refactorings vornehmen. Das Vertrauen in das, was wir tun, muss dann aus einer anderen Quelle kommen. Das kann die Entwicklungsumgebung sein, die Refactorings automatisch durchführen kann, manuelle Tests von sehr kleinschrittigem Vorgehen oder anderes. Aufgrund der zusätzlichen Zeit, die wir dafür brauchen, wird es aber vermutlich doch kosteneffektiver sein, Tests zu schreiben.

1.5 Werkzeuge für den Anfang

Wie schon gesagt, sind die Refactorings, über die wir in diesem Buch sprechen, für den Einsatz in einer objektorientierten Programmiersprache gedacht. Das ist das Hauptwerkzeug, das du brauchst, um dieses Buch zu lesen und zu verstehen. Programmieren und Refactoring sind Fähigkeiten, die wir mit den Händen ausführen. Sie sind deshalb auch am besten mit den Händen zu erlernen – indem du die Beispielen nachvollziehst, experimentierst und Spaß hast, während deine Hände die Bewegungen lernen. Um dem Buch zu folgen, brauchst du die Werkzeuge, die ich nun beschreibe. Wie du sie installierst, liest du im Anhang.

1.5.1 Programmiersprache: TypeScript

Alle Codebeispiele in diesem Buch sind in TypeScript verfasst. Diese Wahl hatte mehrere Gründe. Allen voran: TypeScript hat große Ähnlichkeit mit einigen der verbreitetsten Programmiersprachen – Java, C#, C++ und JavaScript – und alle, die eine dieser Sprachen kennen, sollten den Code problemlos verstehen. TypeScript erlaubt es außerdem, Code von ganz und gar nicht objektorientiert – also ohne einzige Klasse – bis hin zu hochgradig objektorientiert zu schreiben.

Formatierung

Um den Platz auf der gedruckten Seite besser auszunutzen, verwenden die Codebeispiele eine Formatierung, die auf Zeilenumbrüche weitgehend verzichtet, aber trotzdem lesbar bleibt. Das bedeutet nicht, dass du deinen Code auch so formatieren solltest – es sei denn, du schreibst ein Buch mit vielen Beispielen in TypeScript. Das ist auch der Grund, warum Klammern und Einrückungen in diesem Buch manchmal anders sind als in den Projektdateien.

Falls sich jemand in TypeScript nicht auskennt, erkläre ich unerwartetes Verhalten in Kästen wie dem folgenden.

In TypeScript ...

... verwenden wir Identität (`===`), um auf Gleichheit zu prüfen, weil das Verhalten näher an dem ist, was wir von Gleichheit erwarten, als das doppelte Gleichheitszeichen (`==`). So zum Beispiel:

- `0 == ""` ist `true`.
- `0 === ""` ist `false`.

Obwohl alle Beispiele in TypeScript geschrieben sind, gelten die Regeln und Refactorings gleichermaßen für andere objektorientierte Programmiersprachen. In seltenen Fällen hilft uns TypeScript besonders weiter – oder kommt uns in die Quere. Diese Fälle werden jeweils erwähnt und es wird besprochen, wie man in anderen Sprachen mit der Situation umgehen kann.

1.5.2 Editor: Visual Studio Code

Ich gehe für dieses Buch nicht davon aus, dass du eine bestimmte Entwicklungsumgebung benutzt, aber falls du keine spezielle Vorliebe hast, empfehle ich Visual Studio Code. Es funktioniert gut mit TypeScript, und es lässt uns `tsc -w` im Hintergrund ausführen, um unseren Code zu kompilieren, auch wenn wir es vergessen.

Wichtig

Visual Studio Code hat nichts mit Visual Studio zu tun!

1.5.3 Versionskontrolle: Git

Du musst zwar keine Versionskontrolle benutzen, um dem Buch folgen zu können, aber es wird sehr viel einfacher, Änderungen rückgängig zu machen, wenn du etwas falsch gemacht hast.

Zur Beispiellösung zurücksetzen

Du kannst jederzeit zum Beginn eines Abschnitts zurückkehren, indem du einen Befehl wie

```
git reset –hard section-2.1
```

ausführst.

Achtung: Alle Änderungen, die du gemacht hast, gehen dabei verloren.

1.6 Das durchgehende Beispiel: ein 2D-Rätselspiel

Bleibt nur noch zu erläutern, wie ich dir alle diese wunderbaren Regeln und großartigen Refactorings beibringen werde. Das Buch ist um ein durchgehendes Beispiel aufgebaut: ein 2D-Schieberätsel, ähnlich dem klassischen Spiel *Boulder Dash* (Abbildung 1.5).

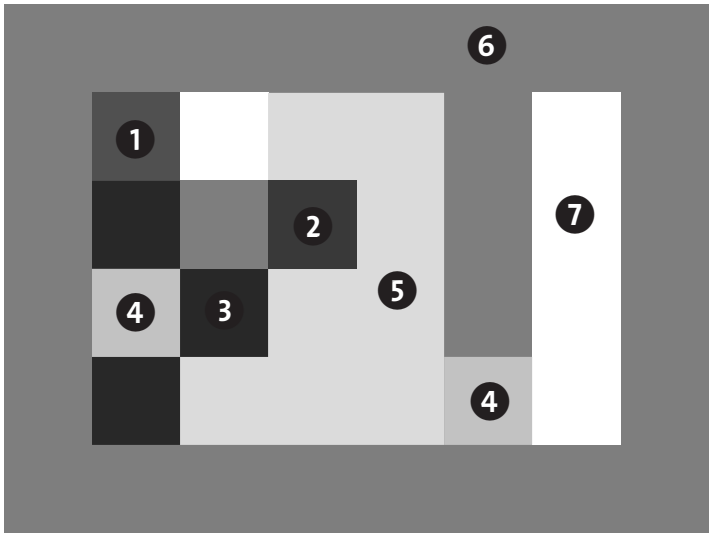

Abbildung 1.5 Ein Bildschirmfoto des unveränderten Spiels »Boulder Dash«

Das gibt uns eine umfangreiche Codebasis, mit der wir im gesamten ersten Teil des Buches spielen können. Ein durchgehendes Beispiel zu haben spart Zeit, weil wir uns nicht in jedem Kapitel in ein neues Beispiel einlesen müssen.

Das Beispiel ist in einem realistischen Stil geschrieben. Der Code ähnelt dem, was man auch in der Softwareindustrie findet. Es ist keine einfache Übung – außer, man hat die Fähigkeiten aus diesem Buch erworben. Der Code folgt bereits den Prinzipien *DRY* (Don't Repeat Yourself – Wiederhol dich nicht) und *KISS* (Keep It Simple, Stupid – Halt es einfach, du Trottel). Trotzdem ist er genauso unangenehm wie ein trockener Kuss.

Für das Beispiel habe ich ein Spiel gewählt, weil beim manuellen Testen leicht auffällt, wenn sich etwas falsch verhält. Wir haben ein intuitives Verständnis dafür, wie es sich verhalten sollte. Ein Spiel zu testen macht außerdem mehr Spaß, als sich zum Beispiel Logdateien aus einem Buchhaltungssystem anzuschauen.

Die Benutzerin oder der Benutzer steuert das Spielerquadrat mit den Pfeiltasten. Ziel des Spiels ist es, die Kiste (in Abbildung 1.5 mit ❷ markiert) in die untere rechte Ecke zu rangieren. Die Farben sind in der Druckvariante nicht enthalten; die Spielelemente sind wie folgt eingefärbt:

❶ Das rote Quadrat ist die Spielerin oder der Spieler.

❷ Braune Quadrate sind Kisten.

❸ Blaue Quadrate sind Steine.

❹ Gelbe Quadrate sind Schlüssel *oder* Schlösser – das ziehen wir später gerade.

❺ Grünliche Quadrate heißen *Flux*.

❻ Graue Quadrate sind Wände.

❼ Weiße Quadrate sind Luft, also leer.

Wenn unter einer Kiste oder einem Stein nichts ist, dann fallen sie. Die Spielerin oder der Spieler kann eine einzelne Kiste oder einen einzelnen Stein schieben, wenn das Feld dahinter frei ist und das Objekt nicht gerade fällt. Der Weg von der Kiste in die untere rechte Ecke ist zunächst durch ein Schloss versperrt, die Spielerin oder der Spieler muss also einen Schlüssel holen, um es zu öffnen. Flux kann von ihr bzw. ihm gegessen (entfernt) werden, indem sie bzw. er das Feld betritt.

Jetzt ist ein guter Zeitpunkt, sich das Spiel zu besorgen und es auszuprobieren:

1. Öffne eine Konsole dort, wo du das Spiel speichern möchtest.
   - Mit `git clone https://github.com/thedrlambda/five-lines` kannst du den Quelltext des Spiels herunterladen.
   - Mit `tsc -w` kompilierst du TypeScript zu JavaScript, nachdem du etwas geändert hast.
2. Öffne die Datei *index.html* in einem Webbrowser.

Du kannst das Leveldesign im Code ändern, indem du das Array in der Variable `map` anpasst (ein Beispiel dafür findest du im Anhang). Viel Spaß dabei!

1. Öffne den Ordner in Visual Studio Code.
2. Wähle Terminal und dann Neues Terminal.
3. Führe den Befehl `tsc -w` aus.
4. TypeScript kompiliert nun deine Änderungen im Hintergrund. Du kannst das Terminal getrost schließen.
5. Warte nach jeder Änderung kurz ab, während der Compiler arbeitet, dann lade die Seite im Browser neu.

Mit demselben Vorgehen wirst du die Beispiele in Teil I nachvollziehen können. Bevor wir aber dazu kommen, bauen wir im nächsten Kapitel ein breiteres Grundwissen über Refactoring auf.

1.6.1 Übung macht den Meister: eine zweite Codebasis

Ich glaube fest an das Lernen durch Üben, deshalb habe ich ein zweites Projekt vorbereitet, dieses allerdings ohne Lösungen. Du kannst dieses Projekt verwenden, wenn du das Buch später noch einmal zur Hand nimmst und eine zusätzliche Herausforderung suchst, oder als Übungen für Studierende, falls du dieses Buch für die Lehre einsetzt. Dieses zweite Projekt ist ein 2D-Actionspiel. Beide Codebasen haben denselben Stil und dieselbe Grundstruktur, sie haben dieselben Elemente, und es braucht dieselben Schritte, sie zu refactorn. Diese zweite Codebasis ist etwas komplexer, aber wenn du den Regeln und Refactorings folgst, solltest du trotzdem zum Ziel gelangen. Um an dieses Projekt zu kommen, folge den oben aufgeführten Schritten mit der URL *https://github.com/thedrlambda/bomb-guy*.

1.7 Ein Wort zu Software aus der echten Welt

Es ist wichtig, im Kopf zu behalten, dass dieses Buch eine Einführung in Refactoring ist. Es versucht nicht, exakte Regeln an die Hand zu geben, die du unter allen Umständen auf produktiven Code anwenden kannst. Du wendest die Regeln an, indem du zuerst ihre Namen lernst und sie dann befolgst. Fällt dir das leicht, dann lerne die Beschreibungen mit den Ausnahmen. Schließlich benutzt du dieses Wissen, um ein Verständnis für das eigentliche Problem, den Code Smell, zu entwickeln.

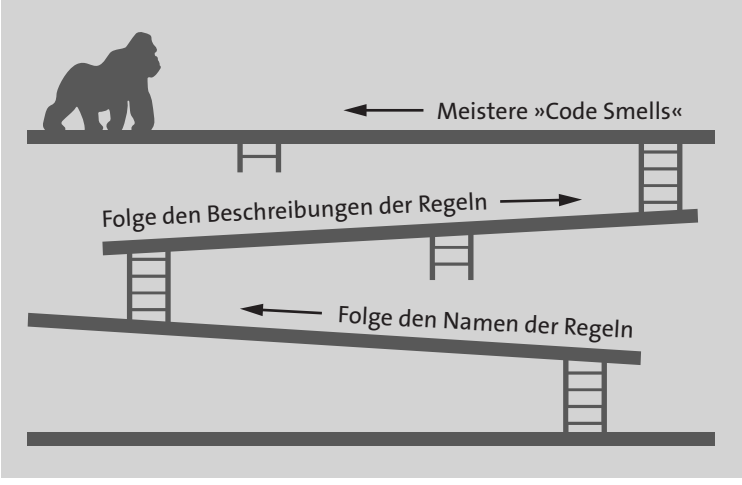

Abbildung 1.6 Wie du die Regeln anwendest

Das erklärt auch, warum wir nicht einfach ein Programm schreiben können, das das Refactoring für uns übernimmt (bestenfalls könnten wir vielleicht ein Plug-in schreiben, das anhand der Regeln Codestellen erkennt, die möglicherweise problematisch sind). Der Zweck der Regeln ist, Verständnis zu entwickeln. Kurz gesagt: Du folgst den Regeln, bis du es besser weißt.

Bedenke auch, dass wir in einer sicheren Umgebung – dem Beispielprojekt – Refactoring lernen und deshalb keine automatisierten Tests brauchen; bei einem echten System trifft das wahrscheinlich nicht zu.

1.8 Zusammenfassung

- Refactoring bedarf einer Kombination aus *Fähigkeiten*, um zu wissen, was ein Refactoring erfahren sollte, *Kultur*, um zu wissen, wann die richtige Zeit für dieses Refactoring ist, und *Werkzeugen*, um zu wissen, wie man ein Refactoring sicher durchführt.
- Üblicherweise werden Code Smells herangezogen, um zu beschreiben, was ein Refactoring erfahren sollte. Für einen Einsteiger sind Code Smells nur schwer anwendbar, weil sie unscharf sind. Dieses Buch enthält konkrete Regeln, die während des Lernens Code Smells ersetzen. Die Regeln haben drei Abstraktionsgrade: sehr konkrete Namen, Beschreibungen, die Nuancen in Form von Ausnahmen hinzufügen, und zuletzt die Absicht des Code Smells, von dem sie abgeleitet wurden.
- Ich glaube, dass Refactoring und automatisiertes Testen getrennt voneinander erlernbar sind und dass so die Einstiegshürde gesenkt wird. Anstelle von automatisierten Tests verwenden wir den Compiler, Versionskontrolle und manuelles Testen.
- Der Arbeitsablauf des Refactorings ist mit der Rot-Grün-Refactoring-Schleife an testgetriebene Entwicklung gekoppelt. Weil darin aber eine Abhängigkeit von automatisierten Tests steckt, schlage ich stattdessen diesen sechsschrittigen Workflow vor: Erkunden, Spezifizieren, Implementieren, Testen, Refactoring, Liefern. Wenn wir vorhandenen Code anpassen, ist die richtige Zeit für Refactoring, bevor wir den Code ändern.
- Im gesamten ersten Teil dieses Buches verwenden wir Visual Studio Code, TypeScript und Git, um den Code eines 2D-Rätselspiels aufzuräumen.

Kapitel 5
Ähnlichen Code zusammenführen

In diesem Kapitel

- Ähnliche Klassen vereinheitlichen mit »Ähnliche Klassen zusammenführen«
- Mit booleschen Operationen Struktur hervorheben
- Einfache UML-Klassendiagramme verstehen
- Ähnlichen Code zusammenführen mit »Strategie einführen«
- Unordnung beseitigen mit »Keine Interfaces mit nur einer Implementierung«

Im letzten Kapitel hatte ich angesprochen, dass wir mit `updateTile` noch nicht fertig sind. Die Funktion verstößt noch gegen mehrere Regeln, allen voran »Benutze niemals `if` mit `else`« (Abschnitt 4.1.1). Zudem haben wir daran gearbeitet, bestimmte Strukturen im Code zu erhalten, weil sie Domänenwissen ausdrücken, z. B. die Oder-Verknüpfung `||`. In diesem Kapitel schauen wir uns an, wie wir noch weitere solche Strukturen herausstellen können.

So sieht `updateTile` im Moment aus:

Listing 5.1 Ausgangszustand

```
01 function updateTile(x: number, y: number) {
02    if ((map[y][x].isStone() || map[y][x].isFallingStone())
03          && map[y + 1][x].isAir()) {
04       map[y + 1][x] = new FallingStone();
05       map[y][x] = new Air();
06    } else if ((map[y][x].isBox() || map[y][x].isFallingBox())
07          && map[y + 1][x].isAir()) {
08       map[y + 1][x] = new FallingBox();
09       map[y][x] = new Air();
10    } else if (map[y][x].isFallingStone()) {
11       map[y][x] = new Stone();
12    } else if (map[y][x].isFallingBox()) {
13       map[y][x] = new Box();
14    }
15 }
```

5.1 Ähnliche Klassen zusammenführen

Als Erstes fällt auf, dass wir auch hier wieder geklammerte Bedingungen haben, die einen Zusammenhang in unserer Domäne darstellen, z. B. (`map[y][x].isStone() || map[y][x].isFallingStone()`). Diesen Ausdruck von Domänenwissen wollen wir nicht nur beibehalten, sondern hervorheben. Unser erster Schritt ist deshalb, für die beiden geklammerten `||` jeweils eine Funktion einzuführen. Wir führen die Begriffe *stony* (steinig) und *boxy* (»kistig«) ein. Sie sollen bedeuten, »etwas verhält sich wie ein Stein« und »etwas verhält sich wie eine Kiste«.

Listing 5.2 Vorher

```
01 function updateTile(x: number, y: number) {
02    if ((map[y][x].isStone()
03          || map[y][x].isFallingStone())
04          && map[y + 1][x].isAir()) {
05       map[y + 1][x] = new FallingStone();
06       map[y][x] = new Air();
07    } else if ((map[y][x].isBox()
08          || map[y][x].isFallingBox())
09          && map[y + 1][x].isAir()) {
10       map[y + 1][x] = new FallingBox();
11       map[y][x] = new Air();
12    } else if (map[y][x].isFallingStone()) {
13       map[y][x] = new Stone();
14    } else if (map[y][x].isFallingBox()) {
15       map[y][x] = new Box();
16    }
17 }
```

Listing 5.3 Nachher

```
01 function updateTile(x: number, y: number) {
02    if (map[y][x].isStony()
03
04          && map[y + 1][x].isAir()) {
05       map[y + 1][x] = new FallingStone();
06       map[y][x] = new Air();
07    } else if (map[y][x].isBoxy()
08
09          && map[y + 1][x].isAir()) {
10       map[y + 1][x] = new FallingBox();
11       map[y][x] = new Air();
12    } else if (map[y][x].isFallingStone()) {
13       map[y][x] = new Stone();
14    } else if (map[y][x].isFallingBox()) {
15       map[y][x] = new Box();
16    }
17 }
18
19 interface Tile {
20    // ...
21    isStony(): boolean;
22    isBoxy(): boolean;
23 }
24 class Air implements Tile {
25    // ...
26    isStony() { return false; }
27    isBoxy() { return false; }
28 }
```

2, 7, 21–22, 26–27: Neue Hilfsmethoden und ihre Aufrufe

Jetzt, wo die beiden Oder-Verknüpfungen erledigt sind, könnten wir den Code in die Klassen schieben. Aber wir können auch noch abwarten und zunächst die Klassen und Methoden betrachten, die wir im letzten Kapitel eingeführt haben. Mit »Versuchsweise löschen und kompilieren« (Abschnitt 4.5.1) können wir jetzt `isStone` und `isBox` entfernen.

Es fällt auf, dass es zwischen `Stone` und `FallingStone` nur zwei Unterschiede gibt: die Ergebnisse von `isFallingStone` und `moveHorizontal`.

Listing 5.4 Stone

```
01 class Stone implements Tile {
02    isAir() { return false; }
03    isFallingStone() { return false; }
04    isFallingBox() { return false; }
05    isLock1() { return false; }
06    isLock2() { return false; }
07    draw(g: CanvasRenderingContext2D,
08          x: number, y: number) {
09       // ...
10    }
11    moveVertical(dy: number) { }
12    isStony() { return true; }
13    isBoxy() { return false; }
14    moveHorizontal(dx: number) {
15       // ...
16    }
17 }
```

Listing 5.5 FallingStone

```
01 class FallingStone implements Tile {
02    isAir() { return false; }
03    isFallingStone() { return true; }
04    isFallingBox() { return false; }
05    isLock1() { return false; }
06    isLock2() { return false; }
07    draw(g: CanvasRenderingContext2D,
08          x: number, y: number) {
09       // ...
10    }
11    moveVertical(dy: number) { }
12    isStony() { return true; }
13    isBoxy() { return false; }
14    moveHorizontal(dx: number) {
15
16    }
17 }
```

Beide Listings, 3, 15: Die einzigen Unterschiede

Gibt eine Methode einen konstanten Wert zurück, nennen wir sie eine *konstante Methode*.

Wir können diese beiden Klassen zusammenführen, weil sie eine konstante Methode teilen, die in beiden einen anderen Wert zurückgibt. Zwei Klassen zusammenzuführen ist ein zweiteiliger Prozess und erinnert ein wenig an die Addition von Brüchen. Der erste Schritt zur Addition von Brüchen ist, den Nenner anzugleichen, und genauso ist der erste Schritt zur Vereinigung zweier Klassen: sie bis auf ihre konstanten Methoden gleichzumachen. Der zweite Schritt zur Addition von Brüchen ist das eigentliche Addieren, für Klassen ist es das eigentliche Zusammenführen. Schauen wir uns das am Beispiel an.

1. Im ersten Teil des Vorgehens gleichen wir die beiden Implementierungen von `moveHorizontal` an.
   a) Umgib im Rumpf beider Implementierungen den bisherigen Code mit einem `if (true) { … }`.

Listing 5.6 Vorher

```
01 class Stone implements Tile {
02    // ...
03    moveHorizontal(dx: number) {
04
05       if (map[playery]
06                [playerx+dx+dx].isAir()
07             && !map[playery+1]
08                [playerx+dx].isAir()) {
09          map[playery][playerx+dx + dx] =
10                this;
11          moveToTile(playerx+dx, playery);
12       }
13
14    }
15 }
16 class FallingStone implements Tile {
17    // ...
18    moveHorizontal(dx: number) {
19
20    }
21 }
```

Listing 5.7 Nachher (1/8)

```
01 class Stone implements Tile {
02    // ...
03    moveHorizontal(dx: number) {
04       if (true) {
05          if (map[playery]
06                [playerx+dx+dx].isAir()
07             && !map[playery+1]
08                [playerx+dx].isAir()) {
09             map[playery][playerx+dx + dx] =
10                   this;
11             moveToTile(playerx+dx, playery);
12          }
13       }
14    }
15 }
16 class FallingStone implements Tile {
17    // ...
18    moveHorizontal(dx: number) {
19       if (true) { }
20    }
21 }
```

4, 19: Neue `if (true)`s

b) Ersetze das `true` jeweils durch `isFallingStone() === true` und `isFallingStone() === false`.

Listing 5.8 Vorher

```
01 class Stone implements Tile {
02    // ...
03    moveHorizontal(dx: number) {
04       if (true) {
05          if (map[playery]
06                [playerx+dx+dx].isAir()
07             && !map[playery+1]
08                [playerx+dx].isAir()) {
09             map[playery][playerx+dx + dx] =
10                   this;
11             moveToTile(playerx+dx, playery);
```

Listing 5.9 Nachher (2/8)

```
01 class Stone implements Tile {
02    // ...
03    moveHorizontal(dx: number) {
04       if (this.isFallingStone()  === false){
05          if (map[playery]
06                [playerx+dx+dx].isAir()
07             && map[playery+1]
08                [playerx+dx].isAir()) {
09             map[playery][playerx+dx + dx] =
10                   this;
11             moveToTile(playerx+dx, playery);
```

```
12          }
13       }
14    }
15 }
16 class FallingStone implements Tile {
17    // ...
18    moveHorizontal(dx: number) {
19       if (true) { }
20    }
21 }
```

```
12          }
12       }
14    }
15 }
16 class FallingStone implements Tile {
17    // ...
18    moveHorizontal(dx: number) {
19       if (this.isFallingStone() === true) { }
20    }
21 }
```

4, 20: Spezialisierte Bedingungen

c) Kopiere den Rumpf beider `moveHorizontal`s und füge ihn mit einem `else` im jeweils anderen `moveHorizontal` ein.

Listing 5.10 Vorher

```
01 class Stone implements Tile {
02    // ...
03    moveHorizontal(dx: number) {
04       if (this.isFallingStone()
05             === false) {
06          if (map[playery]
07                [playerx+dx+dx].isAir()
08             && !map[playery+1]
09                [playerx+dx].isAir()) {
10             map[playery][playerx+dx + dx] =
11                   this;
12             moveToTile(playerx+dx, playery);
13          }
14       }
15
16
17
18    }
19 }
20 class FallingStone implements Tile {
21    // ...
22    moveHorizontal(dx: number) {
23
24
```

Listing 5.11 Nachher (3/8)

```
01 class Stone implements Tile {
02    // ...
03    moveHorizontal(dx: number) {
04       if (this.isFallingStone()
05             === false) {
06          if (map[playery]
07                [playerx+dx+dx].isAir()
08             && !map[playery+1]
09                [playerx+dx].isAir()) {
10             map[playery][playerx+dx + dx] =
11                   this;
12             moveToTile(playerx+dx, playery);
13          }
14       }
15       else if (this.isFallingStone()
16             === true) {
17       }
18    }
19 }
20 class FallingStone implements Tile {
21    // ...
22    moveHorizontal(dx: number) {
23       if (this.isFallingStone()
24             === false) {
```

```
25
26
27
28
29
30
31
32
33
34        if (this.isFallingStone() === true)
35        {
36        }
37    }
38 }
```

```
25          if (map[playery]
26                [playerx+dx+dx].isAir()
27              && !map[playery+1]
28                [playerx+dx].isAir()) {
29              map[playery][playerx+dx + dx] =
30                    this;
31              moveToTile(playerx+dx, playery);
32          }
33        }
34        else if (this.isFallingStone()=== true)
35        {
36        }
37    }
38 }
```

15, 23: Rumpf der anderen Methode

2. Jetzt unterscheiden sich die beiden Klassen nur noch durch die konstante Methode `isFallingStone`, und wir können mit Phase zwei beginnen, indem wir ein neues Feld `falling` einführen und ihm im Konstruktor einen Wert zuweisen.

Listing 5.12 Vorher

```
01 class Stone implements Tile {
02    // ...
03
04
05
06
07   isFallingStone() { return false; }
08 }
09 class FallingStone implements Tile {
10    // ...
11
12
13
14
15   isFallingStone() { return true; }
16 }
```

Listing 5.13 Nachher (4/8)

```
01 class Stone implements Tile {
02    private falling: boolean;
03    constructor() {
04       this.falling = false;
05    }
06    // ...
07    isFallingStone() { return false; }
08 }
09 class FallingStone implements Tile {
10    private falling: boolean;
11    constructor() {
12       this.falling = true;
13    }
14    // ...
15    isFallingStone() { return true; }
16 }
```

2, 10: Neues Feld

4, 12: Weist dem neuen Feld einen Default-Wert zu.

3. Passe `isFallingStone` an und gib das neue Feld `falling` zurück.

Listing 5.14 Vorher

```
01 class Stone implements Tile {
02    // ...
03    isFallingStone() { return false; }
04 }
05 class FallingStone implements Tile {
06    // ...
07    isFallingStone() { return true; }
08 }
```

Listing 5.15 Nachher (5/8)

```
01 class Stone implements Tile {
02    // ...
03    isFallingStone() { return this.falling; }
04 }
05 class FallingStone implements Tile {
06    // ...
07    isFallingStone() { return this.falling; }
08 }
```

3, 7: Gibt jetzt ein Feld zurück anstelle einer Konstanten.

4. Kompiliere, um sicher zu sein, dass wir bisher nichts kaputt gemacht haben.
5. Für jede der Klassen:
   a) Kopiere den Default-Wert von `falling` und mache aus dem Default-Wert einen Parameter.

Listing 5.16 Vorher

```
01 class Stone implements Tile {
02    private falling: boolean;
03    constructor() {
04       this.falling = false;
05    }
06    // ...
07 }
```

Listing 5.17 Nachher (6/8)

```
01 class Stone implements Tile {
02    private falling: boolean;
03    constructor(falling: boolean) {
04       this.falling = falling;
05    }
06    // ...
07 }
```

3–4: Aus `falling` wird ein Parameter.

   b) Korrigiere die Kompilierfehler, indem du den alten Default-Wert als Argument übergibst.

Listing 5.18 Vorher

```
01 /// ...
02    new Stone();
03 /// ...
```

Listing 5.19 Nachher (7/8)

```
01 /// ...
02    new Stone(false);
03 /// ...
```

2: Übergibt jetzt den Default-Wert.

   c) Lösche alle betroffenen Klassen bis auf eine. Behebe die entstehenden Kompilierfehler, indem du überall die verbleibende Klasse verwendest.

Listing 5.20 Vorher

```
/// ...
    new FallingStone(true);
/// ...
```

Listing 5.21 Nachher (8/8)

```
/// ...
    new Stone(true);
/// ...
```

2: Ersetzt die gelöschte Klasse mit dem Ergebnis der Zusammenführung.

Insgesamt bewirkt die Zusammenführung dieses Ergebnis:

Listing 5.22 Vorher

```
function updateTile(x: number, y: number) {
   if (map[y][x].isStony()
         && map[y + 1][x].isAir()) {
      map[y + 1][x] = new FallingStone();
      map[y][x] = new Air();
   } else if (map[y][x].isBoxy()
         && map[y + 1][x].isAir()) {
      map[y + 1][x] = new FallingBox();
      map[y][x] = new Air();
   } else if (map[y][x].isFallingStone()) {
      map[y][x] = new Stone();
   } else if (map[y][x].isFallingBox()) {
      map[y][x] = new Box();
   }
}
class Stone implements Tile {
   // ...
   isFallingStone() { return false; }
   moveHorizontal(dx: number) {
      if (map[playery]
            [playerx+dx+dx].isAir()
         && !map[playery+1]
            [playerx+dx].isAir()){
         map[playery][playerx+dx + dx] =
               this;
         moveToTile(playerx+dx, playery);
      }
   }
}
```

Listing 5.23 Nachher

```
function updateTile(x: number, y: number) {
   if (map[y][x].isStony()
         && map[y + 1][x].isAir()) {
      map[y + 1][x] = new Stone(true);
      map[y][x] = new Air();
   } else if (map[y][x].isBoxy()
         && map[y + 1][x].isAir()) {
      map[y + 1][x] = new FallingBox();
      map[y][x] = new Air();
   } else if (map[y][x].isFallingStone()) {
      map[y][x] = new Stone(false);
   } else if (map[y][x].isFallingBox()) {
      map[y][x] = new Box();
   }
}
class Stone implements Tile {
   constructor(private falling: boolean) { }
   // ...
   isFallingStone() { return this.falling; }
   moveHorizontal(dx: number) {
      if (this.isFallingStone()
            === false) {
         if (map[playery]
               [playerx+dx+dx].isAir()
            && !map[playery+1]
               [playerx+dx].isAir()) {
            map[playery][playerx+dx + dx] =
                  this;
            moveToTile(playerx+dx, playery);
```

```
30 class FallingStone implements Tile {
31    // ...
32    isFallingStone() { return true; }
33    moveHorizontal(dx: number) { }
34 }
```

```
30          }
31       } else if(this.isFallingStone()
32             === true) {
33       }
34    }
35 }
```

19: `isFallingStone` gibt das Feld zurück.
4, 11, 17: Privates Feld, im Konstruktor gesetzt
20: `moveHorizontal` enthält die kombinierten Rümpfe.
33: `FallingStone` wurde gelöscht.

In TypeScript...

... verhalten sich Konstruktoren etwas anders als in den meisten Sprachen. Erstens können wir nur einen Konstruktor haben, und er heißt immer `constructor`.

Zweitens: Die Schlüsselwörter `public` oder `private` vor einem Konstruktorparameter führen dazu, dass ein Feld mit dem gleichen Namen angelegt und mit dem Wert des Parameters initialisiert wird. Die beiden folgenden Beispiele sind äquivalent.

Listing 5.24 Vorher

```
class Stone implements Tile {
   private falling: boolean;
   constructor(falling: boolean) {
      this.falling = falling;
   }
}
```

Listing 5.25 Nachher

```
class Stone implements Tile {

constructor(private falling: boolean) { }
}
```

In diesem Buch bevorzugen wir die Variante aus Listing 5.25.

Schauen wir uns das Ergebnis des Refactorings in `moveHorizontal` an, fallen einige interessante Details auf. Das Offensichtlichste ist eine leere `if`-Anweisung. Wichtiger ist aber, dass die Methode jetzt ein `else` enthält und damit gegen »Benutze niemals `if` mit `else`« verstößt. Ein häufiger Effekt des Zusammenführens von Klassen ist das Aufdecken versteckter Typcodes. In diesem Fall ist der boolesche Wert `falling` ein Typcode. Wir können ihn als solchen sichtbar machen, indem wir ihn in ein Enum umwandeln.

Listing 5.26 Vorher

```
01
02
03
```

Listing 5.27 Nachher

```
01 enum FallingState {
02    FALLING, RESTING
03 }
```

```
/// ...
   new Stone(true);
/// ...
   new Stone(false);
/// ...
class Stone implements Tile {
   constructor(private falling: boolean)
   { }
   // ...
   isFallingStone() {
      return this.falling;

   }
}
```

```
/// ...
   new Stone(FallingState.FALLING);
/// ...
   new Stone(FallingState.RESTING);
/// ...
class Stone implements Tile {
   constructor(private falling: FallingState)
   { }
   // ...
   isFallingStone() {
      return this.falling
            === FallingState.FALLING;
   }
}
```

Schon diese Änderung macht den Code lesbarer, denn statt eines unbenannten Booleans haben wir nun einen aussagekräftigen Enum-Wert. Noch besser ist aber, dass wir schon wissen, wie wir mit Enums umgehen: »Typcodes durch Klassen ersetzen« (Abschnitt 4.1.3).

Listing 5.28 Vorher

```
enum FallingState {
   FALLING, RESTING
}









   new Stone(FallingState.FALLING);
   new Stone(FallingState.RESTING);
class Stone implements Tile {
   constructor(private falling:
FallingState)
   { }
   // ...
```

Listing 5.29 Nachher

```
interface FallingState {
   isFalling(): boolean;
   isResting(): boolean;
}
class Falling implements FallingState {
   isFalling() { return true; }
   isResting() { return false; }
}
class Resting implements FallingState {
   isFalling() { return false; }
   isResting() { return true; }
}
   new Stone(new Falling());
   new Stone(new Resting());
class Stone implements Tile {
   constructor(private falling:
FallingState)
   { }
   // ...
```

```
20     isFallingStone() {
21        return this.falling
22              === FallingState.FALLING;
23     }
24  }
```

```
20     isFallingStone() {
21        return this.falling.isFalling();
22     }
23  }
```

Wenn uns die Performanz von `new` Sorgen macht, können wir auch für `Falling` und `Resting` je eine Konstante einführen und diese verwenden. Aber denke daran, dass Performanz-Optimierung auf den Daten von Profiling-Werkzeugen beruhen sollte.

Wenn wir jetzt `isFallingStone` nach `moveHorizontal` integrieren, sehen wir, dass wir auch »Code in Klassen schieben« aus Abschnitt 4.1.5 anwenden sollten.

Listing 5.30 Vorher

```
01  interface FallingState {
02     // ...
03
04
05  }
06  class Falling implements FallingState {
07     // ...
08
09
10  }
11  class Resting implements FallingState {
12     // ...
13  }
14  class Stone implements Tile {
15     // ...
16     moveHorizontal(dx: number) {
17        if (!this.falling.isFalling()) {
18           if (map[playery]
19                 [playerx+dx+dx].isAir()
20              && !map[playery+1]
21                 [playerx+dx].isAir()) {
22              map[playery][playerx+dx + dx] =
23                    this;
24              moveToTile(playerx+dx, playery);
25           }
26        } else if (this.falling.isFalling()) {
```

Listing 5.31 Nachher

```
01  interface FallingState {
02     // ...
03     moveHorizontal(
04        tile: Tile, dx: number): void;
05  }
06  class Falling implements FallingState {
07     // ...
08     moveHorizontal(tile: Tile, dx: number) {
09     }
10  }
11  class Resting implements FallingState {
12     // ...
13     moveHorizontal(tile: Tile, dx: number) {
14        if (map[playery]
15              [playerx+dx+dx].isAir()
16           && !map[playery+1]
17              [playerx+dx].isAir()) {
18           map[playery][playerx+dx + dx] =
19                 tile;
20           moveToTile(playerx+dx, playery);
21        }
22     }
23  }
24  class Stone implements Tile {
25     // ...
26     moveHorizontal(dx: number) {
```

```
27        }
28    }
29 }
```

```
27        this.falling.moveHorizontal(this, dx);
28    }
29 }
```

Da wir ein neues Interface eingeführt haben, können wir nun zum Abschluss »Versuchsweise löschen und kompilieren« nutzen, um `isResting` zu entfernen. Ich überlasse es dir, das Gleiche für `Box` und `FallingBox` zu tun. Beachte dabei, dass du `FallingState` wiederverwenden kannst. Zwei ähnliche Klassen auf diese Art zusammenzuführen nennen wir »Ähnliche Klassen zusammenführen«.

5.1.1 Refactoring: »Ähnliche Klassen zusammenführen«

Beschreibung

Immer, wenn wir zwei oder mehr Klassen haben, die sich nur durch konstante Methoden unterscheiden, können wir sie durch dieses Refactoring zusammenführen. Die konstanten Methoden nennen wir die *Basis*. Eine Basis mit zwei Methoden nennen wir eine *Zweierbasis*. Wir wollen unsere Basis so klein wie möglich halten. Um *x* Klassen zusammenzuführen, darf unsere Basis nicht mehr als *x*–1 Methoden haben.

Klassen zusammenzuführen ist eine gute Sache, denn weniger Klassen zu haben, bedeutet meistens, mehr Struktur freizulegen.

Vorgehen

1. Im ersten Teil des Vorgehens gleichen wir alle Methoden an, die nicht zur Basis gehören. Führe für jede dieser Methoden die folgenden Schritte durch.
   a) Umgib im Rumpf aller Implementierungen den bisherigen Code mit einem `if (true) { … }`.
   b) Ersetze das `true` jeweils durch einen Ausdruck, der alle Methoden der Basis aufruft und ihren Rückgabewert mit einer Konstanten vergleicht.
   c) Kopiere den Rumpf jeder Implementierung und füge ihn mit einem `else` in allen anderen Versionen ein.
2. Jetzt unterscheiden sich nur noch die Basismethoden und wir können mit Phase zwei beginnen, indem wir für jede Basismethode ein neues Feld einführen und ihm im Konstruktor eine Konstante zuweisen.
3. Passe die Basismethoden an, sodass sie das neue Feld zurückgeben statt einer Konstanten.
4. Kompiliere, um sicher zu sein, dass du bisher nichts kaputt gemacht hast.
5. Führe für jede der Klassen für jedes Feld Folgendes durch:
   a) Kopiere den Default-Wert des Feldes und mache dann aus dem Default-Wert einen Parameter.

b) Korrigiere die Kompilierfehler, indem du den alten Default-Wert als Argument übergibst.

6. Wenn alle betroffenen Klassen gleich sind, lösche sie alle bis auf eine. Behebe die entstehenden Kompilierfehler, indem du überall die verbleibende Klasse verwendest.

Beispiel

Die Ampel in diesem Beispiel hat drei sehr ähnliche Klassen, also versuchen wir, sie zusammenzuführen.

Listing 5.32 Ausgangszustand

```
function nextColor(t: TrafficColor) {
   if (t.color() === "red") return new Green();
   else if (t.color() === "green") return new Yellow();
   else if (t.color() === "yellow") return new Red();
}
interface TrafficColor {
   color(): string;
   check(car: Car): void;
}
class Red implements TrafficColor {
   color() { return "red"; }
   check(car: Car) { car.stop(); }
}
class Yellow implements TrafficColor {
   color() { return "yellow"; }
   check(car: Car) { car.stop(); }
}
class Green implements TrafficColor {
   color() { return "green"; }
   check(car: Car) { car.drive(); }
}
```

Wir folgen dem Vorgehen:

1. Die Basismethode ist `color`, sie gibt in jeder Klasse eine andere Konstante zurück. Wir müssen die `check`-Methoden angleichen, also führe für jede von ihnen diese Schritte durch:

   a) Umgib im Rumpf aller Implementierungen von `check` den bisherigen Code mit einem `if (true) { … }`.

Listing 5.33 Vorher

```
01 class Red implements TrafficColor {
02    // ...
03    check(car: Car) {
04
05       car.stop();
06
07    }
08 }
09 class Yellow implements TrafficColor {
10    // ...
11    check(car: Car) {
12
13       car.stop();
14
15    }
16 }
17 class Green implements TrafficColor {
18    // ...
19    check(car: Car) {
20
21       car.drive();
22
23    }
24 }
```

Listing 5.34 Nachher (1/8)

```
01 class Red implements TrafficColor {
02    // ...
03    check(car: Car) {
04       if (true) {
05          car.stop();
06       }
07    }
08 }
09 class Yellow implements TrafficColor {
10    // ...
11    check(car: Car) {
12       if (true) {
13          car.stop();
14       }
15    }
16 }
17 class Green implements TrafficColor {
18    // ...
19    check(car: Car) {
20       if (true) {
21           car.drive();
22       }
23    }
24 }
```

4, 12, 20: `if (true) { … }` hinzugefügt

b) Ersetze das `true` jeweils durch einen Ausdruck, der die Basismethode aufruft und ihren Rückgabewert mit einer Konstanten vergleicht.

Listing 5.35 Vorher

```
01 class Red implements TrafficColor {
02    color() { return "red"; }
03    check(car: Car) {
04       if (true) {
05          car.stop();
06       }
07    }
08 }
```

Listing 5.36 Nachher (2/8)

```
01 class Red implements TrafficColor {
02    color() { return "red"; }
03    check(car: Car) {
04       if (this.color() === "red") {
05          car.stop();
06       }
07    }
08 }
```

```
09 class Yellow implements TrafficColor {
10    color() { return "yellow"; }
11    check(car: Car) {
12       if (true) {
13          car.stop();
14       }
15    }
16 }
17 class Green implements TrafficColor {
18    color() { return "green"; }
19    check(car: Car) {
20       if (true) {
21          car.drive();
22       }
23    }
24 }
```

```
09 class Yellow implements TrafficColor {
10    color() { return "yellow"; }
11    check(car: Car) {
12       if (this.color() === "yellow") {
13          car.stop();
14       }
15    }
16 }
17 class Green implements TrafficColor {
18    color() { return "green"; }
19    check(car: Car) {
20       if (this.color() === "green") {
21          car.drive();
22       }
23    }
24 }
```

4, 12, 20: Vergleich mit der Basismethode

c) Kopiere den Rumpf jeder Implementierung und füge ihn mit einem `else` in allen anderen Versionen ein.

Listing 5.37 Vorher

```
01 class Red implements TrafficColor {
02    // ...
03    check(car: Car) {
04       if (this.color() === "red") {
05          car.stop();
06       }
07
08
09
10
11    }
12 }
13 class Yellow implements TrafficColor {
14    // ...
15    check(car: Car) {
16
17
18       if (this.color() === "yellow") {
```

Listing 5.38 Nachher (3/8)

```
01 class Red implements TrafficColor {
02    // ...
03    check(car: Car) {
04       if (this.color() === "red") {
05          car.stop();
06       } else if (this.color() === "yellow") {
07          car.stop();
08       } else if (this.color() === "green") {
09          car.drive();
10       }
11    }
12 }
13 class Yellow implements TrafficColor {
14    // ...
15    check(car: Car) {
16       if (this.color() === "red") {
17          car.stop();
18       } else if (this.color() === "yellow") {
```

```
19          car.stop();
20       }
21
22
23    }
24 }
25 class Green implements TrafficColor {
26    // ...
27    check(car: Car) {
28
29
30
31
32       if (this.color() === "green") {
33          car.drive();
34       }
35    }
36 }
```

```
19          car.stop();
20       } else if (this.color() === "green") {
21          car.drive();
22       }
23    }
24 }
25 class Green implements TrafficColor {
26    // ...
27    check(car: Car) {
28       if (this.color() === "red") {
29          car.stop();
30       } else if (this.color() === "yellow") {
31          car.stop();
32       } else if (this.color() === "green") {
33          car.drive();
34       }
35    }
36 }
```

6–9, 16–17, 20–21, 28–31: Die Methoden werden in die jeweils anderen Versionen kopiert.

2. Jetzt sind alle `check`-Methoden gleich, nur noch die Basismethode unterscheidet sich. Wir können mit Phase zwei beginnen, indem wir ein neues Feld `color` einführen und ihm im Konstruktor eine Konstante zuweisen.

Listing 5.39 Vorher

```
01 class Red implements TrafficColor {
02
03
04    color() { return "red"; }
05    // ...
06 }
07 class Yellow implements TrafficColor {
08
09
10    color() { return "yellow"; }
11    // ...
12 }
13 class Green implements TrafficColor {
14
```

Listing 5.40 Nachher (4/8)

```
01 class Red implements TrafficColor {
02    constructor(
03       private col: string = "red") { }
04    color() { return "red"; }
05    // ...
06 }
07 class Yellow implements TrafficColor {
08    constructor(
09       private col: string = "yellow") { }
10    color() { return "yellow"; }
11    // ...
12 }
13 class Green implements TrafficColor {
14    constructor(
```

```
15
16     color() { return "green"; }
17     // ...
18 }
```

```
15         private col: string = "green") { }
16     color() { return "green"; }
17     // ...
18 }
```

3, 9, 15: Neue Konstruktoren

3. Passe die Basismethode an, sodass sie das neue Feld zurückgibt statt einer Konstanten.

Listing 5.41 Vorher

```
01 class Red implements TrafficColor {
02     // ...
03     color() { return "red"; }
04 }
05 class Yellow implements TrafficColor {
06     // ...
07     color() { return "yellow"; }
08 }
09 class Green implements TrafficColor {
10     // ...
11     color() { return "green"; }
12 }
```

Listing 5.42 Nachher (5/8)

```
01 class Red implements TrafficColor {
02     // ...
03     color() { return this.col; }
04 }
05 class Yellow implements TrafficColor {
06     // ...
07     color() { return this.col; }
08 }
09 class Green implements TrafficColor {
10     // ...
11     color() { return this.col; }
12 }
```

3, 7, 11: Gibt ein Feld zurück.

4. Kompiliere, um sicher zu sein, dass du bisher nichts kaputt gemacht hast.
5. Für jede der Klassen, ein Feld nach dem anderen:
   a) Kopiere den Default-Wert des Feldes und mache dann aus dem Default-Wert einen Parameter.

Listing 5.43 Vorher

```
01 class Red implements TrafficColor {
02     constructor(
03         private col: string = "red") { }
04     // ...
05 }
```

Listing 5.44 Nachher (6/8)

```
01 class Red implements TrafficColor {
02     constructor(
03         private col: string) { }
04     // ...
05 }
```

3: Default-Wert entfernt

   b) Korrigiere die Kompilierfehler, indem du den alten Default-Wert als Argument übergibst.

Listing 5.45 Vorher

```
01 function nextColor(t: TrafficColor) {
02    if (t.color() === "red")
03       return new Green();
04    else if (t.color() === "green")
05       return new Yellow();
06    else if (t.color() === "yellow")
07       return new Red();
08 }
```

Listing 5.46 Nachher (7/8)

```
01 function nextColor(t: TrafficColor) {
02    if (t.color() === "red")
03       return new Green();
04    else if (t.color() === "green")
05       return new Yellow();
06    else if (t.color() === "yellow")
07       return new Red("red");
08 }
```

7: Default-Wert eingefügt, um den Fehler zu beheben

6. Wenn alle betroffenen Klassen gleich sind, lösche alle bis auf eine. Behebe die entstehenden Kompilierfehler, indem du überall die verbleibende Klasse verwendest.

Listing 5.47 Vorher

```
01 function nextColor(t: TrafficColor) {
02    if (t.color() === "red")
03       return new Green();
04    else if (t.color() === "green")
05       return new Yellow();
06    else if (t.color() === "yellow")
07       return new Red();
08 }
09 class Yellow implements TrafficColor { ... }
10 class Green implements TrafficColor { ... }
```

Listing 5.48 Nachher (8/8)

```
01 function nextColor(t: TrafficColor) {
02    if (t.color() === "red")
03       return new Red("green");
04    else if (t.color() === "green")
05       return new Red("yellow");
06    else if (t.color() === "yellow")
07       return new Red("red");
08 }
```

3, 5: Klassen `Yellow` und `Green` gelöscht

Jetzt brauchen wir das Interface nicht mehr, und wir sollten `Red` umbenennen. Außerdem sollten wir uns überlegen, wie wir das `if` mit `else`s loswerden können – vielleicht mit einem Refactoring, das wir bald kennenlernen werden. Aber wir haben jetzt schon die drei Klassen erfolgreich zusammengeführt.

Listing 5.49 Vorher

```
01 function nextColor(t: TrafficColor) {
02    if (t.color() === "red")
03       return new Green();
04    else if (t.color() === "green")
05       return new Yellow();
06    else if (t.color() === "yellow")
07       return new Red();
```

Listing 5.50 Nachher

```
01 function nextColor(t: TrafficColor) {
02    if (t.color() === "red")
03       return new Red("green");
04    else if (t.color() === "green")
05       return new Red("yellow");
06    else if (t.color() === "yellow")
07       return new Red("red");
```

```
}
interface TrafficColor {
   color(): string;
   check(car: Car): void;
}
class Red implements TrafficColor {
   color() { return "red"; }
   check(car: Car) { car.stop(); }
}
class Yellow implements TrafficColor {
   color() { return "yellow"; }
   check(car: Car) { car.stop(); }
}
class Green implements TrafficColor {
   color() { return "green"; }
   check(car: Car) { car.drive(); }
}
```

```
}
interface TrafficColor {
   color(): string;
   check(car: Car): void;
}
class Red implements TrafficColor {
   constructor(private col: string) { }
   color() { return this.col; }
   check(car: Car) {
      if (this.color() === "red") {
         car.stop();
      } else if (this.color() === "yellow") {
         car.stop();
      } else if (this.color() === "green") {
         car.drive();
      }
   }
}
```

Es wäre jetzt wahrscheinlich auch sinnvoll, für die drei Farben Konstanten anzulegen, um sie nicht immer wieder instanziieren zu müssen. Das ist zum Glück trivial.

Zum Weiterlesen

Soweit ich weiß, ist dies die erste Beschreibung dieses Vorgehens als Refactoring.

5.2 Einfache Bedingungen zusammenführen

Um `updateTile` weiter zu verbessern, möchten wir als Nächstes die Rümpfe einiger `if`-Zweige angleichen. Schauen wir uns den Code an.

Listing 5.51 Ausgangszustand

```
function updateTile(x: number, y: number) {
   if (map[y][x].isStony()
         && map[y + 1][x].isAir()) {
      map[y + 1][x] = new Stone(new Falling());
      map[y][x] = new Air();
   } else if (map[y][x].isBoxy()
         && map[y + 1][x].isAir()) {
      map[y + 1][x] = new Box(new Falling());
```

Diese Leseprobe haben Sie beim
edv-buchversand.de heruntergeladen.
Das Buch können Sie online in unserem
Shop bestellen.

Hier zum Shop