Kapitel 1

Refactoring: Ein erstes Beispiel

Wie soll ich anfangen, über das Refactoring zu sprechen? Üblicherweise werden die Geschichte des Themas oder die grundlegenden Prinzipien oder Ähnliches erörtert. Wenn dies auf Konferenzen geschieht, werde ich ein wenig schläfrig. Meine Gedanken schweifen ab, und ein Hintergrundprozess meines Gehirns achtet darauf, ob der Redner ein Beispiel ankündigt, um mich dann zu informieren.

Beispiele dagegen beleben meine Aufmerksamkeit, weil ich nachvollziehen kann, was beschrieben wird. Bei den Prinzipien werden viel zu oft Verallgemeinerungen vorgenommen, sodass es schwierig ist, sich vorzustellen, wie etwas praktisch angewendet wird. Ein Beispiel hingegen ist hilfreich, eine konkrete Anwendung zu verdeutlichen.

Ich werde also dieses Buch mit einem Beispiel für Refactoring beginnen. Anhand dessen werde ich die Funktionsweise des Refactorings erläutern und versuchen, Ihnen ein Gespür für den Refactoring-Prozess zu vermitteln. Im sich daran anschließenden Kapitel werde ich wie üblich mit den Grundlagen fortfahren.

Die Wahl eines einführenden Beispiels stellt mich jedoch vor ein Problem. Wenn ich ein umfangreiches Programm auswähle, es beschreibe und Ihnen erkläre, wie das Refactoring vorzunehmen ist, dann ist das für die meisten zu kompliziert, es zu verstehen. (Ich habe das bei der ersten Ausgabe des vorliegenden Buches ausprobiert – und letztlich zwei Beispiele verwerfen müssen, die zwar ziemlich überschaubar waren, deren Beschreibung aber jeweils mehr als hundert Seiten lang war.) Wähle ich hingegen ein Beispiel aus, das einfach genug ist, um gut nachvollziehbar zu sein, könnte das den Eindruck erwecken, dass sich Refactoring überhaupt nicht lohnt.

Ich bin also mit dem klassischen Problem konfrontiert, mit dem man zu kämpfen hat, wenn man Verfahren beschreiben möchte, die für Programme in der Praxis nützlich sind. Offen gestanden ist es tatsächlich nicht der Mühe wert, alle Refactorings, die ich Ihnen zeigen werde, bei dem kleinen Programm anzuwenden, das ich als Beispiel verwende. Wenn der Code jedoch Teil eines größeren Systems ist, dann wird das Refactoring wichtig. Stellen Sie sich beim Lesen meines Beispiels einfach vor, dass es Bestandteil eines sehr viel größeren Systems ist.

1.1 Der Ausgangspunkt

In der ersten Ausgabe dieses Buchs gab das erste Programm die Rechnung einer Videothek aus, was heute dazu führen dürfte, dass sich einige Leser fragen: »Was ist eine Videothek?« Anstatt diese Frage zu beantworten, habe ich das Beispiel überarbeitet, das jetzt auf etwas noch Älterem beruht, das jedoch auch heutzutage noch aktuell ist.

Stellen Sie sich ein Unternehmen vor, das eine Schauspieltruppe beschäftigt, die auf verschiedenen Veranstaltung Theatervorstellungen gibt. Typischerweise beauftragt ein Kunde

einige Vorstellungen. Daraufhin stellt das Unternehmen eine Rechnung aus, deren Betrag von der Größe des Publikums und der Art des aufgeführten Theaterstücks abhängt. Derzeit bietet das Unternehmen zwei Arten von Theaterstücken an: Tragödien und Komödien. Das Unternehmen schreibt aber nicht nur Rechnungen, sondern vergibt auch »Treuepunkte« (`volumeCredits`) an seine Kunden, für die sie bei zukünftigen Vorstellungen einen Preisnachlass erhalten. Sie können sich das als eine Maßnahme zur Kundenbindung vorstellen.

Die Darsteller speichern Informationen über die aufgeführten Theaterstücke in einer einfachen JSON-Datei, die in etwa so aussieht:

```
plays.json...
{
"hamlet": {"name": "Hamlet", "type": "tragedy"},
"as-like": {"name": "As You Like It", "type": "comedy"},
"othello": {"name": "Othello", "type": "tragedy"}
}
```

Die Daten für die Rechnungen werden ebenfalls in Form von JSON-Dateien gespeichert:

```
invoices.json...
[
  {
    "customer": "BigCo",
    "performances": [
      {
        "playID": "hamlet",
        "audience": 55
      },
      {
        "playID": "as-like",
        "audience": 35
      },
      {
        "playID": "othello",
        "audience": 40
      }
    ]
  }
]
```

Der Code zur Ausgabe der Rechnung ist in dieser einfachen Funktion enthalten:

```
function statement (invoice, plays) {
  let totalAmount = 0;
```

```
  let volumeCredits = 0;
  let result = `Abrechnung für ${invoice.customer}\n`;
  const format = new Intl.NumberFormat("en-US",
                        { style: "currency", currency: "USD",
                          minimumFractionDigits: 2 }).format;
  for (let perf of invoice.performances) {
    const play = plays[perf.playID];
    let thisAmount = 0;

    switch (play.type) {
    case "tragedy":
      thisAmount = 40000;
      if (perf.audience > 30) {
        thisAmount += 1000 * (perf.audience - 30);
      }
      break;
    case "comedy":
      thisAmount = 30000;
      if (perf.audience > 20) {
      thisAmount += 10000 + 500 * (perf.audience - 20);
      }
      thisAmount += 300 * perf.audience;
      break;
    default:
      throw new Error(`Unbekannter Typ: ${play.type}`);
    }

    // Treuepunkte hinzufügen
    volumeCredits += Math.max(perf.audience - 30, 0);
    // Zusatzpunkte für jeweils 10 Komödienbesucher
    if ("comedy" === play.type) volumeCredits +=
                              Math.floor(perf.audience / 5);
    // Ausgabezeile für diesen Auftrag
    result += ` ${play.name}: ${format(thisAmount/100)}
                        (${perf.audience} Plätze)\n`;
    totalAmount += thisAmount;
  }
  result += `Rechnungsbetrag ${format(totalAmount/100)}\n`;
  result += `Sie erhalten ${volumeCredits} Treuepunkte\n`;
  return result;
}
```

Die Ausführung dieses Codes mit den obigen Testdaten erzeugt die folgende Ausgabe:

```
Abrechnung für BigCo
  Hamlet: $650.00 (55 Plätze)
  As You Like It: $580.00 (35 Plätze)
  Othello: $500.00 (40 Plätze)
Rechnungsbetrag $1,730.00
Sie erhalten 47 Treuepunkte
```

1.2 Anmerkungen zum ersten Programm

Was halten Sie von dem Design dieses Programms? Ich würde zunächst einmal sagen, dass es in dieser Form akzeptabel ist – ein so kurzes Programm benötigt keine tiefgehende Struktur, um verständlich zu sein. Denken Sie jedoch an meinen früheren Hinweis, dass ich kurze Beispiele verwenden muss. Stellen Sie sich dieses Programm deutlich umfangreicher vor – vielleicht mehrere hundert Zeilen lang. Bei einer derartigen Größe ist eine einzelne Funktion schwer zu verstehen.

Ist eine Aussage über die Struktur des Programms in Anbetracht der Tatsache, dass es funktioniert, nicht lediglich ein ästhetisches Urteil? Die Abneigung gegenüber »hässlichem« Code? Dem Compiler ist es schließlich völlig gleichgültig, ob der Code hässlich oder schön ist. Aber wenn ich Änderungen am System vornehme, sind Menschen beteiligt – und Menschen ist es nicht egal, wie der Code aussieht. Es ist schwierig, ein System mit schlechtem Design zu verändern – weil es schwierig ist, herauszufinden, was geändert werden muss und wie diese Änderungen mit dem vorhandenen Code zusammenwirken, um das gewünschte Verhalten zu erzielen. Und wenn es schwierig ist herauszufinden, was geändert werden muss, dann ist die Wahrscheinlichkeit groß, dass mir Fehler unterlaufen und ich Bugs verursache.

Wenn ich Änderungen an einem Programm vornehmen muss, das aus mehreren Hundert Codezeilen besteht, ist es mir daher lieber, wenn es aus einer Reihe von Funktionen und anderen Programmelementen aufgebaut ist, die es mir ermöglichen, die Funktionsweise des Programms leichter zu verstehen. Wenn das Programm keine Struktur aufweist, fällt es mir für gewöhnlich leichter, das Programm zunächst einmal zu strukturieren und erst dann die erforderlichen Änderungen vorzunehmen.

Tipp

Wenn Sie ein Programm um ein bestimmtes Feature erweitern möchten, der Code jedoch nicht in geeigneter Weise strukturiert ist, sollten Sie zunächst ein Refactoring des Programms vornehmen, um es zu vereinfachen, und anschließend das Feature hinzufügen.

In diesem Fall wünschen die Anwender eine Reihe von Änderungen. Zunächst einmal soll die Abrechnung auch in HTML ausgegeben werden. Überlegen Sie sich, welche Auswirkungen diese Änderung haben würde. Ich müsste bei jedem Befehl, der einen String hinzufügt, eine bedingte Anweisung verwenden. In Anbetracht der Tatsache, dass die Komplexität der Funktion dadurch erheblich zunimmt, ziehen es die meisten vor, die Funktion zu kopieren

und so zu ändern, dass sie HTML ausgibt. Eine Kopie zu erstellen ist zwar nicht besonders mühsam, kann zukünftig aber für eine Vielzahl an Schwierigkeiten sorgen. Jede Änderung des Abrechnungsverfahrens würde mich dazu zwingen, beide Methoden zu aktualisieren – und zu gewährleisten, dass beide Methoden in übereinstimmender Weise geändert werden. Wenn ich ein Programm schreibe, das sich niemals ändern wird, ist diese Art von Copy & Paste unproblematisch. Schreibe ich hingegen ein langlebiges Programm, kann diese Redundanz gefährlich werden.

Das führt mich zu einer zweiten Änderung. Die Schauspieler möchten weitere Arten von Theaterstücken aufführen. Sie planen, ihrem Repertoire Schäferspiele, Pastoralen, Tragikomödien, Einakter usw. hinzuzufügen. Sie haben noch nicht entschieden, was genau sie eigentlich wollen und wann sie es umsetzen werden. Die Änderungen werden sich jedoch auf das Abrechnungsverfahren und die Berechnung der Treuepunkte auswirken. Im Grunde genommen spielt es keine Rolle, welchen Spielplan sie vorlegen, denn als erfahrener Entwickler kann ich davon ausgehen, dass er in den kommenden sechs Monaten ein weiteres Mal geändert wird. Und wenn neue Features implementiert werden sollen, geht es nicht um einige wenige, sondern um raue Mengen.

Wieder müssen Änderungen an der Klassifikation und dem Abrechnungsverfahren in der Methode `statement` vorgenommen werden. Wenn ich den Code von `statement` nach `htmlStatement` kopiere, muss ich gewährleisten, dass alle Änderungen an den beiden Methoden in übereinstimmender Weise vorgenommen werden. Wenn die Komplexität des Abrechnungsverfahrens zunimmt, wird es darüber hinaus schwieriger, herauszufinden, wo die Änderungen vorgenommen werden müssen, und sie fehlerfrei durchzuführen.

Ich möchte an dieser Stelle betonen, dass es genau diese Änderungen sind, die ein Refactoring erforderlich machen. Wenn der Code funktioniert und nie geändert werden muss, ist es absolut in Ordnung, ihn nicht anzutasten. Es wäre zwar schön, ihn zu verbessern, aber wenn niemand den Code verstehen muss, verursacht er auch keine Probleme. Sobald jedoch jemand die Funktionsweise des Codes verstehen muss und Schwierigkeiten hat, diese nachzuvollziehen, müssen Sie etwas unternehmen.

1.3 Der erste Schritt des Refactorings

Wenn ich ein Refactoring vornehme, ist der erste Schritt stets der gleiche. Ich muss mich vergewissern, dass für den betreffenden Codeabschnitt verlässliche Tests existieren. Die Tests sind unverzichtbar, denn obwohl ich Refactorings auf eine strukturierte Weise durchführe, die fast alle Möglichkeiten dafür ausschließt, dass ich Bugs verursache, bin ich auch nur ein Mensch, und mir unterlaufen Fehler. Je umfangreicher ein Programm ist, desto wahrscheinlicher wird es, dass meine Änderungen versehentlich dazu führen, dass irgendetwas nicht mehr funktioniert. Im digitalen Zeitalter ist Fragilität gleichbedeutend mit Software.

Die Methode `statement` liefert einen `String` zurück, daher erstelle ich einige Abrechnungen, indem ich sie mit verschiedenen Arten von Theaterstücken aufrufe und so den String für die Abrechnung erzeuge. Anschließend vergleiche ich die Ergebnisse mit verschiedenen, von Hand erzeugten Referenzstrings. Zur Einrichtung dieser Tests verwende ich ein Test-Framework, sodass ich sie durch einen einfachen Tastendruck in meiner Entwicklungsumgebung ausführen kann. Die Ausführung der Tests dauert nur ein paar Sekunden, und wie Sie sehen werden, führe ich sie häufig aus.

Ein wichtiger Bestandteil der Tests ist die Art und Weise, wie die Ergebnisse dargestellt werden. Sie sind entweder grün, was bedeutet, dass alle Strings mit den Referenzstrings übereinstimmen, oder rot, in Form einer Liste von aufgetretenen Fehlern – die Zeilen, die ein abweichendes Ergebnis enthalten. Die Tests sind also selbsttestend, und das ist unverzichtbar. Anderenfalls müsste ich die Testergebnisse am Schreibtisch von Hand mit den Referenzwerten vergleichen, und das würde mich stark verlangsamen. Aktuelle Test-Frameworks verfügen über alle notwendigen Features, um selbsttestende Tests zu schreiben.

Tipp

Vergewissern Sie sich, dass Ihnen verlässliche Tests zur Verfügung stehen, bevor Sie das Refactoring durchführen. Diese Tests müssen selbsttestend sein.

Bei der Durchführung des Refactorings verlasse ich mich auf die Tests. Ich stelle sie mir als Bug-Detektoren vor, die mich vor meinen eigenen Fehlern schützen. Um mein Ziel zu erreichen, muss ich es zweimal formulieren, sowohl im Code als auch im Test. Ich müsste einen Fehler an beiden Stellen und auf übereinstimmende Weise begehen, um den Detektor zu umgehen. Durch diese doppelte Überprüfung verringere ich die Wahrscheinlichkeit, etwas falsch zu machen. Es kostet zwar etwas Zeit, die Tests zu erstellen, das macht die beim Debugging eingesparte Zeit jedoch mehr als wett. Dieser Teil ist bei der Durchführung eines Refactorings von so großer Bedeutung, dass ich ihm ein eigenes Kapitel gewidmet habe (Kapitel 4, *Tests erstellen*).

1.4 Aufteilung der statement-Funktion

Beim Refactoring einer umfangreichen Funktion wie dieser versuche ich im Geiste bestimmte Punkte zu identifizieren, die verschiedene Teile des Gesamtverhaltens voneinander trennen. Der erste Codeabschnitt, der mir ins Auge fällt, ist die in der Mitte befindliche `switch`-Anweisung.

```
function statement (invoice, plays) {
  let totalAmount = 0;
  let volumeCredits = 0;
  let result = `Abrechnung für ${invoice.customer}\n`;
  const format = new Intl.NumberFormat("en-US",
                       { style: "currency", currency: "USD",
                         minimumFractionDigits: 2 }).format;
  for (let perf of invoice.performances) {
    const play = plays[perf.playID];
    let thisAmount = 0;

    switch (play.type) {
    case "tragedy":
      thisAmount = 40000;
      if (perf.audience > 30) {
```

```
        thisAmount += 1000 * (perf.audience - 30);
      }
      break;
    case "comedy":
      thisAmount = 30000;
      if (perf.audience > 20) {
      thisAmount += 10000 + 500 * (perf.audience - 20);
      }
      thisAmount += 300 * perf.audience;
      break;
    default:
      throw new Error(`Unbekannnter Typ: ${play.type}`);
    }

    // Treuepunkte hinzufügen
    volumeCredits += Math.max(perf.audience - 30, 0);
    // Zusatzpunkte für jeweils 10 Komödienbesucher
    if ("comedy" === play.type) volumeCredits +=
                              Math.floor(perf.audience / 5);
    // Ausgabezeile für diesen Auftrag
    result += ` ${play.name}: ${format(thisAmount/100)}
                       (${perf.audience} Plätze)\n`;
    totalAmount += thisAmount;
  }
  result += `Rechnungsbetrag ${format(totalAmount/100)}\n`;
  result += `Sie erhalten ${volumeCredits} Treuepunkte\n`;
  return result;
}
```

Wenn ich diesen Abschnitt betrachte, komme ich zu dem Ergebnis, dass er die Kosten für eine Vorstellung berechnet. Diese Schlussfolgerung ist eine Erkenntnis über den Code. Ward Cunningham aber würde sagen, dass sich diese Erkenntnis in meinem Kopf befindet – eine bekanntermaßen sehr flüchtige Form eines Speichers. Ich muss diese Erkenntnis dauerhaft speichern, indem ich sie von meinem Kopf zurück in den Code selbst übertrage. Auf diese Weise verrät mir der Code seine Funktionsweise, wenn ich später an diese Stelle zurückkehre – ich muss sie nicht erneut herausfinden.

Diese Erkenntnis wird in den Code übertragen, indem der zugehörige Abschnitt zu einer eigenen Funktion umgewandelt wird, die entsprechend ihrer Aufgabe benannt wird – also etwa `amountFor(aPerformance)`. Wenn ich wie hier einen Codeabschnitt in eine Funktion umwandeln möchte, verwende ich dafür ein Verfahren, das die Wahrscheinlichkeit minimiert, dass ich diese Umwandlung falsch angehe. Ich habe dieses Verfahren dokumentiert und *Funktion extrahieren* (Abschnitt 6.1) genannt, damit man leicht darauf verweisen kann.

Als Erstes muss ich in dem Codeabschnitt nach Variablen suchen, die nicht mehr zum Gültigkeitsbereich gehören, sobald sich der Code in einer eigenen Funktion befindet. In diesem Fall gibt es davon drei: `perf`, `play` und `thisAmount`. Die ersten beiden werden vom extrahierten Code zwar verwendet, aber nicht verändert, also kann ich sie als Parameter übergeben. Bei Variablen, die verändert werden, muss ich ein wenig mehr Aufwand betreiben. Hier gibt es nur eine, also kann ich sie zurückgeben. Zudem kann ich ihre Initialisierung im extrahierten Code vornehmen. Damit ergibt sich Folgendes:

```
function statement...
function amountFor(perf, play) {
  let thisAmount = 0;
  switch (play.type) {
  case "tragedy":
    thisAmount = 40000;
    if (perf.audience > 30) {
      thisAmount += 1000 * (perf.audience - 30);
    }
    break;
  case "comedy":
    thisAmount = 30000;
    if (perf.audience > 20) {
      thisAmount += 10000 + 500 * (perf.audience - 20);
    }
    thisAmount += 300 * perf.audience;
    break;
  default:
      throw new Error(`Unbekannter Typ: ${play.type}`);
  }
  return thisAmount;
}
```

Wenn ich eine kursiv gedruckte Überschrift wie

```
function einName...
```

verwende, bedeutet dies, dass sich der nachfolgende Code innerhalb des Gültigkeitsbereichs der angegebenen Funktion, Datei oder Klasse befindet. Für gewöhnlich gibt es noch weiteren Code, der zu diesem Gültigkeitsbereich gehört, der allerdings nicht abgedruckt wird, weil er für das aktuelle Thema nicht von Bedeutung ist.

Die Funktion `statement` ruft ihrerseits nun diese Funktion auf, um `thisAmount` einen Wert zuzuweisen:

```
Oberste Ebene...
function statement (invoice, plays) {
  let totalAmount = 0;
  let volumeCredits = 0;
  let result = `Abrechnung für ${invoice.customer}\n`;
  const format = new Intl.NumberFormat("en-US",
                       { style: "currency", currency: "USD",
                         minimumFractionDigits: 2 }).format;
  for (let perf of invoice.performances) {
    const play = plays[perf.playID];
    let thisAmount = amountFor(perf, play);

  // Treuepunkte hinzufügen
  volumeCredits += Math.max(perf.audience - 30, 0);
  // Zusatzpunkte für jeweils 10 Komödienbesucher
  if ("comedy" === play.type) volumeCredits +=
                              Math.floor(perf.audience / 5);
  // Ausgabezeile für diesen Auftrag
  result += ` ${play.name}: ${format(thisAmount/100)}
                                ($${perf.audience} Plätze)\n`;
  totalAmount += thisAmount;
}
result += `Rechnungsbetrag ${format(totalAmount/100)}\n`;
result += `Sie erhalten ${volumeCredits} Treuepunkte\n`;
return result;
```

Sofort nach Durchführen dieser Änderung kompiliere ich den Code und führe die Tests aus, um zu prüfen, ob irgendetwas nicht mehr funktioniert. Es ist eine wichtige Angewohnheit, nach jedem Refactoring, mag es auch noch so einfach sein, zu testen. Man macht sehr leicht Fehler – jedenfalls geht es mir so. Sollte mir also ein Fehler unterlaufen, muss ich nur eine kleine Änderung berücksichtigen, um ihn zu finden, wenn ich gewissenhaft nach jeder Änderung die Tests durchführe. Die Fehlersuche und -behebung wird dadurch sehr vereinfacht. Das ist das Entscheidende beim Refactoring: kleine Änderungen vornehmen und nach jeder Änderung testen. Wenn ich versuche, zu große Änderungen umzusetzen, zwingt mich ein Fehler zu einem kniffligen und langwierigen Debugging, das viel Zeit in Anspruch nehmen kann. Kleine Änderungen, die schnelles Feedback ermöglichen, sind der Schlüssel dazu, ein solches Durcheinander zu vermeiden.

Hinweis

Mit dem Begriff »kompilieren« ist hier gemeint, die notwendigen Schritte zu unternehmen, um den JavaScript-Code auszuführen. Da JavaScript direkt ausführbar ist, brauchen ggf. keine weiteren Schritte zu erfolgen. In manchen Fällen ist es jedoch erforderlich, den Code in ein Ausgabeverzeichnis zu verschieben oder einen sprachspezifischen Compiler wie Babel (`https://babeljs.io`) anzuwenden.

Tipp

Das Refactoring ändert Programme in kleinen Schritten, sodass es leichtfällt, einen Bug im Code aufzuspüren, wenn Sie einen Fehler begehen.

Da es sich hier um JavaScript handelt, kann ich `amountFor` als verschachtelte Funktion in `statement` unterbringen. Das hat den Vorteil, dass ich keine Daten aus dem Gültigkeitsbereich der umgebenden Funktion an die neu extrahierte Funktion übergeben muss. In diesem Fall ergibt sich dadurch kein Unterschied, aber immerhin gibt es ein Problem weniger, das gehandhabt werden muss.

Die Tests werden bestanden, also besteht der nächste Schritt darin, die Änderungen in meine lokale Versionsverwaltung einzuchecken. Ich verwende eine Versionsverwaltung wie git oder Mercurial, die es mir ermöglicht, private Commits vorzunehmen. Nach jeder erfolgreichen Durchführung eines Refactorings checke ich die Änderungen ein, sodass ich mühelos zu einer funktionierenden Version zurückkehren kann, falls ich später etwas durcheinanderbringen sollte. Die gesammelten Änderungen fasse ich dann zu umfassenderen Commits zusammen, bevor ich sie in einem gemeinsam genutzten Repository einchecke.

Funktion extrahieren (Abschnitt 6.1) ist ein Refactoring, das häufig automatisiert wird. Wenn ich in Java programmieren würde, hätte ich instinktiv die Tastenkombination in meiner IDE gedrückt, die dieses Refactoring durchführt. Für JavaScript gibt es aktuell noch keine so ausgereifte Unterstützung für dieses Refactoring, daher führe ich es von Hand durch. Das ist zwar nicht schwer, allerdings muss ich bei den Variablen mit lokalem Gültigkeitsbereich sehr aufmerksam sein.

Nach der Verwendung von *Funktion extrahieren* (Abschnitt 6.1) sehe ich mir den Code an, um zu überprüfen, ob es irgendeine schnelle und einfache Möglichkeit gibt, die extrahierte Funktion verständlicher zu machen. Zunächst einmal benenne ich einige Variablen um, damit die Bezeichnungen aussagekräftiger sind. Aus `thisAmount` wird beispielsweise `result`.

```
function statement...
function amountFor(perf, play) {
  let result = 0;
  switch (play.type) {
  case "tragedy":
    result = 40000;
    if (perf.audience > 30) {
      result += 1000 * (perf.audience - 30);
    }
    break;
  case "comedy":
    result = 30000;
    if (perf.audience > 20) {
      result += 10000 + 500 * (perf.audience - 20);
    }
```

```
    result += 300 * perf.audience;
    break;
  default:
      throw new Error(`Unbekannter Typ: ${play.type}`);
  }
  return result;
}
```

Es gehört zu meinem Programmierstil, als Bezeichnung für den Rückgabewert einer Funktion immer »result« zu verwenden. Auf diese Weise ist mir jederzeit klar, welche Rolle die Variable einnimmt. Und wieder kompiliere ich, führe die Tests aus und checke den Code ein. Anschließend geht es mit dem ersten Argument weiter.

```
function statement...
function amountFor(aPerformance, play) {
  let result = 0;
  switch (play.type) {
  case "tragedy":
    result = 40000;
    if (aPerformance.audience > 30) {
      result += 1000 * (aPerformance.audience - 30);
    }
    break;
  case "comedy":
    result = 30000;
    if (aPerformance.audience > 20) {
      result += 10000 + 500 * (aPerformance.audience - 20);
    }
    result += 300 * aPerformance.audience;
    break;
  default:
      throw new Error(`Unbekannter Typ: ${play.type}`);
  }
  return result;
}
```

Auch diese Bearbeitung folgt wieder meinem eigenen Programmierstil. Bei einer dynamisch typisierten Sprache wie JavaScript ist es sinnvoll, die Datentypen im Auge zu behalten – deshalb enthält meine Standardbezeichnung für einen Parameter den Namen des Typs. Außerdem verwende ich einen unbestimmten Artikel, sofern es keine speziellen Informationen über die Rolle des Parameters gibt, die über die Bezeichnung ausgedrückt werden sollte. Ich habe diese Konvention von Kent Beck (Kent Beck: *Smalltalk Best Practice Patterns*. Addison-Wesley, 1997, ISBN 013476904X) übernommen und halte sie für sehr hilfreich.

Tipp

Code schreiben, den ein Computer versteht, kann jeder. Gute Programmierer schreiben Code, den Menschen verstehen.

Ist diese Umbenennung die Mühe wert? Unbedingt. Guter Code sollte deutlich ausdrücken, was er bewirkt, und Variablennamen tragen ganz entscheidend zu verständlichem Code bei. Zögern Sie nie, Bezeichnungen zu ändern, um die Verständlichkeit zu verbessern. Mit geeigneten Suchen-und-Ersetzen-Funktionen ist das für gewöhnlich ganz einfach. Tests und statisch typisierte Programmiersprachen heben die Stellen im Code hervor, die Sie womöglich übersehen haben. Mit automatisierten Refactoring-Tools ist es dann geradezu trivial, selbst sehr häufig verwendete Funktionen umzubenennen.

Als Nächstes sollte bei der Umbenennung der Bezeichnungen der Parameter `play` berücksichtigt werden, dem allerdings ein anderes Schicksal bevorsteht.

1.4.1 Entfernen der Variable play

Da ich mir Gedanken über die Parameter für `amountFor` mache, betrachte ich deren Ursprung. `aPerformance` gehört zur Schleife und ändert sich dementsprechend bei jedem Durchlauf. `play` wird dagegen anhand der jeweiligen Vorstellung berechnet, daher ist es überhaupt nicht nötig, diesen Wert als Parameter zu übergeben – ich kann ihn innerhalb von `amountFor` einfach neu berechnen. Beim Aufteilen einer langen Funktion versuche ich, Variablen wie `play` zu eliminieren, weil temporäre Variablen zahlreiche Bezeichnungen mit lokalem Gültigkeitsbereich erzeugen, die das Extrahieren von Code verkomplizieren. Das hier verwendete Refactoring heißt *Temporäre Variable durch Abfrage ersetzen* (Abschnitt 7.4).

Ich extrahiere zunächst die rechte Seite der Zuweisung und mache daraus eine eigene Funktion.

```
function statement...
function playFor(aPerformance) {
  return plays[aPerformance.playID];
}

Oberste Ebene...
function statement (invoice, plays) {
  let totalAmount = 0;
  let volumeCredits = 0;
  let result = `Abrechnung für ${invoice.customer}\n`;
  const format = new Intl.NumberFormat("en-US",
                       { style: "currency", currency: "USD",
                         minimumFractionDigits: 2 }).format;
  for (let perf of invoice.performances) {
    const play = playFor(perf);
    let thisAmount = amountFor(perf, play);
```

```
  // Treuepunkte hinzufügen
  volumeCredits += Math.max(perf.audience - 30, 0);
  // Zusatzpunkte für jeweils 10 Komödienbesucher
  if ("comedy" === play.type) volumeCredits +=
                              Math.floor(perf.audience / 5);
  // Ausgabezeile für diesen Auftrag
  result += ` ${play.name}: ${format(thisAmount/100)}
                                ($${perf.audience} Plätze)\n`;
  totalAmount += thisAmount;
}
result += `Rechnungsbetrag ${format(totalAmount/100)}\n`;
result += `Sie erhalten ${volumeCredits} Treuepunkte\n`;
return result;
```

Ich kompiliere, teste, checke ein und verwende anschließend das Refactoring *Variable inline platzieren* (Abschnitt 6.4).

```
Oberste Ebene...
function statement (invoice, plays) {
  let totalAmount = 0;
  let volumeCredits = 0;
  let result = `Abrechnung für ${invoice.customer}\n`;
  const format = new Intl.NumberFormat("en-US",
                        { style: "currency", currency: "USD",
                          minimumFractionDigits: 2 }).format;
  for (let perf of invoice.performances) {
    const play = plays[perf.playID];
    let thisAmount = amountFor(perf, playFor(perf));

  // Treuepunkte hinzufügen
  volumeCredits += Math.max(perf.audience - 30, 0);
  // Zusatzpunkte für jeweils 10 Komödienbesucher
  if ("comedy" === playFor(perf).type) volumeCredits +=
                              Math.floor(perf.audience / 5);
  // Ausgabezeile für diesen Auftrag
  result += `${playFor(perf).name}: ${format(thisAmount/100)}
                                ($${perf.audience} Plätze)\n`;
  totalAmount += thisAmount;
}
result += `Rechnungsbetrag ${format(totalAmount/100)}\n`;
```

```
result += `Sie erhalten ${volumeCredits} Treuepunkte\n`;
return result;
```

Ich kompiliere, teste, checke ein. Dank der inline platzierten Variablen kann ich *Funktionsdeklaration ändern* (Abschnitt 6.5) auf `amountFor` anwenden, um den Parameter `play` zu entfernen. Ich erledige diese Aufgabe in zwei Schritten. Zunächst verwende ich innerhalb von `amountFor` die neue Funktion:

```
function statement...
function amountFor(aPerformance, play) {
  let result = 0;
  switch (playFor(aPerformance).type) {
  case "tragedy":
    result = 40000;
    if (aPerformance.audience > 30) {
      result += 1000 * (aPerformance.audience - 30);
    }
    break;
  case "comedy":
    result = 30000;
    if (aPerformance.audience > 20) {
      result += 10000 + 500 * (aPerformance.audience - 20);
    }
    result += 300 * aPerformance.audience;
    break;
  default:
      throw new Error(`Unbekannnter Typ:
                      ${playFor(aPerformance).type}`);
  }
  return result;
}
```

Ich kompiliere, teste, checke ein und lösche den Parameter.

```
Oberste Ebene...
function statement (invoice, plays) {
  let totalAmount = 0;
  let volumeCredits = 0;
  let result = `Abrechnung für ${invoice.customer}\n`;
  const format = new Intl.NumberFormat("en-US",
                        { style: "currency", currency: "USD",
                          minimumFractionDigits: 2 }).format;
```

```
  for (let perf of invoice.performances) {
    let thisAmount = amountFor(perf~~, playFor(perf)~~);

  // Treuepunkte hinzufügen
  volumeCredits += Math.max(perf.audience - 30, 0);
  // Zusatzpunkte für jeweils 10 Komödienbesucher
  if ("comedy" === playFor(perf).type) volumeCredits +=
                              Math.floor(perf.audience / 5);
  // Ausgabezeile für diesen Auftrag
  result += `${playFor(perf).name}:
                             ${format(thisAmount/100)}
                            (${perf.audience} Plätze)\n`;
  totalAmount += thisAmount;
}
result += `Rechnungsbetrag ${format(totalAmount/100)}\n`;
result += `Sie erhalten ${volumeCredits} Treuepunkte\n`;
return result;

function statement...
function amountFor(aPerformance~~, play~~) {
  let result = 0;
  switch (playFor(aPerformance).type) {
  case "tragedy":
    result = 40000;
    if (aPerformance.audience > 30) {
      result += 1000 * (aPerformance.audience - 30);
    }
    break;
  case "comedy":
    result = 30000;
    if (aPerformance.audience > 20) {
      result += 10000 + 500 * (aPerformance.audience - 20);
    }
    result += 300 * aPerformance.audience;
    break;
  default:
      throw new Error(`Unbekannnter Typ:
                      ${playFor(aPerformance).type}`);
  }
  return result;
}
```

Und wieder kompilieren, testen, einchecken.

Dieses Refactoring beunruhigt einige Programmierer. Vorher wurde der Code zum Abrufen der Vorstellung (`play`) einmal pro Schleifendurchlauf aufgerufen, jetzt erfolgt der Aufruf dreimal. Auf die Wechselwirkungen zwischen Refactoring und Performance komme ich später noch zu sprechen. Fürs Erste stelle ich hier nur fest, dass diese Änderung höchstwahrscheinlich keine merkliche Auswirkung auf die Performance haben wird. Und selbst wenn es so wäre, ist es viel einfacher, die Performance einer gut strukturierten Codebasis zu verbessern.

Der größte Nutzen des Entfernens lokaler Variablen besteht darin, dass das Extrahieren von Code viel einfacher wird, weil man sich weniger um den lokalen Gültigkeitsbereich kümmern muss. Tatsächlich entferne ich lokale Variablen für gewöhnlich, bevor ich Code extrahiere.

Nun bin ich mit den Argumenten der Funktion `amountFor` fertig und sehe mir an, wo die Funktion aufgerufen wird. Sie wird dazu verwendet, einer temporären Variable einen Wert zuzuweisen, die nie wieder aktualisiert wird, deshalb platziere ich sie inline.

```
Oberste Ebene...
function statement (invoice, plays) {
  let totalAmount = 0;
  let volumeCredits = 0;
  let result = `Abrechnung für ${invoice.customer}\n`;
  const format = new Intl.NumberFormat("en-US",
                       { style: "currency", currency: "USD",
                         minimumFractionDigits: 2 }).format;
  for (let perf of invoice.performances) {

    // Treuepunkte hinzufügen
    volumeCredits += Math.max(perf.audience - 30, 0);
    // Zusatzpunkte für jeweils 10 Komödienbesucher
    if ("comedy" === playFor(perf).type) volumeCredits +=
                             Math.floor(perf.audience / 5);
    // Ausgabezeile für diesen Auftrag
    result += `${playFor(perf).name}:
                             ${format(amountFor(perf)/100)}
                            (${perf.audience} Plätze)\n`;
    totalAmount += amountFor(perf);
  }
  result += `Rechnungsbetrag ${format(totalAmount/100)}\n`;
  result += `Sie erhalten ${volumeCredits} Treuepunkte\n`;
  return result;
```