

Clean Code Kochbuch

Rezepte für gutes Code-Design und
bessere Softwarequalität

» Hier geht's
direkt
zum Buch

DIE LESEPROBE

Deklarativer Code

Bei Software ist Verhalten entscheidend. Es ist das, worauf die Benutzer angewiesen sind. Benutzer schätzen es, wenn wir ein Verhalten hinzufügen (vorausgesetzt, es ist das, was sie wirklich wollten). Wenn wir jedoch ein Verhalten, auf das sie angewiesen sind, ändern oder entfernen (Fehler einführen), vertrauen sie uns nicht mehr.

– Michael Feathers, *Working Effectively with Legacy Code*
(dt. *Effektives Arbeiten mit Legacy Code: Refactoring und Testen bestehender Software*)

Einführung

Als deklarativer Code wird Programmcode bezeichnet, der beschreibt, *was* ein Programm tun soll, anstatt die Schritte zu spezifizieren, die das Programm zur Erledigung einer Aufgabe ausführen muss. Der Code konzentriert sich dabei also auf das gewünschte Ergebnis (das »Was«) und nicht auf den Prozess zur Erreichung dieses Ergebnisses (das »Wie«). Deklarativer Code ist im Vergleich zu imperativem Code, der die einzelnen Schritte angibt, die ein Programm ausführen soll, lesbarer und verständlicher. Zudem ist der Code prägnanter und fokussiert auf das Endergebnis, nicht auf die konkreten Details, wie dieses Ergebnis erreicht wird.

Deklarativer Code wird häufig in Programmiersprachen eingesetzt, die funktionale Programmierung unterstützen, ein Programmierparadigma, das die Verwendung von Funktionen zur Beschreibung der Berechnungen eines Programms betont. Beispiele für deklarative Programmiersprachen sind *SQL*, das für die Verwaltung von Datenbanken verwendet wird, und *HTML*, das zur Strukturierung und Formatierung von Dokumenten für das Web genutzt wird.

In der Softwareentwicklung herrscht eine gewisse Trägheit, die teilweise noch aus den Zeiten stammt, in denen Programme aus Zeit- und Platzgründen in Low-Level-Sprachen geschrieben werden mussten. Das ist heute nicht mehr der Fall. Moderne Compiler und virtuelle Maschinen sind leistungsfähiger denn je und überlassen den Entwicklern die wichtige Aufgabe, hochsprachlichen, deklarativen und sauberen Code zu schreiben.

Geltungsbereich wiederverwendeter Variablen begrenzen

Problem

Sie verwenden dieselbe Variable in verschiedenen Geltungsbereichen.

Lösung

Sie sollten dieselbe Variable nicht für verschiedene Zwecke verwenden, sondern für alle lokalen Variablen den minimal möglichen Geltungsbereich (und damit eine möglichst kurze Lebensdauer) festlegen.

Diskussion

Die Wiederverwendung von Variablen erschwert die Nachvollziehbarkeit von Geltungsbereichen und deren Grenzen und verhindert, dass Refactoring-Tools unabhängige Codeblöcke extrahieren können. Bei der Programmierung von Skripten ist es durchaus üblich, Variablen wiederzuverwenden. Nach einigen Kopieren-und-Einfügen-Operationen können umfangreiche, fortlaufende Blöcke entstehen. Die Hauptursache des Problems ist das Kopieren von Code. Sie sollten stattdessen das Rezept »Wiederholten Code entfernen« auf Seite 149 anwenden. Als Faustregel gilt, dass Sie den Geltungsbereich so weit wie möglich einschränken sollten, da ein zu umfangreicher Scope zu Verwirrung führt und die Fehlersuche erschwert.

Im folgenden Codebeispiel wird die Variable `total` wiederverwendet:

```
// Zeilensumme ausgeben.
double total = item.getPrice() * item.getQuantity();
System.out.println("Line total: " + total);

// Gesamtbetrag ausgeben.
total = order.getTotal() - order.getDiscount();
System.out.println( "Amount due: " + total );

// Variable "total" wird wiederverwendet.
```

Sie sollten den Geltungsbereich der Variablen eingrenzen und den Code in zwei separate Blöcke bzw. Funktionen aufteilen. Das können Sie erreichen, wenn Sie gemäß »Methode in ein Objekt extrahieren« auf Seite 159 vorgehen:

```
function printLineTotal() {
    double lineTotal = item.getPrice() * item.getQuantity();
    System.out.println("Line total: " + lineTotal);
}

function printAmountTotal() {
    double amountTotal = order.getTotal() - order.getDiscount();
    System.out.println("Amount due: " + amountTotal);
}
```

In der Regel sollten Sie Variablennamen nicht wiederverwenden. Verwenden Sie stattdessen lokale, spezifische und selbsterklärende Namen.

Verwandte Rezepte

- »Wiederholten Code entfernen« auf Seite 149
- »Methode in ein Objekt extrahieren« auf Seite 159
- »Überlange Methoden unterteilen« auf Seite 165
- »Überflüssige Variablen reduzieren« auf Seite 168



Absichtserklärender Code

Absichtserklärender Code (Intention-revealing Code) kommuniziert klar seinen Zweck oder seine Absicht an andere Entwickler, die ihn in der Zukunft lesen oder mit ihm arbeiten werden. Dadurch soll er verhaltensorientierter, deklarativer, lesbarer, verständlicher und wartbarer werden.

Code durch Aufteilung in Funktionen strukturieren

Problem

In Ihrem Code befinden sich längere Codeabschnitte, die durch Leerzeilen voneinander getrennt sind.

Lösung

Verwenden Sie »Methode in ein Objekt extrahieren« auf Seite 159, um durch Leerzeilen getrennte Codeblöcke sinnvoll zu strukturieren.

Diskussion

Kürzere Funktionen begünstigen die Lesbarkeit, verbessern die Wiederverwendbarkeit und folgen dem KISS-Prinzip. Hier ein Beispiel mit übermäßigem Gebrauch von Leerzeilen zur Trennung von Codeblöcken:

```
function translateFile() {
    $this->buildFilename();
    $this->readFile();
    $this->assertFileContentsOk(); // Und viele weitere Zeilen.

    // Eine Leerzeile signalisiert den Beginn eines weiteren Blocks.
    $this->translateHyperlinks();
    $this->translateMetadata();
    $this->translatePlainText();

    // Eine weitere Leerzeile signalisiert den Beginn noch eines Blocks.
    $this->generateStats();
    $this->saveFileContents(); // Und viele weitere Zeilen.
}
```

Indem Sie »Methode in ein Objekt extrahieren« auf Seite 159 anwenden, ergibt sich eine kürzere, besser strukturierte Version:

```
function translateFile() {  
    $this->readFileToMemory();  
    $this->translateContents();  
    $this->generateStatsAndSaveFileContents();  
}
```

Falls Sie einen Linter verwenden, können Sie diesen so einstellen, dass er Sie darauf hinweist, wenn Sie Leerzeilen verwenden oder Methoden zu lang werden. Leerzeilen als solche sind harmlos, bieten aber Anhaltspunkte, wie man den Code in kleinere logische Einheiten zerlegen kann. Falls Sie Ihren Code mit Kommentaren anstelle von (oder zusätzlich zu) Leerzeilen unterbrechen, ist dies ein Code-Smell, der nach einer Umstrukturierung verlangt (siehe »Kommentare innerhalb von Methoden entfernen« auf Seite 134).



Das KISS-Prinzip

Das Akronym *KISS* steht für »Keep It Simple, Stupid«. Das Prinzip besagt, dass Systeme am besten funktionieren, wenn man sie möglichst einfach hält und nicht überkompliziert. Einfachere Systeme sind leichter zu verstehen, zu verwenden und zu warten als komplexe Systeme, sodass die Wahrscheinlichkeit sinkt, dass sie versagen oder unerwartete Ergebnisse liefern.

Verwandte Rezepte

- »Kommentare innerhalb von Methoden entfernen« auf Seite 134
- »Methode in ein Objekt extrahieren« auf Seite 159
- »Überlange Methoden unterteilen« auf Seite 165

Siehe auch

Dieses Rezept wird in Robert Martins Buch *Clean Code* ausführlich erklärt.

Versionierte Methoden entfernen

Problem

Im Code kommen Methoden mit Versionsangaben wie `sort`, `sortOld`, `sort20210117`, `sortFirstVersion`, `workingSort` usw. vor.

Lösung

Entfernen Sie die Versionsbezeichnung aus dem Namen und verwenden Sie stattdessen eine Versionsverwaltung.

Diskussion

Über den Namen versionierte Funktionen beeinträchtigen die Lesbarkeit und Wartbarkeit. Sie sollten nur jeweils eine Arbeitsversion eines Artefakts (Klasse, Methode, Attribut) behalten und die Versionierung einer Versionsverwaltung überlassen. Wenn Sie Code haben, der versionierte Methoden wie diese verwendet:

```
findMatch()  
findMatch_new()  
findMatch_newer()  
findMatch_newest()  
findMatch_version2()  
findMatch_old()  
findMatch_working()  
findMatch_for_real()  
findMatch_20200229()  
findMatch_thisoneisnewer()  
findMatch_themostnewestone()  
findMatch_thisisit()  
findMatch_thisisit_for_real()
```

... sollten Sie alle Varianten der Methode durch eine einzige ersetzen:

```
findMatch()
```

Wie bei vielen anderen Entwurfsmustern können Sie eine interne Richtlinie erstellen und diese klar kommunizieren; oder Sie fügen automatische Regeln hinzu, um versionierte Methodenbezeichnungen anhand von Suchmustern zu identifizieren. In der Softwareentwicklung spielt das Management der zeitlichen Codeevolution immer eine Rolle. Zum Glück gibt es dafür heutzutage ausgereifte Werkzeuge.



Versionsverwaltung

Eine Versionsverwaltung (Version Control System) ist ein Werkzeug, mit dem Entwickler Änderungen am Quellcode eines Softwareprojekts nachverfolgen können. Dadurch können viele Entwickler gleichzeitig an derselben Codebasis arbeiten, was die Zusammenarbeit, das Roll-back von Änderungen und die Verwaltung verschiedener Codeversionen erleichtert. Derzeit ist Git das am meisten verwendete System.

Verwandte Rezepte

»Kommentare in Funktionsnamen umwandeln« auf Seite 133

Doppelte Verneinungen entfernen

Problem

Sie nutzen Methoden, die nach einer negativen Bedingung benannt sind.

Lösung

Verwenden Sie zur Benennung Ihrer Variablen, Methoden und Klassen immer positiv formulierte Bezeichnungen.

Diskussion

Bei diesem Rezept geht es vor allem um die Lesbarkeit: Beim Lesen negativer Bedingungen kann Ihr Gehirn auf eine falsche Fährte gelockt werden. Hier ein Beispiel für eine doppelte Verneinung:

```
if (!work.isNotFinished())
```

Wenn Sie es positiv formulieren:

```
if (work.isDone())
```

Sie können Ihren Linter anweisen, mit regulären Ausdrücken nach Zeichenketten oder Ausdrücken wie `!not` oder `!isNot` zu suchen und Warnungen auszugeben. Sie müssen der Testabdeckung vertrauen und sichere Umbenennungen und andere Refactorings erstellen.

Verwandte Rezepte

- »Übertriebene Raffinesse aus dem Code entfernen« auf Seite 155
- »Boolesche Variablen reifizieren« auf Seite 209
- »Rückgabe boolescher Werte für Bedingungsprüfungen vermeiden« auf Seite 220
- »Mit truthy-Werten umgehen« auf Seite 377

Siehe auch

»Remove Double Negative« auf Refactoring.com (<https://oreil.ly/bR1Sf>)

Falsch zugeordnete Verantwortlichkeiten verschieben

Problem

Methoden befinden sich in den falschen Objekten.

Lösung

Erstellen oder überladen Sie die richtigen Objekte, indem Sie mithilfe des MAPPER die passenden Stelle identifizieren.

Diskussion

Die geeigneten Objekte für bestimmte Zuständigkeiten zu finden, ist eine schwierige Aufgabe. Sie müssen die Frage beantworten: »Wer ist verantwortlich für ...?« Wenn Sie mit Personen außerhalb der Softwarewelt sprechen, können diese Ihnen Hinweise darauf geben, wo Sie die einzelnen Verantwortlichkeiten bzw. Zuständigkeiten platzieren sollten. Entwickler hingegen neigen dazu, Verhalten an sehr seltsamen Orten zu platzieren ... etwa in Hilfsfunktionen!

Hier einige Beispiele für die Zuständigkeit von `add`:

```
Number>>#add: a to: b
  ^ a + b

// Das ist in vielen Programmiersprachen normal, aber nicht im wirklichen Leben.
```

Hier ein anderer Ansatz:

```
Number>>#add: adder
  ^ self + adder

// Diese Variante kann in einigen Programmiersprachen nicht kompiliert werden,
// weil sie die Änderung bestimmter Verhaltensweisen in Basisklassen verbieten.
// Aber es ist der richtige Ort für diese Zuständigkeit.
```

Es gibt einige Sprachen, in denen man primitiven Typen Zuständigkeiten hinzufügen kann. Geschieht dies im geeigneten Objekt, findet man sie zuverlässig an der gleichen Stelle. Hier ein weiteres Beispiel, in dem die Konstante `PI` definiert wird:

```
class GraphicEditor {
  constructor() {
    this.PI = 3.14;
    // Sie sollten die Konstante nicht hier definieren.
  }

  pi() {
    return this.PI;
    // Liegt nicht in der Verantwortung dieses Objekts.
  }

  drawCircle(radius) {
    console.log("Drawing a circle with radius ${radius} " +
      "and circumference " + (2 * this.pi()) * radius);
  }
}
```

Wenn Sie die Zuständigkeit in ein Objekt `RealConstants` verschieben, vermeiden Sie die Wiederholung von Code:

```
class GraphicEditor {
  drawCircle(radius) {
    console.log("Drawing a circle with radius " + radius +
      " and circumference " + (2 * RealConstants.pi()) * radius));
  }
}
// Die Definition von PI liegt in der Zuständigkeit von RealConstants
// (oder Number oder Ähnlichem).
```



```
class RealConstants {
  pi() {
    return 3.14;
  }
}
```

Verwandte Rezepte

»Hilfsfunktionen und -klassen umbenennen und aufteilen« auf Seite 107

»Feature Envy vorbeugen« auf Seite 276

Explizite Iterationen ersetzen

Problem

Vermutlich sind Ihnen Schleifen bereits seit Ihren ersten Programmierschritten bekannt. Aber Enums und Iteratoren sind die nächste Generation, und Sie benötigen eine höhere Ebene der Abstraktion.

Lösung

Verwenden Sie bei der Iteration keine Indizes. Bevorzugen Sie Sammlungen höherer Ordnung.

Diskussion

Indizes verletzen oft die Kapselung und sind weniger deklarativ. Wenn eine Sprache dies unterstützt, sollten Sie `foreach()` oder Iteratoren höherer Ordnung bevorzugen. Sie können `yield()`, *Caches*, *Proxies*, *Lazy Loading* und vieles mehr verwenden, wenn Sie Ihre Implementierungsdetails verbergen.

Hier ein Beispiel, in dem eine strukturelle Iteration über den Index i durchgeführt wird:

```
for (let i = 0; i < colors.length; i++) {
  console.log(colors[i]);
}
```

Das folgende Beispiel ist deklarativer:

```
colors.forEach((color) => {
  console.log(color);
});
```

// Es werden Closures und Pfeilfunktionen verwendet.

Es gibt aber auch Ausnahmen. Wenn die Anwendungsdomäne verlangt, dass die Elemente (wie in Kapitel 2 definiert) auf natürliche Zahlen wie Indizes abgebildet werden, ist die erste Lösung (mit Index i) angemessen. Denken Sie daran, dass Sie

immer Analogien zur realen Welt finden müssen. Dieser Code-Smell fällt vielen Entwicklern nicht negativ auf, weil sie glauben, dass es sich hierbei eher um eine Spitzfindigkeit handelt. Bei der Erstellung von Clean Code geht es aber um diese wenigen deklarativen Dinge, die einen Unterschied machen können.

Verwandte Rezepte

»Abkürzungen ausschreiben« auf Seite 105

Entwurfsentscheidungen dokumentieren

Problem

Sie haben nicht-triviale Entscheidungen bezüglich Ihres Codes getroffen und müssen die Gründe dafür dokumentieren.

Lösung

Verwenden Sie deklarative und absichtserklärende Namen.

Diskussion

Sie sollten Ihre Entwurfs- oder Implementierungsentscheidungen deklarativ formulieren, indem Sie beispielsweise die Entscheidung extrahieren und ihr einen klaren, absichtserklärenden Namen geben. Vermeiden Sie die Verwendung von Kommentaren, da Kommentare »toter Code« sind, der leicht veraltet und nicht kompiliert wird. Seien Sie explizit bezüglich Ihrer Entscheidung oder wandeln Sie den Kommentar in eine Methode um. Manchmal stößt man auf willkürliche Regeln, die nicht so leicht überprüfbar sind. Wenn Sie beispielsweise keinen Test schreiben können, sollten Sie, anstatt einen Kommentar zu verwenden, eine Funktion mit einem besonders aussagekräftigen und deklarativen Namen versehen, um vor zukünftigen Änderungen zu warnen.

Hier ein Beispiel für eine nicht explizit kommunizierte Entwurfsentscheidung:

```
// Der Prozess muss mit mehr Speicher ausgeführt werden.  
set_memory("512k");  
  
run_process();
```

Die folgende Variante ist dagegen eindeutig und gibt einen Hinweis auf die Gründe für den höheren Speicherbedarf:

```
increase_memory_to_avoid_false_positives();  
run_process();
```

Code ist Prosa. Und Entwurfsentscheidungen sollten ihre Geschichte erzählen.

Verwandte Rezepte

»Kommentare in Funktionsnamen umwandeln« auf Seite 133

»Kommentare innerhalb von Methoden entfernen« auf Seite 134

Magische Zahlen durch Konstanten ersetzen

Problem

Sie verwenden eine Methode, die Berechnungen mit vielen Zahlen durchführt, ohne deren Bedeutung zu beschreiben.

Lösung

Vermeiden Sie die Verwendung *magischer Zahlen* ohne dazugehörige Erklärung. Sie kennen deren Quelle nicht und sollten sich davor hüten, sie zu ändern und Fehler zu provozieren.

Diskussion

Magische Zahlen verursachen Kopplung. Sie sind schwer zu testen und zu lesen. Geben Sie Konstanten semantische Namen (bedeutungsvoll und absichtserklärend) und ersetzen Sie sie durch Parameter, damit Sie sie mocken können (siehe »Mock-Objekte durch echte Objekte ersetzen« auf Seite 330). Eine Konstantendefinition ist oft ein anderes Objekt als der Nutzer dieser Konstante, und glücklicherweise können viele Linter Zahlлитerale in Attributen und Methoden erkennen.

Hier eine wohlbekannte Konstante:

```
function energy($mass) {  
    return $mass * (299792 ** 2);  
}
```

Wenn Sie das Beispiel umformulieren, erhalten Sie:

```
function energy($mass) {  
    return $mass * (LIGHT_SPEED_KILOMETERS_OVER_SECONDS ** 2);  
}
```

Verwandte Rezepte

»Variablen als veränderlich deklarieren« auf Seite 74

»Veränderliche Konstanten einfrieren« auf Seite 80

»Übertriebene Raffinesse aus dem Code entfernen« auf Seite 155

»Überflüssige Klammern entfernen« auf Seite 170

»Versteckte Annahmen explizit machen« auf Seite 263

»God Objects aufspalten« auf Seite 268

»Was« und »Wie« trennen

Problem

Ihr Code konzentriert sich mehr auf die interne Funktionsweise als auf das Ergebnis.

Lösung

Kümmern Sie sich nicht um Implementierungsdetails. Seien Sie deklarativ, nicht imperativ.

Diskussion

Die Namensgebung ist wichtig, um versehentliche Kopplungen zu vermeiden. In der Softwarebranche kann die Trennung von Zuständigkeiten (Separation of Concerns) eine schwierige Aufgabe sein, aber funktionale Software besitzt die Fähigkeit, dem Test der Zeit standzuhalten. Implementierungsorientierte Software führt dagegen zu Kopplung und ist schwieriger zu ändern.

Manchmal werden Änderungen in Kommentaren dokumentiert. Da Kommentare nur selten gepflegt werden, ist das jedoch keine gute Lösung (siehe »Kommentare in Funktionsnamen umwandeln« auf Seite 133). Wenn Sie ein Design wählen, das auf Anpassungsfähigkeit und die Vermittlung seiner Absichten ausgerichtet ist, überlebt Ihr Code länger und funktioniert besser.

In diesem Codebeispiel ist die durchzuführende Aktion an die ausstehende Aufgabe von `stepWork` gekoppelt:

```
class Workflow {
    moveToNextTransition() {
        // Die Geschäftsregel wird mit der zufälligen Implementierung gekoppelt.
        if (this.stepWork.hasPendingTasks()) {
            throw new Error('Preconditions are not met yet..');
        } else {
            this.moveToNextStep();
        }
    }
}
```

Hier eine bessere Lösung:

```
class Workflow {
    moveToNextTransition() {
        if (this.canMoveOn()) {
            this.moveToNextStep();
        } else {
            throw new Error('Preconditions are not met yet..');
        }
    }

    canMoveOn() {
        // Die Implementierung (das "Wie") wird hinter
        // dem "Was" verborgen.
    }
}
```

```
        return !this.stepWork.hasPendingTasks();
    }
}
```

Um vorzeitige Optimierung zu vermeiden, sollten Sie gute Namen wählen und bei Bedarf Vermittlungsebenen hinzufügen (siehe Kapitel 16). Das Argument, dass man damit Ressourcen verschwendet und über einen tiefen Einblick verfügen muss, ist nicht stichhaltig. Zudem kann jede moderne virtuelle Maschine diese zusätzlichen Aufrufe cachen oder inline ausführen.

Verwandte Rezepte

»Kommentare in Funktionsnamen umwandeln« auf Seite 133

»Isolierte Klassen umbenennen« auf Seite 313

Reguläre Ausdrücke dokumentieren

Problem

Sie verwenden magische reguläre Ausdrücke, die schwer zu verstehen sind.

Lösung

Zerlegen Sie Ihre komplexen regulären Ausdrücke in kürzere und deklarativere Einheiten.

Diskussion

Reguläre Ausdrücke schaden der Lesbarkeit und erschweren die Wart- und Testbarkeit. Sie sollten sie nur für die Validierung von Zeichenketten verwenden. Wenn Sie Objekte manipulieren müssen, wandeln Sie diese nicht in Strings um: Erstellen Sie stattdessen kleine Objekte gemäß »Small Objects erstellen« auf Seite 56.

Hier ist ein Beispiel für einen regulären Ausdruck, der nicht deklarativ ist:

```
val regex = Regex("^\\+(?:[0-9][ - ]?){6,14}[0-9a-zA-Z]$")
```

Diese Variante ist deklarativer und einfacher zu verstehen und zu debuggen:

```
val prefix = "\\+"
val digit = "[0-9]"
val space = "[ - ]"
val phoneRegex = Regex("^$prefix(?:$digit$space?){6,14}$digit$")
```

Reguläre Ausdrücke sind ein sinnvolles Werkzeug. Es gibt allerdings nicht viele Möglichkeiten, um automatisch fehlerhafte Verwendungen zu entdecken. Eine Whitelist könnte hilfreich sein. Reguläre Ausdrücke sind auch ein hervorragendes Werkzeug für die Validierung von Strings. Sie sollten sie deklarativ und ausschließlich für Zeichenketten verwenden. Gute Bezeichnungen sind sehr wichtig, um die

Bedeutung von Suchmustern verstehen zu können. Wenn Sie Objekte oder Hierarchien manipulieren müssen, sollten Sie dies mithilfe von Objekten tun, es sei denn, es liegt überzeugende, auf Benchmarks basierende Evidenz dafür vor, dass ein Verzicht darauf einen *deutlichen* Leistungsvorteil bietet.

Verwandte Rezepte

- »String-Validierungen in Objekte umwandeln« auf Seite 63
- »Übertriebene Raffinesse aus dem Code entfernen« auf Seite 155
- »Vorzeitige Optimierungen entfernen« auf Seite 248
- »Problematische reguläre Ausdrücke ersetzen« auf Seite 388

Yoda-Conditions neu formulieren

Problem

In Ihren Bedingungen platzieren Sie den zu erwartenden Wert auf der linken Seite des Ausdrucks.

Lösung

Formulieren Sie Ihre Bedingungen so, dass der überprüfende Variablenwert auf der linken Seite und der erwartete Wert auf der rechten Seite steht.

Diskussion

Die meisten Programmierer nennen die Variable oder den zu prüfenden Ausdruck auf der linken Seite und den zu vergleichenden Wert auf der rechten Seite. Das ist auch die richtige Reihenfolge für Assertions. Yoda-Conditions, die diese Reihenfolge umkehren, werden manchmal verwendet, um in Sprachen, die dies erlauben, versehentliche Zuweisungen anstelle von Vergleichen zu verhindern.

Hier ein Beispiel für eine Yoda-Condition:

```
if (42 == answerToLifeMeaning) {  
    // Verhindert eine versehentliche Zuweisung aufgrund eines Tippfehlers  
    // mit nur einem Gleichheitszeichen,  
    // da '42 = answerToLifeMeaning' immer ungültig ist.  
}
```

So sollte es nach der Umformulierung aussehen:

```
if (answerToLifeMeaning == 42) {  
    // Könnte allerdings mit 'answerToLifeMeaning = 42' verwechselt werden.  
}
```

Nennen Sie den konstanten Wert, gegen den geprüft werden soll, immer auf der rechten Seite des Vergleichs.

Verwandte Rezepte

»Argumente je nach Rolle bzw. Aufgabe benennen« auf Seite 122

Scherzhaft benannte Methoden umbenennen

Problem

Es liegt Code vor, der andere Menschen beleidigen oder verletzen könnte.

Lösung

Drücken Sie sich nicht informell aus und seien Sie nicht beleidigend. Seien Sie freundlich zu Ihrem Code und Ihren Lesern.

Diskussion

Sie müssen den Code auf professionelle Weise schreiben und aussagekräftige Namen verwenden. Unser Beruf hat natürlich eine kreative Seite. Vielleicht langweilen Sie sich manchmal und versuchen, den Code witzig zu formulieren. Das schadet allerdings nicht nur der Verständlichkeit des Codes, sondern auch Ihrem Ruf. Hier ein Beispiel für eine unprofessionelle Methodenbezeichnung:

```
function eradicateAndMurderAllCustomers();  
// Das ist unprofessionell und beleidigend.
```

Eine professionellere Benennung der Methode:

```
function deleteAllCustomers();  
// Das ist deklarativer und professionell.
```

Sie können eine Liste mit verbotenen und obszönen Wörtern erstellen und automatisch oder in Code-Reviews daraufhin prüfen. Namenskonventionen sollten allgemein gehalten sein und Bezeichnungen keinen kulturellen Jargon enthalten. Sie sollten Produktionscode so schreiben, dass zukünftige Entwickler (inklusive Ihres eigenen zukünftigen Ichs) ihn leicht verstehen können.

Verwandte Rezepte

»Abstrakte Namen ändern« auf Seite 115

Callback Hell vermeiden

Problem

Es gibt asynchronen Code mit Callbacks, die übermäßig verschachtelt sind, wodurch der Code schwer zu lesen und zu pflegen ist.

Lösung

Verarbeiten Sie keine Aufrufe mit Callbacks. Rufen Sie sie stattdessen sequenziell auf.

Diskussion

In der Callback Hell landen Sie, wenn Ihr Code mehrere Callbacks ineinander verschachtelt, was zu einer komplexen und schwer lesbaren Codestructur führt. Das ist häufig in JavaScript bei der asynchronen Programmierung zu beobachten, wenn eine Callback-Funktion als Argument an eine andere Funktion übergeben wird. Diese tiefe Verschachtelung erzeugt Code, der auch als Pyramide des Verderbens (Pyramid of Doom) bezeichnet wird.

Wenn Sie die innere Funktion aufrufen, gibt diese möglicherweise eine Funktion zurück, die einen Callback empfängt, was zu einer Kette von verschachtelten Callbacks führt, die schwer zu verfolgen und zu verstehen sind.

Hier ist ein kurzes Beispiel für diese »Callback-Hölle«:

```
asyncFunc1(function (error, result1) {
  if (error) {
    console.log(error);
  } else {
    asyncFunc2(function (error, result2) {
      if (error) {
        console.log(error);
      } else {
        asyncFunc3(function (error, result3) {
          if (error) {
            console.log(error);
          } else {
            // Der verschachtelte Callback setzt sich fort ...
          }
        });
      }
    });
  }
});
```

Sie können es so refaktorisieren:

```
function asyncFunc1() {
  return new Promise((resolve, reject) => {
    // Asynchroner Vorgang.
    // ...

    // Bei Erfolg:
    resolve(result1);

    // Bei Fehler:
    reject(error);
  });
}
```



```

function asyncFunc2() {
  return new Promise((resolve, reject) => {
    // Asynchroner Vorgang.
    // ...

    // Bei Erfolg:
    resolve(result2);

    // Bei Fehler:
    reject(error);
  });
}

async function performAsyncOperations() {
  try {
    const result1 = await asyncFunc1();
    const result2 = await asyncFunc2();
    const result3 = await asyncFunc3();

    // Mit weiteren Operationen fortfahren.
  } catch (error) {
    console.log(error);
  }
}

performAsyncOperations();

```

Sie können Promises und `async/await` verwenden, um dieses Problem zu lösen, damit der Code lesbarer wird und leichter zu debuggen ist.

Verwandte Rezepte

»Übertriebene Raffinesse aus dem Code entfernen« auf Seite 155

»Verschachtelten Pfeilcode refaktorisieren« auf Seite 219

Gute Fehlermeldungen formulieren

Problem

Sie müssen gute Fehlerbeschreibungen erstellen, sowohl für die Entwickler, die Ihren Code verwenden (und damit auch für sich selbst), als auch für die Endbenutzer.

Lösung

Verwenden Sie aussagekräftige Formulierungen und schlagen Sie Abhilfemaßnahmen vor. Es bringt viel, wenn Sie Ihren Nutzern mit Freundlichkeit begegnen.

Diskussion

Als Programmierer sind wir in den seltensten Fällen UX-Experten. Nichtsdestotrotz sollten Sie sinnvolle Fehlermeldungen verwenden, dabei an die Endbenutzer denken

und Meldungen mit klaren Exit-Aktionen versehen. Folgen Sie dabei dem Prinzip der geringsten Überraschung (siehe »Veränderliche Konstanten einfrieren« auf Seite 80) für Ihre Benutzer.

Das ist eine schlechte Fehlermeldung:

```
alert("Cancel the appointment?", "Yes", "No");

// Keine Konsequenzen und Maßnahmen.
// Die Optionen sind nicht eindeutig.
```

Ein aussagekräftiger Fehlerhinweis sähe so aus:

```
alert("Cancel the appointment? \n" +
      "You will lose all the history",
      "Cancel Appointment",
      "Keep Editing");

// Die Konsequenzen sind klar.
// Die Auswahlmöglichkeiten bieten Kontext.
```

Verschleiern Sie Fehlersituationen nicht durch die Ausgabe von nichtssagenden internen Domänenwerten, und unterscheiden Sie klar zwischen einer Null und einem Fehler. Betrachten Sie den folgenden Code, der einen Netzwerkfehler verbirgt und fälschlicherweise einen Saldo von 0 anzeigt, was beim Endbenutzer leichte Panik auslösen könnte:

```
def get_balance(address):
    url = "https://blockchain.info/q/addressbalance/" + address
    response = requests.get(url)
    if response.status_code == 200:
        return response.text
    else:
        return 0
```

Diese Fassung ist klarer und deutlicher:

```
def get_balance(address):
    url = "https://blockchain.info/q/addressbalance/" + address
    response = requests.get(url)
    if response.status_code == 200:
        return response.text
    else:
        raise BlockchainNotReachableError("Error reaching blockchain")
```



Deklarative Beschreibungen von Ausnahmefehlern

Die Beschreibungen von Ausnahmefehlern (Exceptions) sollten die Geschäftsregel und nicht den Fehler erwähnen. Eine gute Beschreibung lautet: »Die Zahl sollte zwischen 1 und 100 liegen.« Eine schlechte Beschreibung wäre: »Wert außerhalb des gültigen Bereichs.« Denn wo liegen die Grenzen des gültigen Bereichs?

Sie sollten bei Code-Reviews alle Fehler- und Ausnahmemeldungen prüfen und immer die Perspektive der Endbenutzer mitdenken, wenn Sie Ausnahmen auslösen oder Meldungen anzeigen.

Verwandte Rezepte

- »Nullobjekte erstellen« auf Seite 231
- »Geschäftslogik aus der Benutzeroberfläche entfernen« auf Seite 284
- »Keine Ausnahmen bei erwarteten Fällen auslösen« auf Seite 356
- »Rückgabecodes durch Ausnahmen ersetzen« auf Seite 359

Magische Korrekturen vermeiden

Problem

Sie verwenden Anweisungen, die in einigen Sprachen gültig und auf magische Weise interpretiert bzw. korrigiert werden, die aber expliziter sein und dem Fail-Fast-Prinzip entsprechen sollten.

Lösung

Entfernen Sie magische Korrekturen aus Ihrem Code.

Diskussion

Einige Sprachen kehren Probleme sozusagen unter den Teppich und nehmen magische Korrekturen und obskure Typumwandlungen vor, was gegen das Fail-Fast-Prinzip verstößt. Sie sollten sich eindeutig ausdrücken und alle Unklarheiten beseitigen. Ändern Sie diese Art von magischen Ausdrücken:

```
new Date(31, 02, 2020);  
  
1 + 'Hello';  
  
!3;  
  
// Das ist in den meisten Sprachen gültig.
```

Hier ist eine explizite Lösung:

```
new Date(31, 02, 2020);  
// Lösen Sie eine Ausnahme aus.  
  
1 + 'Hello';  
// Typkonflikt.  
  
!3;  
// Negation ist eine boolesche Operation.
```

In Abbildung 6-1 sehen Sie das unerwartete Ergebnis der Addition einer Zahl zu einer Zeichenkette, ein Vorgang, der in der realen Welt nicht vorkommt und im Code eine Ausnahme auslösen sollte.

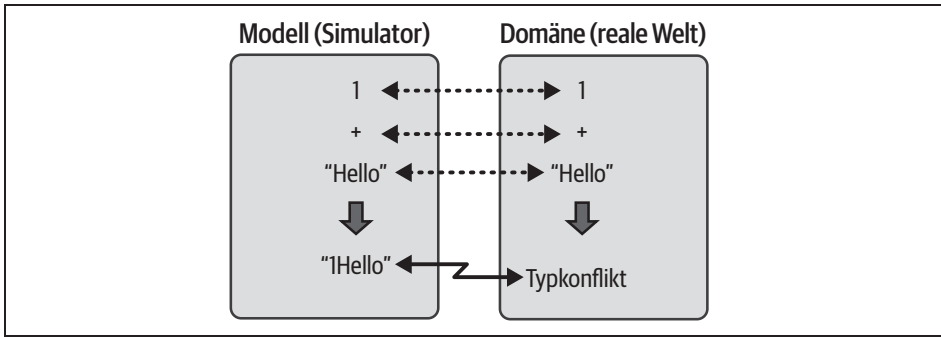


Abbildung 6-1: Die Ausführung der »+«-Methode führt zu unterschiedlichen Ergebnissen im Modell und in der realen Welt.

Viele dieser Probleme werden durch die Programmiersprachen selbst begünstigt. Formulieren Sie möglichst deklarativ und explizit. Verlassen Sie sich nicht auf sprachimmanente magische Lösungen (im Gegensatz zu rationalen). Viele Programmierer wollen besonders schlau erscheinen, indem sie bestimmte Sprachmerkmale ausnutzen. Das führt aber zu unnötig komplexem Code, der nur vortäuscht, besonders clever zu sein, tatsächlich aber das Gegenteil von Clean Code darstellt.

Verwandte Rezepte

»Übertriebene Raffinesse aus dem Code entfernen« auf Seite 155

»Mit truthy-Werten umgehen« auf Seite 377