Engineer

Refactoring

🛠 Codestruktur ändern, ohne das Verhalten zu ändern

Was es ist

Refactoring verändert die innere Struktur von Code, ohne zu ändern, was der Code für seine Nutzer tut. Namen, Funktionsgrenzen und die Anordnung von Modulen ändern sich; Eingaben, Ausgaben und Nebenwirkungen bleiben genau gleich. Das Verb lautet refaktorieren, und auch eine einzelne solche Änderung heißt Refactoring.

Martin Fowler definiert den Begriff in seinem Buch Refactoring und formulierte die Definition 2004 auf seiner Website erneut. Als Substantiv bezeichnet ein Refactoring eine Änderung der inneren Softwarestruktur, die Code leichter verständlich macht und die Kosten für Änderungen senkt, während sein von außen sichtbares Verhalten gleich bleibt. Als Verb bedeutet refaktorieren, eine Folge solcher Änderungen vorzunehmen und das äußere Verhalten währenddessen unverändert zu lassen. Beide Teile zählen. Das Ziel ist eine günstigere nächste Änderung. Die Bedingung ist, dass von außen niemand bemerken kann, dass etwas geschehen ist.

Fowler zieht eine klare Grenze: „Refactoring ist kein anderes Wort für das Aufräumen von Code“, schreibt er auf refactoring.com. Es bezeichnet eine bestimmte Technik. Restrukturierung verwendet er als weiter gefassten Begriff für die Neuordnung von Code mit beliebigen Mitteln, auch solchen, die das Verhalten nicht bei jedem Schritt erhalten. Zur Technik wird Refactoring durch seine Schrittgröße. Jeder Schritt ist eine kleine Umformung, die allein wenig bewirkt und bei der wenig schiefgehen kann. Das gesamte System funktioniert nach jedem Schritt weiter. Eine lange Folge kann dennoch eine große Strukturänderung ergeben.

Die Schritte haben Namen. Der Katalog auf refactoring.com begleitet die zweite Buchauflage und führt sie mit den früheren Namen auf, die sie ersetzen: Extract Function (eine Funktion extrahieren), auch als Extract Method bekannt, und die Umkehrung Inline Function (einen Funktionsaufruf durch den Funktionskörper ersetzen); Inline Variable (eine Variable durch ihren Ausdruck ersetzen); Rename Variable (eine Variable umbenennen); Move Function (eine Funktion verschieben); Split Phase (Phasen trennen) und Dutzende weitere. Ein gemeinsamer Name lässt einen Entwickler in zwei Wörtern sagen, was eine Änderung war. Wer sie prüft, kann sie mit einer bekannten Anleitung abgleichen.

Das Konzept für eine Entscheidung nutzen

Nutzen Sie „Was ist Refactoring?“ für eine konkrete Entscheidung. Wir können helfen, das Konzept mit dem System in Verbindung zu bringen, das Sie entwickeln oder einkaufen. Ein Team führt eine Idee von der Skizze bis zu einem Produkt, bei dem Ihre Kunden sich anmelden können, statt sie zwischen drei Dienstleistern weiterzureichen.

Unsere Leistungen entdecken Kontakt aufnehmen →

Die Annahmen hinter dem Begriff prüfen

Nutzen Sie die Erklärung und die verlinkten Arbeiten, um Fragen für Ihr Projekt zu formulieren. Wir können die Annahmen gemeinsam mit Ihnen prüfen, bevor ein Werkzeug ausgewählt wird. GIS-Kartenprodukt zu 10-fachem Umsatz geführt — 10x Umsatzwachstum durch die Leitung des Kartenprodukts

GIS-Kartenprodukt zu 10-fachem Umsatz geführt Kontakt aufnehmen →

Warum es wichtig ist

Erfolgreiche Software verändert sich weiter. Neue Funktionen und Korrekturen bauen aufeinander auf. Jede macht die nächste tendenziell etwas schwieriger, bis, wie Fowler es beschreibt, jede Erweiterung quälend lange dauert. Refactoring ist das Gegengewicht: Es formt den Code so um, dass Ergänzungen keine unnötige Komplexität mitbringen. Sein Nutzen bemisst sich an den Kosten der nächsten Änderung, nicht am Aussehen des Codes.

Deshalb behandelt Fowler es als Teil gewöhnlicher Programmierarbeit statt als Aufgabe im Projektplan. Er nennt drei Zeitpunkte. Vor einer neuen Funktion prüft er, ob der Code ihre Ergänzung einfach macht. Falls nicht, refaktoriert er zuerst. Das ist nach seiner Erfahrung meist schneller, als die Funktion hineinzuzwingen. Sobald die Funktion läuft, refaktoriert er das Ergebnis so, dass jemand nach einigen Wochen ohne Rätselraten wieder folgen kann. Muss er fremden Code zum Verständnis lesen, refaktoriert er ihn, damit die nächste Person nicht an derselben Stelle kämpfen muss. Der erste Fall hat einen eigenen Namen: vorbereitendes Refactoring. Die zugehörige Regel schreibt Fowler Kent Beck zu: Vor jeder Änderung zuerst den Code so umformen, dass die Änderung einfach wird, was der schwierige Teil sein kann, und sie erst dann vornehmen.

Refactoring hält Risiken außerdem auf eine Weise klein, die beim Ergänzen von Funktionen nicht möglich ist. Fowler gibt eine weitere Metapher von Beck weiter: die zwei Hüte. Mit dem Refactoring-Hut erhält jede Änderung das Verhalten und alle Tests bleiben durchgehend grün. Beim Ergänzen von Funktionalität kommen Tests hinzu und vorhandene schlagen fehl. Das ist belastender und riskanter. Wer weiß, welcher Hut gerade aufgesetzt ist, weiß auch, was ein fehlgeschlagener Test bedeutet: Beim Refactoring war der letzte Schritt falsch.

Auf dieselbe Weise wird der größte Teil technischer Schulden abgetragen. Das Google-Papier über verborgene Schulden in Systemen des maschinellen Lernens nennt Refactoring von Code zuerst unter den Wegen zur Schuldentilgung. Werkzeuge helfen: Viele Sprachen haben inzwischen Editoren, die übliche Refactorings automatisieren. Fowler hält solche Werkzeuge für wertvoll, aber nicht unverzichtbar. Fehlen sie, verlässt er sich auf kleine Schritte und häufige Tests.

Wie es in der Praxis aussieht

Nehmen wir eine für diesen Artikel geschriebene Funktion, die an einer Stelle den Gesamtbetrag einer Bestellung berechnet und einen Beleg ausgibt:

```
function printReceipt(order) {
  let total = 0;
  for (const line of order.lines) {
    total += line.price * line.quantity;
  }

  // print the receipt lines
  console.log(`Order ${order.id}`);
  console.log(`Total: ${total}`);
}
```

Eine neue Anforderung kommt hinzu: Derselbe Gesamtbetrag muss auf einer Rechnung stehen. Statt die Schleife zu kopieren, refaktoriert der Entwickler zuerst. Schritt eins ist Extract Function: Die Schleife wandert in eine eigene, nach ihrem Ergebnis benannte Funktion, und die Tests laufen. Schritt zwei extrahiert die Ausgabe ebenso, sodass ein Funktionsname den Kommentar ersetzt. Die Tests laufen erneut. Schritt drei ist Inline Variable: Die Variable `total` wird nur

noch einmal verwendet, also tritt der Aufruf an ihre Stelle. Nach jedem Schritt tut das Programm genau dasselbe wie vorher, und jeder Schritt könnte für sich committet werden:

```
function printReceipt(order) {
  printLines(order.id, orderTotal(order));
}

function orderTotal(order) {
  let total = 0;
  for (const line of order.lines) {
    total += line.price * line.quantity;
  }
  return total;
}

function printLines(id, total) {
  console.log(`Order ${id}`);
  console.log(`Total: ${total}`);
}
```

Erst jetzt kommt die Funktionserweiterung hinzu: eine Zeile im Rechnungscode, die `orderTotal` aufruft. Keiner der drei Refactoring-Schritte benötigte einen neuen Test, weil sich nichts Beobachtbares geändert hat. Die vorhandenen Tests für den Beleg sind der Nachweis dafür.

Auch Infrastrukturcode wird refaktoriert. Dort steht mehr auf dem Spiel, weil Namen auf laufende Ressourcen verweisen. Die Terraform-Dokumentation warnt: Eine neue Ressourcenadresse in der Konfiguration weist Terraform standardmäßig an, die bestehende Ressource zu zerstören und eine neue anzulegen. `aws_instance.a` zur besseren Lesbarkeit in `aws_instance.web` umzubenennen würde deshalb einen Server ersetzen. Seit Terraform v1.1 hält stattdessen ein `moved` -Block die alte Adresse fest. Der nächste Plan aktualisiert dann die Adresse, ohne etwas zu zerstören:

```
resource "aws_instance" "web" {
  # (configuration unchanged)
}

moved {
  from = aws_instance.a
  to   = aws_instance.web
}
```

Für frühere Versionen verweist die Dokumentation für denselben Umzug auf `terraform state mv`. In beiden Fällen ist das Refactoring erst sicher, wenn der Plan gelesen wurde und eine Verschiebung statt einer Zerstörung und Neuanlage zeigt.

Solche Änderungen von neuen Funktionen getrennt zu halten ist einfacher, wenn der Verlauf beides benennt. Der Commitstandard im Portfolioeintrag über einen gemeinsamen Standard für die Historie von vier Repositories beginnt jede Commitnachricht mit einem konventionellen Präfix für die Änderungsart. Ein Commit-Message-Hook setzt dies durch. Die Seite über Arbeiten vom Juli 2026 zählt im größten Repository 26 Refactorings neben 156 Featurecommits und 133 Korrekturen. Sie sagt ebenso offen, dass der Hook das Format und nicht den Inhalt prüft. Auch ein als Refactoring bezeichneter Commit muss durch Tests und Review nachweisen, dass er das Verhalten unverändert gelassen hat.

Wo es schiefgeht

Verhaltensänderungen schleichen sich ins Refactoring ein. Ein Entwickler räumt eine Funktion auf und behebt darin im selben Commit einen Fehler. Schlägt ein Test fehl, ist unklar, welche der beiden Änderungen die Ursache war. Im Review muss jede Zeile als mögliche Verhaltensänderung gelesen werden. Tragen Sie nur einen Hut zugleich: Refaktorieren Sie in einem Commit und ändern Sie das Verhalten im nächsten. Schlägt beim Refactoring ein Test fehl, nehmen Sie den letzten Schritt zurück, statt weitere Korrekturen auf den fehlerhaften Schritt aufzubauen.

Eine große Restrukturierung heißt Refactoring. Ein Branch verbringt Wochen mit fehlschlagendem Build, während der Code „refaktoriert wird“. Das ist nach Fowlers Begriffen Restrukturierung und trägt das Risiko, das Refactoring gerade vermeiden soll. Teilen Sie die Arbeit in Schritte, die das System jeweils funktionsfähig lassen und einzeln zusammengeführt werden können. Ist das unmöglich, nennen Sie es eine Neufassung und planen Sie es entsprechend.

Es gibt keine Tests als Nachweis unveränderten Verhaltens. Erhaltenes Verhalten ist eine Behauptung und ohne Tests eine Vermutung. Schreiben Sie vor Eingriffen in ungetesteten Code Tests, die sein heutiges Verhalten festhalten, auch wenn es falsch aussieht. Korrigieren Sie die Merkwürdigkeiten anschließend getrennt. Hier gilt Fowlers Faustregel aus seinem Beispiel für vorbereitendes Refactoring: Sobald der Wunsch entsteht, Code auszuführen und seine Ausgabe mit dem Auge zu prüfen, schreiben Sie stattdessen einen Test.

Eine Umbenennung im Infrastrukturcode ersetzt die Ressource. Ein ordentlicher neuer Terraform-Name liest sich im Review wie Refactoring, während der Plan leise vorschlägt, eine Datenbank zu zerstören und eine leere anzulegen. Halten Sie jede Umbenennung und jeden Umzug in ein Modul hinein oder aus ihm heraus mit einem `moved` -Block fest. Werten Sie jede Zerstörung im Plan einer als Refactoring gedachten Änderung als fehlgeschlagenen Schritt.

Ein alter `moved` -Block wird gelöscht. Die Blöcke sammeln sich an; sie zu entfernen sieht nach Aufräumen aus. Die Terraform-Dokumentation nennt dies eine inkompatible Änderung: Jede Konfiguration, die noch die alte Adresse verwendet, erhielte dann einen Plan, der das Objekt zerstört, statt es zu verschieben. Behalten Sie die historischen Blöcke, verketten Sie sie bei zweimaliger Umbenennung und entfernen Sie sie nur aus einem privaten Modul, nachdem alle Nutzer die Version angewendet haben, die sie enthält.

Refactoring wartet auf eine eigene Phase. Ein Team plant einen „Refactoring-Sprint“ für später und ergänzt weiter Funktionen in Code, der sich dagegen sperrt. Ein für später geplanter Sprint lässt sich leicht streichen, wenn sich ein Termin verschiebt. Fowler empfiehlt die Gegenrichtung: Refaktorieren Sie als Teil jeder Änderung, zuerst um sie einfach zu machen und danach um ein klares Ergebnis zu hinterlassen. Dann braucht die Arbeit keinen eigenen Platz im Plan.

Weiterführende Lektüre

- Definition von Refactoring — Martin Fowlers Definitionen als Substantiv und Verb aus seinem Buch.
- Refactoring — Fowlers Website zum Buch, mit der Begründung für Refactoring als alltägliche Programmierarbeit und dem vollständigen Katalog.
- Extract Function — Katalogeintrag zu einem der häufigsten Refactorings, mit Umkehrung und Bezeichnung aus der ersten Auflage.
- Ein Beispiel für vorbereitendes Refactoring — Fowler formt eigenen Code vor einer neuen Funktion in kleinen Schritten um und erklärt die zwei Hüte.
- Module refaktorieren — Terraform-Dokumentation zu `moved` -Blöcken, Umbenennungen, Modulaufteilungen und dem Grund, alte Blöcke zu behalten.

Glossar

Für Ihre tatsächlichen Rahmenbedingungen auswählen

Beschreiben Sie den Arbeitsablauf, die vorhandenen Schnittstellen und die Nutzer, die die Veränderung benötigen. Wir können eine prüfbare Lieferung definieren und die Übergabe so gestalten, dass Ihr Team die gelieferten Bestandteile warten kann.

Preise ansehen Kontakt aufnehmen →

Die Begriffe praktisch nutzbar machen

Fragen Sie, wie „Was ist Refactoring?“ auf Ihre Organisation zutrifft. Ein konkretes Beispiel aus Ihrem bestehenden System reicht aus, um das Gespräch zu beginnen.

Unsere Leistungen entdecken

welcome@engineer.company +45 50 10 61 84 Kopenhagen, Dänemark

Nehmen Sie diese Seite mit

Kontaktkarte

Mit Sorgfalt und Liebe gestaltet von D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

Copyright © 2025 – heute · Engineer ApS · Alle Rechte vorbehalten.

Gedruckt von https://engineer.company/de/glossary/refactoring/