Engineer

Testgetriebene Entwicklung

Tests vor dem Code schreiben, einen kleinen nach dem anderen

Was es ist

Testgetriebene Entwicklung ist eine Art zu programmieren, bei der der Test für ein Verhalten vor dem Code entsteht, der es bereitstellt: jeweils ein kleiner Test nach dem anderen. Kent Beck, der seinen eigenen Anteil daran als Wiederentdeckung beschreibt, nennt es einen Programmierablauf. Nachdem er festgestellt hatte, dass Programmierer sich über die Bedeutung des Begriffs uneinig waren, hielt er den Ablauf 2023 in einem kurzen Beitrag namens Canon TDD fest. Darin verwendet er fast überall die Abkürzung TDD.

Der Ablauf hat fünf Schritte:

- Die Fälle auflisten, die die Änderung behandeln muss.
- Genau einen Fall aus der Liste auswählen und als konkreten, ausführbaren Test schreiben.
- Diesen Test zum Bestehen bringen, ohne dass ein früherer Test fehlschlägt, und jeden neu entdeckten Fall auf die Liste setzen.
- Falls der Implementierungsentwurf besser werden kann, jetzt refaktorieren; dieser Schritt ist optional.
- Zu Schritt 2 zurückkehren, bis die Liste leer ist.

  Die Liste ist eine Verhaltensanalyse, kein Entwurfsdokument: zuerst der Grundfall, dann die Varianten und schließlich die Fälle, in denen die Änderung Bestehendes nicht beschädigen darf. Der Test aus Schritt 2 ist ein echter automatisierter Test mit Vorbereitung, Aufruf und Zusicherungen. Beim ersten Lauf schlägt er fehl, weil das beschriebene Verhalten erst noch zu schreiben ist. Beck nennt Schritt 3 den Weg von Rot zu Grün, vom fehlgeschlagenen zum bestandenen Test.

Ebenso hilfreich ist zu wissen, was der Ablauf nicht ist. Alle Tests vor jeglichem Code zu schreiben ist ein häufiger Vorwurf gegen TDD. Beck antwortet direkt: Das ist kein TDD; wer es kritisiert, kritisiert etwas anderes. In die andere Richtung ist er genauso gelassen. Ein Team, das anders arbeitet und gute Ergebnisse erzielt, kann das gern tun. Für die genaue Befolgung der Schritte gebe es keinen goldenen Stern.

Das Konzept für eine Entscheidung nutzen

Nutzen Sie „Was ist testgetriebene Entwicklung?“ 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

Beck erklärt das Ziel des Ablaufs. Wer das Verhalten eines Systems ändert, sollte am Ende Folgendes erreicht haben: Die Änderung erhält bestehendes Verhalten und erfüllt zugleich die neuen Anforderungen, der Code ist für die nächste Änderung bereit, und die programmierende Person und ihre Kollegen vertrauen allen drei Punkten. Weil jeder Test entsteht, solange seine Änderung noch klein und frisch ist, wächst das Sicherheitsnetz im selben Tempo wie der Code, statt für später versprochen zu werden.

Der zweite Grund ist der Entwurf. Beck trennt zwei Arten von Entwurfsentscheidung: wie ein Verhalten aufgerufen wird und wie das System es implementiert. Den Test zuerst zu schreiben erzwingt die Schnittstellenentscheidungen, weil der Test etwas aufrufen und sagen muss, was zurückkommt. Die Implementierungsentscheidungen warten bis zum Refactoring, wenn bestandene Tests sie absichern. Ein Test, der sich umständlich schreiben lässt, warnt früh vor einer umständlich zu benutzenden Schnittstelle.

Der dritte Grund ist die Gegenprüfung. Den erwarteten Wert eines Tests leitet eine Person aus der Anforderung ab; der Code gelangt auf einem zweiten Weg dorthin. Stimmen beide überein, haben zwei unabhängige Wege dieselbe Antwort erreicht. Laut Beck stammt ein großer Teil des Werts von TDD als Prüfverfahren aus dieser Gegenprüfung.

Auch der Endpunkt wird erkennbar. Beck rät, so lange zu testen und zu programmieren, bis die Angst vor dem Verhalten des Codes in Langeweile umgeschlagen ist. Die Gewohnheit ist viel älter als der Beitrag. Beck ordnet seine Wiederentdeckung zwischen 1994 ein, als er SUnit für einen Beratungskunden schrieb, und Oktober 1995, als er Ward Cunningham auf der OOPSLA-Konferenz in Austin TDD vorführte. Sein Aufsatz Simple Smalltalk Testing beschreibt ein einfaches Testframework für Smalltalk. Die Python-Dokumentation zu unittest verweist noch immer darauf als Becks ursprünglichen Aufsatz über das Muster, das auch unittest verwendet.

Wie es in der Praxis aussieht

Als Beispiel dient eine Inhaltsprüfung, die verbotene Modewörter in den Texten einer Website melden soll. Die Arbeit beginnt mit einer Testliste in einfachen Notizen:

```
- ein verbotenes Wort wird gemeldet
- ein verbotenes Wort mit einem Großbuchstaben wird gemeldet
```

Schritt 2 macht aus genau einem Eintrag einen Test. Dass der erste zuerst kommt, hat einen Grund: Er gibt der Prüfung ein bekanntes verbotenes Wort und prüft, ob es gemeldet wird. Der Test schlägt fehl, bis die Prüfung es finden kann. Er kommt in eine Datei namens `test_wording.py`, die pytest einsammelt, weil ihr Name dem Muster `test_*.py` entspricht:

```
from wording import find_banned


def test_reports_a_banned_word():
    assert find_banned("we leverage it", {"leverage"}) == ["leverage"]
```

`pytest -q` schlägt sofort fehl, weil sich noch kein `find_banned` importieren lässt. Genau darum geht es: Der Test wurde beim Fehlschlagen beobachtet, damit ein späteres Bestehen etwas bedeutet. Vorerst genügt der einfachste Code, der ihn bestehen lässt, in `wording.py`:

```
def find_banned(text, banned):
    return [word for word in text.split() if word in banned]
```

Der Test besteht. Der nächste Listeneintrag wird zum nächsten Test:

```
def test_reports_a_capitalised_banned_word():
    assert find_banned("Leverage it", {"leverage"}) == ["Leverage"]
```

Er schlägt fehl. Weil pytest die Zwischenwerte einer Zusicherung ausgibt, zeigt die Ausgabe eine leere Liste, wo `["Leverage"]` erwartet wurde. Der Vergleich von `word.lower()` statt `word` lässt ihn bestehen. Bei dieser Änderung fällt auf, dass `text.split()` Satzzeichen am Wort lässt. „leverage,“ mit Komma würde deshalb durchrutschen. Das kommt auf die Liste statt sofort in den Code und wird zum dritten Test:

```
def test_reports_a_banned_word_before_a_comma():
    assert find_banned("Use it, leverage, win", {"leverage"}) == ["leverage"]
```

Er schlägt fehl, und eine weitere Zeile lässt alle drei bestehen:

```
def find_banned(text, banned):
    words = [word.strip(".,;:!?") for word in text.split()]
    return [word for word in words if word.lower() in banned]
```

Schritt 4 ist optional. Hier lässt sich wenig verbessern: zwei Zeilen mit je einer Aufgabe. Jeder Listeneintrag hat nun einen bestandenen Test, also ist die Arbeit fertig. Jeder Erwartungswert wurde aus dem jeweiligen Listeneintrag abgeleitet und nie aus einem Lauf kopiert.

Die Reihenfolge der Liste war bewusst gewählt; die eigene Dokumentation der Website zeigt, warum. Ein Projekt, das Tests für die eigenen Prüfprogramme schrieb, durchgeführt im August 2026, stellte fest, dass seine eigenen Prüfungen, darunter Rechtschreibung, Erzählperspektive und Zahlenkonsistenz, monatelang Grün gemeldet hatten. Eine Prüfung, die nur saubere Eingaben gesehen hat, lässt sich nicht von einer unterscheiden, die nichts liest. Die Lösung war, genau den Defekt einzubauen, den jedes Prüfprogramm finden soll: Die Rechtschreibprüfung erhält ein absichtlich falsch geschriebenes Wort, und der Test schlägt fehl, wenn der Lauf nichts meldet. Die Prüfung der Erzählperspektive erhält Prosa in der ersten Person und muss sie ablehnen. Die Modewortprüfung erhält ein verbotenes Wort. Zwei der Prüfprogramme lasen

tatsächlich weniger Dateien, als ihre Dokumentation behauptete. Die Seite beschreibt kein TDD. Ihre Lehre ist aber die, die TDD vom ersten Schritt an einbaut: Ein Test hat erst dann einen Wert, wenn er nachweislich einmal fehlgeschlagen ist.

Wo es schiefgeht

Alle Tests werden zuerst geschrieben. Jeden Listeneintrag in einen Test zu verwandeln, bevor einer besteht, ist genau die Fehlinterpretation, die Beck herausgreift. Erzwingt der erste Test ein Umdenken, müssen alle spekulativ geschriebenen Tests überarbeitet werden. Wer beim sechsten Test noch keinen bestandenen gesehen hat, erlebt laut Beck Niedergeschlagenheit oder Langeweile. Einen Test schreiben, ihn zum Bestehen bringen, dann den nächsten wählen. Zeigt ein neuer Test, dass die bisherige Arbeit einen Fall nicht behandeln kann, bleibt die Wahl zwischen Weitermachen und Neubeginn. Beck rät, neu anzufangen und die Fälle anders anzuordnen.

Der Test enthält keine Zusicherung. Ein Test, der Code aufruft und nichts prüft, erhöht die gemessene Testabdeckung und beweist nichts. Jeder Test braucht Vorbereitung, Aufruf und Zusicherung. Beck schlägt vor, gelegentlich einen Test rückwärts von seinen Zusicherungen aus zu schreiben.

Die Zusicherung wird gelöscht. Wenn ein Test nicht besteht, täuscht das Löschen seiner Zusicherung nur ein Bestehen vor. Ihn wirklich zum Bestehen bringen oder den Fall auf die Liste zurücknehmen und die Reihenfolge überdenken.

Erwartungswerte werden aus der Ausgabe kopiert. Den Code auszuführen und seine Rückgabe in den Test zu kopieren macht aus zwei Prüfwegen einen: Der Test bestätigt dann nur, dass der Code tut, was er schon getan hat. Den erwarteten Wert aus der Anforderung ableiten, nötigenfalls von Hand.

Refactoring beim Wechsel von Rot zu Grün. Struktur und Verhalten im selben Schritt zu ändern nennt Beck das Tragen zweier Hüte. Erst zum Laufen bringen, dann richtig machen: mit der einfachsten Änderung Grün erreichen und erst aufräumen, wenn die Tests bestehen.

Zu viel aufräumen oder zu früh abstrahieren. Aufräumen fühlt sich gut an und kann sicherer wirken als der nächste Test. Aufhören, sobald der Code für diese Sitzung gut genug ist. Zwei ähnliche Codestellen unverändert lassen, bis das Muster klar ist. Beck fasst es knapp: „Duplizierung ist ein Hinweis, kein Befehl."

Die Tests kennen die Implementierung. Tests, die private Methoden aufrufen oder die getätigten Aufrufe statt des Ergebnisses prüfen, brechen bei jeder internen Änderung. Erik Kueflers Kapitel über Unit-Tests in *Software Engineering at Google* nennt sie brüchige Tests: Sie schlagen bei einer unabhängigen Änderung fehl, die keinen wirklichen Fehler einführt. In TDD kostet das doppelt, weil das Refactoring von Tests abhängt, die bei internen Umbauten weiter Grün bleiben. Über die öffentliche Schnittstelle testen, den entstandenen Zustand statt der Interaktionen prüfen und jeden Test nach dem geprüften Verhalten benennen.

Weiterführende Lektüre

- Canon TDD — Kent Becks eigene Beschreibung des Ablaufs von 2023: fünf Schritte, die Trennung von Schnittstellen- und Implementierungsentwurf sowie die Fehler in jedem Schritt.
- Simple Smalltalk Testing: With Patterns — Becks Aufsatz über ein einfaches Smalltalk-Testframework mit Testvorbereitungen, Testfällen und Testsammlungen, in einer archivierten Fassung.
- Unit Testing — Erik Kueflers Kapitel in *Software Engineering at Google* über Tests, die Refactoring überstehen, über das Testen über öffentliche APIs und über Verhaltenstests.
- Get Started — die Einführung in pytest: Installation, ein erster fehlgeschlagener Test samt Bericht und die Suche nach Testdateien.
- unittest: Unit testing framework — das Framework der Python-Standardbibliothek, das Becks Aufsatz als ursprüngliche Beschreibung seines Testmusters nennt.

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 testgetriebene Entwicklung?“ 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/test-driven-development/