WHITEPAPER

Beweisen statt vermuten

Ein testgestütztes Verfahren zum Nachweis semantisch redundanter Methoden

> Ein Beweis, der scheitern kann, ist ein Beweis. Eine Ähnlichkeitszahl ist eine Meinung.

Manfred Wolff

Konzeptpapier zu twin-kit · Stand August 2026

INHALT

Inhaltsverzeichnis

Zusammenfassung

KI-gestützte Entwicklung erzeugt eine Form von Redundanz, die kein verbreitetes Werkzeug findet: Methoden, die dasselbe tun, aber verschieden aussehen. Ich schlage ein Verfahren vor, das solche Paare nicht nur aufspürt, sondern die Gleichheit nachweist. Der Nachweis funktioniert so: Der Rumpf der einen Methode wird in einer Wegwerf-Kopie des Projekts durch einen Aufruf der anderen ersetzt, dann läuft die vollständige Testsuite. Bleibt sie grün, ist die Austauschbarkeit belegt, und zwar genau so weit, wie die Testsuite trägt. Wie weit sie trägt, sagt das Verfahren gleich mit, über den Mutation Score der betroffenen Klassen. Dieses Papier beschreibt das Verfahren, erklärt die nötigen Begriffe, grenzt es von der Forschung zur Klonerkennung ab und legt das Design der Validierung offen. Ergebnisse gibt es noch nicht; die Erfolgskriterien sind vorab festgelegt, damit ich sie hinterher nicht passend machen kann.

KAPITEL 01

1. Das Problem

Ein Sprachmodell löst jedes Issue für sich, und jede Lösung ist für sich richtig. Was fehlt, ist das Gedächtnis für die Codebasis: das Wissen, dass dieselbe Fähigkeit drei Pakete weiter schon existiert. Ein Mensch, der zehn Jahre an einem System arbeitet, hat dieses Gedächtnis. Ein Modell hat ein Kontextfenster, und das endet mit der Sitzung.

Das Ergebnis sind Methoden, die dasselbe tun, aber anders heißen, andere Parameter in anderer Reihenfolge nehmen und an anderer Stelle liegen. Kein Copy-Paste, keine Textähnlichkeit, trotzdem Redundanz. Der Schaden ist konkret: Ein Fehler wird an einer Stelle behoben und bleibt an der anderen stehen. Eine fachliche Änderung wird an einer Stelle nachgezogen und an der anderen vergessen. Juergens und Kollegen haben schon 2009 gezeigt, dass inkonsistent geänderte Klone eine messbare Fehlerquelle sind [2]. Damals entstanden Klone durch Kopieren. Heute entstehen sie durch unabhängiges Neuerzeugen, und das macht sie unsichtbar für alles, was auf Textähnlichkeit prüft.

Auf die Idee zu diesem Papier hat mich ein Nebensatz gebracht. Ich hatte in einer laufenden Sitzung nach etwas anderem gefragt, und das Modell meldete beiläufig:

> Der Befund zu `repo-name` ist schärfer als gedacht, es gibt **drei** `getRepoName()`-Implementierungen mit drei verschiedenen Rückgabeformen. Das geht in Issue B ein.
>
> Claude, während der Arbeit an einer Codebasis

Drei Implementierungen derselben Fähigkeit, drei verschiedene Rückgabeformen, entstanden ohne eine einzige Kopie. Gefunden hat sie kein Werkzeug, sondern ein Modell, das zufällig an der richtigen Stelle vorbeikam. Genau diesen Zufall wollte ich reproduzierbar machen. Das Werkzeug dazu heißt twin-kit.

KAPITEL 02

2. Vier Begriffe als Handwerkszeug

Wer die folgenden vier Begriffe kennt, kann den Rest des Papiers ohne Spezialwissen lesen.

Klontypen

Die Forschung zur Klonerkennung unterscheidet vier Typen [1]. Typ 1 ist identischer Code. Typ 2 ist identische Struktur mit umbenannten Bezeichnern. Typ 3 erlaubt eingefügte oder entfernte Zeilen. Typ 4 ist semantische Gleichheit bei syntaktischer Verschiedenheit: gleiche Wirkung, anderer Code. Werkzeuge wie SonarQube oder PMD CPD vergleichen Token, also die kleinsten Bausteine des Quelltexts nach dem Zerlegen, etwa Schlüsselwörter, Namen und Operatoren. Das findet Typ 1 bis 3 zuverlässig und Typ 4 prinzipbedingt nie, denn bei Typ 4 gibt es keine Tokenfolge, die sich gleicht.

Die Testsuite als Orakel

In der Testforschung heißt die Instanz, die richtig von falsch unterscheidet, das Orakel. Mein Orakel ist die vorhandene Testsuite des geprüften Projekts. Das ist eine pragmatische Wahl mit einer klaren Konsequenz: Der Nachweis gilt immer relativ zur Suite. Was kein Test beobachtet, kann auch kein Test widerlegen.

Mutation Testing

Wie gut eine Suite beobachtet, misst Mutation Testing [3]. Ein Werkzeug wie PIT [4] baut absichtlich kleine Fehler in den Produktivcode ein, sogenannte Mutanten, etwa ein `>` durch `>=` ersetzt oder eine Bedingung negiert. Dann läuft die Suite. Schlägt mindestens ein Test fehl, gilt der Mutant als getötet. Überlebt er, hat die Suite den eingebauten Fehler nicht bemerkt. Der Mutation Score ist der Anteil getöteter Mutanten. Er misst, was Zeilenabdeckung nicht misst: nicht ob Code ausgeführt wird, sondern ob sein Verhalten geprüft wird.

Embeddings

Ein Embedding ist ein Vektor aus einigen hundert Zahlen, den ein neuronales Netz aus einem Text berechnet, hier aus einem Methodenrumpf. Die Pointe: Bedeutungsähnliche Texte bekommen ähnliche Vektoren, auch wenn sie wörtlich verschieden sind. Ähnlichkeit misst man über die Kosinusähnlichkeit, anschaulich den Winkel zwischen zwei Vektoren; 1 heißt gleiche Richtung, 0 heißt nichts gemein. Damit lässt sich semantische Nähe billig schätzen. Schätzen, nicht beweisen.

KAPITEL 03

3. Verwandte Arbeiten und ihre Lücke

Für Typ 4 gibt es Forschungsansätze, etwa über Programmabhängigkeitsgraphen, die Daten- und Kontrollfluss abbilden, oder neuerdings über Sprachmodelle als Klassifikator. Die Benchmarks dazu, allen voran BigCloneBench [6], zeigen seit Jahren steigende Erkennungsraten. Zwei aktuelle Beobachtungen dämpfen das Bild. Erstens brechen viele Verfahren ein, wenn sie Funktionalität sehen, die nicht im Trainingsmaterial vorkam; Sprachmodelle brechen weniger stark ein als klassische Verfahren, fehlerfrei sind auch sie nicht [8]. Zweitens fragt eine Arbeit von 2026 schon im Titel, ob die gemessenen Erfolge echte semantische Äquivalenz abbilden oder nur Muster im Datensatz [7].

Für mein Vorhaben ist die genaue Antwort zweitrangig, denn alle diese Verfahren teilen eine Eigenschaft: Sie liefern Verdacht. Eine Ähnlichkeitszahl, ein Klassifikator-Urteil, eine Wahrscheinlichkeit. Entwickler haben gelernt, Verdachtslisten wegzuklicken, und zwar zu Recht, denn die Prüfung jedes einzelnen Verdachts kostet mehr als der Verdacht wert ist. Die Lücke ist nicht das Finden. Die Lücke ist der Schritt vom Verdacht zum Nachweis, den bisher der Mensch gehen muss. Genau diesen Schritt automatisiere ich.

KAPITEL 04

4. Das Verfahren

twin-kit arbeitet in fünf Stufen. Jede liest die Ausgabe der vorherigen von der Platte, jede ist einzeln wiederholbar.

4.1 Extraktion

Ein sprachabhängiger Extraktor, für Java auf Basis von Spoon [5], für TypeScript auf Basis von ts-morph, erzeugt einen sprachneutralen Methodenkatalog. Je Methode stehen darin Signatur, Dokumentationszeile, ein normalisierter Rumpf mit vereinheitlichten Bezeichnern, die Folge der Knotentypen des abstrakten Syntaxbaums (der Baumdarstellung, die ein Compiler aus dem Quelltext baut, ohne Formatierung und Namen), Metriken sowie die Menge der aufgerufenen Methoden. Ab hier kennt das Werkzeug keine Programmiersprache mehr.

Ausschlussregeln entfernen, was massenhaft und zu Recht gleich ist: Getter, Setter, `equals`, `hashCode`, generierten Code, Testcode, triviale Delegationen. Ebenso überschriebene Methoden, denn Polymorphie ist keine Redundanz, und Überladungen derselben Klasse, denn die sind meist absichtliche Bequemlichkeit.

4.2 Kandidatensuche

Aus dem Katalog entstehen Kandidatenpaare. Hart ausgeschlossen wird nur, was sich von vornherein nicht zusammenlegen lässt, etwa unvereinbare Rückgabetypen. Alles andere fließt gewichtet in einen Ähnlichkeitswert ein: die Kosinusähnlichkeit der Embeddings, ein struktureller Fingerabdruck über den Syntaxbaum, die Jaccard-Ähnlichkeit der Aufrufmengen (die Größe der Schnittmenge geteilt durch die Größe der Vereinigung, ein Standardmaß für Mengenähnlichkeit) und die Nähe der zyklomatischen Komplexität (grob: die Zahl der unabhängigen Pfade durch eine Methode).

Die Reihenfolge ist Absicht. Merkmale der Implementierung wie Komplexität und Aufrufmengen dürfen nicht hart filtern, denn Typ 4 bedeutet gerade: gleiche Semantik, andere Implementierung. Eine Stream-Lösung und eine Schleife unterscheiden sich genau in diesen Merkmalen.

4.3 Triage

Ein Sprachmodell sichtet die Kandidatenpaare und urteilt: äquivalent, teilweise überlappend, verschieden, mit zwei Sätzen Begründung und einer Vermutung zu Randfällen. Zwei Regeln sind nicht verhandelbar. Das Modell erzeugt eine Hypothese, niemals eine Änderung. Und es ist ein Modell anderer Bauart als das, das den geprüften Code geschrieben hat, denn ein Modell ist gegenüber der eigenen Lösung systematisch unkritisch. Für Audits in fremden Codebasen läuft diese Stufe gegen ein lokales Modell, damit kein Quelltext den Rechner des Kunden verlässt.

4.4 Der Beweis durch Delegation

Das Herzstück. Für eine Hypothese „B ist äquivalent zu A“ wäre der naheliegende Nachweis, alle Aufrufer von B auf A umzubiegen und B zu löschen. Das ist mechanisch schwierig, weil Parameterreihenfolgen, Typen und

Rückgabewerte auseinandergehen. Stattdessen ein kleinerer Eingriff mit gleichem Beweiswert: Der Rumpf von B wird durch einen Aufruf von A ersetzt. B bleibt stehen, alle Aufrufer bleiben unverändert.

```
// Zwei Methoden, vermutlich semantisch gleich:

static String join(List<String> parts, String sep) {
    StringBuilder sb = new StringBuilder();
    for (int i = 0; i < parts.size(); i++) {
        if (i > 0) sb.append(sep);
        sb.append(parts.get(i));
    }
    return sb.toString();
}

static String concatWith(String delimiter, List<String> items) {
    return items.stream().collect(Collectors.joining(delimiter));
}

// Der Eingriff: Der Rumpf von concatWith wird zur Delegation an join.
// Man beachte die vertauschte Parameterreihenfolge; die Abbildung
// übernimmt das Werkzeug.

static String concatWith(String delimiter, List<String> items) {
    return join(items, delimiter);
}
```

Der Ablauf um diesen Eingriff herum:

1. **Baseline.** Je Commit läuft die Suite zweimal auf dem unveränderten Stand. Ist sie nicht beide Male grün, gibt es keine Beweise, denn ein späteres Rot wäre nicht interpretierbar.
2. **Arbeitskopie.** Der Eingriff passiert in einem `git worktree`, einem zweiten Arbeitsverzeichnis desselben Repositories auf einem Wegwerf-Zweig. Der Arbeitsstand des Entwicklers wird nie angefasst.
3. **Kompilieren zuerst.** Scheitert schon der Compiler, lautet das Urteil `nicht abbildbar`, und es hat Sekunden gekostet. Kompilierfehler und Testfehler sind verschiedene Aussagen: Das eine heißt, der Umbau ist mechanisch unmöglich. Das andere heißt, die Methoden verhalten sich verschieden.
4. **Testlauf und Urteil.** Bleibt die volle Suite grün, ist die Hypothese unter der vorhandenen Suite belegt. Fällt ein Test, wird der Lauf einmal wiederholt. Bestätigtes Rot heißt widerlegt, und der fehlgeschlagene Test ist das Gegenbeispiel; für den Entwickler oft nützlicher als ein Fund. Wechselt das Ergebnis, war der Test wackelig, und das Urteil lautet `ungeprüft`.
5. **Aufräumen.** Worktree und Zweig verschwinden, unabhängig vom Ausgang.

Eine Grenze gehört hierher, nicht ins Kleingedruckte: Die Delegation braucht einen Empfänger für den Aufruf von A. Bei statischen Methoden, Methoden derselben Klasse und zustandslos konstruierbaren Klassen ist er da. Ist A dagegen die Methode einer Komponente mit Abhängigkeiten, etwa eines Spring-Beans, gibt es keinen mechanisch sauberen Empfänger, und das Urteil lautet `nicht abbildbar (Empfänger)`. Solche Paare gehen als unbewiesene Hypothese an den Menschen. Die Erwartung dahinter: KI-erzeugte Redundanz entsteht überproportional bei statischen und zustandslosen Helfern, also genau in der beweisbaren Menge. Ob diese Erwartung stimmt, ist eine Messgröße der Validierung.

4.5 Konfidenz

Ein grüner Lauf ist nur so viel wert wie die Suite, die ihn erzeugt. Deshalb liefert jedes Urteil seine eigene Belastbarkeit mit:

Stufe	Bedingung
bewiesen	Suite grün, Mutation Score der betroffenen Klassen 100 Prozent, und ein gezielter Mutationslauf auf A lässt Tests fallen, die zu B gehören
wahrscheinlich	Suite grün, aber mindestens eine Bedingung darüber fehlt; der Score wird genannt
ungeprüft	keine verwertbare Abdeckung im betroffenen Bereich, oder wackeliger Test
widerlegt	Suite rot, mit dem fehlgeschlagenen Test als Gegenbeispiel

Die dritte Bedingung der obersten Stufe verdient einen Satz. Nach der Delegation laufen Bs Tests faktisch gegen A. Mutiert man nun A und fallen dabei Tests, die für B geschrieben wurden, dann ist belegt, dass Bs Verhaltenserwartungen jetzt an A geprüft werden. Ohne diese Prüfung hieße Grün bei einem B ohne eigene Tests nur: niemand hat etwas gemerkt.

4.6 Bericht

Am Ende stehen ein maschinenlesbarer JSON-Bericht, ein lesbarer Markdown-Bericht und optional SARIF, ein standardisiertes Austauschformat für Analysebefunde, das GitHub direkt als Annotationen am Code anzeigt. Je Befund: Paar, Urteil, Konfidenz, Beweisprotokoll und ein Patch-Vorschlag als Diff. Dazu ein Blindstellen-Bericht: die Bereiche, in denen kein Beweis möglich war, weil die Testabdeckung es nicht hergibt. Dort steht implizit der Satz: Hier musst du testen, damit sich Aussagen beweisen lassen.

KAPITEL 05

5. Was der Beweis beweist, und was nicht

Präzision an dieser Stelle schützt vor Überverkauf. Der grüne Lauf beweist nicht die semantische Äquivalenz von A und B im mathematischen Sinn. Er beweist eine gerichtete Ersetzbarkeit: B kann durch A ersetzt werden, ohne dass die Testsuite es merkt. Das ist eine Aussage relativ zu einem Orakel, und ihre Stärke hängt an der Stärke des Orakels. Genau dafür gibt es die Konfidenzstufen.

Zwei Lücken bleiben auch bei voller Mutation Coverage. Seiteneffekte, die kein Test beobachtet, etwa ein Logeintrag oder ein Zählerstand, können sich unterscheiden. Und Nichtdeterminismus in der Suite kann ein Urteil verfälschen; dagegen stehen die doppelte Baseline und die Wiederholung roter Läufe. Beide Lücken stehen in jedem Bericht als Hinweis.

> Die Gegenrichtung ist verlässlicher: Ein bestätigtes Rot ist ein hartes Gegenbeispiel. Der fehlgeschlagene Test benennt konkret, worin sich die beiden Methoden unterscheiden. Widerlegen kann das Verfahren also stärker als belegen, und das ist für ein Werkzeug, das Vorschläge macht, die richtige Asymmetrie.

KAPITEL 06

6. Geplante Validierung

Prüfstein ist kanban-kit, ein Projekt mit 2.171 Tests, 100 Prozent Zeilen- und Zweigabdeckung als Build-Gate und 1.021 von 1.021 getöteten Mutanten. Dort bedeutet ein grüner Lauf etwas. Zugleich ist es der schwerste Prüfstein für die Trefferquote, denn eine diszipliniert gebaute Codebasis sollte wenig Redundanz enthalten.

Ein Nullbefund allein wäre deshalb mehrdeutig: Codebasis sauber, oder Detektor blind? Die Auflösung sind geimpfte Zwillinge: zehn bis zwanzig konstruierte Paare, die per Skript in einen Zweig eingebaut werden, in fünf Schwierigkeitsklassen von „gleiche Struktur, andere Namen“ bis „anderes Paket, andere Schicht, andere Datenstruktur“. Gemessen wird der Recall je Klasse, also der Anteil der eingebauten Paare, den das Werkzeug wiederfindet. Erst dieser Wert macht den Befund auf dem echten Code interpretierbar.

Die Erfolgskriterien sind vorab festgelegt. Die Idee trägt, wenn mindestens drei vorher unbekannte Paare belegt werden und mindestens eines davon nach Ansicht des Codes zusammengelegt gehört. Sie trägt nicht, wenn bei mehr als zwanzig Kandidaten nichts belegt wird, obwohl die geimpften Paare gefunden werden; dann folgt genau eine zweite, fremde Codebasis, und danach ist Schluss. Werden schon die geimpften Paare nicht gefunden, ist die Suchstufe defekt, nicht die Idee, und wird nachgeschärft, bevor geurteilt wird.

KAPITEL 07

7. Grenzen

Das Verfahren funktioniert am besten dort, wo das Problem am kleinsten ist: in gut getesteten Codebasen. Diese Schwäche ist konstitutiv, und die Antwort darauf ist der Konfidenzwert samt Blindstellen-Bericht, der aus der Schwäche eine Handlungsanweisung macht. Die beweisbare Menge ist durch das Empfänger-Problem begrenzt; wie stark, ist eine offene Messfrage. Jeder Beweis kostet einen Testlauf, bei vier bis sechs Minuten je Lauf ist das eine nächtliche Übung mit Obergrenze, keine interaktive. Und die Kandidatensuche wird zu viel vorschlagen; das ist erträglich, weil der Beweis den Rest erledigt, und genau deshalb erscheint kein unbewiesener Verdacht ohne seine Zahl im Bericht.

KAPITEL 08

8. Einordnung

Der Beitrag dieses Verfahrens liegt nicht in einer besseren Erkennungsrate. Er liegt im Wechsel der Aussagekategorie: von „diese Methoden könnten dasselbe tun“ zu „ich habe die eine durch die andere ersetzt, 2.171 Tests bleiben grün, hier ist der Pull Request, und hier steht, wie belastbar das ist“. Verdacht wird zu Nachweis, und der Mensch entscheidet nur noch ja oder nein. Leitplanken müssen scheitern können, nicht argumentieren.

> Ein Beweis, der scheitern kann, ist ein Beweis. Eine Ähnlichkeitszahl ist eine Meinung.

Literatur

[1] C. K. Roy, J. R. Cordy, R. Koschke: *Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach*. Science of Computer Programming 74(7), 2009.

[2] E. Juergens, F. Deissenboeck, B. Hummel, S. Wagner: *Do Code Clones Matter?* Proceedings of ICSE 2009.

[3] R. A. DeMillo, R. J. Lipton, F. G. Sayward: *Hints on Test Data Selection: Help for the Practicing Programmer.* IEEE Computer 11(4), 1978.

[4] H. Coles, T. Laurent, C. Henard, M. Papadakis, A. Ventresque: *PIT: A Practical Mutation Testing Tool for Java*. Proceedings of ISSTA 2016.

[5] R. Pawlak, M. Monperrus, N. Petitprez, C. Noguera, L. Seinturier: *Spoon: A Library for Implementing Analyses and Transformations of Java Source Code.* Software: Practice and Experience 46, 2016.

[6] J. Svajlenko, J. F. Islam, I. Keivanloo, C. K. Roy, M. M. Mia: *Towards a Big Data Curated Benchmark of Inter-project Code Clones*. Proceedings of ICSME 2014.

[7] *Semantic Code Clone Detection: Are We There Yet?* arXiv:2606.25272, 2026.

[8] K. Kitsios et al.: *Detecting Semantic Clones of Unseen Functionality.* arXiv:2510.04143.

Manfred Wolff · mwolff.org · Konzeptpapier, Stand August 2026. Ergebnisse der Validierung folgen in einer zweiten Fassung.