Testen

Prof. Sven Apel

Universität des Saarlandes

Basierend auf Material von Andreas Zeller und Norbert Siegmund

Teil IV

Testtaktiken: Strukturell

Testtaktiken

Tests basieren auf *Spezifikation*

Test deckt soviel *spezifiziertes* Verhalten wie möglich ab

Tests basieren auf *Code*

Test deckt soviel *implementiertes* Verhalten wie möglich ab

Warum Strukturell?

Wenn ein Teil des Programms *nie* ausgeführt wird, bleiben Defekte dort unentdeckt

"Teile" = Anweisungen, Module, Bedingungen…

Attraktiv, da *automatisierbar*

Warum Strukturell?

Ergänzt funktionale Tests

Erst funktionale Tests ausführen, dann nicht abgedeckten Code testen

Kann *Details* abdecken, die in der abstrakten Spezifikation fehlen

Herausforderung

```
class Roots {
    // Löse ax² + bx + c = 0
    public roots(double a, double b, double c)
    { … }

    // Ergebnis: Werte für x
    double root_one, root_two;
}
```

Welche Werte für *a*, *b* und *c* sollten wir testen?

Wenn *a*, *b*, *c* 32-Bit Integers sind, haben wir $(2^{32})^3 \approx 10^{28}$ mögliche Eingaben.
Mit 1.000.000.000.000 Tests/s brauchen wir immer noch 2.5 Mrd. Jahre!

Der Code

```
// Löse ax² + bx + c = 0
public roots(double a, double b, double c)
{
    double q = b * b - 4 * a * c;
    if (q > 0 && a ≠ 0) {
        // Code für zwei Lösungen
    }

    else if (q == 0) {
        // Code für eine Lösung
    }

    else {
        // Code für keine Lösungen
    }
}
```

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Teste diesen Fall

und diesen

und diesen!

Testfälle

```
// Löse ax² + bx + c = 0
public roots(double a, double b, double c)
{
    double q = b * b - 4 * a * c;
    if (q > 0 && a ≠ 0) {
        // Code für zwei Lösungen
    }

    else if (q == 0) {
        // Code für eine Lösung
    }

    else {
        // Code für keine Lösungen
    }
}
```

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

(a, b, c) = (3, 4, 1)

(a, b, c) = (0, 0, 1)

(a, b, c) = (3, 2, 1)

Ein Fehler

```
// Löse ax² + bx + c = 0
public roots(double a, double b, double c)
{
    double q = b * b - 4 * a * c;
    if (q > 0 && a ≠ 0) {
        // Code für zwei Lösungen
    }

    else if (q == 0) {
        x = (-b) / (2 * a);
    }

    else {
        // Code für keine Lösungen
    }
}
```

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Code muss a = 0 berücksichtigen

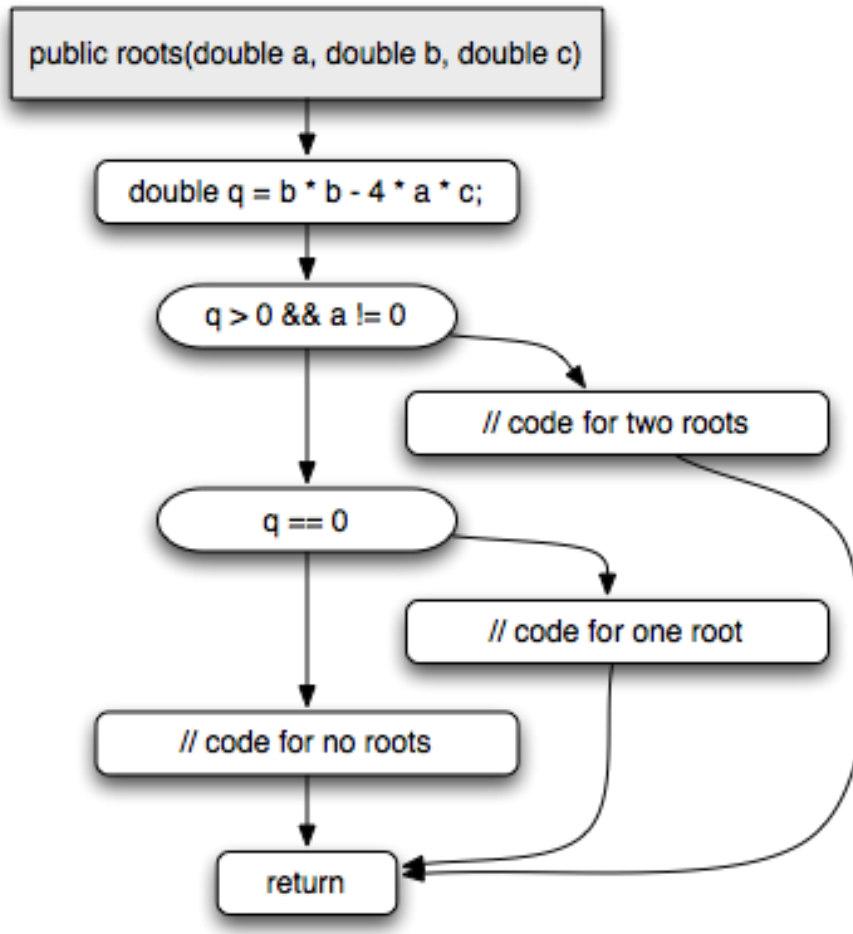
(a, b, c) = (0, 0, 1)

Struktur ausdrücken

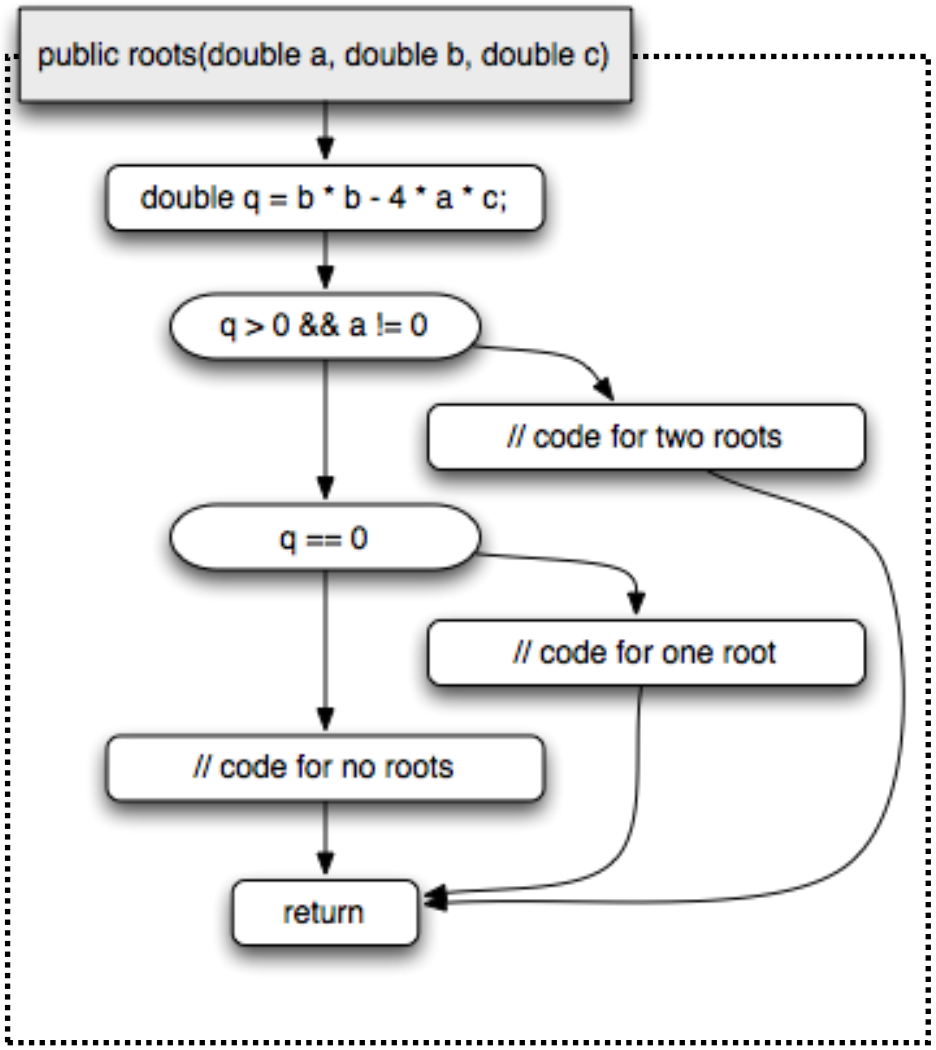
```
// Löse ax² + bx + c = 0
public roots(double a, double b, double c)
{
    double q = b * b - 4 * a * c;
    if (q > 0 && a ≠ 0) {
        // Code für zwei Lösungen
    }

    else if (q == 0) {
        x = (-b) / (2 * a);
    }

    else {
        // Code für keine Lösungen
    }
}
```



Kontrollflussgraph

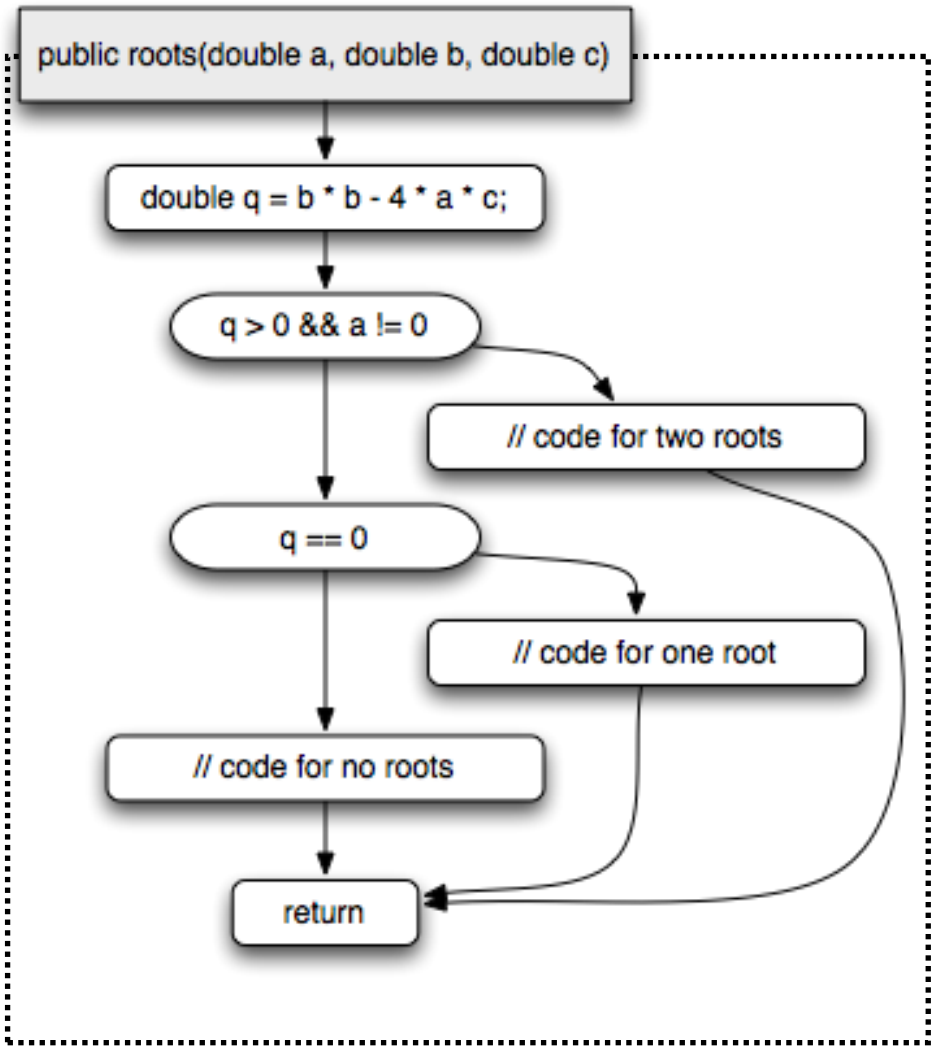


Ein *Kontrollflussgraph* (CFG) drückt mögliche Ausführungspfade eines Programms aus

Knoten sind *Basic Blocks* – Anweisungsfolgen mit 1 Eingang und 1 Ausgang

Kanten stellen *Kontrollfluss* dar – die Möglichkeit, dass ein Basic Block nach einem anderen ausgeführt wird

Strukturelles Testen



Aus dem CFG lassen sich *Testkriterien* ableiten

Je mehr Teile abgedeckt (ausgeführt) werden, um so höher die Chance einen Fehler zu finden

“Teile” können sein: Knoten, Kanten, Pfade, Bedingungen…

Testkriterien

Wie wissen wir, ob ein Test "gut genug" ist?

Ein Testkriterium ist ein Prädikat, das für ein Paar ⟨*Programm, Tests*⟩ erfüllt oder nicht erfüllt ist

Gewöhnlich als Regel ausgedrückt –
z.B. "alle Anweisungen müssen abgedeckt sein"

Anweisungsabdeckung

Testkriterium: Jede Anweisung (oder Knoten im CFG) muss wenigstens 1x ausgeführt werden

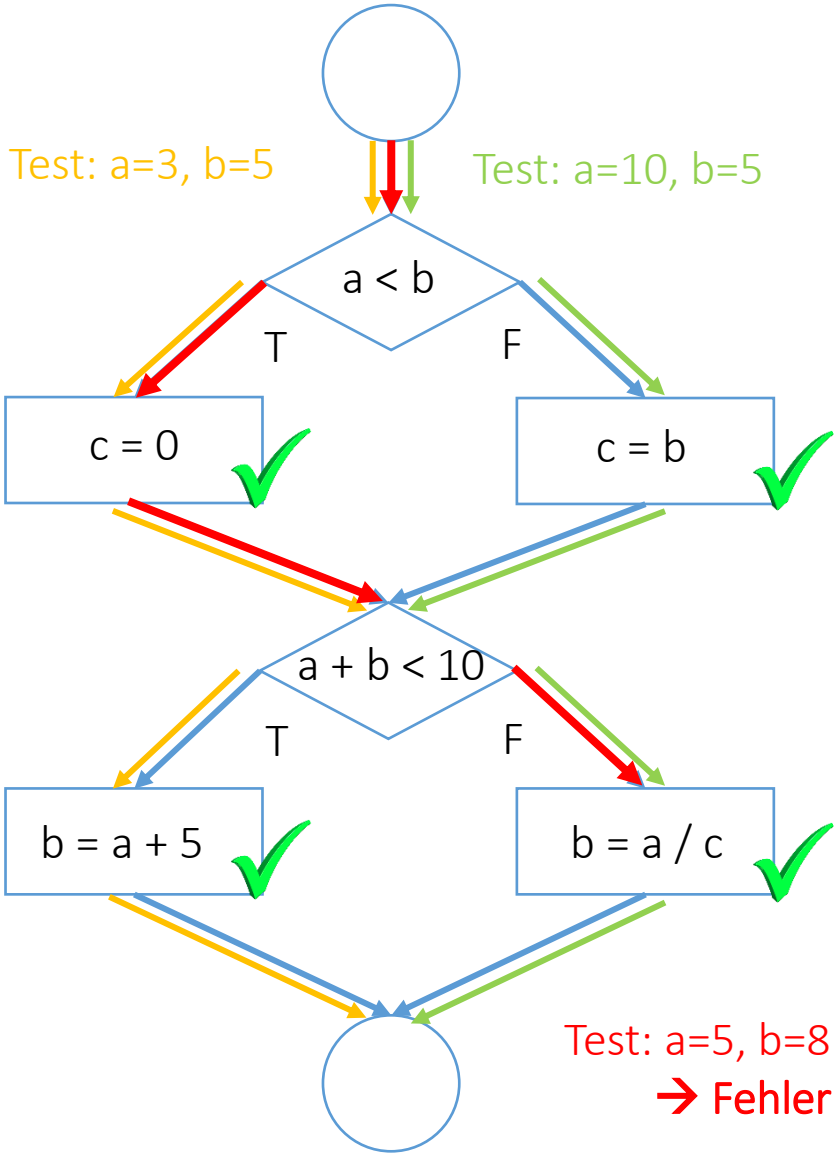
Hintergrund: Ein Defekt in einer Anweisung kann nur gefunden werden, wenn die Anweisung ausgeführt wird

Abdeckung: $\frac{\#\ ausgeführte\ Anweisungen}{\#\ Anweisungen}$

Beispiel

Nicht alle Wege müssen geprüft werden

Schleifen werden unzureichend geprüft



Abdeckung/Überdeckung berechnen

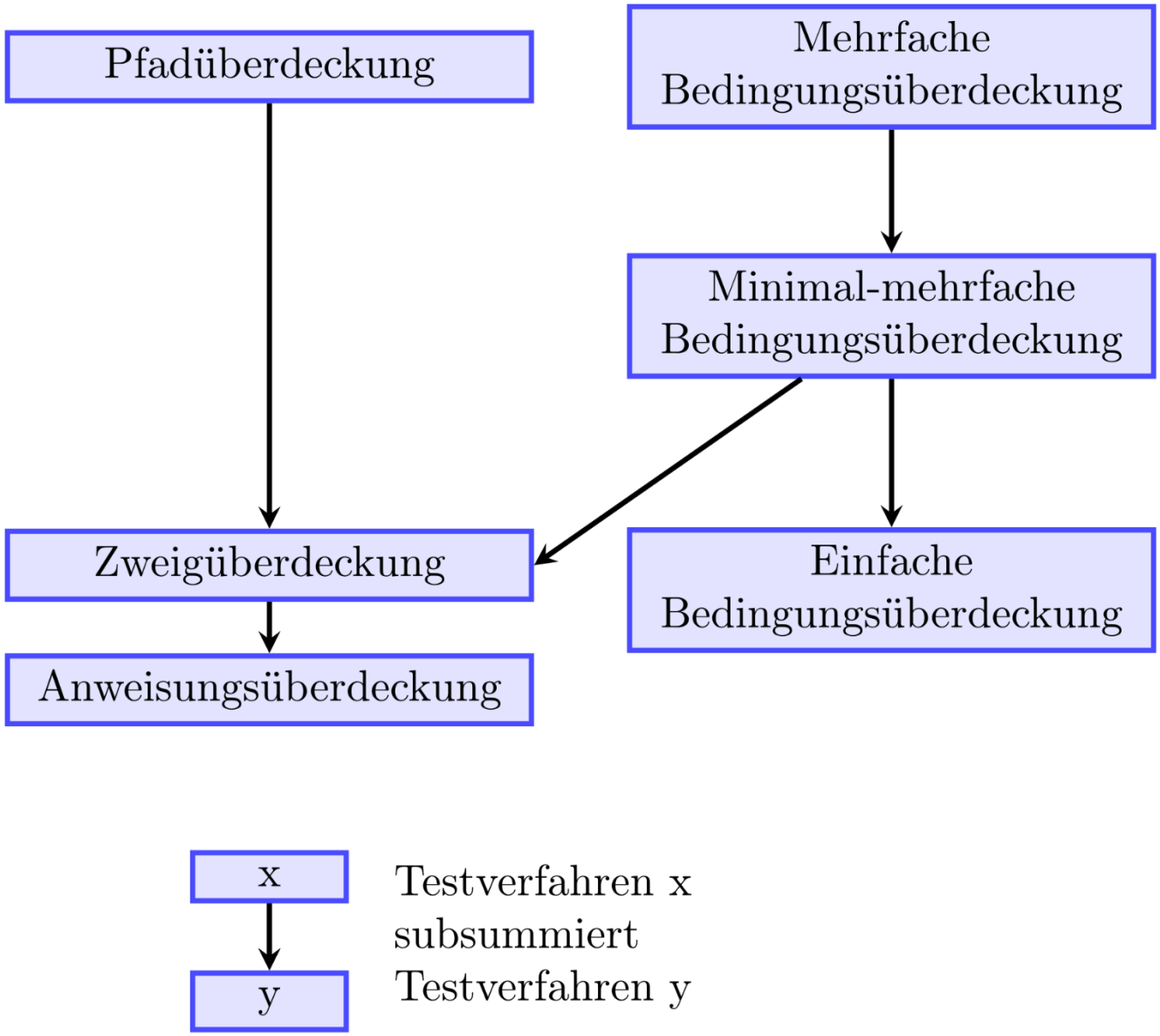
Die Abdeckung wird automatisch berechnet, während das Programm läuft

Benötigt *Instrumentierung* zur Übersetzungszeit

Mit GCC etwa: –ftest-coverage –fprofile-arcs

Nach Ausführung prüft ein *Abdeckungswerkzeug* die Ergebnisse

Mit GCC etwa: “gcov source-file”erzeugt .gcov-Dateien mit Abdeckung


Pfadüberdeckung
Mehrfache Bedingungsüberdeckung
Minimal-mehrfache Bedingungsüberdeckung
Zweigüberdeckung
Einfache Bedingungsüberdeckung
Anweisungsüberdeckung
x
y
Testverfahren x subsummiert Testverfahren y

Zweigabdeckung

Testkriterium: Jeder Zweig im CFG muss mindestens 1x ausgeführt werden

Abdeckung: $\frac{\#\ \text{ausgeführte Verzweigungen}}{\#\ \text{Verzweigungen}}$

Umfasst Anweisungsabdeckung

da bei Erreichen aller Zweige auch alle Knoten erreicht werden

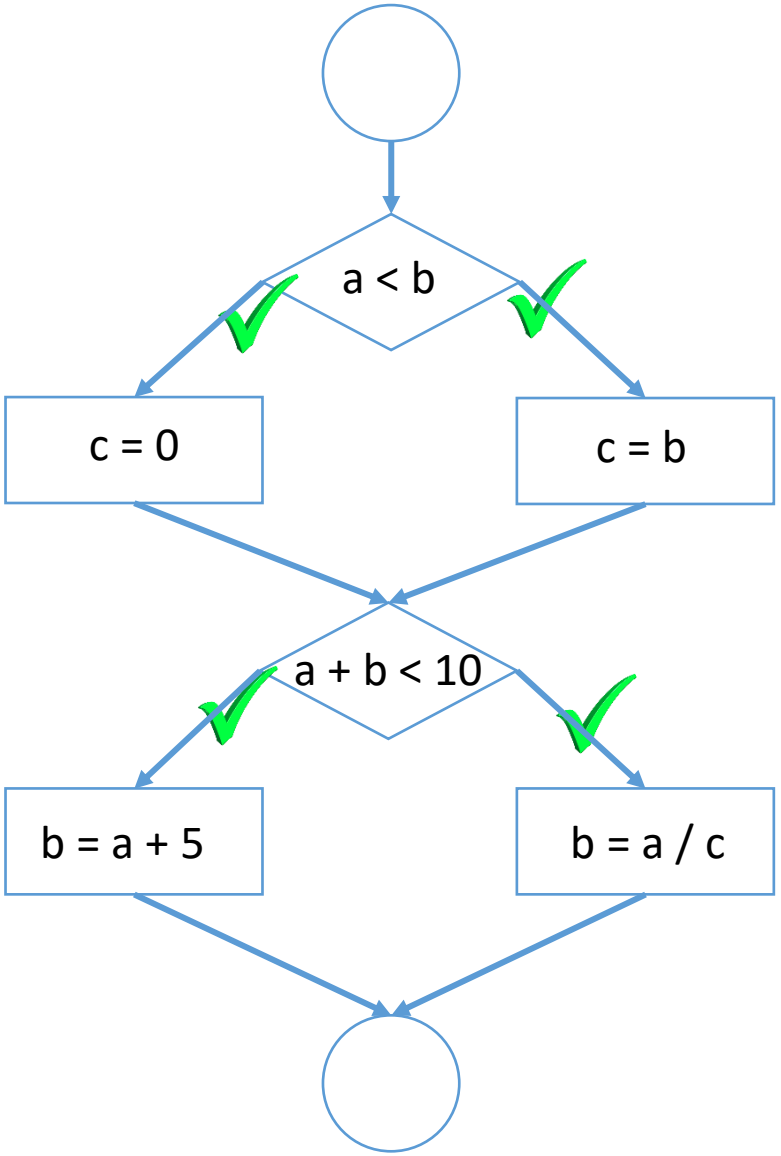
In Industrie am häufigsten genutzt

Beispiel

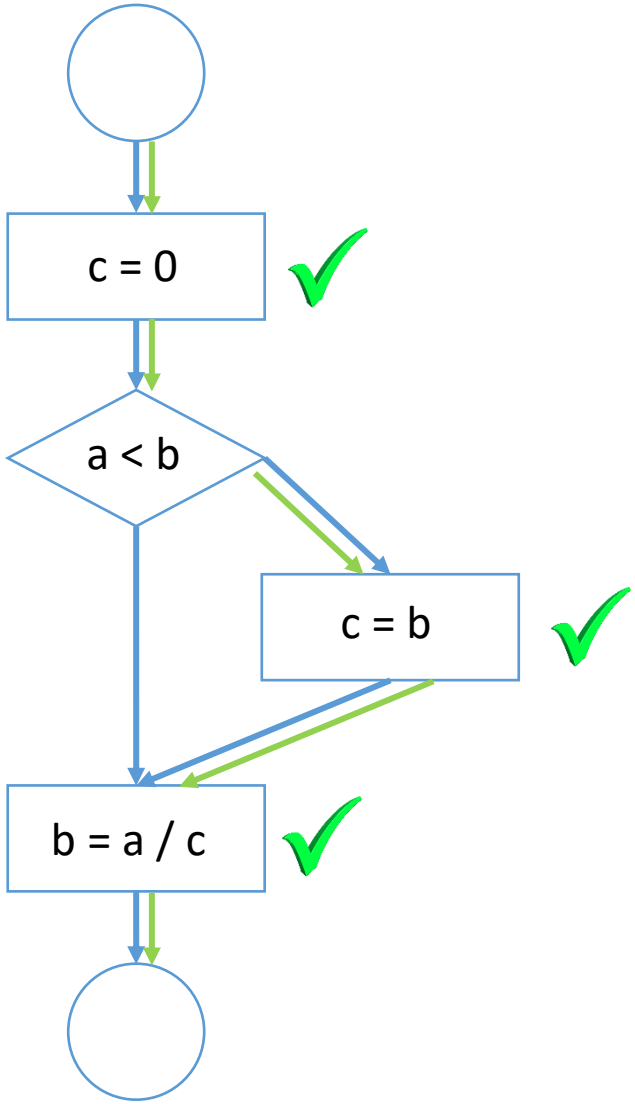
Auch Zweige ohne Anweisungen!

Nicht alle Wege müssen geprüft werden

Schleifen werden unzureichend geprüft

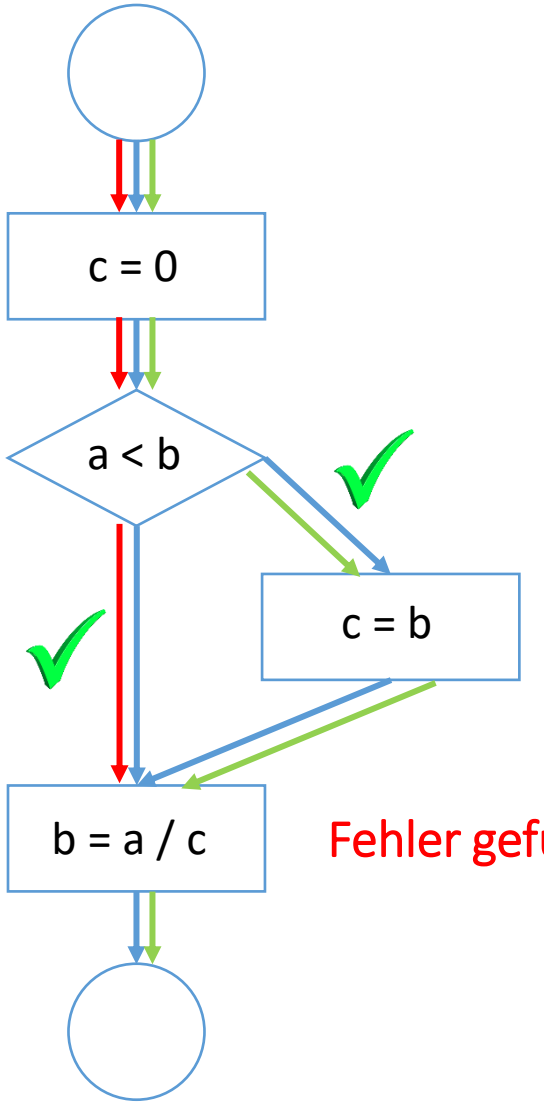


Anweisungsabdeckung



Fehler nicht gefunden!

Zweigabdeckung



Fehler gefunden!

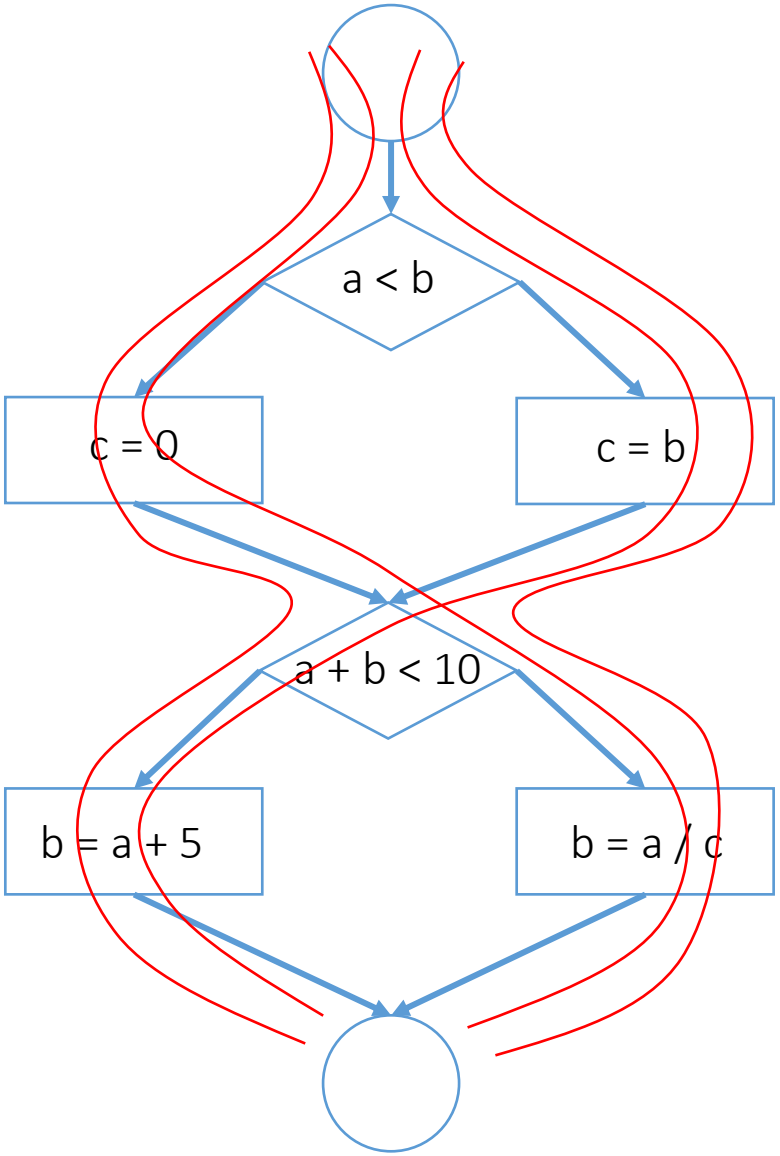
Pfadabdeckung

über einzelne Verzweigungen hinaus

Grundidee: *Folgen von Verzweigungen* im Kontrollfluss abdecken

Weit mehr Pfade als Verzweigungen

impliziert Kompromisse



Weyukers Hypothese

Die Angemessenheit eines
Abdeckungskriteriums
kann nur intuitiv definiert werden.

Annahmen

Struktur des Programms folgt der *Struktur der Aufgabenstellung* – deshalb (und dann) funktioniert strukturiertes Testen

Beispiel: *Fallunterscheidungen*

Ist das nicht gegeben, müssen eigene Kriterien her

Kriterien erfüllen

Manchmal sind Kriterien nicht erfüllbar:

Anweisungen werden u.U. nicht ausgeführt wegen *defensiver Programmierung* oder *Code-Wiederverwendung*

Bedingungen sind u.U. nicht erfüllbar wegen *voneinander abhängigen Bedingungen*

Pfade sind u.U. nicht erfüllbar wegen *voneinander abhängiger Zweige*

Kriterien erfüllen

Eine bestimmte Stelle im Code zu erreichen kann sehr schwer sein!

Selbst die besten Programme enthalten unerreichbaren Code

Große Mengen an unerreichbarem Code sind ein schweres *Wartungsproblem*

In der Praxis: Erlaube Abdeckung < 100%

Mehr Kriterien

Objektorientierte Abdeckung

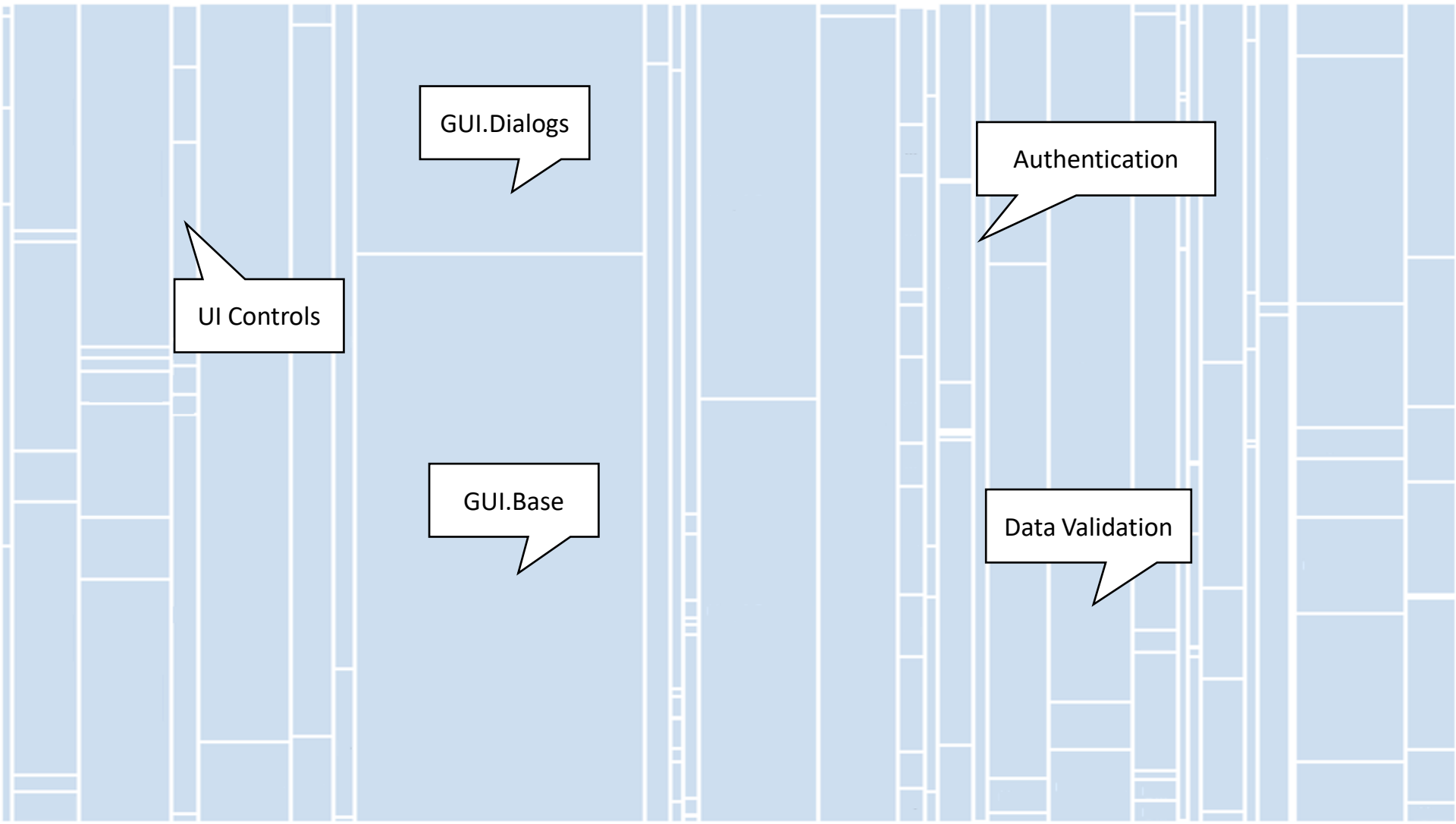
z.B. “Jeder Übergang im endlichen Automaten des Objekts muss abgedeckt sein” oder “Jedes Methodenpaar in der Folge von Aufrufen muss abgedeckt sein”

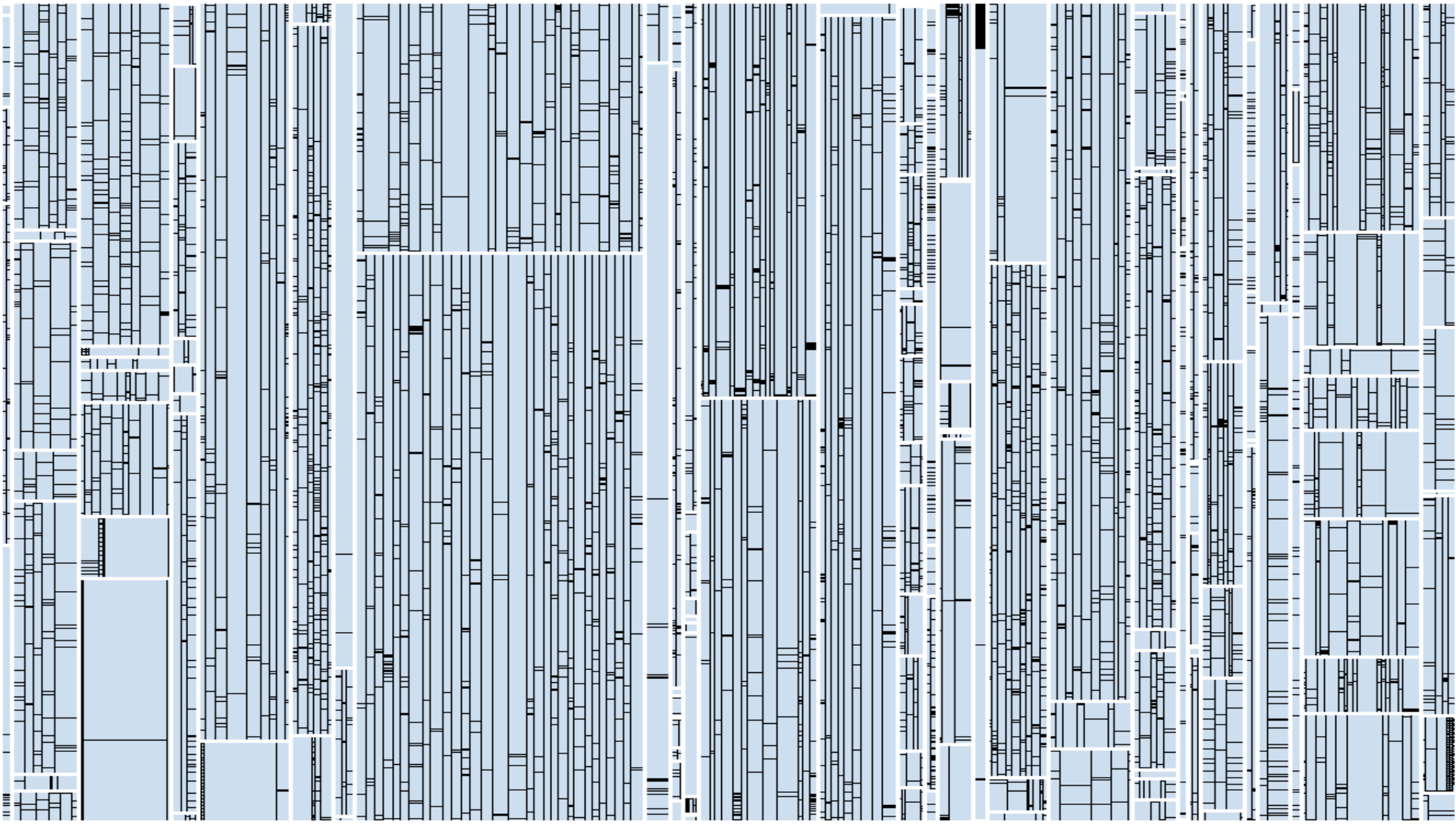
Inter-Class-Abdeckung

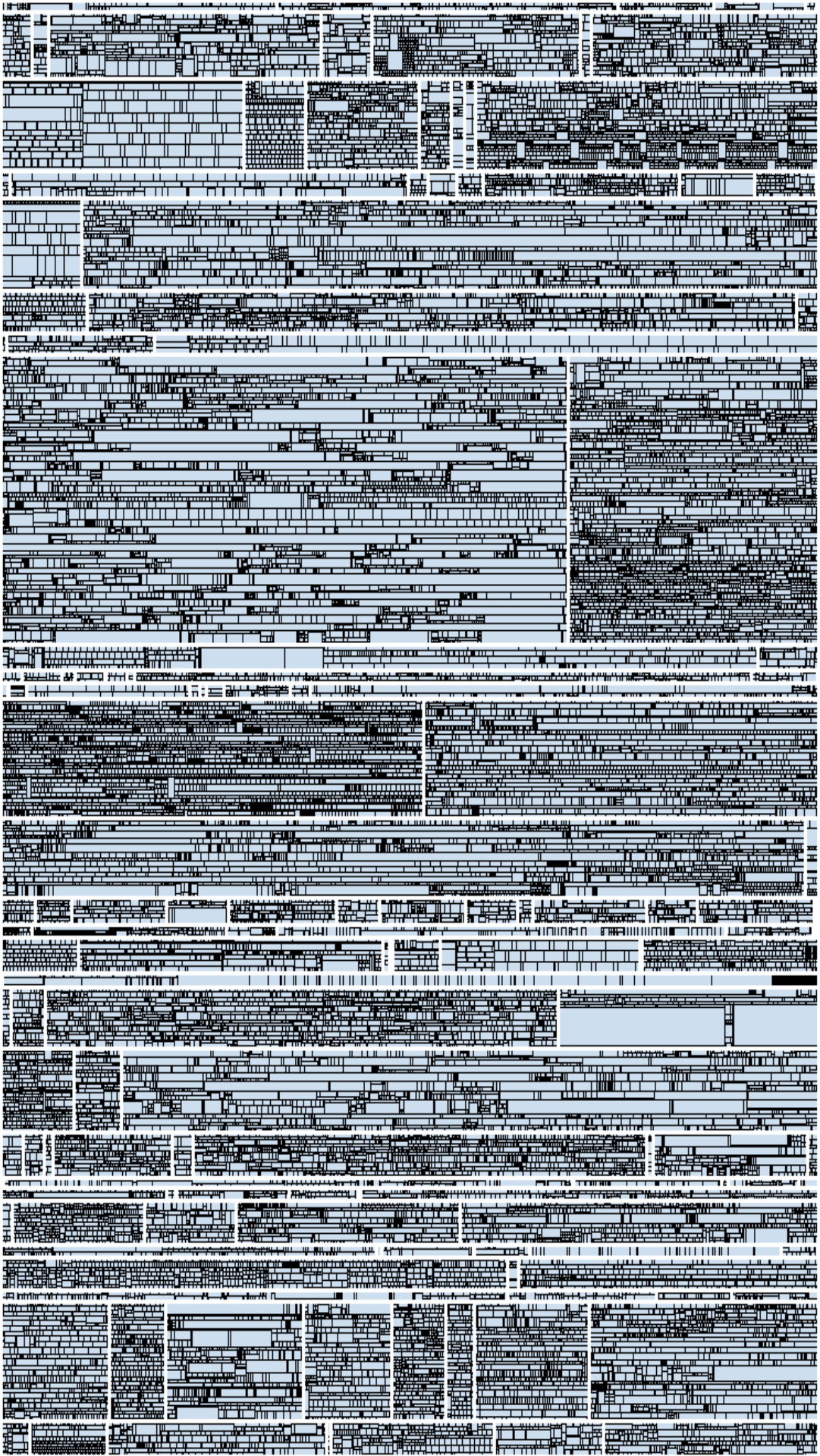
z.B. “Jede Interaktion zwischen zwei Objekten muss abgedeckt sein”

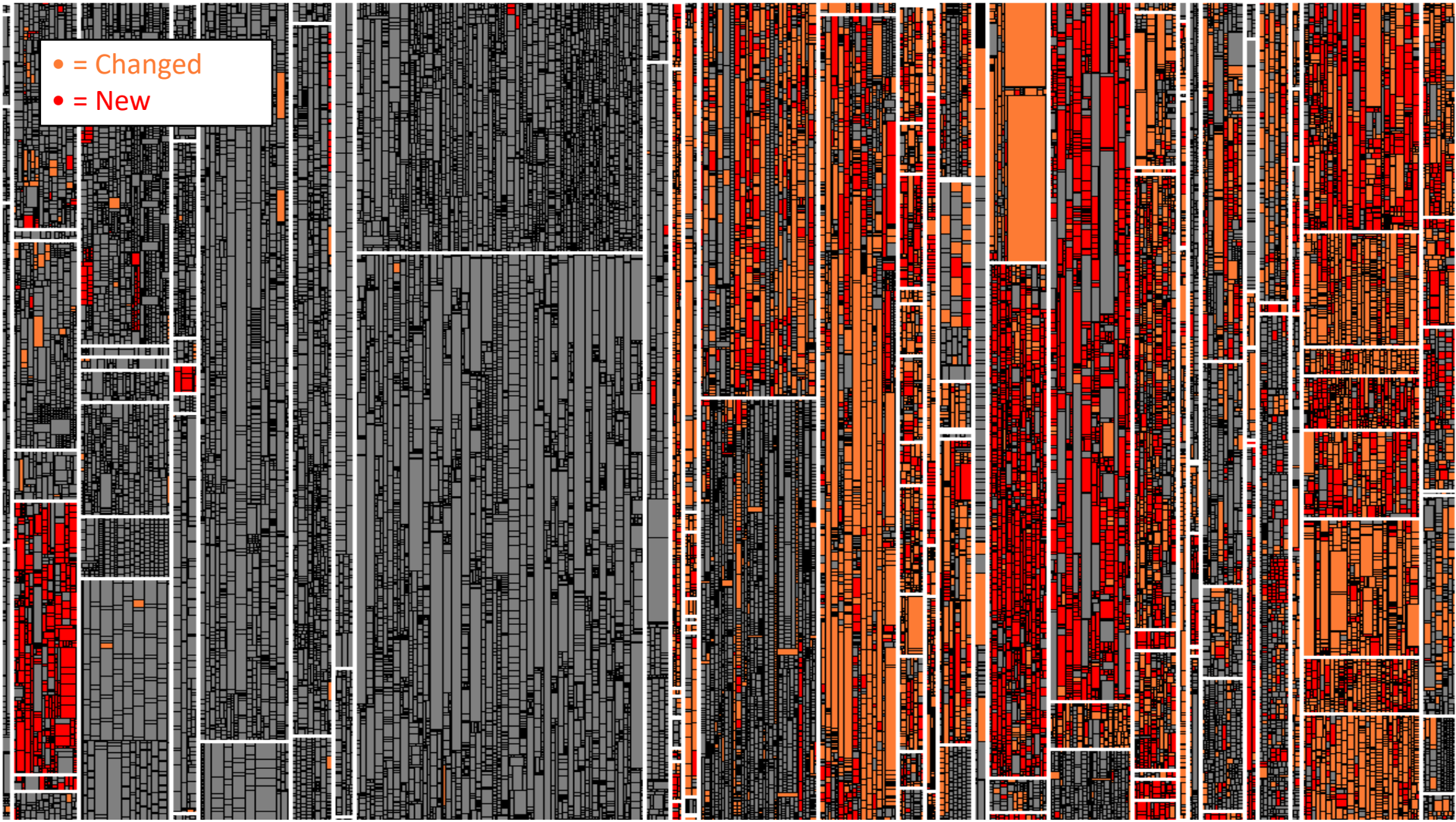
Datenflussabdeckung

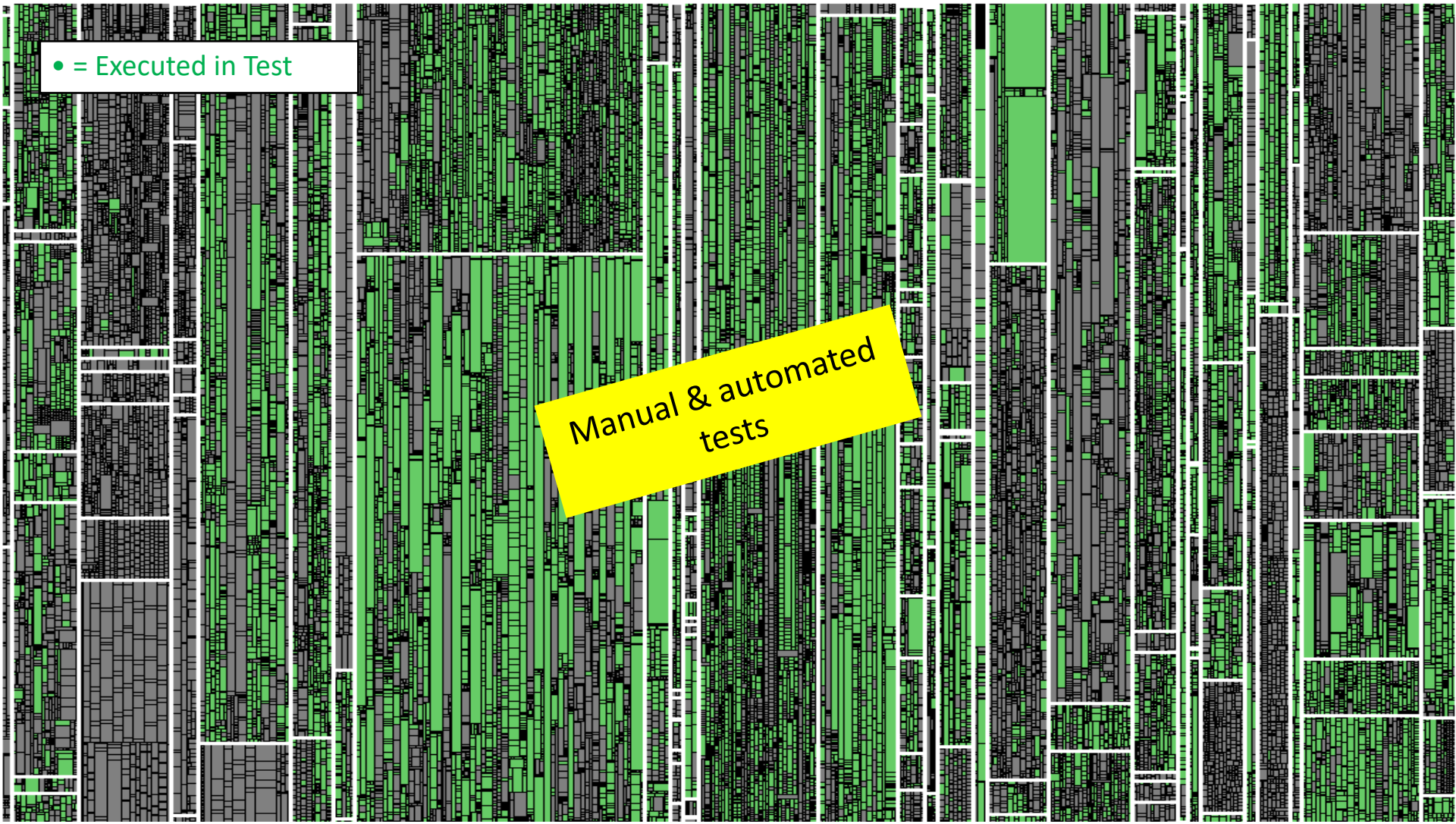
z.B. “Jedes Paar aus Definition und Benutzung einer Variablen muss abgedeckt sein”

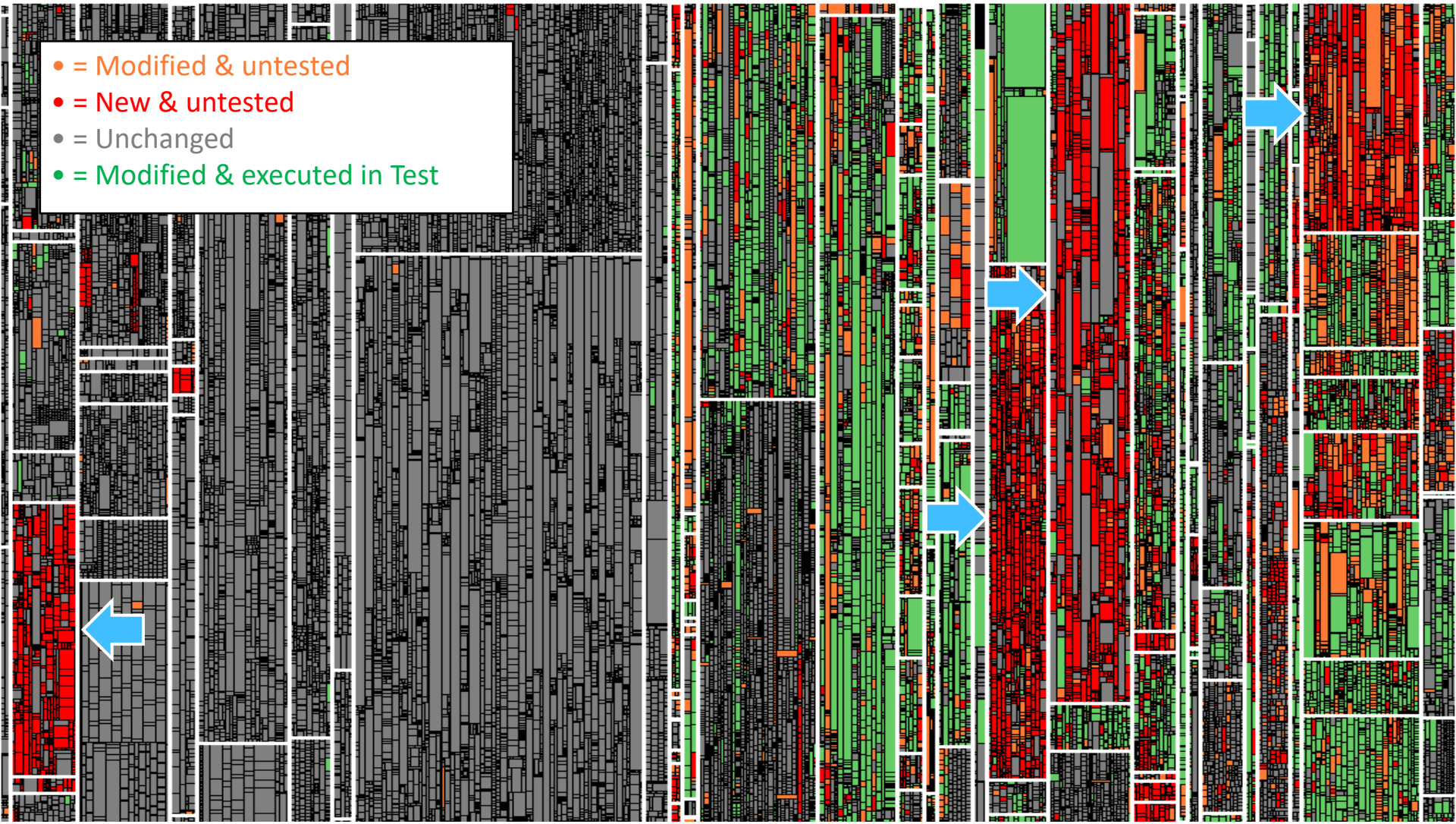

GUI.Dialogs
Authentication
UI Controls
GUI.Base
Data Validation

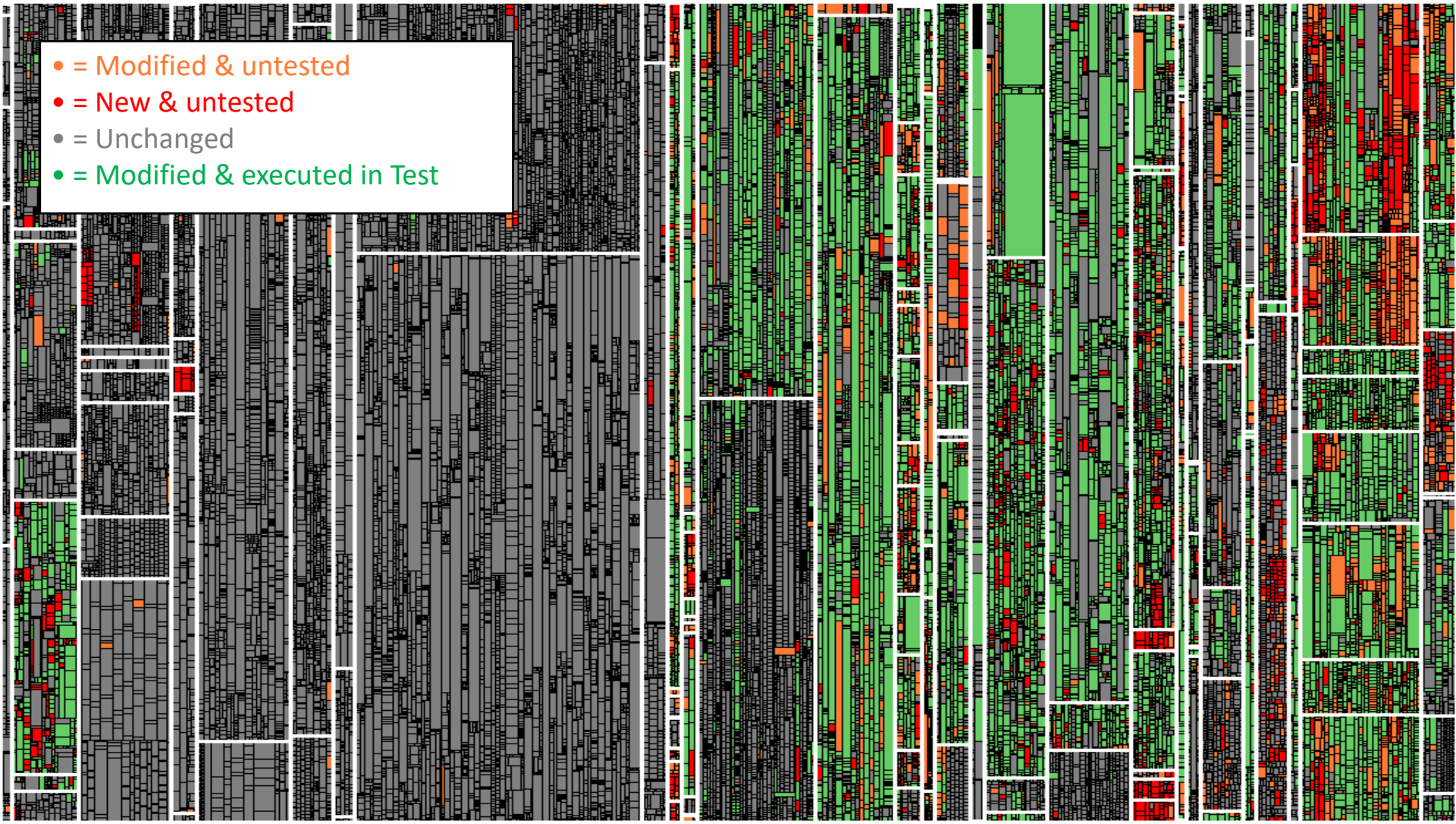





• = Changed
• = New


• = Executed in Test
Manual & automated tests


• = Modified & untested
• = New & untested
• = Unchanged
• = Modified & executed in Test


• = Modified & untested
• = New & untested
• = Unchanged
• = Modified & executed in Test