Regressionstesten

Software-Test: Verfahren und Werkzeuge

Sommersemester 2005

Dominik Schindler

Übersicht

1. Einleitung
   - Definition: Regressionstest
   - Zeitpunkt
   - Motivation
   - Probleme
2. Reduzierung des Testaufwands
   - Motivation
   - Testfälle reduzieren
   - Testfälle auswählen
     - TestTube für prozedurale Programmiersprachen
     - „Graph Traversal Algorithm“ für C++
     - „RETEST“ für Java und andere objektorientierte Sprachen
3. Fazit
4. Werkzeuge

1. Definition

Regressionstest:

Der Regressionstest bezeichnet die Wiederholung bereits durchgeführter Tests um,

- sicherzustellen, dass der Fehler behoben wurde und
- um auszuschließen, dass die Änderung irgendwelche Auswirkungen auf andere Teile der Software hat.

- Grundlage sind die Black-Box-Tests und White-Box-Tests aus dem Modultest

1. Definition

2 Arten von Regressionstests:

- **progressiver Regressionstest:** Spezifikation hat sich geändert, z.B. aufgrund „*adaptive maintenance*“ oder „*perfective maintenance*“, d.h., modifiziertes Programm gegen modifizierte Spezifikation testen → geeignete Testfälle müssen hinzugefügt werden

- **korrigierender Regressionstest:** Spezifikation unverändert („*corrective maintenance*“), d.h., nur die geänderten Anweisungen müssen mit bestehenden Testfällen getestet werden

1. Zeitpunkt (1)

- Regressionstests werden immer dann ausgeführt, wenn sich die Software geändert hat
- Sie können bereits im kompletten Software-Entwicklungszyklus durchgeführt werden
- Größtenteils werden Regressionstests aber in der Wartungsphase durchgeführt

1. Zeitpunkt (2)

mögliche Änderungen in der Wartungsphase:

„corrective maintenance“

- **Entfernen aktueller Fehler:** Korrigiert Software- und Implementationsfehler, damit System korrekt funktioniert

„perfective maintenance“

- **Anpassung an neue Benutzeranforderung:** z.B. Hinzufügen neuer Funktionalität, Verbesserung der Performance, …

„preventive maintenance“

- **vorbeugende Maßnahmen:** Verbesserung der SW-Qualität, Anpassung der Dokumentation, …

„adaptive maintenance“

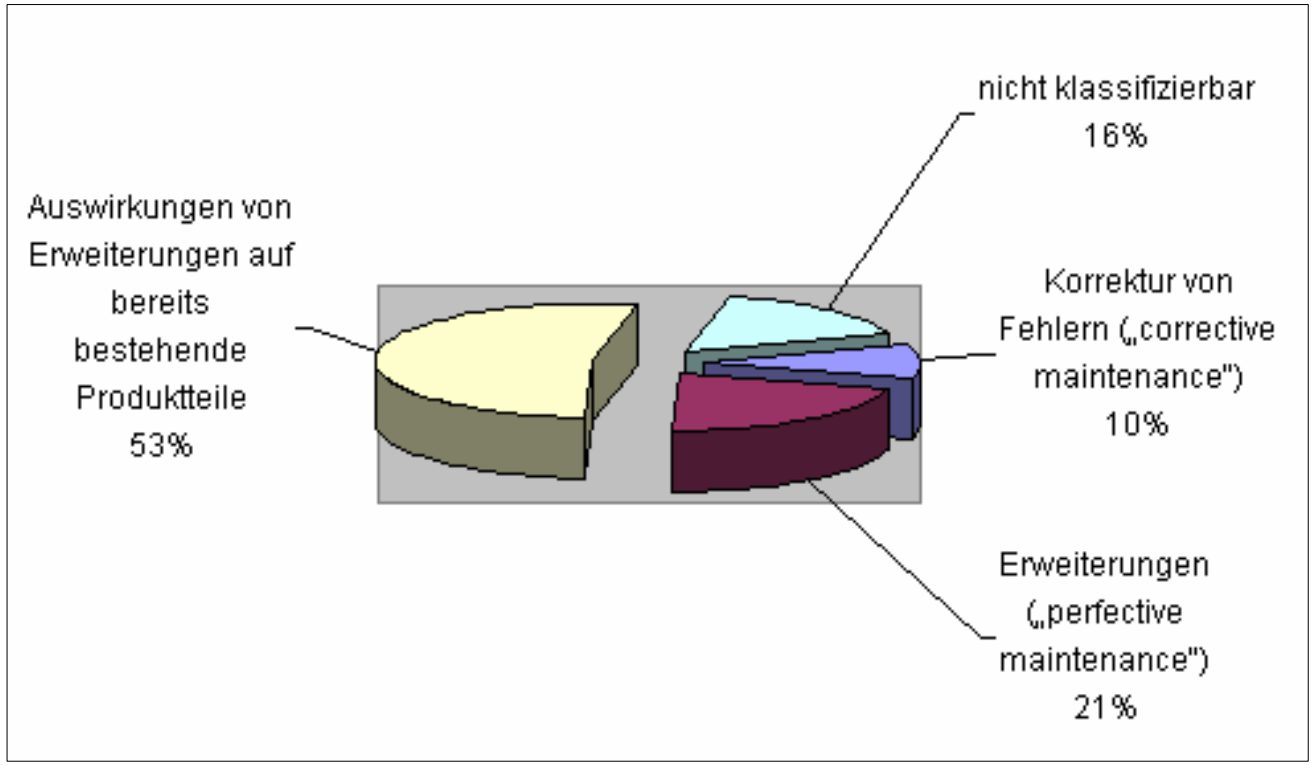
- **Anpassung an neue Umgebung:** z.B. neuer Übersetzer, neue Hardware, neues Betriebssystem, …

1. Motivation (1)

- Regressionstests werden durchgeführt, um Vertrauen in die geänderte Software zu gewinnen
- **Hauptproblem bei Software-Wartung:** Erzeugen von Fehlern durch Änderung, Erweiterung und Fehlerkorrekturen
- Wahrscheinlichkeit, dass neue Fehler hinzugefügt werden:
  - 20% - 50%, laut Sharp (1993)
  - in manchen Fällen sogar 50% - 80%, laut Hetzel (1984)
- In vielen Software-(Qualitäts-)Standards gefordert

1. Motivation (2)

Wahrscheinlichkeit klassifiziert nach Art der Änderung



Quelle: Wallmüller, 1990

1. Probleme

- Welche Testfälle für die geänderte Software auswählen?

  („*regression-test-selection problem*“)

- Welche Testfälle müssen hinzugefügt werden, um neue Funktionalität zu testen?

  („*test-suite-augmentation problem*“)

- Manuelle Durchführung von Regressionstests ist Unsinn!

Übersicht

1. Einleitung
   - Definition: Regressionstest
   - Zeitpunkt
   - Motivation
   - Probleme
2. Reduzierung des Testaufwands
   - Motivation
   - Testfälle reduzieren
   - Testfälle auswählen
     - TestTube für prozedurale Programmiersprachen
     - „Graph Traversal Algorithm“ für C++
     - „RETEST“ für Java und andere objektorientierte Sprachen
3. Fazit
4. Werkzeuge

2. Reduzierung des Testaufwands

Problem:

- Die Wiederholung aller vorhandenen Testfälle ist wirtschaftlicher Unsinn.
- Bei konsequenter Durchführung wäre bei vielen Projekten kein Ende in Sicht.
- **Aber:** Zu wenig Testfälle können unter Umständen die Software nicht ausreichend testen!

Lösung:

- Anzahl der vorhandenen Testfälle verringern
- Testfälle aus den vorhandenen Testfällen auswählen, um nur die Änderung(en) testen

2. Testfälle verringern

- **Entfernung von veralteten Testfällen:** Diese Testfälle wurden hinzugefügt, um spezielle Änderungen und Erweiterungen zu testen.
- **Weglassen von redundanten Testfällen:** Testen dieselben Teile der Software; kann z.B. auftreten wenn mehrere Tester Testfälle erstellen
- **evtl. Zusammenfassen von semantisch gleichartigen Testfällen** zu einem Testfall

2. Testfälle auswählen

- Allen Selektionsverfahren gemeinsam ist die statische und dynamische Analyse des Quelltextes:
  - **statische Analyse:** Suche nach den Änderungen
  - **dynamische Analyse:** Überdeckung der Testfälle bestimmen

- **Definitionen:**
  - `P:` Programm/-teil, das gestestet wird
  - `T:` eine Menge von Testfällen (sog. Testpaket)
  - `P(i):` Ausführung von `P` mit Eingabe `i`
  - `P‘:` geändertes Programm `P`
  - `t:` ein Testfall aus der Menge der Testfälle `T`, `t = (i, o)`, mit `o` = erwartetes Ergebnis

2. Bewertung der Selektionsverfahren (1)

- Bewertung der Selektionsverfahren anhand folgender Kriterien (nach Polak 2004):
  - **„Inclusiveness“:** bestimmt, ob ein Verfahren sicher ist
  - **„Precision“:** gibt an, wie präzise die Auswahl eines Verfahren ist
  - **„Efficiency“:** wie Effizient ist ein Verfahren
  - **„Generality“:** Anwendbarkeit auf andere Programmierumgebung

- **Definition „modifikation-traversierend“:** Einen Testfall t bezeichnet man als modifikation-traversierend („modification-traversing“), wenn er neuen oder geänderten Code in P’ ausführt, bzw. wenn er Code in P ausführte, der in P’ gelöscht wurde.

2. Bewertung der Selektionsverfahren (2)

- **„Inclusiveness“**-Kriterium
  - Misst die Anzahl der selektierten modifikations-traversierenden Testfälle
  - Angenommen, T enthält genau n modifikations-traversierende Testfälle für P und P‘, und ein Verfahren V selektiert m von n Testfällen, dann ist die „Inclusiveness“ von V bzgl., P, P‘ und T:

$$I = \frac{m}{n} \cdot 100, \quad \text{für } n > 0, \text{ sonst } 100\%$$

  - Beispiel: T hat 10 modifikations-traversierende Testfälle für P und P‘, und ein Verfahren V selektiert davon 9 Testfälle, dann ist die „Inclusiveness“ von V bzgl. P, P‘ und T:

$$I = \frac{9}{10} \cdot 100 = 90\%$$

  - Verfahren mit einer „Inclusiveness“ von 100% bezeichnet man als **sicher**

2. Bewertung der Selektionsverfahren (3)

- **„Precision“**-Kriterium:
  - Mist die Anzahl der nicht modifikations-traversierenden Testfälle, die nicht selektiert werden
  - Angenommen, T enthält genau n nicht modifikations-traversierende Testfälle für P und P‘, und ein Verfahren V selektiert m von den n Testfällen nicht, dann ist die „Precision“ von V bzgl., P, P‘ und T:

$$P = \frac{m}{n} \cdot 100, \quad \textit{für } n > 0, \textit{ sonst } 100\%$$

  - Beispiel: T hat 30 nicht modifikations-traversierende Testfälle für P und P‘, und ein Verfahren V selektiert davon 6 Testfälle, d.h., 24 wurden nicht selektiert, dann ist die „Precision“ von V bzgl. P, P‘ und T:

$$P = \frac{24}{30} \cdot 100 = 80\%$$

2. Bewertung der Selektionsverfahren (4)

- **„Efficiency“**-Kriterium: Betrachtet den Zeit- und Speicherplatzbedarf

- **„Generality“**-Kriterium: Betrachtet die Breite der Anwendbarkeit eines Verfahrens, z.B. betrachtet diese Kriterium ein Verfahren als schlecht,
  - das nicht alle Konstrukte einer Sprache berücksichtigt
  - das nicht mit realistischen Modifikationen umgehen kann
  - das von einer bestimmten Test- und Wartungsumgebung abhängig ist
  - das auf bestimmte Analysewerkzeuge angewiesen ist

Übersicht

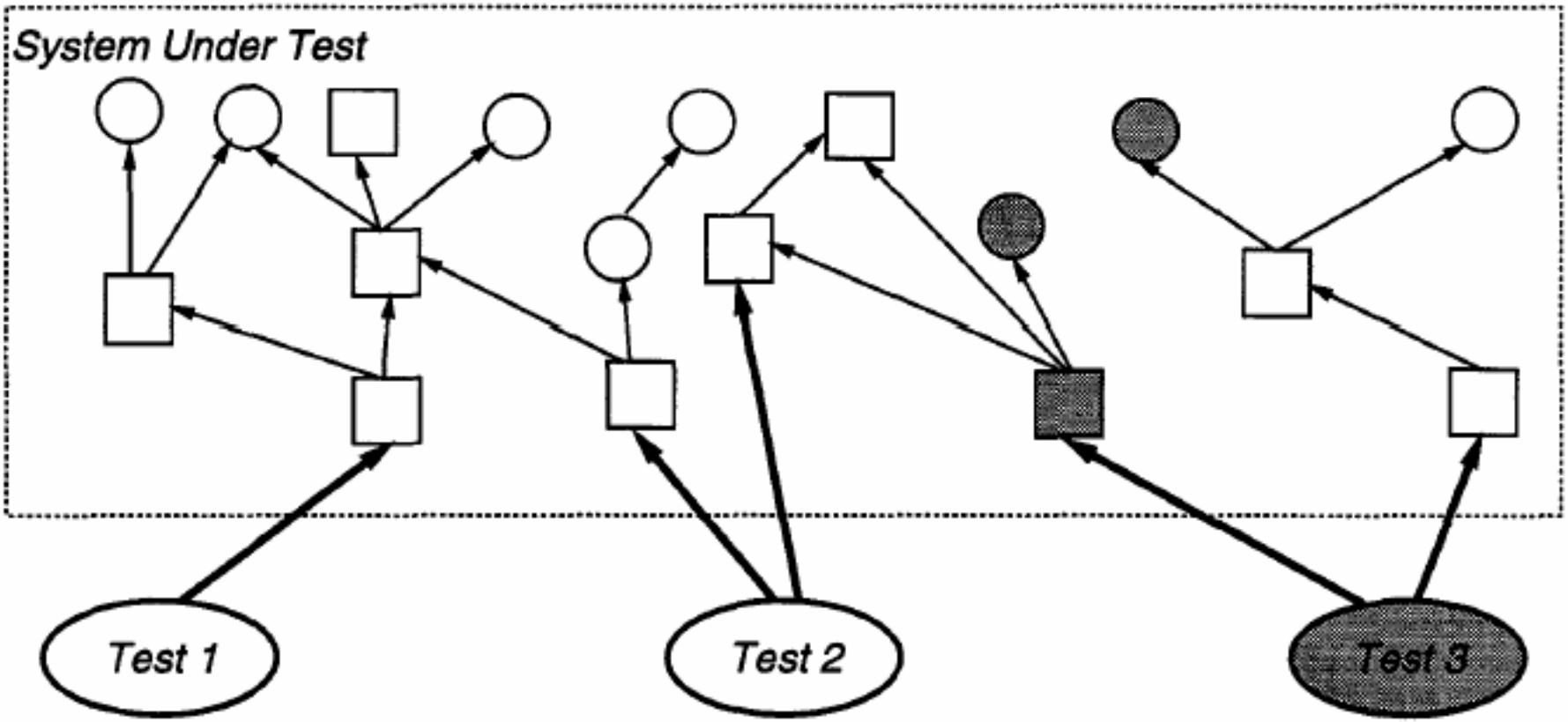
1. Einleitung
   - Definition: Regressionstest
   - Zeitpunkt
   - Motivation
   - Probleme
2. Reduzierung des Testaufwands
   - Motivation
   - Testfälle reduzieren
   - Testfälle auswählen
     - TestTube für prozedurale Programmiersprachen
     - „Graph Traversal Algorithm“ für C++
     - „RETEST“ für Java und andere objektorientierte Sprachen
3. Fazit
4. Werkzeuge

2.1. TestTube (1)

- entwickelt von Yih-Farn Chen et al, 1994
- unterstützt prozedurale Programmiersprachen, wie C, Pascal, …
- Firewall-Verfahren von White und Abdullah ähnelt diesem Verfahren und wurde auf das Klassenkonzept übertragen
- **statische Analyse** sucht nach veränderten Funktionen bzw. Variablen und die davon abhängigen Funktionen/Variablen
- **dynamische Analyse** ermittelt die Überdeckung der einzelnen Testfälle

2.1. TestTube (2)

Quelle: Yih-Farn Chen et al, 1994



- Rechtecke stellen Funktionen und Kreise Variablen dar
- Kanten zeigen Abhängigkeiten zw. Funktionen und Variablen
- graue Elemente stellen modifizierte Funktionen und Variablen, bzw. ausgewählte Testfälle dar

2.1. TestTube (3)

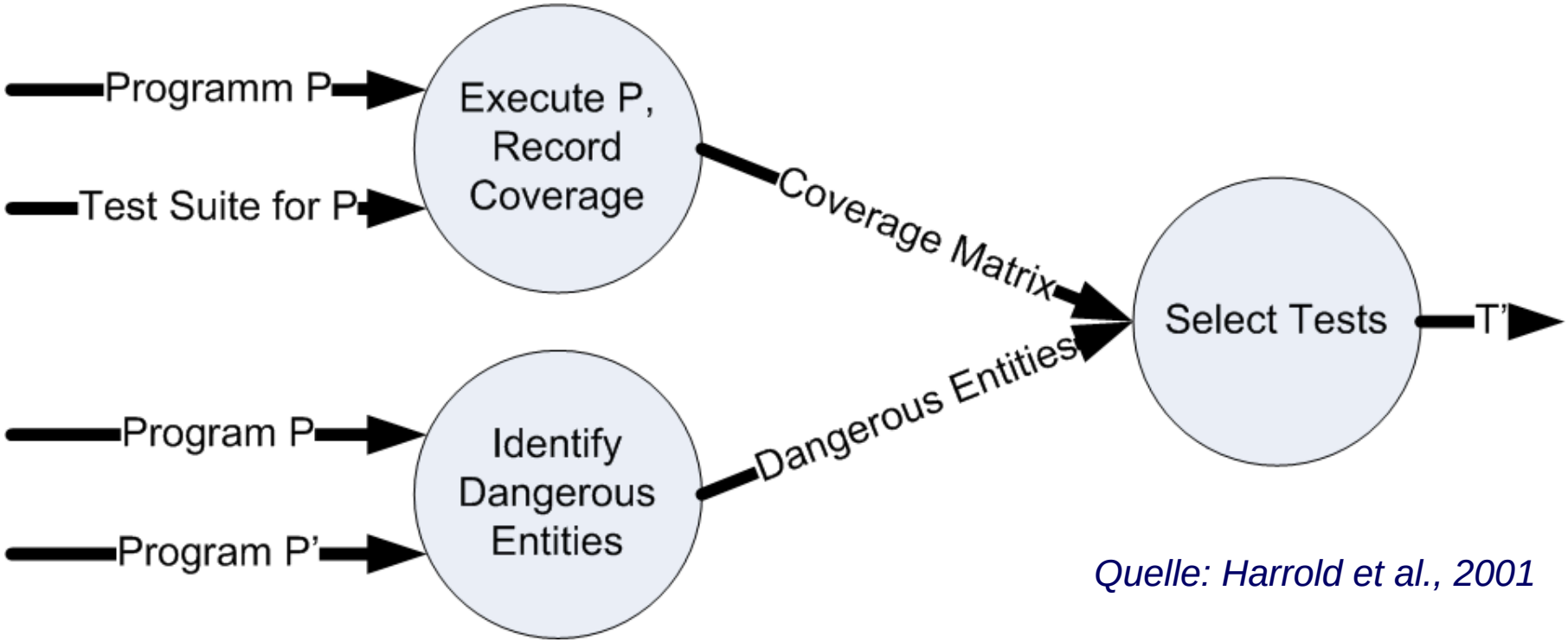
- Da es keine objektorientierten Features wie Polymorphismus, dynamisches Binden, Overloading, usw. in prozeduralen Programmiersprachen gibt, ist das Verfahren simple sowie effizient.

- **Bewertung:**
  - **„Inclusiveness“:** 100%, d.h., Verfahren ist sicher
  - **„Precision“:** 100%
  - **„Efficiency“:** gut, wg. die Einfachheit des Verfahrens
  - **„Generality“:** schlecht, weil es nicht für „moderne“ Programmiersprachen, wie z.B. objektorientierte Sprachen geeignet ist

2.2. „Graph Traversal Algorithm“ (1)

- Entwickelt von Gregg Rothermel und Mary Jean Harrold, 2000
- Wurde für die Programmiersprache C++ entwickelt, da es aber auf Kontrollflussgraphen basiert, ist es leicht portierbar
- **statische Analyse** erstellt für die ursprüngliche und geänderte Version einen Kontrollflussgraphen und bestimmt anschließend die → gefährlichen Kanten durch den Vergleich der beiden Kontrollflussgraphen
- **dynamische Analyse** bestimmt die (Methoden-) Überdeckung jedes einzelnen Testfalles und erstellt eine Überdeckungsmatrix

2.2. „Graph Traversal Algorithm“ (2)



Quelle: Harrold et al., 2001

- Die **„Select-Tests“ Komponente** gleicht die gefährlichen Kanten mit den Kanten aus der Überdeckungsmatrix ab und wählt die Testfälle T‘ aus
- T‘ sind die Testfälle aus der ursprünglichen Testfallmenge T, um speziell die Modifikationen zu Testen

2.2. „Graph Traversal Algorithm“ (3)

Definition „gefährliche Kante“:

- Eine gefährliche Kante e ist eine Kante, so dass für jede Eingabe `i` die Anweisung e von `P` überdeckt wird und `P(i)` und `P’(i)` sich unterschiedlich verhalten.

- Dieses unterschiedliche Verhalten kann entweder durch eine Kante hervorgerufen werden, die in `P’` auf einen anderen Knoten zeigt als in `P`, oder in `P’` nicht mehr vorhanden ist.

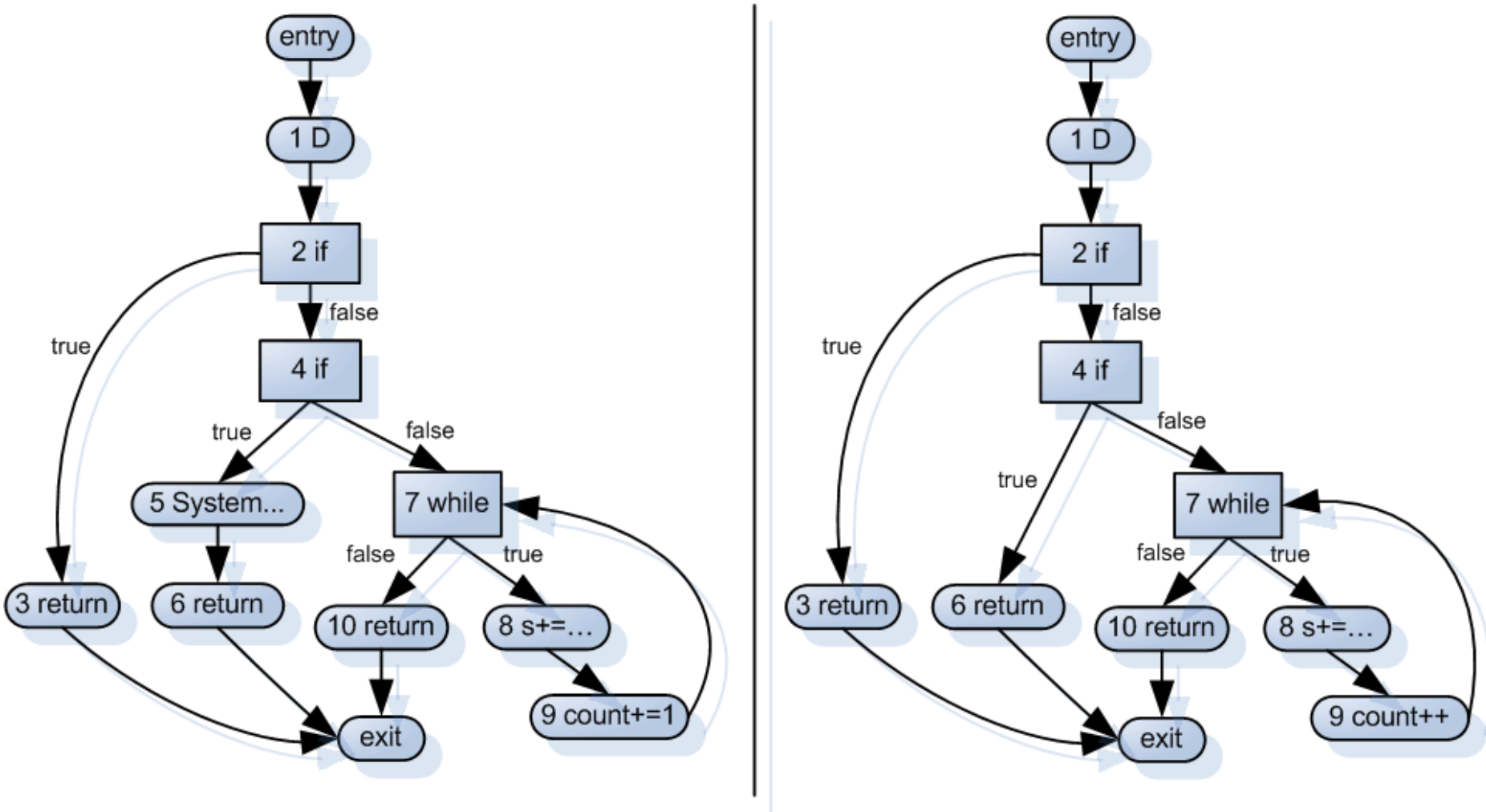
2.2. „Graph Traversal Algorithm“ (4)

Beispiel:

```
public static float avg(Float[] floats) {
(1)  int count = 0
(1)   float s = 0;

(2)  if (floats.length == 0) {
(3)        return 0;
     }
(4)    if (floats.length > 100) {
(5)  System.out.println(“Zu viele Elemente im
                                              Array!”);
(6)          return -1;
     }
(7)    while (count < floats.length) {
(8)        s += floats[count].floatValue();
(9)        count += 1;
     }

(10) return s / (float)count;
}
```

```
public static float avg’(Float[] floats) {
(1)  int count = 0,
(1) float s = 0;

(2)  if (floats.length == 0) {
(3)        return 0;
     }
(4)    if (floats.length > 100) {
(5)
(6)          return -1;
     }
(7)    while (count < floats.length) {
(8)        s += floats[count].floatValue();
(9)        count++;
     }

(10)  return s / (float)count;
}
```

2.2. „Graph Traversal Algorithm“ (5)



2.2. „Graph Traversal Algorithm“ (6)

- Beispiel-Testpaket T für avg

Testfall	Eingabe	Erwartete Ausgabe
1	[]	0
2	[0,1,2,…,99,100]	„Zu viele Elemente im Array!“, -1
3	[1]	1.0
4	[1, 2]	1.5

- Kantenüberdeckungsmatrix für Testpaket T von avg

Testfall	Überdeckte Kanten
1	(entry,1), (1,2), (2,3), (3,exit)
2	(entry,1), (1,2), (2,4), **(4,5)**, (5,6), (6,exit)
3	(entry,1), (1,2), (2,4), (4,7), (7,8), **(8,9)**, (9,7), (7,10), (10, exit)
4	(entry,1), (1,2), (2,4), (4,7), [(7,8), **(8,9)**, (9,7)]², (7,10), (10, exit)

- Die Testfälle 2, 3 und 4 müssen bei `avg‘` wiederholt werden

2.2. „Graph Traversal Algorithm“ (6)

- **Bewertung:**
  - **„Inclusiveness“:** 100%, d.h., Verfahren ist sicher
  - **„Precision“:** 100%
  - **„Efficiency“:** schlecht, da externe Klassen (z.B. Bibliotheken) ebenfalls analysiert werden müssen
  - **„Generality“:** schlecht, da das Verfahren keine Exceptions unterstützt und nur für einen Teil des Sprachumfangs von C++ implementiert wurde

2.3. „RETEST“ (1)

- Entwickelt von Harrold et al, 2001
- Wurde für Java entwickelt, das Konzept wurde aber so allgemein gehalten, so dass es leicht auf andere objektorientierte Sprachen übertragen werden kann
- Erweitert das „GTA“-Verfahren um objektorientierte Features wie Vererbung, Polymorphismus und dynamisches Binden
- Behandelt nicht nur den Kontrollflussgraphen innerhalb einer Methode, sondern auch den Kontrollfluss über Methodengrenzen hinweg
- Um diese Erweiterung darstellen zu können, wurde der Kontrollflussgraph zum JIG („Java Interclass Graph“) erweitert.
- Es ist eines des ersten Verfahren, das Java komplett unterstützt!

2.3. „RETEST“ (2)

Variablen und Objekttypen (1)

- Bei jeder Instantiierung einer Variablen (z.B. mittels **new()**) wird zusätzlich zum Objekttyp die Klassenhierarchie durch einen global klassifizierenden Klassennamen mit dargestellt.

- Dieser global klassifizierende Klassenname beinhaltet auch alle implementierten Schnittstellen in alphabetischer Reihenfolge.

- Beispiel: Eine Klasse `B` aus dem Paket `foo` erbt von Klasse `A` aus dem gleichen Paket und implementiert die Schnittstelle `I` aus dem Paket `bar`:

  ➔ `java.lang.Object:bar.I:foo.A:foo.B`

2.3. „RETEST“ (3)

Variablen und Objekttypen (2)

- Im vorherigen Verfahren führte eine Änderung innerhalb des Deklarationsknoten `D` dazu, dass alle Testfälle ausgewählt werden

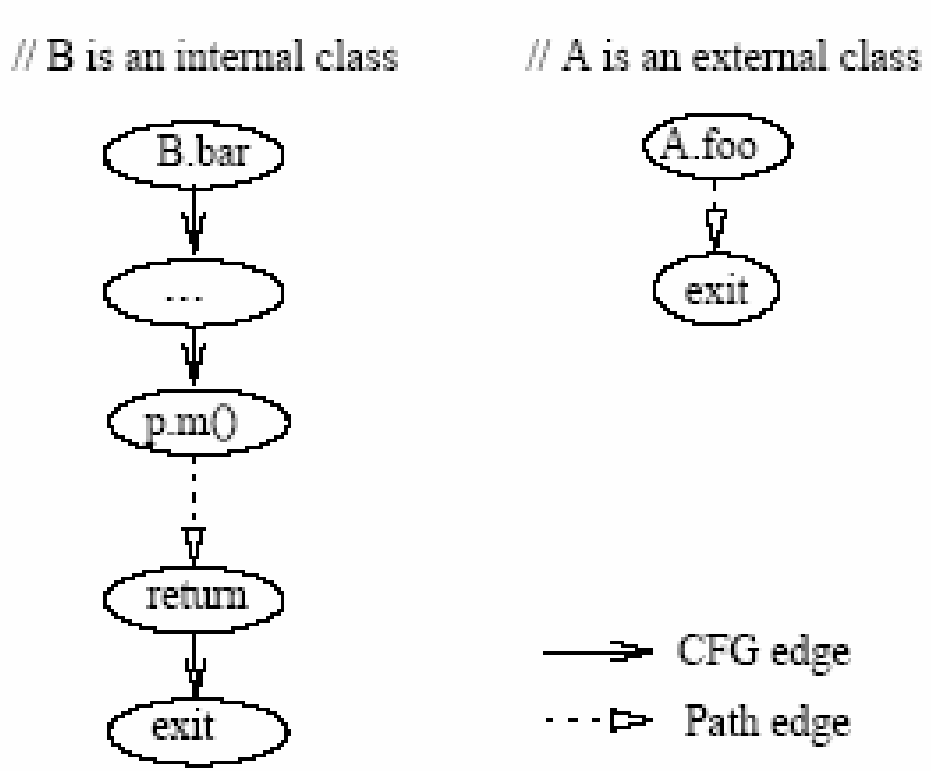
- Deshalb wird jeder Variablen primitiven Typs (`int`, `boolean`, …) sein Typ an den Variablennamen angehängt, z.B. zaehler_int, wahr_boolean, …

- Die Erkennung einer Änderung des Variablentyps bzw. in der Klassenhierarchie wird an die Stelle verschoben, an der die Variable benutzt bzw. instantiiert wird.

2.3. „RETEST“ (4)

Interne und externe Methoden

- Jede interne Methode wird durch einen eigenen Kontrollflussgraphen dargestellt.
- Der Kontrollflussgraph auf der Aufruferseite wird mit einem `call`-Knoten und einem `return`-Knoten erweitert (z.B. `p.m()`).
- Jede externe Methode, die von einer internen Methode aufgerufen wird, wird durch einen zusammengeklappten Kontrollflussgraphen dargestellt.

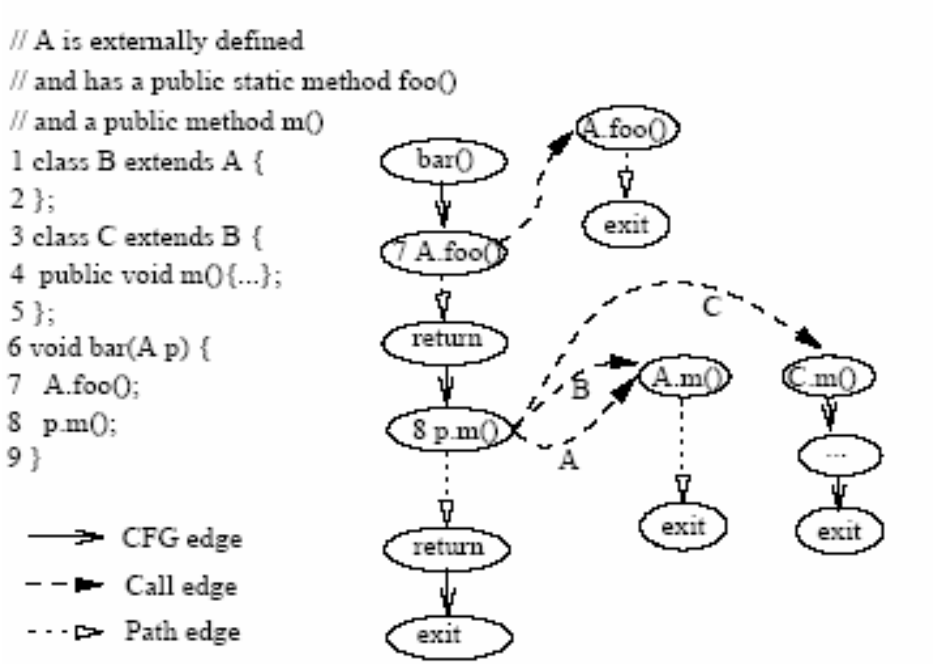


Quelle: Harrold et al, 2001

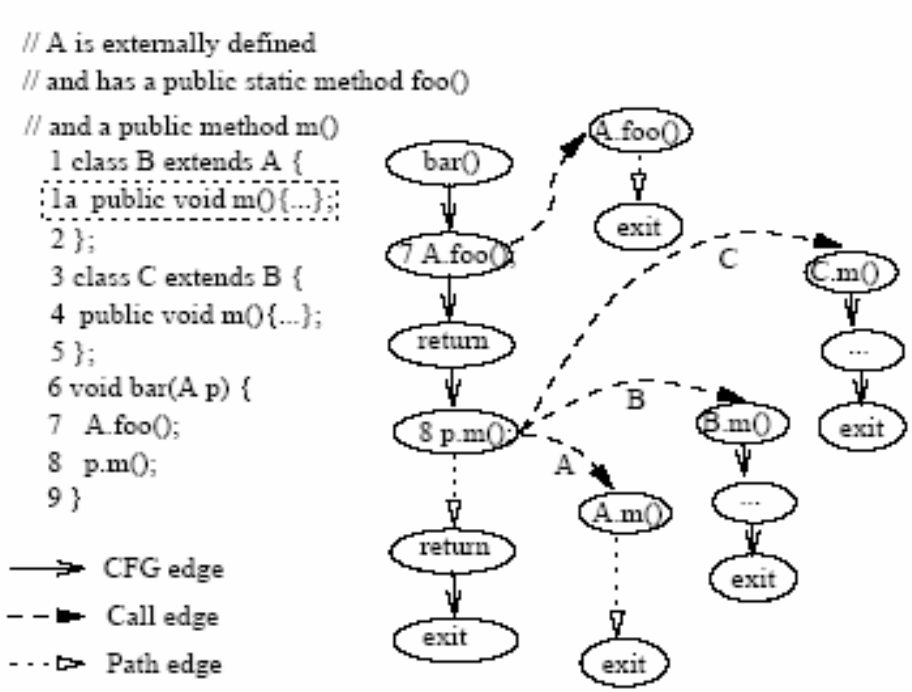
2.3. „RETEST“ (5)

Interprozedurale Interaktionen zwischen internen Methoden

- **nicht virtueller Aufruf:** Der aufrufende Knoten hat nur eine ausgehende `call`-Kante.
- **virtueller Aufruf:** Der aufrufende Knoten ist mit dem Eingangsknoten aller in Frage kommender Methoden über eine `call`-Kante verbunden.



(a) representing internal method calls in foo() that uses B and C



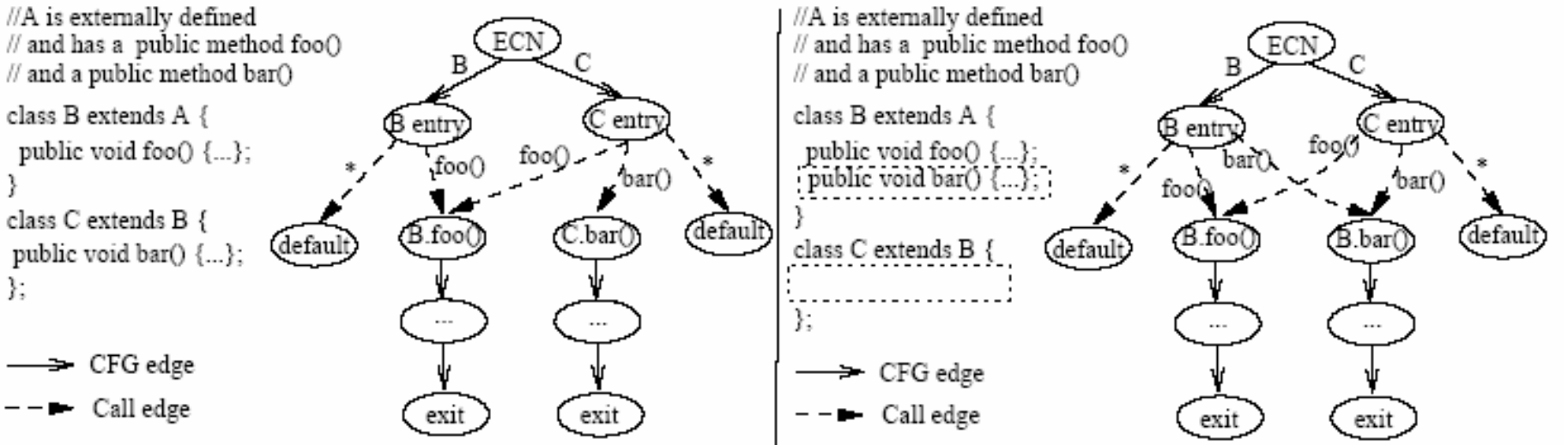
(b) representing internal method calls in foo() that uses modified B and C

Quelle: Harrold et al, 2001

2.3. „RETEST“ (6)

Interprozedurale Interaktionen durch externe Methoden

- Die einzigen internen Methoden, die von Außen aus aufgerufen werden können, sind Methoden, die Methoden externer Klassen (z. B. Klasse A) überschreiben.
- ECN-Knoten = „External Code Node“, repräsentiert Aufruf von Außen
- Für jede von einer externen Klasse aus zugängliche interne Klasse wird ein sog. Klasseneingangsknoten erstellt.
- Ein `default`-Knoten steht für alle Methoden der Klasse A, die durch ein Objekt A aufgerufen werden können, aber extern definiert sind, z.B. Methode `bar()` bei B.



(a) representing external method calls for B and C

(b) representing external method calls for modified B and C

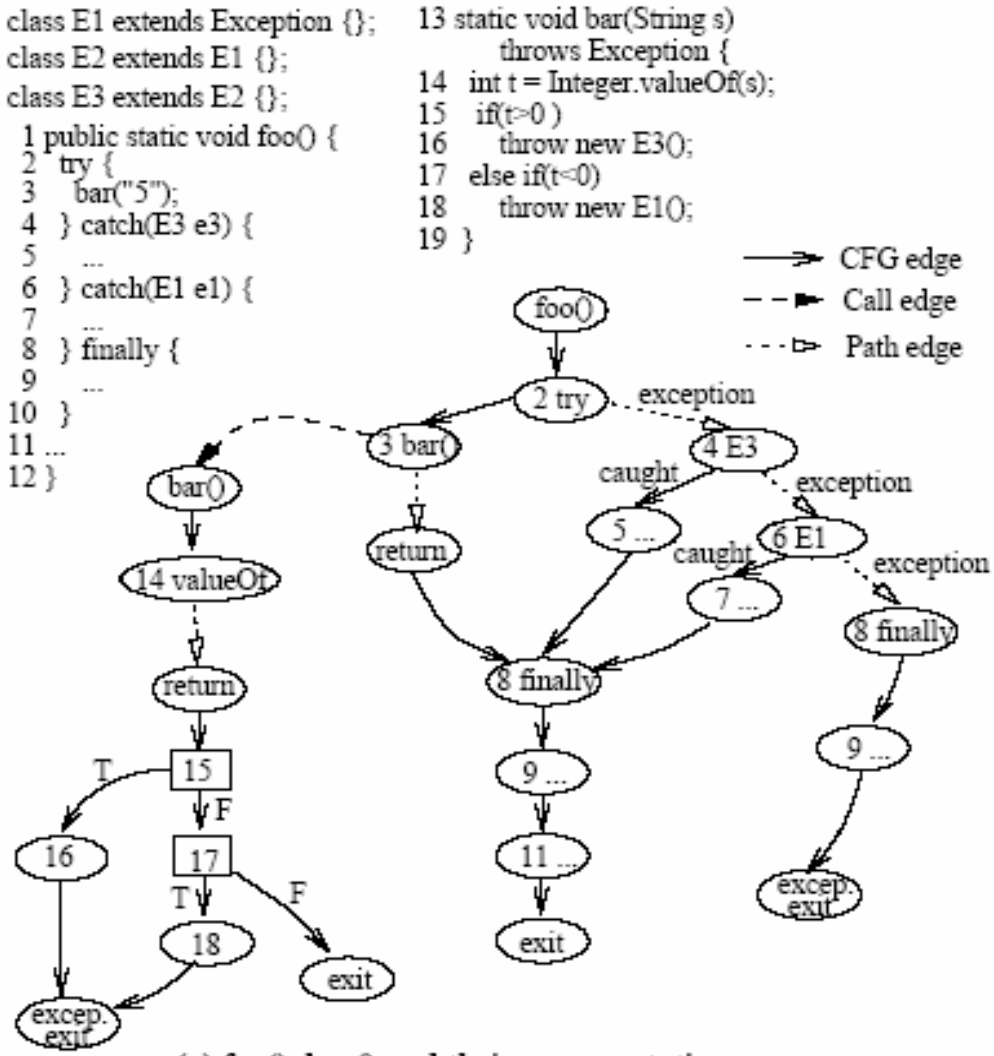
Quelle: Harrold et al, 2001

2.3. „RETEST“ (7)

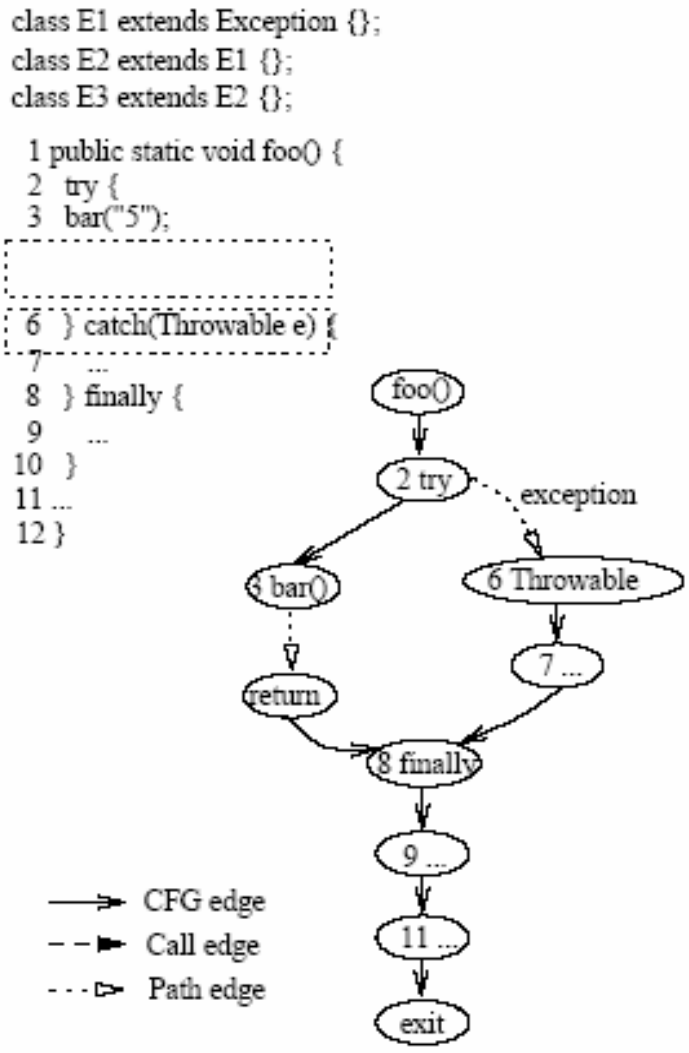
Exceptions

- Der `try`-Block, die `catch`-Blöcke und der eventuelle vorhandene `finally`-Block werden im JIG durch einen eigenen Knoten dargestellt.
- Eine Pfadkante vom `try`-Knoten zum ersten `exception`-Knoten, die über alle `exception`-Knoten bis hin zum finally-Knoten führt, stellt die Weiterreichung der Exception dar.
- Der Sprung bei einer Exception wird nicht expliziert durch eine Kante dargestellt.
- Der `exception-exit`-Knoten stellt den Fall dar, dass die Exception nicht behandelt wurde und somit die Methode umgehend verlassen wird.

2.3. „RETEST“ (8)



(a) foo(), bar() and their representations



(b) a modified version of foo() and its representation

Quelle: Harrold et al, 2001

2.3. „RETEST“ (9)

Annahmen (1)

- **keine Verwendung von Reflection:**
  - Durch Reflection erhält man Zugriff auf Felder, Methoden und Konstruktoren geladener Klassen und kann diese manipulieren.
  - ➢ Solche Änderungen überall zu finden bedarf eines riesigen Aufwands und würde den Algorithmus ineffizient machen.

- **unabhängige externe Klassen:**
  - externe Klassen können ohne interne Klassen kompiliert werden
  - externe Klassen können keine internen Klassen explizit durch einen Klassenlader laden
  - ➢ Es wird sichergestellt, dass externe Klassen nur über definierte virtuelle Methoden mit internen Klassen interagieren.
  - ➢ Durch wird die Anzahl der Interaktionsmöglichkeiten beschränkt, die der Algorithmus analysieren muss.

2.3. „RETEST“ (10)

Annahmen (2)

- **deterministische Testläufe:**
  - Jeder unveränderte Testlauf muss deterministisch sein, d.h. das unter gleichen Vorbedingungen die gleichen Ergebnisse erzielt werden müssen.
  - ➢ Es wird sichergestellt, dass das Ergebnis bei der Ausführung eines Testfalls, der modifizierte Stellen im Code nicht überdeckt, beim Original und der veränderten Version gleich sind.

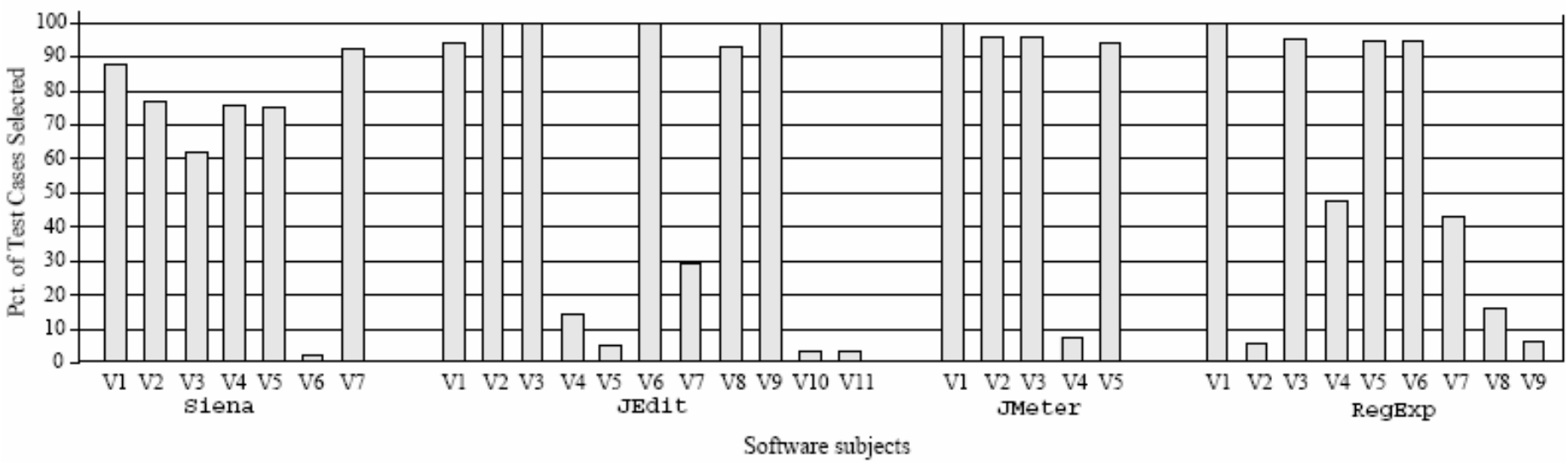
- **Bewertung:**
  - **„Inclusiveness“:** 100%, d.h., Verfahren ist sicher
  - **„Precision“:** gut
  - **„Efficiency“:** gut (bei o.g. Annahmen), da externe Klassen (z.B. Bibliotheken) nicht analysiert werden müssen
  - **„Generality“:** gut, das Verfahren unterstützt alle Konstrukte von objektorientierten Programmiersprachen (Exceptions, Polymorphie, usw.) und ist nicht nur für Java anwendbar

3. Fazit (1)

- Eine empirische Studie von RETEST zeigt, dass eine Reduzierung der Testfälle von ca. 40% möglich ist.
- In Umgebungen, in denen die Kosten für Testfälle groß sind, z.B. das Testen von Software in der Luftfahrt, kann das Weglassen eines einzigen Testfalls bereits Tausende von Euro sparen.
- Solche empirischen Studien lassen sich aber nur eingeschränkt auf reale Programme bzw. Programmänderungen übertragen.
- Die Effizienz hängt überwiegend vom Ort der Änderung ab!

3. Fazit (2)

Pro-gramm	#Methoden	#Versionen	#Test-fälle	Methoden-überdeckung	∅Einsparung
Siena	185	7	138	70%	33,0%
JEdit	3495	11	189	75%	41,6%
JMeter	109	5	50	67%	21,8%
RegExp	168	9	66	46%	43,9%



Quelle: Harrold et al, 2001

4. Werkzeuge (1)

MERCURY WINRUNNER

- Mercury WinRunner
  - Bietet automatische funktionale Tests und Regressionstests
  - Kann Aktionen aufzeichnen und abspielen
  - Unterstützt die Sprachen Visual Basic, Java, Delphi
  - Preis nicht ermittelbar
  - WinRunner Quickstart Standard Seminar: 9900€ für 5 Tage

4. Werkzeuge (2)

TestComplete™ automate your tests

- AutomatedQA TestComplete
  - Bietet automatisierte funktionale Tests (UI), Modul-, Regressions-, verteilte Tests und HTTP-Last- und Stresstests, Datenbanktest und „Data-Driven“ Test in einem Werkzeug
  - Unterstützt die Sprachen .NET, Java, Visual Basic, C++, Delphi und Web
  - Skriptsprache zur Definition von Testfällen
  - Für die gebotene Funktionalität recht günstig: ab 350$
  - Die Firma AutomatedQA hat für ihre Tools viele Auszeichnungen bekommen

4. Werkzeuge (3)

aRTS – a Regression Test Selection Tool

- Wurde im Rahmen einer Studienarbeit am Lehrstuhl entwickelt
- Implementiert vollständig das Selektionsverfahren RETEST von Harrold et al für Java
- Reines Selektionstool
- Arbeitet auf der Basis von Java-Bytecode

Fragen?