Esra Ünal
Seminar: Beiträge zum Software Engineering, Institut für Informatik

Freie Universität Berlin

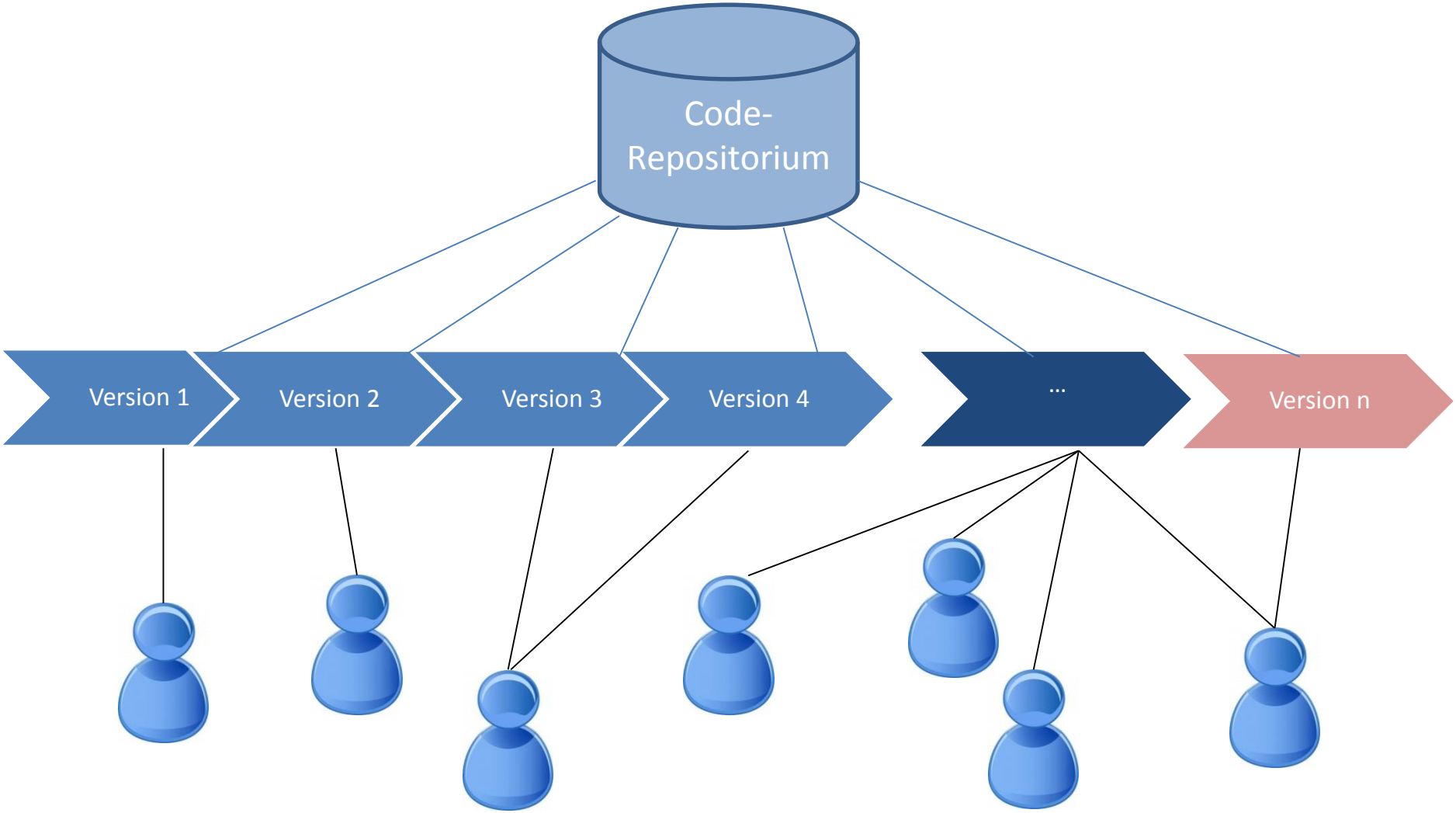
Software Change Classification

Automatisierte Klassifikation von Software-Änderungen als Defektfrei oder als Fehlerbehaftet

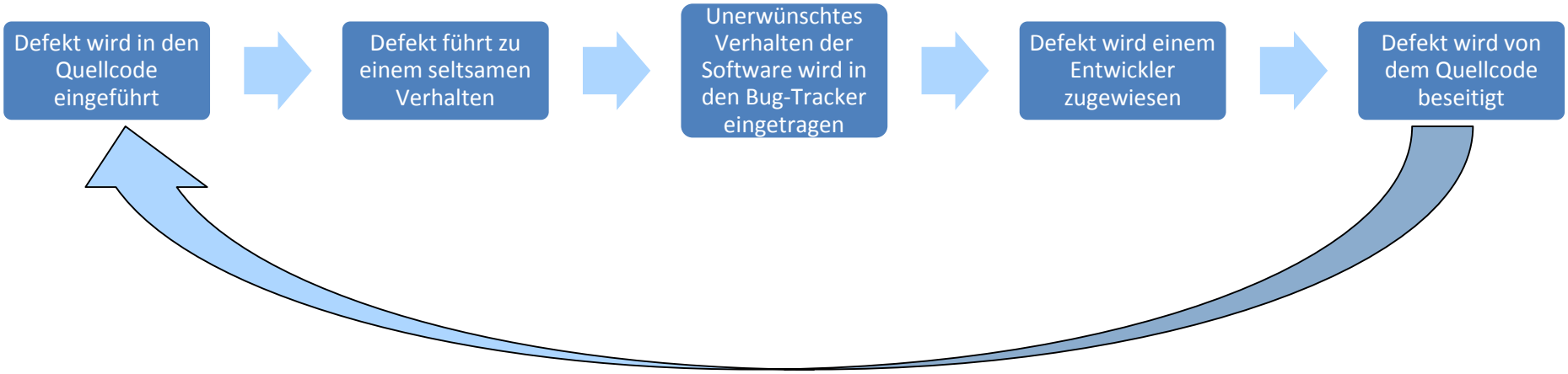
Gliederung

1. Motivation
2. Aufgabe
3. Lösungen
    1. Klassifizierung von Software-Änderungen mithilfe von Änderungslogs (1)
    2. Klassifizierung von Software-Änderungen mithilfe von Änderungslogs (2)
    3. Klassifizierung von Software-Änderungen mithilfe von Testfällen
4. Fazit
5. Quellen

1. Motivation

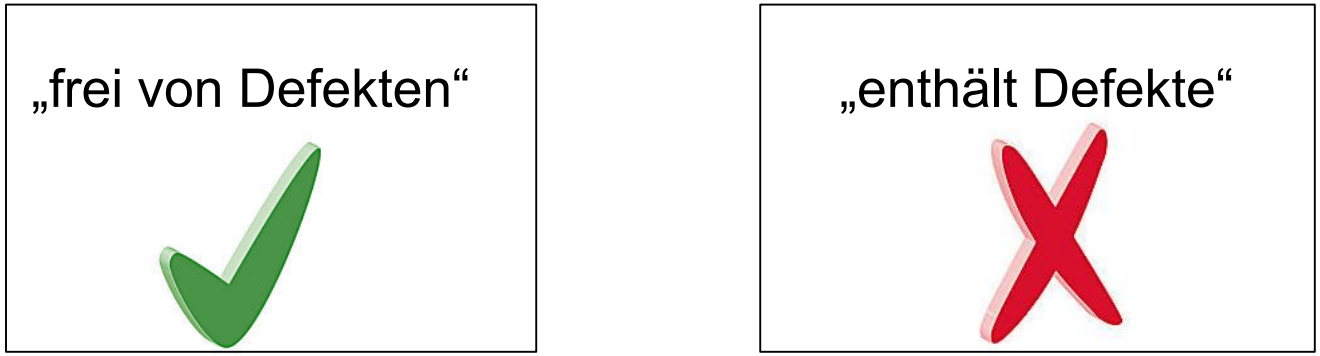


1. Motivation



2. Aufgabe

1. Software-Änderungen in eines der beiden Kategorien einordnen:



2. Identifikation von für den Defekt verantwortlichen Codezeilen

3. Lösungen

1. Klassifizierung von Software-Änderungen mithilfe von Änderungslogs (1)
   - Classifiying Software Changes: Clean or Buggy?
   - Verfahren basiert auf Maschinelles Lernen

2. Klassifizierung von Software-Änderungen mithilfe von Änderungslogs (2)
   - Software Change Classification using Hunk Metrics
   - Erweiterung von dem ersten Verfahren

3. Klassifizierung von Software-Änderungen mithilfe von Testfällen
   - Finding Failure-Inducing Changes in Java Programs using Change Classification
   - Testfälle dienen als Grundlage für die Klassifikation

3.1 Klassifizierung von Software-Änderungen mithilfe von Änderungslogs (1) - Allgemein

- Classifiying Software Changes: Clean or Buggy?

- Sunghun Kim
- E. James Whitehead
- Yi Zhang

- University of California
- IEEE Transaction on Software Engineering April 2008

3.1 Klassifizierung von Software-Änderungen mithilfe von Änderungslogs (1) - Ansatz

- Verfahren ist vierstufig aufgebaut:
  1. Bugfixes aus den Änderungslogs filtern
  2. Defekteinführende Commits identifizieren
  3. Merkmale extrahieren
  4. Klassifikator mit gefundenen Merkmalen trainieren
- Schritt 1 und 2 worden aus den Ergebnissen der Studie von Zeller, Zimmermann und Silwerski aus „When Do Changes Induce Fixes?“ entnommen

3.1 Klassifizierung von Software-Änderungen mithilfe von Änderungslogs (1) – Bugfixes identifizieren

- Änderungslogs zeichnen auf:
  - Entwickler
  - Datum
  - Uhrzeit
  - kurze Beschreibung von den Änderungen

- Bugreports enthalten:
  - kurze Beschreibung
  - Kommentare
  - zugewiesene Entwickler
  - aktueller Bearbeitungszustand

3.1 Klassifizierung von Software-Änderungen mithilfe von Änderungslogs (1) - Bugfixes identifizieren

Änderungslog:

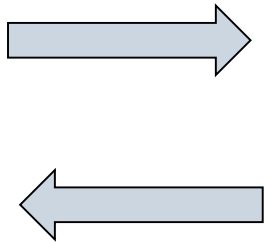
Author: Bob

Datum: 12.01.13

Uhrzeit: 16:44

Version: 1.3

Beschreibung: fixed Bug #110759



Bugreport:

ID: 110759

zugwiesen an: Bob

Datum: 26.01.13 14:32

Status: zugewiesen

Beschreibung: NullpointerException in method foo

3.1 Klassifizierung von Software-Änderungen mithilfe von Änderungslogs (1) - defekteinführende Commits identifizieren

- Codezeilen aus den Bugfixes filtern
- die Version vor dem Bugfix wählen
- in dieser Version die letzte Änderung an den Codezeilen aus dem Bugfix ermitteln
- die resultierende Version ist für den Defekt verantwortlich

- Ausnahme:
  - die Versionen, die erst nach dem Bugreport entstanden sind, werden aus der Liste gestrichen

3.1 Klassifizierung von Software-Änderungen mithilfe von Änderungslogs (1) - Merkmale extrahieren

Version 1:

```
1 Bob  1: public void bar() {
1 Bob  2:     //print report
1 Bob  3:     if (report == null){
1 Bob  4:         println(report);
1 Bob  5:     }
1 Bob  6:}
```

Version 2:

```
2 Carl 1: public void foo() {
1 Bob  2:     //print report
1 Bob  3:     if (report == null){
2 Carl 4:         println(report.message);
1 Bob  5:     }
1 Bob  6:}
```

Version 3:

```
2 Carl   1: public void foo() {
1 Bob    2:     //print report
3 Alice  3:     if (report != null){
2 Carl   4:         println(report.message);
1 Bob    5:     }
1 Bob    6:}
```

3.1 Klassifizierung von Software-Änderungen mithilfe von Änderungslogs (1) - Merkmale extrahieren

- untersuchete Metriken:
  - Anzahl an modifizierten Datein pro Commit
  - Anzahl an Enwticklern, die bisher an der jeweiligen Datei gearbeitet haben
  - hinzugefügte Zeilen
  - gelöschte Zeilen
  - Beschreibungstext im Änderungslog
  - Metadaten (Author, Datum, Uhrzeit, Anzahl bisheriger Änderungen, Anzahl bisher entdeckter Bugs)
  - Komplexitätsmetriken wie (LOC, maximale Verschachtelungslänge, Anzahl an Kommentarzeilen, ...)
  - Dateinamen und Dateipfade
  - alle Wörter, Zahlen, Operatoren in Quelltext Anzahl bisheriger Änderungen

3.1 Klassifizierung von Software-Änderungen mithilfe von Änderungslogs (1) - Klassifikator mit gefundenen Merkmalen trainieren

- mit den gesammelten Daten (250 – 500 Commits) den Klassifikator trainieren
  - welche Merkmale sind besonders ausgeprägt bei fehlerhaftem Quelltext?
  - welche Merkmale sind besonders ausgeprägt bei korrekt vermutetem Quelltext?

- nach dem Training soll der Klassifikator automatisiert zwischen defektfreien und fehelerbehaftetetn Commits unterscheiden können

3.1 Klassifizierung von Software-Änderungen mithilfe von Änderungslogs (1) - Evaluation

- Accuracy:

$$\text{Accuracy} = \frac{n_{b\to b} + n_{c\to c}}{n_{b\to b} + n_{b\to c} + n_{c\to c} + n_{c\to b}}$$

- Defekt-Precision:

$$P(b) = \frac{n_{b\to b}}{n_{b\to b} + n_{c\to b}}$$

- Defekt-Recall:

$$R(b) = \frac{n_{b\to b}}{n_{b\to b} + n_{b\to c}}$$

3.1 Klassifizierung von Software-Änderungen mithilfe von Änderungslogs (1)– Evalutation

- Defektfrei-Precision:

$$P(c) = \frac{n_{c \to c}}{n_{c \to c} + n_{b \to c}}$$

- Defektfrei-Recall:

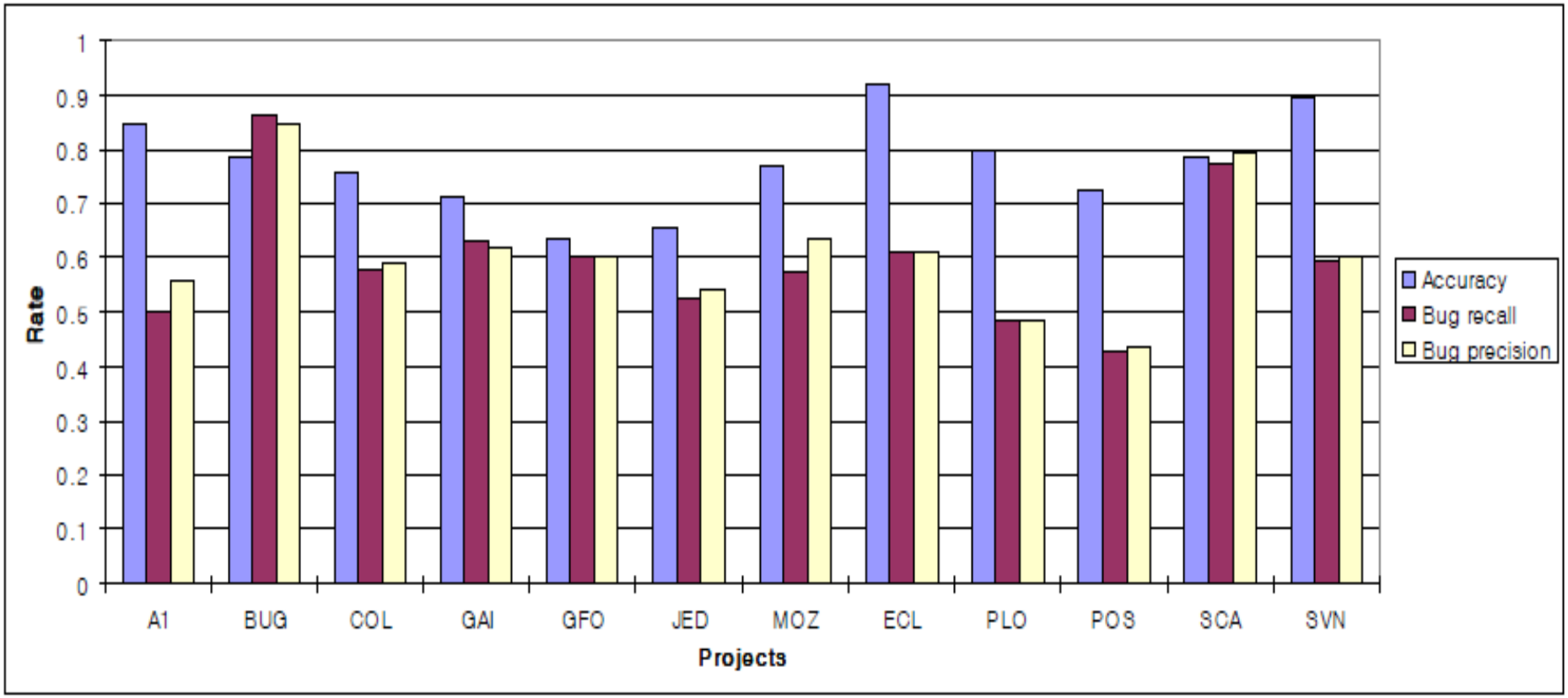
$$R(c) = \frac{n_{c \to c}}{n_{c \to c} + n_{c \to b}}$$

3.1 Klassifizierung von Software-Änderungen mithilfe von Änderungslogs (1) – Evalutation

- Ansatz wurde an 12 Open Source Projekten getestet:

Project	Revisions.	# of clean changes	# of buggy changes	% of buggy changes
Apache HTTP 1.3 (A1)	500-1000	579	121	17.3
Bugzilla (BUG)	500-1000	149	417	73.7
Columba (COL)	500-1000	1,270	530	29.4
Gaim (GAI)	500-1000	742	451	37.8
GForge(GFO)	500-1000	339	334	49.6
Jedit (JED)	500-750	626	377	37.5
Mozilla (MOZ)	500-1000	395	169	29.9
Eclipse(ECL)	500-750	592	67	10.1
Plone(PLO)	500-1000	457	112	19.6
PostgreSQL (POS)	500-1000	853	273	24.2
Scarab (SCA)	500-1000	358	366	50.5
Subversion (SVN)	500-1000	1,925	288	13.0
Total	N/A	8,285	3,505	29.7

3.1 Klassifizierung von Software-Änderungen mithilfe von Änderungslogs (1) – Evaluation



3.1 Klassifizierung von Software-Änderungen mithilfe von Änderungslogs (1) – Evaluation

Project	Accuracy	Buggy change recall	Buggy change precision	Clean change recall	Clean change precision
A1	0.85	0.5	0.56	0.92	0.9
BUG	0.78	0.86	0.85	0.56	0.6
COL	0.76	0.58	0.59	0.83	0.83
GAI	0.71	0.63	0.62	0.76	0.77
GFO	0.64	0.6	0.60	0.67	0.67
JED	0.65	0.53	0.54	0.73	0.72
MOZ	0.77	0.57	0.63	0.86	0.83
ECL	0.92	0.61	0.61	0.96	0.96
PLO	0.8	0.48	0.49	0.89	0.87
POS	0.73	0.43	0.44	0.82	0.82
SCA	0.79	0.78	0.8	0.8	0.78
SVN	0.9	0.59	0.6	0.94	0.94

3.1 Klassifizierung von Software-Änderungen mithilfe von Änderungslogs (1) – Evaluation

- Verfahren sehr langsam
  - jedes Wort, Zahl und Operator im Quelltext wird auf Häufigkeit geprüft
  - Vorgehensweise sichert zwar die Sprachunabhängigkeit, aber sehr aufwendig
  - Kommentare werden nicht ausgeschlossen
  - fehelende Vorbereitung der Datenmenge
    - entfernen von Stopwörtern fehlt
    - Reduzierung auf den Wortstamm fehlt
    - alle Wörter werden in den Index aufgenommen
      - „add“ und „added“ werden als zwei verschiedene Wörter in den Index erfasst

3.2 Klassifizierung von Software-Änderungen mithilfe von Änderungslogs (2) – Ansatz

- Software Change Classification using Hunk Metrics

- Sunghun Kim
- E. James Whitehead
- Kai Pan

- Graz University of Technology
- ICSM 2009

3.2 Klassifizierung von Software-Änderungen mithilfe von Änderungslogs (2) – Verfahren

- Verfahren identisch mit dem ersten bis auf Schritt 3:
  - 27 Metriken zur Identifizierung von charakterisitschen Merkmalen besteht aus:
    - Anzahl von bedingten Anweisungen (if, else)
    - Anzahl von Schleifen (while, for)
    - Anzahl von Funktionsaufrufen
    - Anzahl von Funktionsdeklarationen
    - Anzahl von Variablendeklarationen
    - Anzahl von Zuweisungen
    - Anzahl von Arrays
    - Anzahlvon Return-Anwesiungen
    - Anzahl von logischen Operatoren
    - Anzahl von Klassen
    - Anzahl von bisher identifizierten Defekten
    - …

3.2 Klassifizierung von Software-Änderungen mithilfe von Änderungslogs (2) – Verfahren

- an 7 Open Source Software-Projekten getestet
- 4 der Projekte (Eclipse, Apache HTTP, Mozilla und PostgreSQL) wurden auch mit dem ersten Verfahren evaluiert
- die Beschränkung auf eine feste Metrikliste bringt bessere Ergbnisse hervor

3.2 Klassifizierung von Software-Änderungen mithilfe von Änderungslogs (2) – Evaluation

Project	A	Buggy Hunk		Bug-free Hunk	
		P	R	P	R
Apache	0.76	0.75	0.65	0.76	0.84
Eclipse	0.87	0.84	0.78	0.89	0.92
Epiphany	0.74	0.66	0.51	0.77	0.86
Evolution	0.75	0.70	0.53	0.77	0.85
Mozilla-C	0.86	0.81	0.62	0.88	0.95
Mozilla-J	0.84	0.83	0.76	0.85	0.90
Nautilus	0.77	0.79	0.78	0.75	0.76
PostgreSQL	0.83	0.81	0.72	0.84	0.89

3.2 Klassifizierung von Software-Änderungen mithilfe von Änderungslogs (2) - Probleme

- Problematisch:
  - was passiert mit den Defekte, die verteilt über mehrer Versionen beseitigt wurden?
  - was passiert mit den Codzeilen, die für Defekte keine Bedeutung tragen (Kommentare, leere Zeilen, Klammern)?
  - Was passiert mit Namensänderungen?

3.2 Klassifizierung von Software-Änderungen mithilfe von Änderungslogs (2) - Probleme

Version 1:

```
1: public void bar() {
2:     if (report == null){
3:         println(report);
4:     }
5:}
```

Version 2:

```
1: public void bar(){
2:     if (report==null) {
3:         println(report);
4:     }
5:}
```

Version 3:

```
1: public void foo() {
2:     if (report!=null){
3:       println(report);
4:     }
5:}
```

Änderung hat keine semantische Bedeutung

- nur Leerzeichen entfernt

3.2 Klassifizierung von Software-Änderungen mithilfe von Änderungslogs (2) - Probleme

Version 1:

```
1: private void bar() {
2:     if (report == null){
3:         println(report);
4:     }
5:}
```

Version 2:

```
1: private void foo() {
2:     if (report == null)
3:     {
4:         println(report);
5:     }
6:}
```

Version 3:

```
1: public void foo() {
2:     if (report != null)
3:     {
4:       println(report);
5:     }
6:}
```

Änderung hat keine semantische Bedeutung

- nur Namensänderung
- Klammerposition verschoben

3.3 Klassifizierung von Software-Änderungen mithilfe von Testfällen - Allgemein

- Finding failure-inducing changes in java programs using change classification

- Maximilian Stoerzer
- Barbara G. Ryder
- Xiaoxia Ren
- Frank Tip

- Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering 2006

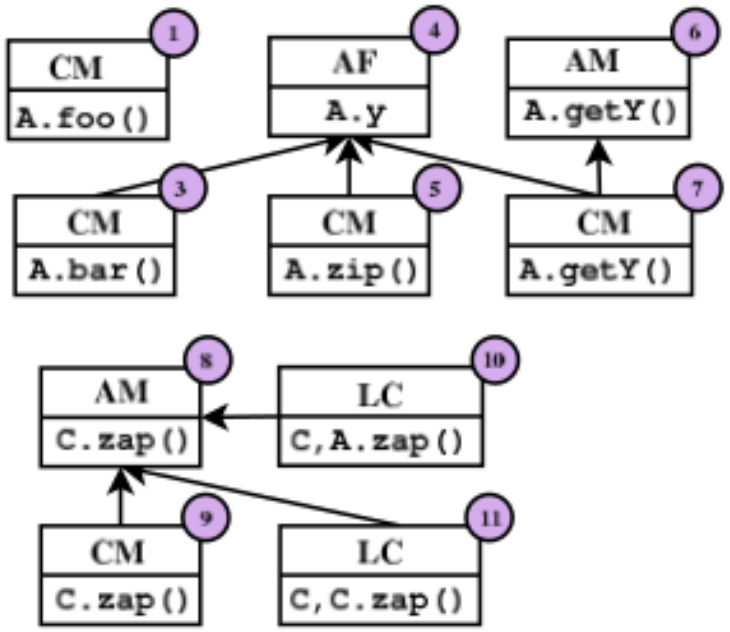
3.3 Klassifizierung von Software-Änderungen mithilfe von Testfällen – Verfahren

- Verfahren:
  1. Änderungen an dem Code in atomare Anwesiungen zerlegen
  2. atomare Anweisungen klassifizieren
     - AC (added classes)
     - DC (deleted classes)
     - AM (added methods)
     - DM (deleted methods)
     - CM (changed methods)
     - AF (added fields)
     - DF (deleted fields)
     - LC (lookup changes)
  3. betroffene Tests identifizieren
  4. verantwortliche atomare Anwesiungen identifizieren
  5. alle atomaren Anweisungen klassfizieren

3.3 Klassifizierung von Software-Änderungen mithilfe von Testfällen – Verfahren

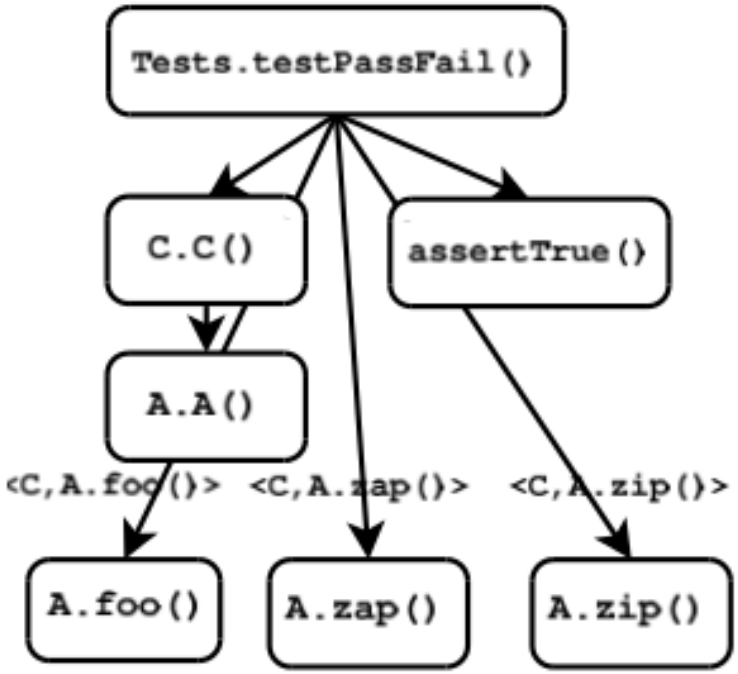
```
public class A {
    public A(int i) { x = i;}
    public void foo() { x = x + 0; }
    public void bar() { y = x; }
    public void zap() { }
    public void zip() { y = x; }
    public int x;
    public static int y;
    public static int getY() {return y;}
}
public class C extends A{
    public C(int k) {super(k);}
    public void zap(){x = 5;}
}
```

```
public class testPassFail(){
    A a = new C(7);
    a.foo(); a.zap(); a.zip();
    Assert.assertTrue(a.x == 7);
}
```

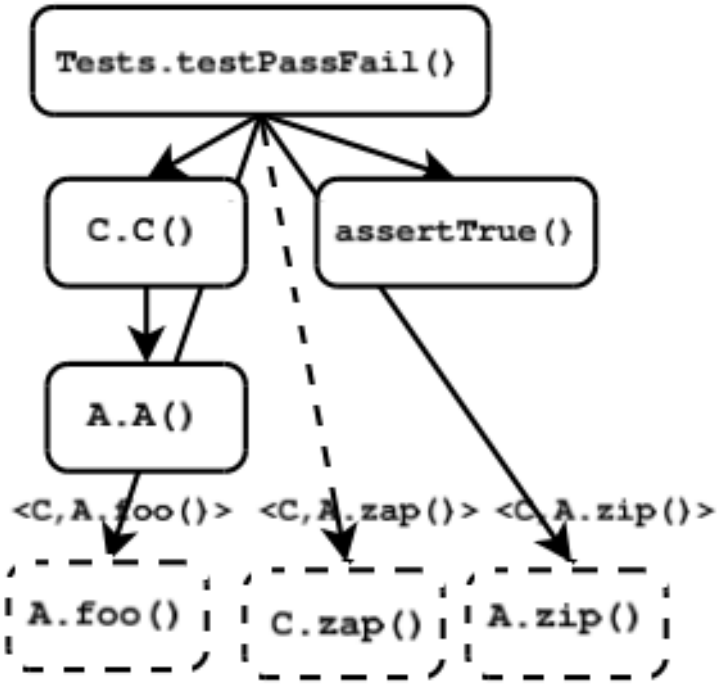


3.3 Klassifizierung von Software-Änderungen mithilfe von Testfällen – Verfahren

Aufrufgraph von der älteren Version



Aufrufgraph von der neuen Version



3.3 Klassifizierung von Software-Änderungen mithilfe von Testfällen – Verfahren

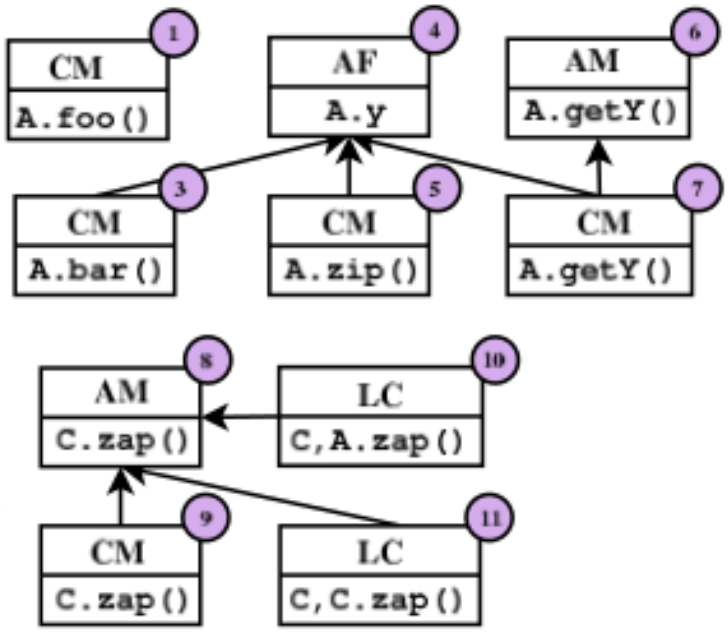
Klassifikationsmethode (idealistisch):

- Grün: atomare Anweisung hat keinen oder einen positiven Einfluss auf den Test
- Rot: atomare Anweisung führt zum Fehlschalgen des Testes
- Gelb: atomare Anweisung hat einen positiven als auch negativen Einfluss auf den Test
- Grau: es können keine Aussagen gemacht werden

3.3 Klassifizierung von Software-Änderungen mithilfe von Testfällen – Verfahren

```
public class A {
    public A(int i) { x = i;}
    public void foo() { x = x + 0; }
    public void bar() { y = x; }
    public void zap() { }
    public void zip() { y = x; }
    public int x;
    public static int y;
    public static int getY() {return y;}
}
public class C extends A{
    public C(int k) {super(k);}
    public void zap(){x = 5;}
}
```

```
public class testPassFail(){
    A a = new C(7);
    a.foo(); a.zap(); a.zip();
    Assert.assertTrue(a.x == 7);
}
```



3.3 Klassifizierung von Software-Änderungen mithilfe von Testfällen – Klassen

Farbklassen	nicht-strikt	strikt	simpel
Rot	$T_B \rightarrow T_F$ (obligatorisch) $T_F \rightarrow T_F$ (optional) $T_B \rightarrow T_B$ (optional)	$T_B \rightarrow T_F$ (obligatorisch) $T_F \rightarrow T_F$ (optional)	$T_{B/F} \rightarrow T_F$
Grün	$T_B \rightarrow T_B$ (alle) ODER $T_F \rightarrow T_B$ (obligatorisch) $T_F \rightarrow T_F$ (optional) $T_B \rightarrow T_B$ (optional)	$T_B \rightarrow T_B$ (alle) ODER $T_F \rightarrow T_B$ (obligatorisch) $T_B \rightarrow T_B$ (optional)	$T_{B/F} \rightarrow T_B$
Gelb	weder in der roten noch in der grünen Klasse		

B = Bestanden

F = Fehlgeschlagen

$T_{\text{Testausgang vor Änderung}} \rightarrow T_{\text{Testausgang nach Änderung}}$

3.3 Klassifizierung von Software-Änderungen mithilfe von Testfällen – Verfahren

- entsprechend der Aufteilung der Klassen wurden für 5 Klassifizierer definiert:

  1. $R_{n\text{-}strikt}/G_{n\text{-}strikt}$
  2. $R_{strikt}/G_{n\text{-}strikt}$
  3. R_{strikt}/G_{strikt}
  4. $R_{n\text{-}strikt}/G_{strikt}$
  5. Simpel

3.3 Klassifizierung von Software-Änderungen mithilfe von Testfällen– Evaluation

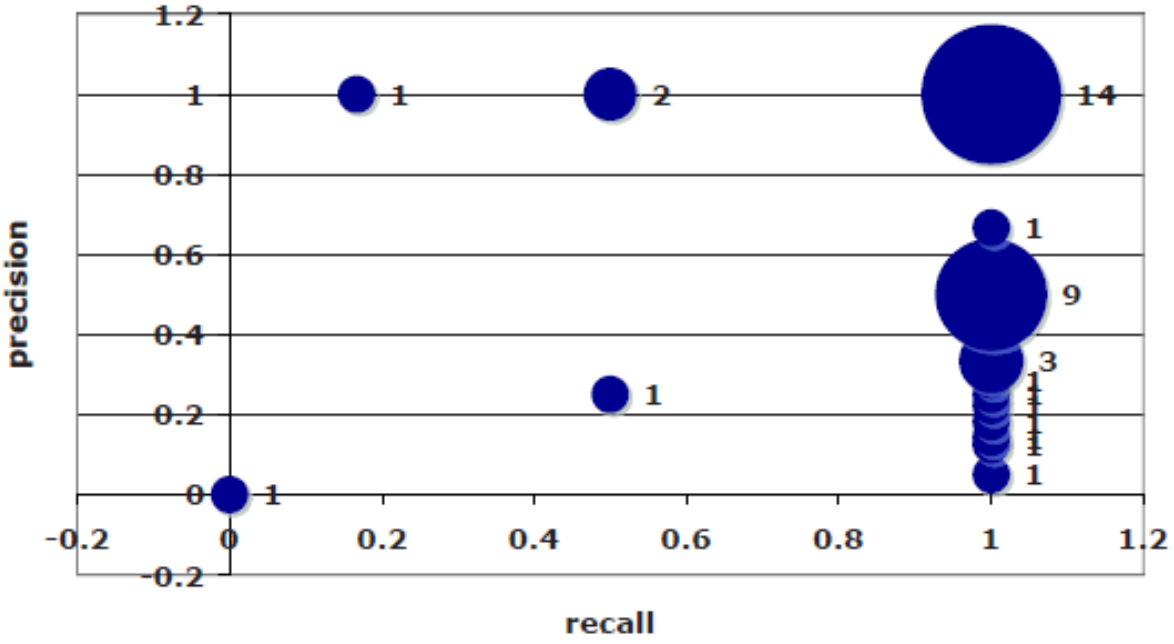
Fallstudie 1: Studentisches Projekt:

- 40 Bachelor-Studenten beteiligt
- Dinics Algorithmus zur Berechnung des maximalen Flusses in einem Netzwerk
- Vorgaben: Black-Box-Tests und Schnittstellen
- für das Bestehen des Kurses dürfen die Tests nicht fehlschlagen
- Studenten hatten keinen Kenntnis darüber für welchen Zweck die Daten erhoben worden
- Programme umfassten durchschnittlich 950 Zeilen Quelltext (mit Kommentaren)

- Erhobenen Daten wurden nachbereitet:
  - Von den 1175 Versionspaaren wurden 61 herangezogen
  - Bei der Erkennung von Defekten hat der $R_{n\text{-}strikt}$–Klassifizierer am erfolgreichsten

3.3 Klassifizierung von Software-Änderungen mithilfe von Testfällen– Evaluation

Ergebnisse des $R_{n\text{-}strikt}$–Klassifizierers:



3.3 Klassifizierung von Software-Änderungen mithilfe von Testfällen– Evaluation

Fallstudie 2: Daikon

- durch dynamische Analyse werden in Softwaresystemen Invarianten ermittelt

- Repositroy enthält zu keinem Zeitpunkt Tests, die bei einer Ausführung fehlschlagen

- für die Evalutation werden die Versionen Daikon/2002-11-11 und Daikon/2002-11-19 herangezogen
  - Testfälle wurde in der aktuelleren Version erneuert
  - für die Evalutation wurden die älteren Tests genommen

3.3 Klassifizierung von Software-Änderungen mithilfe von Testfällen– Evaluation

Fallstudie 2: Daikon

- 61 Tests
- 40 Tests waren von den Änderungen zwischen den beiden Versionen betroffen
- 6093 atomare Anweisungen identifiziert
  - „Grau“: 5715
  - „Grün“: 338
  - „Gelb“ : 33
  - „Rot“: 7

3.3 Klassifizierung von Software-Änderungen mithilfe von Testfällen– Evaluation

Fallstudie 2: Daikon

- Beispiele:
  - Test *testXor* war von 35 atomaren Anweisungen betroffen
    - nur zwei davon waren für das Fehlschalgen verantwortlich
    - Klassifiaktionsverfahren hat 4 als „Rot“ gemeldet
  - Test *testMinus* war von 34 atomaren Anweisungen betroffen
    - nur einer davon führte zum Fehlschalgen des Tests
    - Klassifiaktionsverfahren hat 3 als „Rot“ gemeldet

3.3 Klassifizierung von Software-Änderungen mithilfe von Testfällen– Evaluation

- Aussreichend viele Testsfälle müssen vorhanden sein
- Verfahren verlangsamt die Ausführung der Tests, um einen Faktor von 10
- keine Angaben über die richtige Zurodnung von „grünen“ atomaren Zuweisungen
- verschiedene Klassfikationsmethoden möglich
  - für die Auswahl einer effizienten Methode müssen die Ergbnisse aller Methoden manuell geprüft werden

4. Fazit

- Idealvorstellung von einem automatisierten Kassifizierungswerkzeug zur Defektlokalisierung sehr nützlich

- Erfolg der verschiedenen Klassifizierungsverfahren ist für jedes Projekt unterschiedlich
  - Werte variieren stark (40% - 90%)

- alle Lösungsansätze benötigen weitere manuelle Eingriffe
  - Ergebnisse müssen manuell geprüft werden, um geeigneten bzw. effizienten Klassifikator zu finden

- alle Verfahren sind zeitintensiv

5. Quellen (1)

[1] Kim, Sunghun; Whitehead, Jim James, Jr.; Zhang, Yi: *Classifying Software Changes: Clean or Buggy?* IEEE Transactions on Software Engineering. March-April 2008. Pages: 181 - 196

[2] Śliwerski, Jacek; Zimmermann, Thomas; Zeller, Andreas: *When Do Changes Induces Fixes?* Proceedings of the 2005 international workshop on Mining software repositories. Volume 30 Issue 4. July 2005. Pages 1 - 5

[3] Ferzund, Javed; Ahsan, Syed Nadeem; Wotawa, Franz: *Software Change Classification using Hunk Metrics*, IEEE International Conference on Software Maintenance, ICSM 2009. 20-26 Sept. 2009. Pages 471 - 474

[4] Kim, Sunghun; Whitehead, Jim James, Jr.; Zimmermann, Thomas; Pan, Kai. *Automatic Identification of Bug-Introducing Changes*. 06 Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering. 2006. Pages 81-90

5. Quellen (2)

[5] Stoerzer, Maximilian; Ryder, Barbara; Ren, Xiaoxia; Tip, Frank. *Finding Failure-Inducing Changes in Java Programs using Change Classification*. 06 Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering. 2006. Pages 57 - 68

Diskussionsfrage:

Können automatisierte Klassifizierungsverfahren tatsächlich die Entwickler bei der Auffindung von Defekten unterstützen

oder

behindern sie sogar die Entwickler durch falsche Warnmeldungen?

Freie Universität Berlin

Vielen Dank für Ihre Aufmerksamkeit