Softwarereengineering

Prof. Dr. Rainer Koschke

07.12.2006

Software-Redundanz

- Duplikation von Quelltext durch Copy&Paste
  (Code Cloning)
- Nummer 1 auf Beck und Fowlers „Stink Parade of Bad Smells“
- Konsistente Änderungen an Klonen schwierig
  $\Rightarrow$ erhöhter Wartungsaufwand

Gegenmaßnahme

Abstraktion durch Funktionen oder Makros
$\Rightarrow$ Code wird kürzer, klarer und besser wartbar!

Software-Redundanz

foo.c

```
...
for (i = 1; i < MAX; i++) {
        x = func() * y + x;
}

...

for (j = 1; j < MAX; j++) {
        x = func() * y + x;
}

...
```

bar.c

```
...

for (i = 1; i < MAX; i++) {
        x = func() * y + x;
}
...
```

fred.c

```
...
for (i = 1; i < MAX; i++) {
        x1 = func() * y1 + x1;
}

...


for (i = 1; i < MAX; i++) {
        x2 = func() * y2 + x2;
}

...
```

Typisch: 5 – 30 % des Codes sind redundant...

Software-Redundanz

foo.c

```
...

set_coordinate (&x, y);

...

set_coordinate (&x, y);

...
```

bar.c

```
...

set_coordinate (&x, y);

...
```

```
void set_coordinate (
    float *x, float y) {
  int i;
  for (i = 1; i < MAX; i++) {
        *x = func() * y + *x;
  }
}
```

fred.c

```
...

set_coordinate (&x1, y1);

...

set_coordinate (&x2, y2);

...
```

. . . und können abstrahiert werden

Copy&Paste-Programmierung

- Szenario 1:
  - Neue Funktionalität soll hinzugefügt, die ähnlich ist zu einer existierenden.
  - Auswirkungen sind nicht bekannt:
    - Wer benutzt die existierende Funktionalität bereits?
    - Was benutzt die existierende Funktionalität alles?
  - Häufige Folge: Funktion wird kopiert und leicht modifiziert.
- Szenario 2:
  - Wegen unzureichender Versionskontrolle existieren mehrere Versionen derselben Funktion im selben Code oder in verschiedenen Systemen.
- Szenario 3:
  - Programmiersprache unterstützt keine Vererbung und keine generischen Einheiten. Compiler bietet kein Inlining.
  - Folge: Code wird dupliziert

Folgen der Copy&Paste-Programmierung

Die Folgen sind duplizierter Code und damit höherer Aufwand für Wartung, Verstehen und Test:

- Schwierigere Fehlerkorrektur
- Quelle subtiler Fehler
- Höherer Aufwand beim Verstehen
- Mehr Daten zu analysieren und visualisieren
- Änderbarkeit leidet

$\Longrightarrow$ Untersuchungen: 5–30 % Code sind redundant.

Arten von Klonen

Die Kopie eines Programmfragments wird Klon genannt.

Typ 1 Exakte Kopie

- Keinerlei Veränderung an der Kopie (bis auf White Space und Kommentaren).
- Z.B. Inlining von Hand.

Typ 2 Kopie mit Umbenennungen (parametrisierte Übereinstimmung)

- Bezeichner werden in der Kopie umbenannt.
- Z.B. "'Wiederverwendung"' einer Funktion, generische Funktion von Hand

Typ 3 Kopie mit weiteren Modifikationen

- Code der Kopie wird abgeändert, nicht nur Bezeichner.
- Z.B. "'Erweiterung"' einer Funktion.

Typ 4 Semantische Klone

- Verschiedene Implementierungen desselben Konzepts.

Beispiele

`i = length (l); if (i < 3) {...}`	`i = length (l); if (i < 3) {...}`	Typ 1
`i = length (l); if (i < 3) {...}`	`k = length (j); if (k < x) {...}`	Typ 2
`i = length (l); if (i < 3) {...}`	`k = length (j); if (j < 3) {...}`	zwei Typ 2 bzw. ein Typ 2 mit inkonsistenter Umbenennung
`i = length (l); if (i < 3) {...}`	`k = length (j); f(); if (k < x) {...}`	Typ 3
`x = x + 1;`	`x++;`	primitiver Typ 4

Granularität der Klone

Das Ziel der Klonerkennung bestimmt Granularität. Je feiner die Granularität, desto höher der Aufwand.

- Sequenzen von Anweisungen
  - Beseitigung von manuellem Inlining.
  - Kapselung von Operationen in Unterprogrammen.
- Unterprogramme und Datenstrukturen
  - Identifikation von Kandidaten für die Einführung generischer Konzepte.
  - Vereinigung von Varianten.
  - "'Objektifizierung"'
- Pakete
  - Wie oben, jedoch im größeren Stil.
  - Säuberung nach gescheiterter Versionskontrolle.

Ansätze zur Klonerkennung

Die Ansätze unterscheiden sich in

- Granularität des berücksichtigten Wissens
  - Anweisungssequenzen
  - Funktionen
  - Pakete
- und der Abstraktionsebene der Analyse
  - Text
  - Token-Ebene
  - Syntax
  - quantifizierbare Merkmale
  - Programmabhängigkeiten

Existierende Ansätze

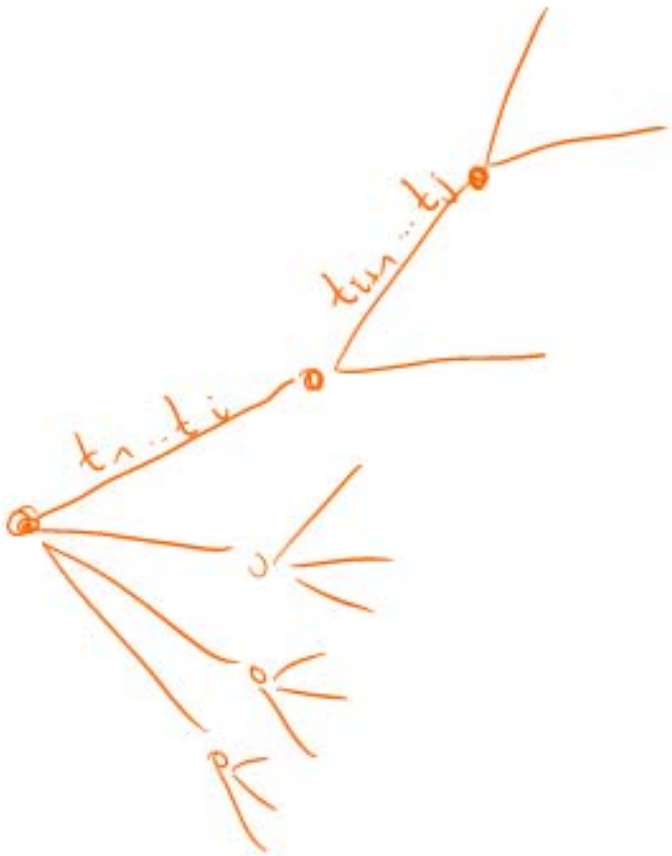
- Anzahl unterschiedlicher Zeilen des Unix-Diff-Programmes
- Pattern Matching auf Ebene der Lexeme (Baker)
- Matching auf dem abstrakten Syntaxbaum (Baxter et al.)
- Vergleich quantifizierter Merkmale (Metriken) des Codes (Mayrand et al.)

Probleme bei der Erkennung

- Jede Zeile/Funktion/Datei muss mit jeder anderen Zeile/Funktion/Datei verglichen werden:
  - Wie kann quadratischer Aufwand vermieden werden?
- Wie kann von Bezeichnern geeignet abstrahiert werden?
  - Soll die Umbenennung konsistent sein?
- Typ-3-Klone: Geklonte Codefragmente von Typ 1 und Typ 2 können zu größeren Klonen zusammengefasst werden.
  - Codefragmente müssen nicht direkt zusammenhängend sein.
  - Code muss nicht identisch, nur ähnlich sein: Ähnlichkeitsmaß?

Verfahren nach Baker (1995)

- Verfahren basiert auf Lexemen.
- Vermeidung des quadratischen Aufwands:
  - quasi-parallele Suche in einer Programmrepräsentation, die jedes mögliche Anfangsstück des Programms enthält.
- Abstraktion von Bezeichnern:
  - Bezeichner werden auf relative Positionen abgebildet.
- Typ-3-Klone:
  - Separater Schritt am Ende


t1..ti
ti+1..tj

Parameter-String

- Für jede Codezeile wird eine Zeichenkette aus Parametersymbolen und Nichtparametersymbolen generiert (so genannter **Parameter-String** oder **P-String**):
  - Struktur der Zeile (genauer: Token-Sequenz) wird auf eindeutiges Nichtparametersymbol abgebildet (Funktor)
  - Bezeichner werden in Argumentliste erfasst
  - Ergebnis ist: (Funktor Argumentliste)
  - Beispiel: x = x + y $\rightarrow$ (P = P + P; x,x,y) $\rightarrow$ α x x y
- Die Konkatenation der P-Strings aller Codezeilen repräsentiert das Programm.

```
j = length(l);
if (j < 3) {x = x + y;}
```

$\rightarrow$ α j length l β j 3 x x y

Prev-String

- Kodierung $prev(s)$ jedes P-Strings s:
  - Erstes Vorkommen eines Bezeichners erhält Nummer 0.
  - Jedes weitere Vorkommen erhält den relativen Abstand zum vorherigen Vorkommen (Funktoren mitgezählt).
  - Beispiel:
    α j length l β j 3 x x y
    $\rightarrow$ α 0 0 0 β 4 0 0 1 0
  - Abstraktion der Bezeichner, jedoch nicht ihrer Reihenfolge.
  - S ist ein **P-Match** von T genau dann, wenn $prev(S) = prev(T)$
  - Beispiel: $x = x + y$ und $a = a + b$ sind ein P-Match wegen $\gamma 010 = \gamma 010$

P-Suffix-Baum

- Sei $S_i = s_i s_{i+1} \ldots s_n\$$ das i'te Suffix von S ($\$$ ist das Endezeichen). Der **P-Suffix-Baum** von S enthält alle $prev(S_i)$ zu den Suffixen des P-Strings S.

  Beispiel: $S = \alpha y \beta y \alpha x \alpha x$

  $prev(S_1) = \alpha 0 \beta 2 \alpha 0 \alpha 2\$$

  $prev(S_2) = 0 \beta 2 \alpha 0 \alpha 2\$$

  $prev(S_3) = \beta 0 \alpha 0 \alpha 2\$$

  $prev(S_4) = 0 \alpha 0 \alpha 2\$$

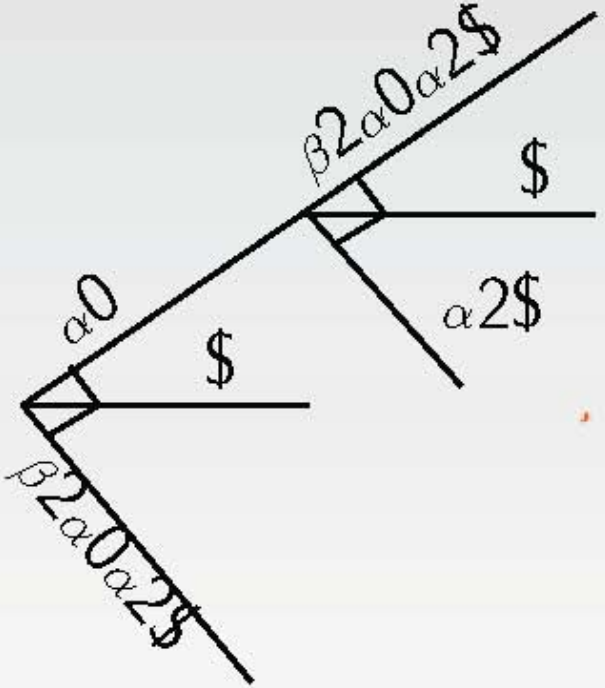
  $prev(S_5) = \alpha 0 \alpha 2\$$

  $prev(S_6) = 0 \alpha 2\$$

  $prev(S_7) = \alpha 0\$$

  $prev(S_8) = 0\$$

  $prev(S_9) = \$$



```
x = x + 1;
a = b + a;
z = z + x;
```

$\alpha x x 1 \alpha a b a \alpha z z x \$$

$S_1 = \alpha 0 1 0 \alpha 0 0 2 \alpha 0 1 9 \$$

$S_2 = 0 1 0 \alpha 0 0 2 \alpha 0 1 9 \$$

$S_3 = 0 0 \alpha 0 0 2 \alpha 0 1 9 \$$

$S_4 = 0 \alpha 0 0 2 \alpha 0 1 0 \$$

$S_5 = \alpha 0 0 2 \alpha 0 1 0 \$$

$S_6 = 0 0 2 \alpha 0 1 0 \$$

$S_7 = 0 0 \alpha 0 1 0 \$$

$S_8 = 0 \alpha 0 1 0 \$$

$S_9 = \alpha 0 1 0 \$$

$S_{10} = 0 1 0 \$$

$S_{11} = 0 0 \$$

$S_{12} = 0 \$$

$S_{13} = \$$