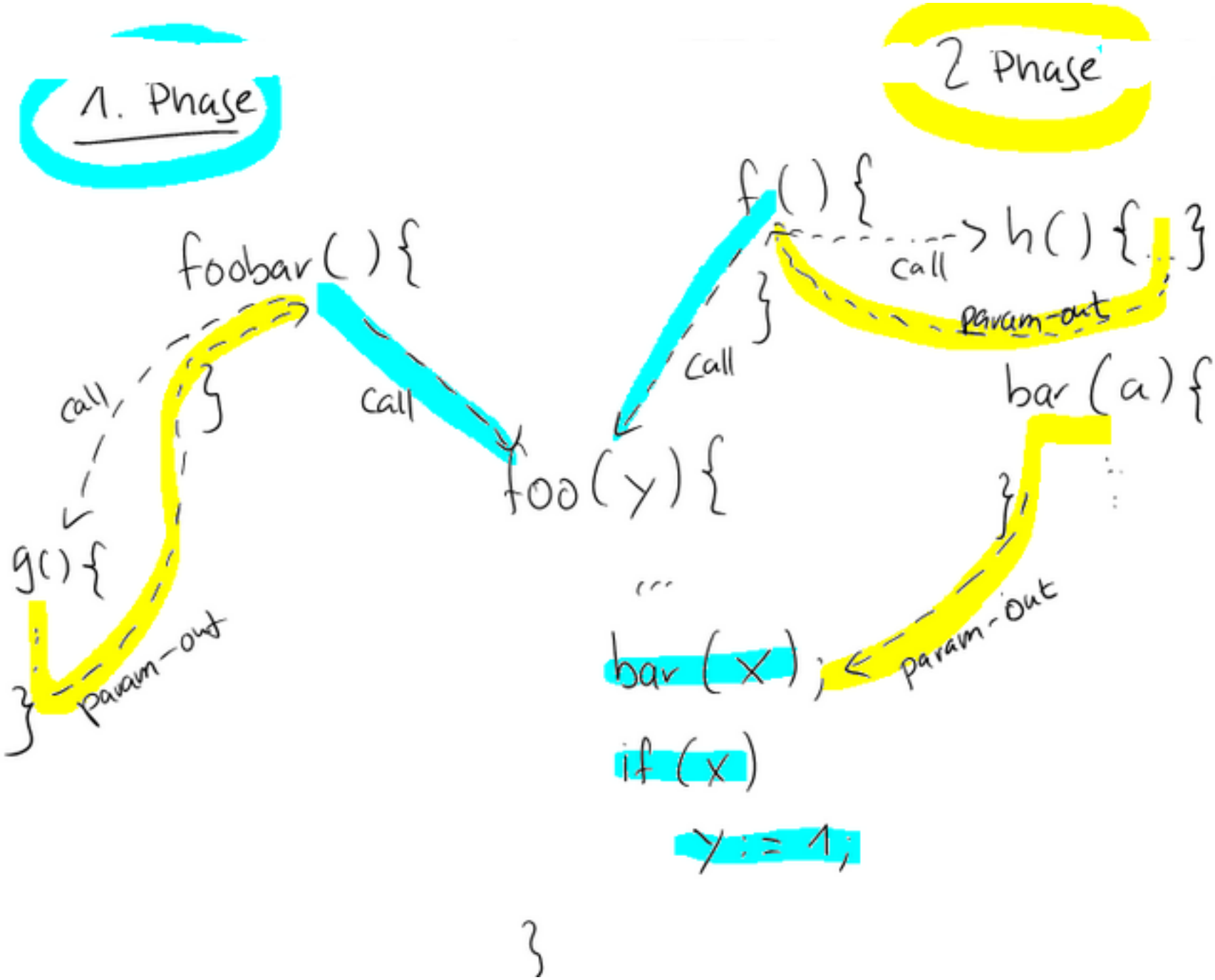
Software-Reengineering

Prof. Dr. Rainer Koschke

Montag, 20.11.2006


1. Phase
2 Phase
foobar( ){
call
call
g(){
param-out
}
foo(y){
...
bar(x);
if(x)
y := 1;
}
f(){
call
}
h(){ ... }
param-out
bar(a){
param-out

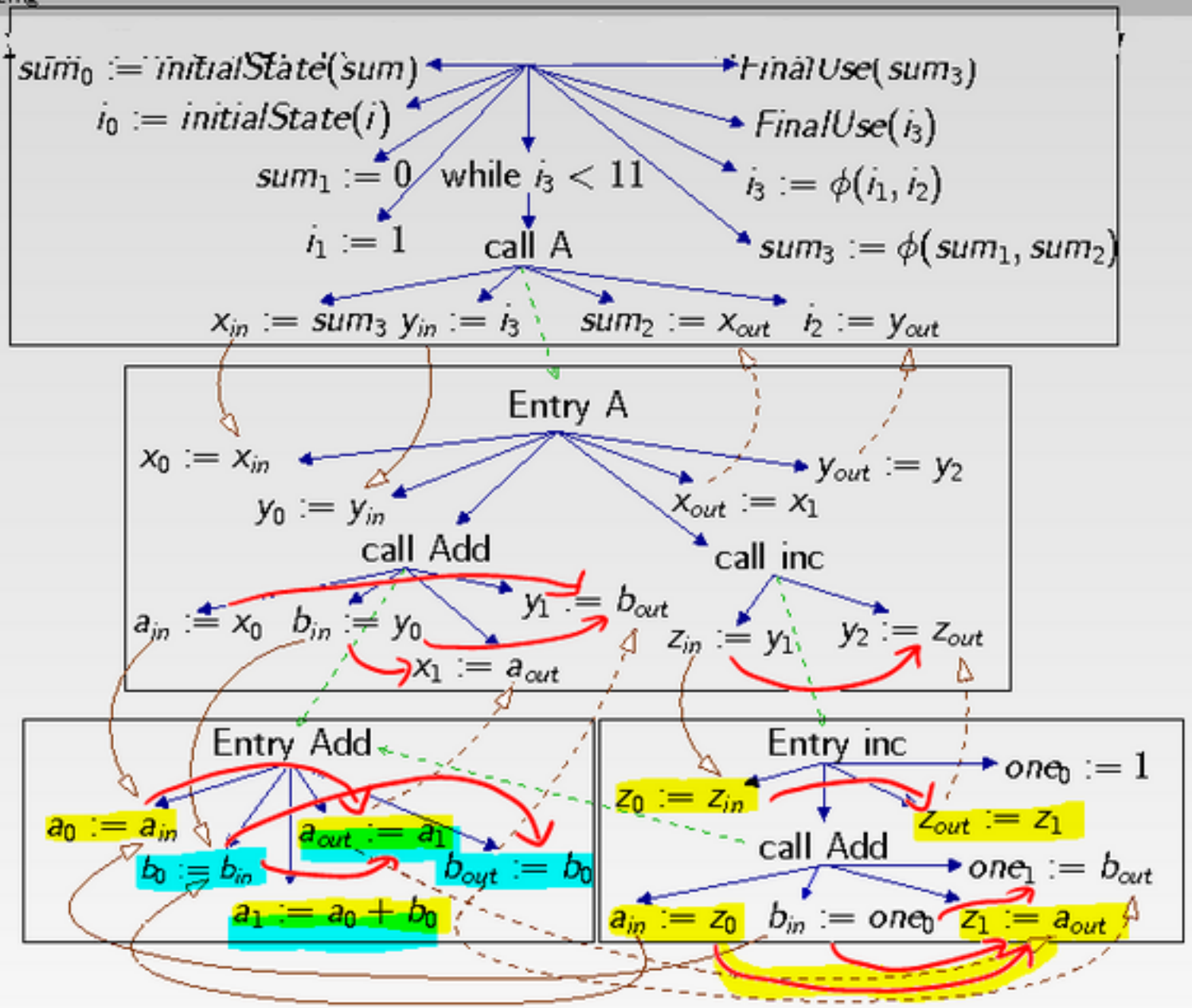
Summary-Edges

- repräsentieren (transitive) Abhängigkeiten der aktuellen In- und Out-Parameter an einer gegebenen Aufrufstelle
- helfen während des Slicings, unnötige Abstiege in aufgerufene Prozeduren zu vermeiden
- werden in zwei Schritten ermittelt:
  - direkte Abhängigkeiten zwischen den formalen Parametern innerhalb der aufrufenden Prozedur
  - indirekte Abhängigkeiten, die sich transitiv durch Aufruf anderer Prozeduren ergeben
- werden hergeleitet mit bekannten Techniken zu Abhängigkeiten in Attributgrammatiken.

Herleitung von Summary-Edges (1)

(1) Für jedes Unterprogramm UP:

- Ermittle alle transitiven Abhängigkeiten zwischen formalen Parameter-Knoten innerhalb des PDGs von UP (liefert intraprozedural induzierte Abhängigkeiten).
- Für jede eingefügte Kante (a, b) zwischen Parametern von UP füge eine unmarkierte Summary-Edge zwischen a und b ein.


sum0 := initialState(sum)
i0 := initialState(i)
sum1 := 0
while i3 < 11
i1 := 1
call A
i3 := φ(i1, i2)
sum3 := φ(sum1, sum2)
finalUse(sum3)
finalUse(i3)
xin := sum3
yin := i3
sum2 := xout
i2 := yout
Entry A
x0 := xin
y0 := yin
call Add
call inc
yout := y2
xout := x1
ain := x0
bin := y0
y1 := bout
x1 := aout
zin := y1
y2 := zout
Entry Add
a0 := ain
b0 := bin
a1 := a0 + b0
aout := a1
bout := b0
Entry inc
z0 := zin
one0 := 1
call Add
ain := z0
bin := one0
one1 := bout
z1 := aout
zout := z1

Herleitung von Summary-Edges (2)

(2) Solange eine unmarkierte Summary-Edge (a, b) eines Unterprogramms UP existiert:

- Markiere (a, b)
- Für jeden Aufruf von UP innerhalb eines Unterprogramms UP':
  - Füge eine Kante zwischen den zu a und b korrespondierenden aktuellen Parametern in UP' ein.
  - Bilde die transitive Hülle in UP' (liefert interprozedural induzierte Abhängigkeiten).
  - Für jede eingefügte Kante (a', b') zwischen aktuellen Parametern in UP' füge eine unmarkierte Summary Edge zwischen a' und b' ein, sofern sie noch nicht existiert.

Slicing-Variante: Chopping

Abhängigkeiten zwischen zwei Punkten

```
read (n);
i:= 1;
sum := 0;
product := 1;
while i <= n loop
            sum := sum + i;
            product := product * i;
            i := i + 1;
end loop;
write (sum);
write (product);
```

Slicing-Variante: Amorphes Slicing (Harmann u. a. 2003)

```
-- Original
if p = q then
  x := 18;
else
  x := 17;
end if;
if p /= q then
  y := x;
else
  y:= 2;
end if;
put (y);
```

```
-- gestalterhaltendes Slice
if p = q then
  x := 18;
else
  x := 17;
end if;
if p /= q then
  y := x;
else
  y:= 2;
end if;
put (y);
```

Slicing-Variante: Amorphes Slicing (Harmann u. a. 2003)

```
-- Original
if p = q then
  x := 18;
else
  x := 17;
end if;
if p /= q then
  y := x;
else
  y:= 2;
end if;
put (y);
```

```
-- Transformation
if p = q then
  x := 18; y := 2;
else
  x := 17;
  y := x;
end if;
put (y);
```

```
-- amorphes Slice
if p = q then
  x := 18; y := 2;
else
  x := 17;
  y := 17 *;
end if;
put (y);
```

Slicing-Varianten

- Backward-/Forward-Slicing, Chopping
- ausführbare Slices/nicht-ausführbare Slices
- statisches/dynamisches Slicing
- gestalterhaltendes (syntax-preserving)/amorphes Slicing
  - amorph: Slicing mit vereinfachender Transformation

Anwendungen von Slicing

"The spiritual symbol slicer attachment can turn any potato salad into a ministry."

Anwendungen von Slicing

- Programmverstehen
  - Reduzierung des Codes auf das für das Verständnis notwendige Maß
- Änderungsanalyse
  - alle von einer Änderung betroffenen Stellen
- Regressionstesten
  - Reduktion des zu testenden Codes
- Restrukturierung
  - z.B. Aufteilung von Unterprogrammen, die mehrere logisch verschiedene Funktionen implementieren Bewertung der Änderbarkeit (und somit der Wartbarkeit) eines Systems
- Messung von Kohäsion

Refactorings

2. Refactoring
   - Refactoring
   - Bad Smells
   - Refactorings von Fowler
   - Pull-Up Field
   - Extract Method
   - Bewertung Refactorings

Refactorings

Refactorings sind semantikerhaltende, restrukturierende Code-Transformationen für objekt-orientierte Programme (zur Verbesserung der Wartbarkeit)

Beschreibung nach Fowler (2000):

- Name
- Anwendbarkeit
- Motivation
- mechanische Schritte (die eigentliche Transformation), die von Hand ausgeführt werden
- Beispiel

Sehr viele dieser Refactorings sind genauso auf prozedurale Programme anwendbar.

Refactoring-Prozess

Angestoßen von Änderungswunsch.

Prozess (inkrementell, iterativ):

1. Identifikation eines "'schlechten Geruchs"' (bad smell)
2. Refactoring
3. Compile & Test
4. Eigentliche Änderung
5. Compile & Test

Stink Parade of Bad Smells by Fowler (2000)

- duplizierter Code
- lange Methoden
- große Klassen
- lange Parameterlisten
- divergente Änderung
  - eine Klasse wird stets geändert in verschiedener Weise und für unterschiedliche Gründe
- Schrotflinten-Chirurgie (Shotgun Surgery)
  - kleine Änderungen überall
- Feature-Neid
  - sehr viele Attribute einer anderen Klasse werden für eine Berechnung benutzt

Stink Parade of Bad Smells by Fowler (2000)

- Datenklumpen (Data Clumps)
  - eine Menge von Datenelementen, die häufig gemeinsam benutzt werden
  - z.B. Attribute einer Klasse, Parameter in Methodensignaturen
- Fixierung aufs Primitive (Primitive Obsession)
  - einfache Typen werden nicht als Klasse sondern als primitive Datentypen deklariert
- Switch-Anweisungen
  - händisches dynamisches Binden
- Parallele Vererbungshierarchien
- Faule Klassen
  - Klassen, die nichts Nützliches (mehr) tun
- ...

70 Refactorings von Fowler (2000)

- Methodenzusammensetzung
  - z.B. Extraktion von Methoden
- Eigenschaften zwischen Klassen bewegen
  - z.B. Verschiebung von Attributen oder Methoden
- Organisation von Daten
  - z.B. Verbergung von Attributen
- Vereinfachung bedingter Ausdrücke
  - z.B. Zerlegung komplexer Bedingungen
- Vereinfachung von Methodenaufrufen
  - z.B. Separierung von bloßem Zugriff von Manipulation
- Generalisierungen
  - z.B. Attribute oder Methoden in der Hierarchie auf- oder abwärts bewegen

Beispiel: Pull-Up Field

Gleiches Attribut in Unterklassen wird nach Oberklasse verlegt.

Motivation für Pull-Up Field nach Fowler (2000)

"'If subclasses are developed independently, or combined through refactoring, you often find that they duplicate features. In particular, certain fields can be duplicates. Such fields sometimes have similar names but not always. The only way to determine what is going on is to look at the fields and see how they are used by other methods. If they are being used in a similar way, you can generalize them."'

"'Doing this reduces duplication in two ways. It removes the duplicate data declaration and allows you to move from the subclassses to the superclass behavior that uses the field."'

Mechanics für Pull-Up Fields

1. Inspect all uses of the candidate fields to ensure they are used in the same way.

Mechanics für Pull-Up Fields

1. Inspect all uses of the candidate fields to ensure they are used in the same way.
2. If the fields do not have the same name, rename the fields so that they have the name you want to use for the superclass field.

Mechanics für Pull-Up Fields

1. Inspect all uses of the candidate fields to ensure they are used in the same way.
2. If the fields do not have the same name, rename the fields so that they have the name you want to use for the superclass field.
3. Compile and test.

Mechanics für Pull-Up Fields

1. Inspect all uses of the candidate fields to ensure they are used in the same way.
2. If the fields do not have the same name, rename the fields so that they have the name you want to use for the superclass field.
3. Compile and test.
4. Create a new field in the superclass.

Mechanics für Pull-Up Fields

1. Inspect all uses of the candidate fields to ensure they are used in the same way.
2. If the fields do not have the same name, rename the fields so that they have the name you want to use for the superclass field.
3. Compile and test.
4. Create a new field in the superclass.
5. If the fields are private, you will need to protect the superclass field so that the subclasses can refer to it.

Mechanics für Pull-Up Fields

1. Inspect all uses of the candidate fields to ensure they are used in the same way.
2. If the fields do not have the same name, rename the fields so that they have the name you want to use for the superclass field.
3. Compile and test.
4. Create a new field in the superclass.
5. If the fields are private, you will need to protect the superclass field so that the subclasses can refer to it.
6. Delete the subclass fields.

Mechanics für Pull-Up Fields

1. Inspect all uses of the candidate fields to ensure they are used in the same way.
2. If the fields do not have the same name, rename the fields so that they have the name you want to use for the superclass field.
3. Compile and test.
4. Create a new field in the superclass.
5. If the fields are private, you will need to protect the superclass field so that the subclasses can refer to it.
6. Delete the subclass fields.
7. Compile and test.

Mechanics für Pull-Up Fields

1. Inspect all uses of the candidate fields to ensure they are used in the same way.
2. If the fields do not have the same name, rename the fields so that they have the name you want to use for the superclass field.
3. Compile and test.
4. Create a new field in the superclass.
5. If the fields are private, you will need to protect the superclass field so that the subclasses can refer to it.
6. Delete the subclass fields.
7. Compile and test.
8. Consider using *Self Encapsulate Field* on the new field.

Beispiel: Pull-Up Field

Gleiches Attribut in Unterklassen wird nach Oberklasse verlegt.