Universität Paderborn
Fakultät für Elektrotechnik, Informatik und Mathematik
Arbeitsgruppe *Programmiersprachen und Übersetzer*

Refactoring in eXtreme Programming
Cross the Rubicon: Extract Method

Seminarausarbeitung für den integrierten Studiengang Informatik

Verfasser: Jacek Bandyk
Betreuer: Jochen Kreimer

Datum: 31.01.2003

Inhaltsverzeichnis

1 Refactoring in eXtreme Programming

Hinter dem Begriff *eXtreme Programming* (XP) verbirgt sich eine Softwareentwicklungsmethode. Sie umfasst neben Prinzipien wie Einfachheit, Teamarbeit auch Programmierparadigmen und Programmiertechniken. Unter *Refactoring* versteht man Verfahren, Methoden und Werkzeuge zur Änderung (Verbesserung) der Struktur eines Programms ohne die von Außen beobachtbare Semantik zu verändern. Es existiert aktuell sehr viel Software, die nur schlecht strukturiert ist und sich somit nur mit hohen Aufwand verändern, erweitern und warten lässt.

Der Hauptgrund für die Entstehung schlecht strukturierten Quellcodes und ungeschickter Architekturen sind die hohen Anfangskosten für die Entwicklung durchdachter und strukturierter Software. Adhoc- Lösungen können schneller und günstiger produziert werden.

1.1 Cross the Rubicon: Extract Method

Automatisierte Methodenextraktion ist ein wichtiges und nützliches Werkzeug des *Refectoring.* Aus diesem Grund wird daher auch die Umsetzung brauchbarer Lösungen als „Cross the Rubicon“ bezeichnet, zu deutsch „Einen entscheidenden Schritt tun“.

Ziel des Verfahrens ist es, ausgewählte, nicht zwingend zusammenhängende Codefragmente automatisch unter Beibehaltung der Semantik des Programms in eine eigene Funktion zu extrahieren. Die Codefragmente werden dabei für gewöhnlich im ursprünglichen Programm durch einen Aufruf der neu generierten Funktion ersetzt.

Mit Hilfe dieser Technik lassen sich große, monolithische Programme schneller und leichter modularisieren. Dabei können folgende Ziele verfolgt und Vorteile erreicht werden:

- Lange Code-Sequenzen enthalten oft Berechnungen vieler verschiedener Aufgaben, die konzeptuell nicht zusammenhängen. Die Aufteilung dieser Aufgaben in einzelne Funktionen verbessert das Verständnis und die Wartung dieser Codesegmente.
- Eliminierung von duplizierten Codesegmenten
  Wird an mehreren Stellen die gleiche Funktionalität benötigt, so wird sehr oft der betroffene Quellcode per *Copy&Paste* dupliziert. Enthält dieses Fragment einen Fehler oder muss es mal erweitert werden, müssen die Änderungen an allen Kopien durchgeführt werden. Wird so ein Segment dagegen in eine Funktion verpackt, beschränkt sich die Wartung auf diese Funktion. Ein Werkzeug zur automatischen Extraktion von Methoden ist in diesem Fall bereits zum Zeitpunkt der Entwicklung sehr nützlich. Der Entwickler kann beim ersten Bedarf der Codeduplizierung das Werkzeug nutzen und ohne viel Zeitverlust im Vergleich zu *Copy&Paste* besser strukturierten Code produzieren.
- Die Kapselung logisch abstrahierbarer Aufgaben in Funktionen fördert das Verständnis und die Wiederverwendung im Code. Ein klassisches Beispiel sind abstrakte Datentypen wie Listen oder Bäume. Diese Funktionen lassen sich dann oft einem anderen Kontext verwenden.
- *Code scavenging* – Code-Plünderung
  Dahinter steckt die Idee, brauchbaren Code mit Hilfe automatisierter Tools im großen Stil zu extrahieren. Dabei wird davon ausgegangen, dass bereits sehr viele Aufgaben und Algorithmen auf der Welt fertig programmiert sind. Code- Plünderung ist übrigens im kleinen Stil unter Entwicklern weit verbreitet (z.B. *Copy&Paste*).

1.1.1 Beispiel zur Methodenextraktion

Es sei folgende Methode in PASCAL- Notation gegeben:

```
procedure readAndProcess(days: integer, var arr: int_array, var sum:
    integer, process: boolean)
var i: integer;
begin
  i := 0;
  sum := i;
  while i < days do begin
    inc(i);
    read(arr[i]);
  end;
  if process = True then begin
    for i := 1 to days do
      sum := sum + arr[i];
  end;
end;
```

Die gegebene Methode gliedert sich in zwei konzeptuell unterschiedliche Teile:

1. Einlesen einer Menge von `days` Zahlen in ein Array
2. Ist Variable `process` wahr, wird die Summe der Zahlen im Array berechnet

Eine Beispielextraktion der Funktionalität zur Eingabe von `days` Zahlen in ein Array würde folgendermaßen aussehen:

```
procedure readAndProcess(days:
integer, var arr: int_array, var
sum: integer, process: boolean)
var i: integer;
begin
  readArr(days, arr);
  i := 0;
  sum := i;
  if process = True then begin
    for i := 1 to days do
      sum := sum + arr[i];
  end;
end;
```

```
procedure readArr(days: integer, var
arr: int_array)
var i: integer;
begin
  i := 0;
  while i < days do begin
    inc(i);
    read(arr[i]);
  end;
end;
```

Es ist zu erkennen, dass alle relevanten Anweisungen in eine neue Funktion `readArr` extrahiert sind. Diese Anweisungen sind in der ursprünglichen Funktion durch den Aufruf der neu erzeugten Funktion ersetzt.

Im folgenden Text wird ein Verfahren vorgestellt, das diesen Vorgang automatisch durchführt.

2 Verfahren zur Methodenextraktion

2.1 Problemstellung

Die Aufgaben bei der Extraktion von Methoden gliedern sich grob in drei Schritte:

1. Identifikation der zu extrahierenden Anweisungen
   Diese können vom Benutzer vollständig angegeben sein. Werden zur Extraktion mehr als die vom Benutzer angegebenen Anweisungen benötigt, kann das Werkzeug durch *Program Slicing* die zu extrahierende Menge ergänzen.
2. Ist die Menge der zu extrahierenden Anweisungen nicht zusammenhängend, müssen diese zusammengeführt werden, ohne die Semantik des Programms zu verändern. Dieser Vorgang gehört zu dem schwierigsten Teil bei der Methodenextraktion.
3. Die Anweisungen werden extrahiert und durch den Aufruf der neu generierten Methode ersetzt. Dabei müssen entsprechende Parameter und Rückgabewerte identifiziert werden.

Die vorgestellte Methode ist zum Teil theoretischer Natur und behandelt keine sprachspezifischen Eigenschaften.

2.2 Program Slicing

Unter dem Begriff *Program Slicing* verbirgt sich eine Technik zur Programmdekomposition mit dem Ziel, alle für eine angegebene Aufgabe oder Berechnung relevanten Anweisungen im Programm zu lokalisieren. Diese können dabei beliebig im Programm verstreut sein. *Program Slicing* basiert auf der statischen Analyse des Datenflussgraphen eines Programms.

Als Eingabe wird ein leicht modifizierter Kontrollflussgraph mit Datenabhängigkeitsinformationen benötigt. Das Verfahren liefert für das sogenannte *Slice Criterion*, C=(Befehl, Variablenmenge) alle für diesen Befehl und zur Definition der gegebenen Variablen notwendigen Programmstellen.

Die einzelnen Schritte des Verfahrens werden im folgenden Textverlauf sukzessiv definiert.

Definition: *Kontrollflussgraph CFG* ist ein gerichteter Graph G = (N, E) mit eindeutigen Start- und Endknoten ε_G, $\chi_G \in N$, so dass jeder Pfad in G von ε_G nach χ_G führt. Jeder Knoten des Graphen steht für eine atomare Anweisung im Programm. Zu jedem Knoten sind Variablennutzungs- und Definitionen verfügbar.

Hinweis: Es handelt sich hierbei nicht um einen klassischen Kontrollflussgraphen, da dieser Codeabschnitte ohne Sprünge zu einem einzigen Knoten zusammenfasst.

Definition: *Postdominator*. Es seien $v, w \in N$. Ein Knoten w postdominiert v, wenn alle von v zu χ_G verlaufenden Pfade w beinhalten.

Definition: *Kontrollabhängigkeit*. Es seien n_i, n_k, c, c‘ $\in N$. Ein Knoten n_k ist kontrollabhängig von der Kante (n_i, c), wenn

1. n_k postdominiert das Ziel von (n_i, c)
2. es ex. eine Kante (n_i, c‘), so dass n_k das Ziel von (n_i, c‘) nicht postdominiert

Definition: *Datenabhängigkeit.*; Es seien n_i, $n_k \in N$. Ein Knoten n_k ist datenabhängig von n_i, wenn

1. n_k nutzt eine in n_i definierte Variable v
2. es gibt mindestens einen Pfad von n_i nach n_k, auf dem v nicht definiert wird

Definition: *Abhängigkeit*. Es seien n_k, n_i, $c \in N$. Ein Knoten n_k ist von n_i abhängig, wenn es von n_i datenabhängig oder von (n_i, c) kontrollabhängig ist.

Definition: *Slice.* Ein *Slice* für das *Slice Criterion* C=($n_k \in N$, Variablenmenge V) ist die Menge der Knoten $n_i \in N$, von denen n_k für die Variablenmenge V transitiv abhängig ist.

```
procedure readAndProcess(days:
   integer, var arr: int_array,
   var sum: integer, process:
   boolean)
var i: integer;
begin
  i := 0;
  sum := i;
  while i < days do begin
    inc(i);
    read(arr[i]);
  end;
  if process = True then begin
    for i := 1 to days do
      sum := sum + arr[i];
  end;
end;
```

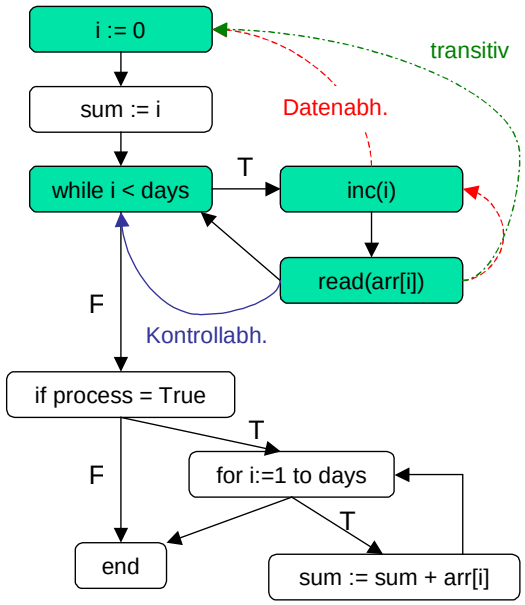

Abbildung 1: Kontrollflussgraph für eine Beispielanwendung

Im vorangehenden Beispiel sind im gegebenen *CFG* beispielhaft *Datenabhängigkeit, Kontrollabhängigkeit* sowie eine transitive *Abhängigkeit* eingezeichnet. Die grün markierten Knoten sind das *Slice* für das *Slice Criterion* C = (read, arr).

Durch die explizite Angabe der relevanten Variablen im *Slice Criterion* ist es möglich, das Berechnungsziel, zu dem das *Slicing*- Verfahren alle berechnungsrelevanten Codestellen liefern soll, präzise anzugeben. So werden im nachfolgenden Beispiel die Stellen vom *Slicing*- Verfahren geliefert, die lediglich an der Berechnung der Variable **total** beteiligt sind. Diese Unterscheidung ist notwendig, wenn z.B. die Berechnung von **total** und **sum** in zwei verschiedene Methoden extrahiert werden soll.

Die im folgenden Beispiel *kursiv* markierten Stellen zeigen das *Slice* für das *Slice Criterion* C = (write, total):

```
begin
      read(x, y);
      total := 0.0;
      sum := 0.0;
      if (x <= 1)
            then sum := y
      else begin
            read(z);
            total := x * y;
      end;
      write(total, sum);
end.
```

Es gibt eine Vielzahl weiterer Analysetechniken, die auf *Program Slicing* basieren sowie eine breite Palette von Anwendungsgebieten. Eines davon ist Methodenextraktion.

2.3 Tucking statements into functions

Tucking statements into functions (Tuck) ist ein vollständiges Verfahren zur automatisierten, semantikerhaltenden Methodenextraktion. Es beinhaltet somit die in Abschnitt 2.1 beschriebenen Schritte, in die sich der Vorgang der Methodenextraktion gliedern lässt.

2.3.1 Hilfsdefinitionen, Graphen

Im Vorfeld werden wichtige Definitionen aufgeführt, die im weiteren Verlauf für die Erklärung des Verfahrens benötigt werden. Zunächst werden Graphen mit speziellen Eigenschaften vorgestellt.

Definition: *Teilgraph*: Ein *CFG* G' = (N', E') ist *Teilgraph* von G = (N, E) wenn $N' \subseteq N$ und $E' \subseteq (N' \times N')$.

Definition: *SESE Teilgraph*. Ein *Teilgraph* G' = (N', E') ist ein *single- entry, single- exit Teilgraph* von G = (N, E), wenn:

$\exists\ \varepsilon_{G'}, \chi_{G'} \in N';\ \forall\ q \in (N - N'), q' \in N'$: $(q, q') \in E \Rightarrow q' = \varepsilon_{G'}$ und $(q', q)\ E \Rightarrow q' = \chi_{G'}$

In Worten: Alle Kanten, die in den *Teilgraphen* G' führen, haben den Startknoten $\varepsilon_{G'}$ und alle Kanten, die ihn verlassen, gehen von $\chi_{G'}$ aus.

Definition: *Faltbarer Teilgraph.* Ein Graph G' = (N', E') ist ein *faltbarer Teilgraph* von G = (N, E), wenn er ein *SESE Teilgraph* von G ist. Zusätzlich darf eine Kante von $\chi_{G'}$ nach $\varepsilon_{G'}$ existieren. Ein *faltbarer Teilgraph* kann vollständig aus einem *CFG* extrahiert werden, ohne die Kontrollflusseigenschaften des ursprünglichen Graphen zu verletzen. Codesegmente, die im vorgestellten Verfahren extrahiert werden sollen, sind immer f*altbare Teilgraphen*.

Die folgenden Definitionen stellen Operationen auf Graphen dar, die vom *Tuck*- Verfahren benötigt werden.

Definition: [V] *Einführung neuer Variablennamen*. Es seien V eine Variablenmenge. [V] repräsentiert ein Menge geordneter Paare (v, v') mit $v \in V$. v' ist dabei ein neuer, noch nicht benutzter Variablenname.

Definition: [V]G*, Variablenumbenennung im Teilgraphen G*: Es sei V eine Menge von Variablen in G. [V]G ist ein neuer Graph, in dem alle Variablen $v \in V$ durch v', $(v, v') \in [V]$ ersetzt sind.

Definition: I[V] *Einführung neuer Zuweisungen*: Es sei V eine Variablenmenge. I[V] repräsentiert einen neuen Graphen, dessen Knoten Zuweisungen für alle $v \in V$ der Form v' = v enthalten.

Die folgende Abbildung zeigt einen einfachen Beispielgraphen, in dem für die Variablen a, b und c Variablen mit neuen Namen eingeführt und ihnen die Werte ihrer korrespondieren Variablen zugewiesen werden.

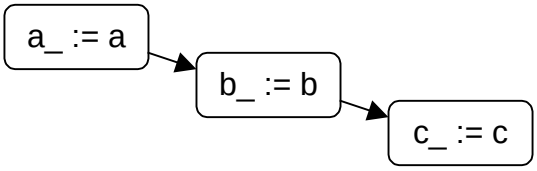

Abbildung 2: Beispiel Einführung neuer Zuweisungen

Definition: $G_1;G_2$, *Sequentielle Graphkomposition durch hinzufügen von Kanten.* Es seien G1, G2 *faltbare Teilgraphen*.

$G1; G2 = G1 \cup G2 \cup (\varnothing, \{\chi_{G1}, \varepsilon_{G2}\})$.

Das folgende Beispiel illustriert die *sequentielle Graphkomposition durch hinzufügen von Kanten*. Es ist zu erkennen, dass ausgehend vom Endknoten 2 des ersten Graphen eine neue Kante zum Startknoten 3 des zweiten Graphen verläuft und die Graphen verbindet. Start- und Endknoten sind eindeutig bestimmt, da es sich um *faltbare Teilgraphen* handelt.

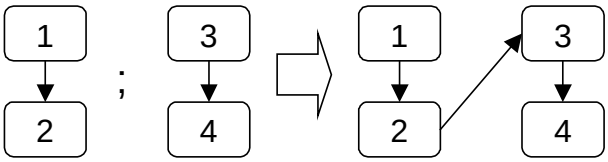

Abbildung 3: Sequentielle Graphkomposition durch hinzufügen von Kanten

Definition: $G1 \oplus G2$, *Sequentielle Graphkomposition durch ersetzen des Endknotens*. Es seien G1, G2 *faltbare Teilgraphen*.

$G1 \oplus G2 = ([\varepsilon_{G2}/\chi_{G1}]G1) \cup G2$

Im folgenden Beispiel ist zu erkennen, dass der Endknoten 2 des ersten Graphen entfernt wird. Alle Kanten die ursprünglich zu diesem Knoten verliefen, verlaufen im komponierten Graphen zum Startknoten des zweiten Graphen. Start- und Endknoten sind wiederum eindeutig bestimmt.

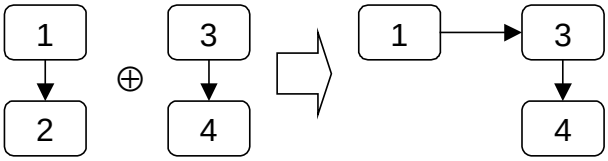

Abbildung 4: Beispielkomposition durch ersetzen des Endknotens

2.3.2 Hilfsdefinitionen, Variablen

Definition: *Region*. Eine *Region* ist eine Menge von Anweisungen in *Kontrollflussgraphen*.

Definition: *IN?(G, X, v)*. Es seien G ein *CFG*, X eine *Region* in G und v eine Variable in X. Eine Variable v ist Eingabevariable für eine Region X, wenn es mindestens eine Definition von v außerhalb von X gibt, von der ein Pfad zu einem Knoten in X führt, der v nutzt.

Definition: *OUT?(G, X, v)*: Es seien G ein *CFG*, X eine *Region* in G und v eine Variable in X. Eine Variable v ist Ausgabevariable für eine Region X, wenn es mindestens eine Definition von v innerhalb von X gibt, von der ein Pfad zu einem Knoten außerhalb von X führt, der v nutzt.

Definition: *Value(G, X)*. Es seien G ein *CFG* und X eine *Region* in G. *Value* liefert die Menge aller Eingabevariablen für die *Region* X.

Value(G, X) = {v | IN?(G, X, v) und nicht OUT?(G, X, v), v ist Variable in X}

Definition: *Outvar(G, X)*. Es seien G ein *CFG* und X eine *Region* in G. *Outvar* liefert die Menge aller Ausgabevariablen für die *Region* X.

Outvar(G, X) = {v | OUT?(G, X, v), v ist Variable in X }

Definition: *Local(G, X)*. Es seien G ein *CFG* und X eine *Region* in G. *Local* liefert die Menge aller lokalen Variablen für die *Region* X.

Local(G, X) = {v | nicht OUT?(G, X, v) und nicht IN?(G, X, v), v ist Variable in X }

Die vorgestellten Definitionen werden zur Identifikation der entsprechenden Variablenmengen für das zu extrahierende Codesegment benötigt. Die folgende Abbildung zeigt die Variablenmengen für die Region X, dargestellt als grün markierte Knoten.

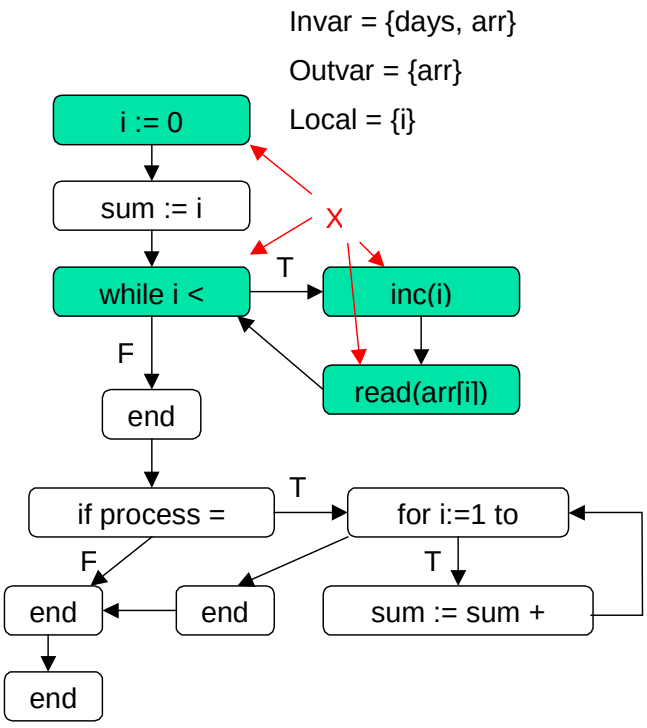

Abbildung 5: Beispiel Variablenmengen

2.3.3 Definition Tuck

Definition*: Wedge.* Seien S eine Menge von Anweisungen und G_S ein *faltbarer Teilgraph* von G, so dass G_S alle Anweisungen von S enthält.

Wedge(G, G_S, S) = *Slice*(G, S) $\cap$ N_S

Wedge ist der erste Schritt im Verfahren und dient der automatischen Vervollständigung aller für die zu extrahierende Funktionalität relevanter Codestellen. Durch die Angabe von G_S lässt sich der Bereich, in dem nach relevanten Codestellen gesucht werden soll, einschränken. Ist so eine

Einschränkung nicht erwünscht, so kann der *CFG* der vollständigen Funktion angegeben werden. Die Identifizierung von G_S ist Sache von entsprechenden Werkzeugen.

Im folgenden Beispiel ist ein *faltbarer Teilgraph* sowie der *Wedge* für das *Slice* bestehend aus den grün markierten Knoten eingezeichnet. *Wedge* und *Slice* sind in diesem Beispiel gleich.

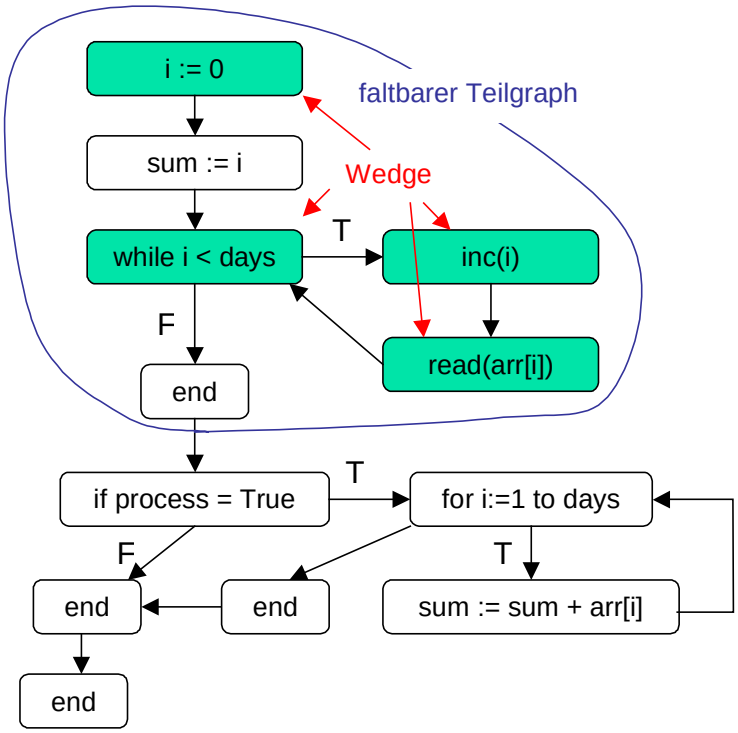

Abbildung 6: Beispiel eines Wedge im gegebenen, faltbaren Teilgraphen

Definition: *Split.* Der *Split*- Schritt enthält Transformationen, die alle zu extrahierenden Codestellen an ein zusammenhängendes Stück bringen und sie somit extrahierbar machen.

```
Split(G, GS, S) = Gnew:
  X  = Wedge(G, GS, S)
  X‘ = NS – Exitnode(GS) – X
  Y  = Wedge(G, GS, X‘)
  Y‘ = NS – Exitnode(GS) – Y
  if Outvar(G, X) ∩ Outvar(G, Y) ≠ ø then
    conflict
  else
    GX = GS / X‘
    GY = GS / Y‘
    V = Local(G, Y) ∪
          (Invar(G, Y) - Outvar(G, Y))
    GXY = I[V]; GX ⊕ [V]GY
    Gnew = [GXY / GS] G
```

Es werden die Mengen X und Y gebildet. X ist die vom *Wedge* gelieferte Menge, die alle extraktionsrelevanten Codestellen enthält. Y ist die Menge der Codestellen, die notwendig sind, um die verbleibende Funktionalität noch richtig berechnen zu können. Besitzen beide dieser Codeblöcke gemeinsame Ausgabevariablen, ist eine semantikerhaltende Extraktion nicht möglich und das Verfahren bricht ab.

In der folgenden Abbildung ist eine Beispielangabe der Knotenmengen X, X’, Y und Y’ dargestellt.

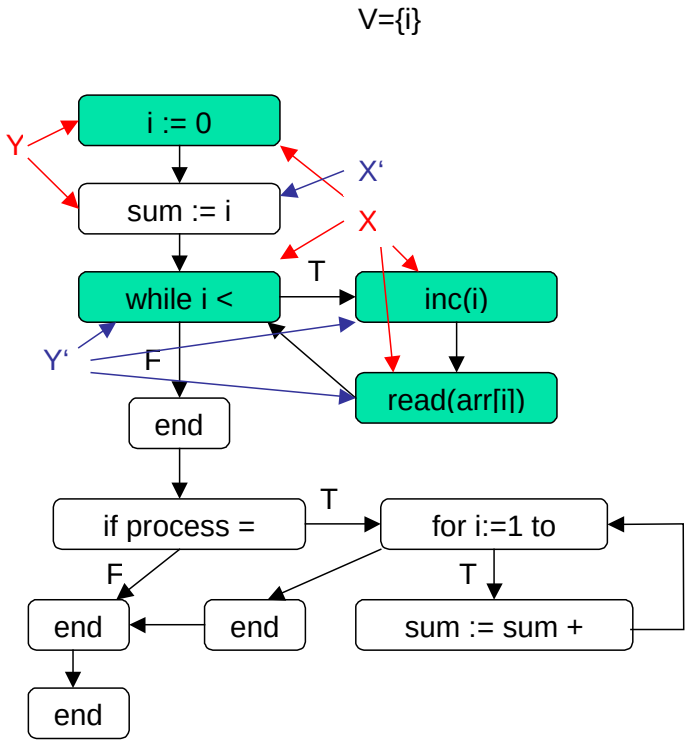

Abbildung 7: Bildung der Knotenmengen im Split- Schritt

Nun werden alle von Y benötigten Eingabevariablen in temporären Variablen gesichert. In Y werden dann diese Kopien genutzt. Diese Variablen werden also vor der Ausführung von X gerettet und stehen somit den Anweisungen in Y noch richtig initialisiert zur Verfügung.

Damit ist G_{new} semantisch äquivalent zu G, da keine der durchgeführten Transformationen die Semantik verändert. Nachfolgend sind beispielhaft Graphoperationen zur Bildung von G_{XY} und der resultierende Graph G_{new} dargestellt.

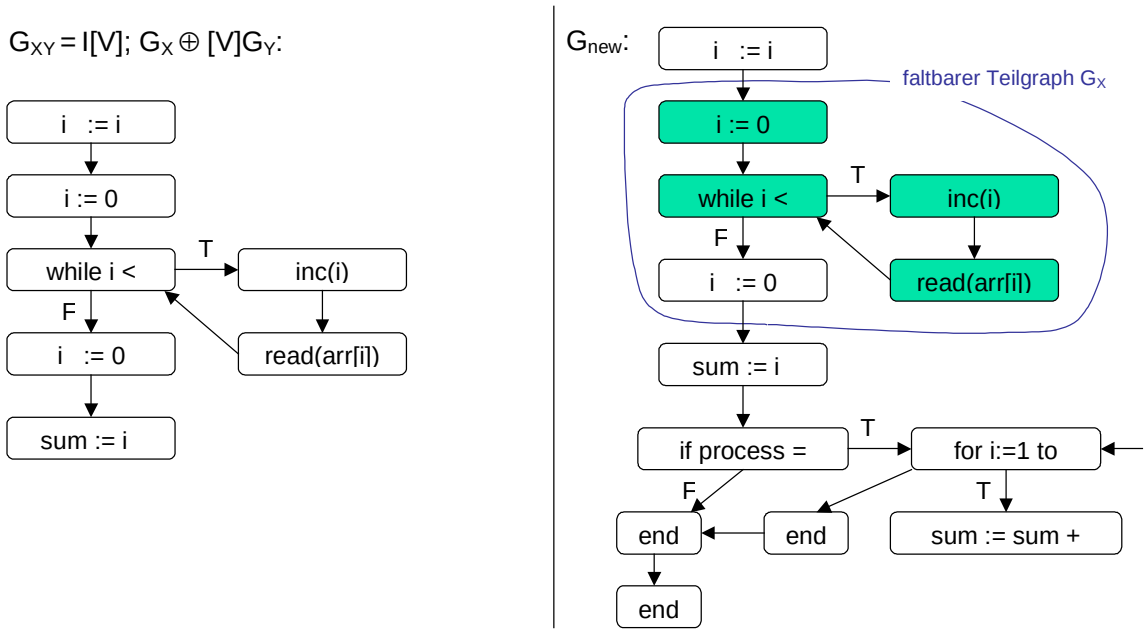

Abbildung 8: Graphoperationen zur Bildung von G_{XY} und resultierender Graph G_{new}

Definition: *Fold.* Dieser Schritt ist für die Extraktion des im *Split*- Schritt entstandenen *faltbaren Teilgraphen* G_X in eine neue Funktion zuständig. Abschließend wird an die ursprüngliche Stelle der Aufruf der neuen, extrahierten Funktion hinzugefügt.

Es seien G, G1, G2 *CFGs,* G_X sei *faltbarerer Teilgraph* in G.

Fold(G, G_X) = <G1, G2>:

$N' = (N_X - \chi_{Gx})$
e2, r2 seien neue Start- und Endknoten in G2
$N2 = N' \cup \{e2, r2\}$
$E2 = (E \cap N' \times N') \cup \{(e2, \varepsilon_{Gx})\} \cup \{(n_X, r2) | (n_X, \chi_{Gx}) \in E_X\}$
$G2 = (N2, E2)$ ist CFG mit $\varepsilon_{G2} = e2$ und $\chi_{G2} = r2$
*f*1 ist Aufruf für G2
$N1 = (N - N') \cup \{f1\}$
$E1 = (E \cap N1 \times N1) \cup \{(f1, \chi_{Gx})\}) \cup \{(n, f1) | (n, \varepsilon_{Gx}) \in E\}$
$G1 = (N1, E1)$ ist CFG mit $\varepsilon_{G1} = \varepsilon_G$ und $\chi_{G1} = \chi_G$

Zusätzlich seien:

Ref(G2) = *Ref*(G1, *f*1) = *Outvar*(G1, N')

Value(G2) = *Value*(G1, *f*1) = *Value*(G1, N')

Local(G2) = *Local*(G1, N')

Der *Fold* Schritt erfolgt auf dem durch *Split* definierten Graphen G_{new}. Dieser enthält für den Teilgraphen G_Y noch temporäre Variablennamen sowie vorangehende Zuweisungen an diese Variablen. Nach dem *Fold*- Schritt werden diese Zuweisungen entfernt und die alten Variablennamen wieder hergestellt. Das im Verlauf des Texts genutzte Beispiel resultiert nach der automatischen Extraktion unter Verwendung des *Tuck*- Verfahrens in die bereits aus Abschnitt 1.1.1 bekannten Funktionen:

```
procedure readAndProcess(days:
integer, var arr: int_array, var
sum: integer, process: boolean)
var i: integer;
begin
  readArr(days, arr);
  i := 0;
  sum := i;
  if process = True then begin
    for i := 1 to days do
      sum := sum + arr[i];
  end;
end;
```

```
procedure readArr(days: integer, var
arr: int_array)
var i: integer;
begin
  i := 0;
  while i < days do begin
    inc(i);
    read(arr[i]);
  end;
end;
```

2.3.4 Vorbehandlung und externe Werkzeuge

Beim vorgestellten Verfahren ist eine gewisse Vorbehandlung des Quellcodes notwendig. Dazu müssen unter Umständen externe Werkzeuge eingesetzt werden. Das kann folgenden Zielen dienen:

- Das *Tuck*- Verfahren benötigt einen *Kontrollflussgraphen* als Eingabe, der *Wedge*- Schritt benötigt einen *faltbaren Teilgraphen*. Die Erhebung dieser Daten wird dem Anwender oder externen Werkzeugen überlassen.
- Nicht initialisierte Variablen oder tote Variablendefinitionen sind für die Auffindung von *Datenabhängigkeiten* störend und können durch eine entsprechende Analyse und Vorbehandlung eliminiert werden.
- Die Analyse interprozeduraler Zugriffe von Zeigervariablen ergänzt die Variablennutzungs- und Definitonsangaben.
- Sprachspezifische Eigenschaften können durch eine Vor- oder Nachbehandlung behandelt werden.

2.4 Semantikerhaltende Methodenextraktion

Das vorgestellte *Tuck*- Verfahren kann Konstrukte nicht extrahieren, wenn die Datenabhängigkeiten komplexer sind. Es bricht ab, wenn der zu extrahierende Teil sowie der verbleibende Teil (Teilgraphen G_X und G_Y, siehe Abschnitt 2.3.3, *Split*) gemeinsame Ausgabevariablen besitzen.

Es gibt zu diesem Thema eine weiterführende Arbeit von Raghavan Komondoor und Susan Horwitz [3]. Diese Arbeit konzentriert sich auf ein Verfahren, das komplexere *Daten*- und *Kontrollabhängigkeiten* in einem *Kontrollflussgraphen* auflösen kann, ohne die Semantik des Programms zu verändern.

Dieses Verfahren nutzt eine Reihe von Ordnungsregeln, die eine semantikerhaltende Restrukturierung des *Kontrollflussgraphen* ermöglichen.

3 Zusammenfassung

Die Automatisierung der Methodenextraktion ist ein sehr wichtiger und nützlicher Schritt im *Refactoring*. Normalerweise werden Transformationen im Rahmen eines *Refactorings* manuell vorgenommen und ein Erhalt der Semantik durch Tests sichergestellt. Durch eine Automatisierung vermindert sich der Aufwand für die Durchführung dieser Transformation erheblich. Das Verfahren behält die beobachtbare Semantik des Programms in jedem Fall bei. Damit sind Fehler ausgeschlossen.

Methodenextraktion ist somit nicht nur theoretisch interessant, sondern praktisch sehr nützlich. Einige Entwicklungsumgebungen stellten bereits Werkzeuge zur automatischen Methodenextraktion bereit.

Das vorgestellte Verfahren liefert eine Grundlage für automatisierte Methodenextraktion und lässt sich, wie auch schon angedeutet, durch ergänzende oder weiterführende Arbeiten und externe Werkzeuge ergänzen.

Das Problem der semantischen Äquivalenz ist unentscheidbar, es werden jedoch brauchbare Näherungslösungen genutzt.

4 Anhang

4.1 Abbildungsverzeichnis

4.2 Literaturverzeichnis

[1] *Restructuring prgrams by tucking statements into functions*, A. Lakhotia and J. Deprez

[2] *Extracting Reusable Function by flow Graph- Based Program Slicing*, F. Lanubile and G. Visaggio

[3] *Semantics- Preserving Procedure Extraction*, R. Kommondoor and S. Horwitz