

Schrittweise Programmentwicklung

1.4 Schrittweise Programmentwicklung

Nutzen der Verifikationsmethodik:

- „Konstruktion“ von Programmen durch präzise Definition (den Zusicherungen!) von Programmezuständen
- Dokumentation des Codes

Systematisches Durchdenken der Bedingungen und Formulieren in anderer Sprache als Programmiersprache hilft Fehler erkennen. (aber nicht alle....: Rundungsfehlern Gleitkommaarithmetik, Wertebereich von Typen, Typkorrektheit)

hs / fub – alp3-1.4/5 1

Fallstudie Programmentwicklung (1)

Beispiel für systematische Programmentwicklung

Anforderungsdefinition:

Gegeben eine Liste von Tageskursen einer Aktie für n_0 Tage.
Gesucht: der maximale Spekulationsgewinn, das ist die größte Differenz von zwei Aktienkursen, wobei der Verkaufstag größer als der Tag des Kaufs ist.

(n_0 ist hier eine Konstante)

hs / fub – alp3-1.4/5 2

Fallstudie Programmentwicklung (2)

Spezifikation: (P,Q) mit

$$P \equiv a.length = n0, \forall 0 \leq i < n0 \ a[i] = A[i]$$

(alle Werte definiert) $\wedge \ n0 > 1$

← mindestens 2 Tage

$$Q \equiv g = \max(a[j] - a[i]) \wedge 0 \leq i < j < n0$$
$$0 \leq i < j < n0$$

hs / fub - alp3-1.4/5 3

Fallstudie Programmentwicklung (3)

Invariante bestimmen

Methode B:

Konstante in Q durch Variable ersetzen,
n0 durch Variable n , also $2 \leq n \leq n0$

$$INV : g = \max a[j] - a[i], 0 \leq i < j < n \wedge 2 \leq n \leq n0$$

g enthält immer den maximalen Kursgewinn der ersten n Tage

Schleifenbedingung :

folgt aus $INV \wedge n = n0 \Rightarrow Q$, also $n \neq n0$

hs / fub - alp3-1.4/5 4

Fallstudie Programmentwicklung (4)

Variablen: g, n, a, i, j
Invariante herstellen:

```
g = a[1] - a[0]
```

Erste Teillösung:

```
{n0 >= 2}
g = a[1] - a[0];
n = 2;
{INV}
while (n != n0) { // {INV ∧ n != n0}
    bestimme Maximum der Differenz für  $0 \leq i < j \leq n$ 
    und erhöhe n um 1
}
{INV}
```

hs / fub - alp3-1.4/5 5

Fallstudie Programmentwicklung (5)

Schrittweise Verfeinerung: Schleifenrumpf konstruieren

Zerlegen :

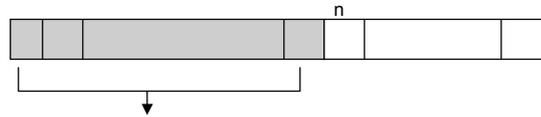
```
{Inv ∧ n != n0}
„ n berücksichtigen“
{INVn+1} // Invariante, bei der n durch n+1 ersetzt
n = n+1;
{INV}
```

„ n berücksichtigen “

$INV_{n+1} : g = \max a[j] - a[i], 0 \leq i < j < n+1 \wedge 2 \leq n+1 \leq n0$

hs / fub - alp3-1.4/5 6

Fallstudie Programmentwicklung (6)



$$g = \max a[j] - a[i], 0 \leq i < j < n$$

g vergrößert sich nur bei Erweiterung auf $n+1$ Feldelemente, wenn eine der Differenzen $a[n] - a[i] > g, i < n$

$$\begin{aligned} g_{n+1} &= \max (a[j] - a[i]), 0 \leq i < j < n+1 \\ &= \max_2 (\max (a[j] - a[i]), 0 \leq i < j < n, \max (a[n] - a[i], 0 \leq i < n)) \end{aligned}$$

hs / fub - alp3-1.4/5 7

Fallstudie Programmentwicklung (7)

$$\begin{aligned} \text{Maximum } \max (a[n] - a[i], 0 \leq i < n) &= a[n] - \min (a[i], 0 \leq i < n) \\ \Rightarrow \\ g &= \max_2 (\max (a[n] - a[i], 0 \leq i < n), a[n] - \min (a[i], 0 \leq i < n)) \end{aligned}$$

Minimum der $a[i]$ merken, $0 \leq i < n$, Variable mina !

$$\Rightarrow g = \max_2 (g, a[n] - \text{mina})$$

Zusätzliche Invariante: $\text{INV}' = \text{mina} = \min a[i], 0 \leq i < n$

$$\Rightarrow \text{Erweiterte Invariante: } \text{INV}_{\text{ges}} = \text{INV} \wedge \text{INV}'$$

hs / fub - alp3-1.4/5 8

Fallstudie Programmentwicklung (5)

Herstellen von INV':

```
mina = min2(a[0], a[1])
```

Herstellen von INV' beim Übergang $n \rightarrow n+1$:

```
mina = min a[i], 0 <= i < n
```

rekursiv: `mina = min2(mina, a[n-1])`

hs / fub - alp3-1.4/5 9

Fallstudie Programmentwicklung (6): Java-Methode

```
static float maxWin(float a[]) {  
    //requires: a.length = n0 > 1,  
    //          for all 0 <= i < n0 a[i] = A[i]  
    //effects:  returns g = max a[j] - a[i] , 0 <= i < j < n0  
    int n0 = a.length;  
    int n = 2;  
    float mina = min2( a[1], a[0]); // min2: Minimum of 2 values  
    float g = a[1] - a[0];  
    //assert: g = max ( a[j] - a[i]  0 <= i < j < n, 2 <= n <= n0,  
    //          mina = min a [i], 0 <= i < n  
    while ( n!=n0) { // {INV and n != n0}  
        mina = min2(mina, a[n-1]);  
        g = max2(g, a[n] - mina);  
        n = n+1;  
    }  
    return g;  
}
```

Linearer Algorithmus!

hs / fub - alp3-1.4/5 10

Fallstudie Programmentwicklung (7)

◆ Quadratischer Algorithmus

```
static float maxWin(float a[]) {  
    //requires: a.length = n0 > 1,  
    //          for all 0 <= i < n0 a[i] = A[i]  
    //effects:  returns g = max a[j] - a[i] , 0 <= i < j < n0  
    int n0 = a.length;  
    float g = a[1]-a[0];  
    //assert:  Inv: 1<= j < n0 , 0 <= i < j,  
    //          diff = max (a[j]-a[i])  
    for (int j = 1; j < n0; j++) {  
        for (int i = 0; i < j; i++) {  
            float diff= a[j]-a[i];  
            if (diff>g) g=diff;  
        }  
    }  
    return g;  
}
```

Quadratischer Algorithmus !

hs / fub - alp3-1.4/5 11

Zusammenfassung Spezifikation und Verifikation

1.5 Spezifikation, Verifikation, Testen

Semantik einer Programmiersprache dient u.a

- zur **Definition** der Programmiersprache (hier: axiomatisch), ist Grundlage zur Implementierung von Übersetzer / Interpretierer
- **Programmspezifikation**
was soll das Programm tun?
Präzisierung der Anforderungsdefinition
- **Programmverifikation**
Beweis, dass Programm tut, was es soll

hs / fub - alp3-1.4/5 12

Verifizieren oder Testen?

- **Programmfehler:** Programm tut nicht das, was es soll oder tut etwas, was es nicht soll
- Es gibt praktische **keine (sehr großen) fehlerfreien Programme**
häufig völlig unabhängige Entwicklung (Betriebssystem, Anwendungsprogramm),
Probleme im Zusammenwirken der Teile!
- Für manche Programme ist formale Verifikation ein Qualitätsmerkmal
Zertifizierung von Programmen nach formaler Verifikation,
Bsp.: **Steuerprogramme in Verkehrsleitsystemen, ABS-Software,...**

hs / fub – alp3-1.4/5 13

Verifizieren oder Testen?

- **Begleitende Verifikation** vermindert Fehler und unterstützt die **Konstruktion von Algorithmen**
- Ohne Spezifikation keine Verifikation,
.... nicht einmal ein sinnvolles Programm
- meist aufwendig
Verifikationsprogramme wünschenswert
- Tests beweisen (ggf.) die Anwesenheit von Fehlern,
nie die Abwesenheit
(wenn unendlich viele mögliche Eingaben)

hs / fub – alp3-1.4/5 14

Anmerkungen zum Testen(1)

Klassifikation

Schnittstellen- (Funktions-) test (blackbox test)

Ein- / Ausgabe-Relation auf Konformität zur Spezifikation prüfen

Programmabhängiger Test (whitebox/glassbox test)

Überprüfung möglichst großer Teile aller Pfade durch das Programm

Möglichst große Überdeckung (des Programmcodes) wünschenswert

Hauptproblem: kombinatorische Explosion der Testfälle, deshalb Vollständigkeit meist illusorisch.

hs / fub – alp3-1.4/5 15

Anmerkungen zum Testen(2)

Systematische Auswahl von Testfällen:

Schnittstellentest (Funktionstest)

pro spezifizierter Bedingung mindestens ein Testfall
Randbereiche (ggf. von beiden Seiten) prüfen, Maximal-, Minimalwerte
Genügende Anzahl von Normalfällen
Zufällige Erzeugung von Testfällen

Überdeckungstest erwünscht, aber kaum machbar:

Wegüberdeckung: jeden Weg einmal durchlaufen

Abschwächung: jede Anweisung einmal durchlaufen

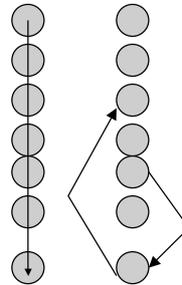
(**Anweisungsüberdeckung [C0]**)

stärker : **zweigüberdeckend [C1]:** jeden Zweig mindestens einmal durchlaufen

hs / fub – alp3-1.4/5 16

Beispiel

```
count=0;
cin = read...;
while (!in.eof){
  count++;
  if ((cin=='a') || (cin == 'e')
      ||...)) vowel++;
cin = read...;
}
```



anweisungsüberdeckend zweigüberdeckend

Wegüberdeckend: jeder mögliche Pfad im Programm wird einmal durchlaufen, etwa bei der Fallunterscheidung 2ⁿ Möglichkeiten bei n Schleifendurchläufen!

hs / fub - alp3-1.4/5 17

Werbung.....

Tja... test
test

"Tja..." könnte die Antwort vieler Programmierer auf die Frage sein, wie sie denn ihre Programme getestet haben.
Aber Tja ist auch...

Ein leicht bedienbares, vielfältig einsetzbares Testwerkzeug für Java-Klassen

Tja unterstützt das bequeme Testen einzelner Java-Klassen (Unit-Tests). Es muss kein zusätzlicher Code geschrieben werden. Tja dient als Testtreiber und ermöglicht Regressionstests. Benötigt wird nur der Byte-Code.

Intro

Features Tja ist bei der Softwareentwicklung im kleineren Rahmen vielseitig einsetzbar:

Download

Screenshots

Tipps zur Benutzung

Mehr Information zum Testen

Ad-hoc-Testen: Eigene oder fremde Klassen, die für sich selbst nicht ausführbar sind, können spontan ausprobiert werden. Nach dem Programmieren einer neuen Methode kann sie ohne zusätzlichen Aufwand sofort mit Tja ausgeführt werden. Da Tja keinen Quellcode der auszuprobierenden Klassen benötigt, können so auch spielerisch fremde Klassen "erforscht" werden, wie z.B. die Java-Standardbibliotheken.

Regressionstests: Mit tja einmal aufgezeichnete Tests können jederzeit wieder ablaufen gelassen werden, um sicherzustellen, dass bereits implementierte, korrekte Funktionalität durch Weiterentwicklung oder Beheben von Fehlern nicht zerstört wurde.

Vergleichende Tests: tja kann auch genutzt werden, um verschiedene Implementierungen einer Schnittstelle zu vergleichen. Anwendung findet sich zum Beispiel in der Lehre bei der Korrektur von Programmierübungen.

Tja wurde am **Institut für Informatik der Freien Universität Berlin** im Rahmen einer Studienarbeit von

...von Miriam Busch

Internet