Programmeren – Blok B

http://www.win.tue.nl/~wstomv/edu/2ip05/

College 9

Tom Verhoeff

Technische Universiteit Eindhoven
Faculteit Wiskunde en Informatica
Software Engineering & Technology

Opmerkingen aan T.Verhoeff@TUE.NL

Onderwerpen

- Specificaties (opnieuw)
- Refactoring
- Systematisch testen van programma's (een begin)

Specificaties

```
1 procedure P ( const c: T1; out v: T2; var x: T3; a: T4 );
2   { pre: predikaat dat aanname vooraf vastlegt
3     post: predikaat dat effect na afloop vastlegt }
4
5 function f ( const c: T1; a: T4 ): T0;
6   { pre: predikaat dat aanname vooraf vastlegt
7     ret: formule die resultaat na afloop vastlegt }
```

Predikaat is geen actie, dus niet iets als $v := a$

Formule is ook geen actie, maar een uitdrukking van het juiste type

Vastleggen: hoeft niet eenduidig, er kan nog vrijheid zijn

Specificaties

Juiste vorm (predikaten, formules)

Niet te zwak, niet te sterk

Zo formeel mogelijk

Kennis en notatie van Logica en Verzamelingen toepassen

Eigen notatie eventueel uitleggen

Specificatie voorbeeld: Sorteren

```
1  const
2    MinIndex = 1;
3    MaxIndex = 1000;
4
5  type
6    TIndex = MinIndex .. MaxIndex;
7    TGetalRij = array [ TIndex ] of Integer;
8
9  procedure SorteerGetalRij ( var gr: TGetalRij );
10   { Sorteer elementen van gr oplopend. }
11   { pre:  ...
12     post: ...
13   }
```


Specificatie voorbeeld (slecht/beter): Sorteren

```
1  procedure SorteerGetalRij ( var gr: TGetalRij );
2    { 1. pre:  alle waarden verschillen
3         post: gr wordt oplopend gesorteerd }
4
5    { 2. pre:  True
6         post: (A i, j: MinIndex <= i < j <= MaxIndex: gr[i] < gr[j]) }
7
8    { 3. pre:  True
9         post: Set ( gr ) = Set ( old gr )
10          and (A i, j: MinIndex <= i < j <= MaxIndex: gr[i] < gr[j]) }
11
12   { 4. pre:  (A i, j: MinIndex <= i < j <= MaxIndex: gr[i] <> gr[j])
13        post: Set ( gr ) = Set ( old gr )
14          and (A i, j: MinIndex <= i < j <= MaxIndex: gr[i] < gr[j]) }
15
16   { where Set ( gr ) = [ n | (E i: i in TIndex: gr[i] = n) ] }
```


Specificatie voorbeeld: toelichting

1. pre: Welke waarden? Die in gr. Maak dat dan expliciet.
   post: Is zo geen predikaat (maar een actie).

2. post: Is te zwak, want `gr[i] = i` voldoet, ongeacht `old g`.

3. pre/post is problematisch: wat als een waarde vaker vóórkomt?
   Dan kan postconditie niet gerealiseerd worden.

4. Dit ziet er aardig uit.

   Let wel dat het wat anders specificeert dan

```
1   { 5. pre:  True
2        post: Set ( gr ) = Set ( old gr )
3          and (A i, j: MinIndex <= i < j <= MaxIndex: gr[i] <= gr[j]) }
```


Refactoring

Refactoring is het aanpassen van een bestaand programma onder behoud van functionaliteit. Bijvoorbeeld:

- Variabele een andere naam geven

  Lazarus kan hierbij assisteren: rechts-klik op naam en kies Refactoring ▷ Rename Identifier

- Letterlijke constante (Eng.: literal) een naam geven

- Type uitdrukking een naam geven

- Groep opdrachten afkorten tot aanroep van een nieuwe procedure

Refactoring: Constante benoemen

Vóór:

```
1 type
2   Speler = 0 .. 5; { speler 0 = niemand }
```

Na:

```
1 const
2   NSpelers  =  5; { aantal spelers, 1 <= NSpelers }
3   Niemand   =  0; { als er geen winnaar is }
4
5 type
6   Speler = Niemand .. NSpelers;
```

Ook relevante andere voorkomens vervangen door de nieuwe naam.

Refactoring: Type benoemen

Vóór:

```
1 var
2   winst: array [ Speler ] of Cardinal; { winstellingen (uitvoer) }
3     { winst[i] = aantal keer dat speler i wint }
```

Na:

```
1 type
2   WinstTelling = array [ Speler ] of Cardinal; { telling per speler }
3
4 var
5   winst: WinstTelling; { winsttellingen (uitvoer) }
6     { winst[i] = aantal keer dat speler i wint }
```

Ook relevante andere voorkomens vervangen door de nieuwe naam.

Refactoring: Procedure extraheren

Groep opdrachten S vervangen door aanroep van nieuwe procedure met S als implementatie. Lazarus kan hierbij *een beetje* assisteren.

Zorgen bij deze vorm van refactoring:

- welke parameters
- welke types (N.B. zo'n type moet een naam hebben)
- welke soort parameters (**const**, **out**, **var**, value)
- welke lokale variabelen
- welke specificatie (contract met pre/post/ret)

Refactoring: Vóór Extract Procedure (handmatig)

```
1 var
2   winst: WinstTelling; { winsttellingen (uitvoer) }
3     { winst[i] = aantal keer dat speler i wint }
4   i: Speler; { om de spelers te doorlopen }
5
6 ...
7
8 ; for i := 0 to NSpelers do begin
9     winst [ i ] := 0
10  end { for i }
```

Refactoring: Na Extract Procedure (handmatig)

```
1  procedure InitWinstTelling ( out wt: WinstTelling );
2    { pre: True; post: (A i: i in speler: wt[i] = 0) }
3  var
4    i: Speler; { om de spelers te doorlopen }
5  begin
6
7    for i := 0 to NSpelers do begin
8      wt [ i ] := 0
9    end { for i }
10
11 end; { InitWinstTelling }
12
13 var
14   winst: WinstTelling; { winsttellingen (uitvoer) }
15     { winst[i] = aantal keer dat speler i wint }
16 ...
17
18 ; InitWinstTelling ( winst )
```


Anonymous Quotes

To err is human. To foul things really up takes a computer.

Computers allow us to make more mistakes faster than ever before.

Voorbeeld van een fout bij de toets 2XP05

```
36   while not Eof(bestand) do begin
37     readln ( bestand, regel );
38     frasewaarde := 0;
39
40     for teller:=1 to Length(regel) do begin
41       if (regel[teller] <> ' ') and (regel[teller] <> '-') then begin
42         frasewaarde := frasewaarde + ord(regel[teller]) - ord('a') + 1;
43       end;
44       if frasewaarde = 100 then aantal100 := aantal100 + 1;
45       if frasewaarde > maxfrase then maxfrase := frasewaarde;
46     end; { for teller }
47
48     writeln ( nieuwbestand, frasewaarde : 3, ' ', regel );
49   end;
```

Wat is de fout? Hoe vind je zo'n fout? Hoe voorkom je zo'n fout?

Voorbeeld van een fout bij de toets 2XP05

Waardeberekening in routine onderbrengen had geholpen:

```
1  procedure WaardeerFrase ( const frase: String; out waarde: Cardinal );
2    { pre: frase bevat alleen maar kleine letters, spaties en koppeltekens
3      post: waarde = Kabalah waarde van frase }
4    var
5      i: Integer; { doorloopt frase }
6    begin
7      waarde := 0
8
9    ; for i := 1 to Length ( frase ) do begin
10       if frase [ i ] in [ 'a' .. 'z' ] then begin
11         Inc ( waarde, ord( frase[i] ) - ord ( 'a' ) + 1
12       end { if }
13     end { for i }
14
15   end; { WaardeerFrase }
```

Voorbeeld van een fout bij de toets 2XP05

```
1   while not Eof(bestand) do begin
2     readln ( inbestand, regel )
3   ; WaardeerFrase ( regel, frasewaarde )
4
5   ; if frasewaarde = 100 then Inc ( aantalGezocht )
6   ; if frasewaarde > maxwaarde then maxwaarde := frasewaarde
7
8     writeln ( uitbestand, frasewaarde : 3, ' ', regel )
9   end { while }
```

Nu is de kans veel kleiner dat de **if**-opdrachten binnen de **for**-lus verdwalen.

Britcher: The Limits of Software

"Programmers will always make errors. No advance in formal [methods] will . . . prevail over human fallibility.

[T]here are two approaches to software errors:

- one accepts them as inevitable and steers work toward removing faults that errors produce;
- the other ignores errors, the resulting faults, and the failures they may cause, and replaces testing, discovery, and repair with legal and business maneuvers."

Robert N. Britcher. *The Limits of Software*. Addison-Wesley, 1999.

Terminology

Failure = When a product in use actually fails its requirements.

E.g. incorrect behavior of a program.

Fault (defect, bug) = A mistake in a product.

E.g. missing assignment statement, or incorrect comment.

Mistake = Human action resulting in a fault.

E.g. overlooking a special case, or mistyping an identifier.

Mistake may result in a **Fault** may result in a **Failure**

Failure is caused by a **Fault** is caused by a **Mistake**

Testing in Software Life Cycle

Traditional place:

Describe – specify – design – construct – test – use – maintain

Better:

Testing (in a broad sense) is done *throughout* the life cycle.

Classical Testing versus Debugging

Classical Testing =

The process of executing a program with the intent of finding the presence of faults.

Debugging =

The act of fault diagnosis and correction.

Verification

Modern point of view:

Verfication =

The act of

reviewing, inspecting, testing, checking, auditing, or otherwise establishing and

documenting whether or not

items, processes, services, or documents

conform to specified requirements.

Limits of Execution-Based Testing

"Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence.

The only effective way to raise the confidence level of a program significantly is to give a convincing proof of its correctness."

E. W. Dijkstra. "The Humble Programmer", *Communications of the ACM*, **15**(10):859–866 (1972).

Some Testing Principles

- A necessary part of a test case is a definition of the expected output or result.
- Thoroughly inspect the result of each test.
- Avoid throw-away test cases unless the program is truly a throw-way program.
- Do not plan a testing effort under the tacit assumption that no faults will be found.
- Testing is an extremely creative and intellectually challenging task.