4EE11 – Project
Programmeren voor W

College 4, 2008–2009, Blok D

Tom Verhoeff, Software Engineering & Technology, TU/e

Onderwerpen

- Controleren (verificatie) van software
  - Pair programming
  - Code review (nalezen)
  - Testen (detecteren, kwaliteit meten)
  - Regressietesten (“terugval” voorkomen)
- “Debuggen” (diagnose, localisatie, reparatie)

Benodigdheden

- Requirements (eisenpakket, opdracht)
- Ontwerpbeschrijving (opgeschreven!)
- Code (leesbare programmatekst)

Ontwerp vastleggen

- Welke data types? Naam, inhoud, doel
- Welke operaties erop? Naam, parameters, types, doel
- Hoe gaat hoofdprogramma er uitzien?
- In tekstbestand beschrijven; doornemen met assistent

Voorbeeld: Graansilo

- State of single particle: floats sx, sy, vx, vy
- State of set of particles: struct with number of defined particles, and array of particle states (does the order of particles matter?)
- Read single particle state, read entire set
- Write single particle, write set (for testing)
- Write single particle as SVG, write set

Graansilo (vervolg)

- data types: Particle, Particle_set
- operaties: read_particle, read_particle_set, write_particle, write_particle_set, write_particle_SVG, write_particle_set_SVG
- Hoofdprogramma: lees particle set uit file, schrijf deze set als SVG naar file

Pair programming

- Driver (pilot): types at keyboard
- Navigator (co-pilot): reflects
- Regelmatig van rol wisselen (elk half uur)
- Dialoog (continu)
- Twee paar ogen zien meer dan één paar
- Foutpreventie: voorkomen is beter dan …
- Kennis over code verspreiden
- Nadeel (voordeel?): tempo van langzaamste

Code review

- Vóór eerste compile
- Met ontwerpbeschrijving ernaast
- Checklist hanteren
- Motivatie: compile+execute is niet geschikt voor het detecteren van alle fouten
- Kan op het scherm of met afdruk (listing)

Checklist

- Is codeerstandaard gevolgd?
- Zijn alle datatypes en operaties uit het ontwerp opgenomen?
- "Magische" constanten alleen via #define?
- Zijn precondities via assert gecontroleerd?
- Zijn alle variabelen geïnitialiseerd?
- Blijven alle indexeringen binnen de grenzen?
- Eindigen alle herhalingen (lussen)?
- Zijn functies niet langer dan ca. 50 regels?

Debuggen

- Ná detecteren van fout
- Diagnose stellen, fout localiseren, en herstellen
- (Debugger: breakpoint, step, inspectie)
- fprintf opdrachten tussenvoegen
- Conditional compilation: #ifdef TEST … #endif

Debug output

```
#include <stdio.h>

...

#ifdef TEST
  fprintf ( stderr, “count = %d\n”, pq.count );
#endif
```

Project > Project options … > Compiler > Define preprocessor symbols: TEST

TEST “uitzetten” als je geen testoutput meer wilt; #ifdef ... #endif laten staan

Test driver

- Hoofdprogramma biedt user interface voor het uitvoeren van berekeningen t.b.v. de opdracht; is minder geschikt voor testen
- Aparte test driver(s) om functionaliteit, los van het user interface, te kunnen testen; ook later nog kunnen herhalen
- Test driver roept functies aan met "slimme" invoer; dit kan hard-coded, of via inlezen uit stdin of file; en controleerbare uitvoer, naar stdout of file, of automatisch controleren

Test cases

- Black box: kies test cases op basis van de eisen, niet op basis van de code
  - grensgevallen: op, vlak vóór, vlak erover
  - elke equivalentieklasse afdekken
- White box: op basis van code, niet de eisen
  - 100% statement coverage, 100% branch coverage

Black box test cases

- Voorbeeld: sorteren van lijsten met maximaal 1000 integers
  - grensgevallen: lijstlengtes 0, 1, 2, 3, 4, 5, 999, 1000, 1001
  - equivalent: 2 1 3 en 6 4 8, wel anders zijn 1 2 3, 1 3 2, 2 3 1, 3 1 2, 3 2 1
  - wel of niet duplicaten: 1 2 1

White box test cases

- "Raak" elke opdracht en elke "vertakking"
- Niet haalbaar om elk pad te "raken"
- Bij sorteren kan het zijn dat je weet dat lijsten van lengte $\leq$ 12 apart worden afgehandeld; dan ook rond 12 testen

Coverage

```
if ( C ) {
  v = 1;
}
if ( D ) {
  w = 2;
}
else {
  w = 3;
}
```

5 (!) statements, $2 + 2$ branches, $2 * 2$ paths

Test Cases				Coverage		
1	2	3	4	Statement	Branch	Path
$\neg C, \neg D$				60%	50%	25%
C, D				80%	50%	25%
C, D	$C, \neg D$			100%	75%	50%
C, D	$\neg C, \neg D$			100%	100%	50%
C, D	$C, \neg D$	$\neg C, D$	$\neg C, \neg D$	100%	100%	100%

Built-in tests

- Er zijn fouten die niet meteen aanleiding geven tot problemen en die pas later tijdens uitvoering van het programma aan het licht komen
- De kunst is zo vlug mogelijk te constateren dat er iets mis is
- Principe: "Fail early"
- #include <assert.h>
- assert (*conditie*);

Voorbeeld met assert

```
#include <assert.h>

...

void remove_min_pq ( PriorityQueue *pq, OBJ *b )
  // pre: not is_empty ( *pq ), d.w.z. *pq is niet leeg
  // ...
{
  assert ( ! is_empty ( *pq ) );
  ...
}
...

  PriorityQueue pq;
  OBJ b;

  clear_pq ( &pq );
  remove_min_pq ( &pq, &b );
```

Als preconditie niet geldt, dan volgt foutmelding:

```
Assertion failed: (! is_empty_pq ( *pq )), function remove_min_pq, file
pq.c, line 51.
Abort trap
```

Adviezen

- Project > Project options … > Compiler > Warnings: Level 2
- Leg aan testcode dezelfde kwalititeitseisen op als aan de productcode
- Bewaar “known correct” testuitvoer
- Herhaal alle tests bij wijzigingen (regressie)
- Test Driven Development (TDD):
  Ontwikkel test cases en test driver vóór coderen van functionaliteit; dit helpt bij het voorkomen van fouten in de implementatie

Afwegingen bij functies

- Functie of procedure (void return type)?
  - Functie als nodig in formule
  - Procedure als berekening "duur" is
- Gewone versus reference parameter
  - void copy (int a, int *b); // post: *b == a
    { *b = a; }
  - p (x, &y);
  - resultaat komt in y (elders gedeclareerd)

Procedures zonder argument

- void p (void);
- Oude stijl C: void p ( ); mag niet meer
- echte functies zonder argumenten zijn verdacht: float f (void);

Help

- Studentassistenten: spreekuur, email
- Technisch adviseurs: ook per email
- Zie projectopdracht in peach voor emailadressen
- FAQ op webpagina