Programmeertechnieken
Week 7

Tim Cocx, Kristian Rietveld

http://liacs.leidenuniv.nl/~rietveldkfd/courses/pt2016/

Universiteit Leiden
The Netherlands

"Confidence in Code"

- "Ons programma / onze module is af, maar hoe weten we nu of alles werkt?"
- Ofwel: hoe krijgen we het vertrouwen dat de code naar behoren functioneert?
- Vraag:
  - Hoe doen jullie dit nu?
  - Heeft iemand concrete ideeën om dit voor elkaar te krijgen?

Unit testing

- Iedereen heeft waarschijnlijk wel eens een aantal aanroepen naar een net-geschreven functie in `main` gezet, de code gecompileerd en daarna gedraaid.
- Waarom? Om te kijken of de functie doet wat we verwachten.
- Als de functie dan werkte, werden dit soort testaanroepen vaak weer weggehaald, want deze "waren niet meer nodig".
- Hoe pakken we dit op een beter gestructureerde manier aan?

Unit testing (2)

- Het antwoord is *Unit Testing*.
- We schrijven kleine testjes die verschillende aspecten van functies testen.
- De tests worden vaak per klasse of module georganiseerd.
- Er kan dan worden nagegaan of een klasse of module naar behoren functioneert.

Unit testing (3)

- Tegenhanger: *functionele tests*.

- Dit zijn tests die het gehele softwarepakket testen, dus de combinatie van alle klassen & modules.

  - Werken alle klassen goed samen? (=integratietest)

Waarom testen?

- Er zijn vele redenen te noemen waarom het belangrijk is om unit tests te schrijven.
- Ten eerste: vaststellen of de klasse naar behoren werkt / werkt volgens de specificatie.
- Belangrijk bij het modulair opbouwen van software:
  - je wilt zeker weten dat een bepaalde module goed werkt, voordat je verder gaat met het implementeren van de volgende module.

Waarom testen? (2)

- Bugs voor zijn.
- Door van te voren goede tests te schrijven, kun je vaak al bugs tegenkomen waar je anders in de toekomst tegenaan zou lopen. Dit proces heet *test-driven development*.
- Dit scheelt een hoop werk! In de toekomst zou er vaak een lange debugsessie aan vooraf gaan voordat je de bug vindt.

Waarom testen? (3)

- Er voor zorgen dat de code op dezelfde manier blijft werken.
- Op deze manier kun je veranderingen doorvoeren aan een klasse en er zeker van zijn dat de klasse naar behoren blijft werken.
  - "*Refactoring*": de architectuur van de code verbeteren.
  - Performance improvements, bijvoorbeeld een algoritme vervangen om de software sneller te maken.
- Als er door een verandering iets niet meer werkt, spreken we over een "*regressie*".

Waarom testen? (4)

- Documentatie van bugs.
- Als je een bug hebt gevonden is het een goed gebruik om eerst een unit test te schrijven die de bug veroorzaakt.
- Volgens verhelp je de bug.
- Omdat de unit test blijft bestaan, heb je nu een toekomstbestendige check dat de bug niet meer zal terugkeren.

Voorbeeld

- Wat moeten we nu allemaal testen? Hoe schrijven we een test?
- Laten we beginnen met een voorbeeld: een stapel (stack) gebaseerd op een enkel gelinkte lijst.

```
template <typename T>
class MyStack
{
  ...
  void push(T item);
  T peek(void) const;
  T pop(void);
  void clear(void);
  bool is_empty(void) const;
  std::size_t size(void) const;
  ...
};
```

Voorbeeld (2)

- Wat tests zouden jullie opstellen voor de volgende methode?

```
void push(T item)
{
  if (is_empty())
    head = tail = new Link(item);
  else
    {
      tail->next = new Link(item);
      tail = tail->next;
    }
}
```

Assertions

- Unit tests worden geschreven door een reeks van operaties uit te voeren en tussentijds te controleren of de staat van het object is zoals je zou verwachten.
- De controles worden opgenomen door "test assertions" te schrijven.
- Bijvoorbeeld (verschilt per 'taal'):

```
assert(stack.size() == 0);
assert(stack.is_empty() == true);
```

Voorbeeld (3)

- We noteren dat als volgt (in pseudocode, werkend voorbeeld later):

```
void test_push(void)
{
  MyStack<int> stack;

  assert(stack.is_empty() == true);

  stack.push(10);
  assert(stack.is_empty() == false);
  assert(stack.size() == 1);
  assert(stack.peek() == 10);
  stack.push(20);
  assert(stack.size() == 2);
  assert(stack.peek() == 20);
  stack.push(30);
  assert(stack.size() == 3);
  assert(stack.peek() == 30);
}
```

Voorbeeld (4)

- Wat zouden jullie testen voor de volgende methode?
  - Dit keer zonder eerst de implementatie te zien.

```
template <typename T>
class MyStack
{
  ...
  void clear(void);
  ...
};
```

Voorbeeld (5)

- Bijvoorbeeld, test de gevallen:
  - gegeven een lege stack.
  - gegeven een stack met 1 element
  - gegeven een stack met 2 elementen
  - gegeven een stack met 10 elementen

Code coverage

- "Code coverage" is een maat voor de hoeveelheid source code die door unit tests wordt gedekt.
- Er zijn verschillende grondigheidsvarianten van coverage.
- Bij het uitvoeren van de unit tests houden unit test frameworks vaak bij welke regels source code wel en niet worden uitgevoerd.
  - 100% code coverage is dan vaak: elke regel source code wordt door ten minste 1 test uitgevoerd.

Code coverage varianten

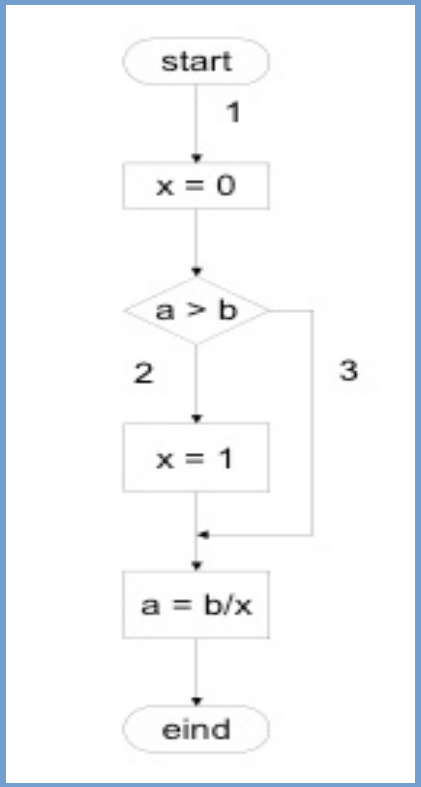
Variant	Wat is het?
Statement Coverage	Alle regels code worden 'geraakt'
Condition Coverage	Alle conditie uitkomsten worden getest
Decision Coverage	Alle beslissingen worden getest
(Modified) Condition / Decision Coverage	Alle conditie uitkomsten en beslissingen worden getest (waarbij condities invloed hebben op beslissing)
Decision-table Test	Alle combinaties van conditieuitkomsten worden getest
Path Coverage	Alle paden worden getest
Path Coverage Measure 2	Alle padcombinaties worden getest

Code coverage varianten

Variant	Wat is het?
Statement Coverage	Alle regels code worden 'geraakt'
Condition Coverage	Alle conditie uitkomsten worden getest
Decision Coverage	Alle beslissingen worden getest
(Modified) Condition / Decision Coverage	Alle conditie uitkomsten en beslissingen worden getest (waarbij condities invloed hebben op beslissing)
Decision-table Test	Alle combinaties van conditieuitkomsten worden getest
Path Coverage	Alle paden worden getest
Path Coverage Measure 2	Alle padcombinaties worden getest

Statement Coverage

- Elke regel code wordt door een test geraakt.
  - Probleem: 100% coverage zorgt er niet perse voor dat 'pad' 3 ook getest wordt...



Condition Coverage

- Elke conditie-uitkomst wordt een keer getest.
  - Maar niet elke uitkomst (beslissing).

```
numberOfBooks< 5 and openFine < 25
```

logical testcases	number < 5	fine < 25	result
Case 1	1	0	0 (no lease)
Case 2	0	1	0 (no lease)

Decision Coverage

- Elke beslissing wordt een keer getest.
  - Maar niet elke conditie.

```
numberOfBooks< 5 and openFine < 25
```

logical testcases	number < 5	fine < 25	result
Case 1	1	1	1(lease)
Case 2	0	1	0 (no lease)

Condition /Decision Coverage

- Elke conditie uitkomst & beslissing wordt een keer getest.

```
numberOfBooks< 5 and openFine < 25
```

logical testcases	number < 5	fine < 25	result
Case 1	1	1	1(lease)
Case 2	0	0	0 (no lease)

Modified Condition /Decision Coverage

- Elke conditie is minimaal 1 keer bepalend voor de beslissing.

```
numberOfBooks< 5 and openFine < 25
```

logical testcases	number < 5	fine < 25	result
Case 1	0	1	0 (no lease)
Case 2	1	0	0 (no lease
Case 3	1	1	1 (lease)

Decision-table test

- Alle combinaties van conditie-uitkomsten worden getest.
  - Volledige waarheidstabel

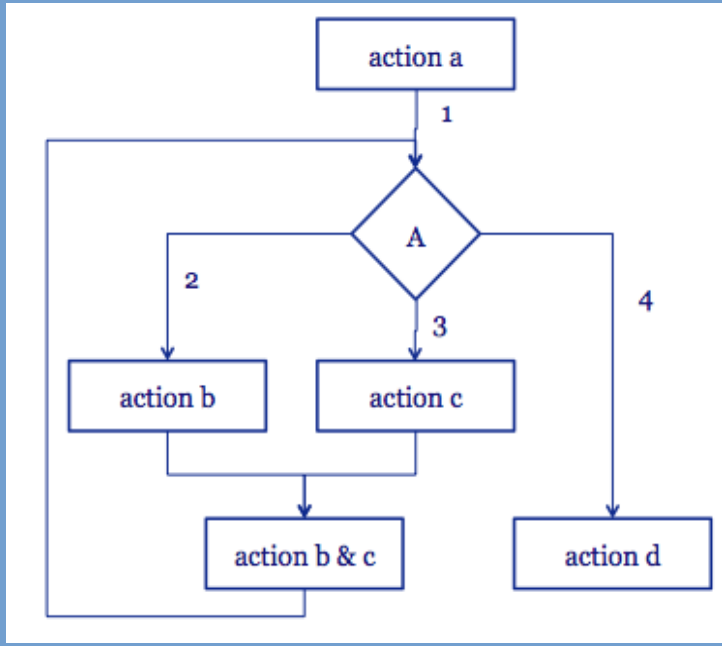
```
numberOfBooks< 5 and openFine < 25
```

logical testcases	number < 5	fine < 25	result
Case 1	0	0	0 (no lease
Case 2	0	1	0 (no lease)
Case 3	1	0	0 (no lease
Case 4	1	1	1 (lease)

Path coverage

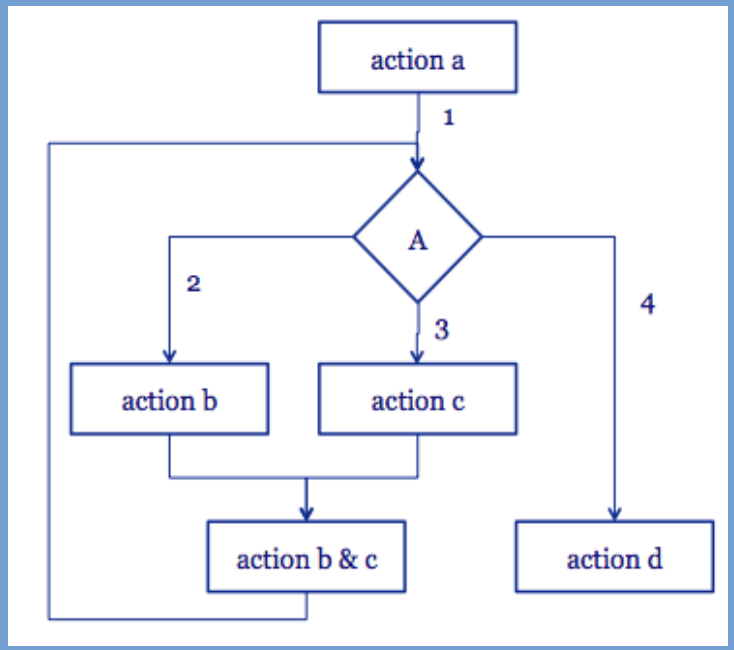
- Alle paden worden een keer gevolgd.

```
action a
while (A)
        if (A)
                action b
        else
                action c
        action b & action c
action d
```



Path coverage; test measures

- Test-measure 1:
  - Test alle paden: 1,2,3,4
  - Laat de loops 0 keer doen
  - Laat de loops >=1 keer doen
- Test-measure 2:
  - Test alle padcombinaties: 12,13,14,22,23,24,32,33,34
  - Laat de loops 0 keer doen
  - Laat de loops 1 keer doen
  - Laat de loop >=2 keer doen



Hoeveel tests?

- Welke test coverage moeten we hanteren? Hoeveel tests moeten we schrijven?
  - Dit hangt af van hoe belangrijk kwaliteit is en hoeveel tijd/budget je dus kan investeren in testen.
  - Minimaal is:
    - 100% statement coverage
    - Condition coverage
    - Boundaries
  - Adviezen:
    - Schrijf ook "randomized tests".
    - Veel if's: decision-table test
    - Complexe loops (algoritme): path coverage

Code coverage (2)

- Demo: voorbeeld code coverage report MyStack.

Boundary conditions

- Bugs zijn vaak te vinden in de zogenaamde "boundary conditions" van een programma.
- Dit zijn de uitzonderingen op de generieke code, of de randgevallen.
- Identificeer boundary conditions in code en schrijf hiervoor aparte tests.
- Bijvoorbeeld bij de stack:
  - clear() op een lege stack.
  - push() op een lege stack is anders geïmplementeerd dan push() op een stack waarin al een element zit.
  - enz.

Randomized testing

- Unit tests werken vaak op specifieke data en geconstrueerde voorbeelden.
- Ook al zijn alle boundary cases afgedekt en is de code coverage 100%, dan nog kunnen er bugs in de code zitten.
- Oplossing: testen met random inputs.

Randomized testing (2)

- Om random te kunnen testen, moeten we ervoor zorgen dat:
  - we een manier hebben om vele verschillende inputs te genereren.
  - we generieke asserties hebben die werken op alle inputs.

Voorbeeld

- Voorbeeld: gegeven een sorteeralgoritme dat werkt op arrays van integers.
- We kunnen een functie schrijven om vele verschillende arrays te genereren met willekeurige inhoud:
  - Kleine arrays, grote arrays.
  - Voornamelijk kleine getallen, voornamelijk grote getallen.
  - Zowel positieve als negatieve getallen.
  - Getallen die meerdere keren in de array voorkomen.
  - enz.

Voorbeeld (2)

- Wat moet er gelden voor de array na het uitvoeren van het sorteeralgoritme?
  - Precies dezelfde getallen uit de input array moeten ook in de output array zitten.
  - De getallen moeten in oplopende volgorde in de output array staan.

Voorbeeld (3)

- Wat zouden we kunnen verzinnen voor ons "MyStack" voorbeeld?

Randomized testing (3)

- Hoe vaak moeten de tests worden gedraaid?
  - We kunnen niet alle mogelijke inputs genereren.
- Bepaal een vast aantal verschillende inputs dat je wilt genereren en testen.
  - Bijvoorbeeld 100 of 1000.
  - De test moet wel in redelijk tijd kunnen worden uitgevoerd.

Andere adviezen

- Schrijf eerst tests, begin daarna de implementatie, oftewel; gebruik *test-driven development*.
- Als je een bug hebt gevonden, schrijf eerst een test die de bug produceert, en verhelp daarna de bug.
- Tests moeten volledig geautomatiseerd zijn en moeten automatisch nagaan of de resultaten van het programma correct zijn.

Andere adviezen (2)

- Blijf testen draaien tijdens het implementeren.
  - Gevolg: testen moeten snel zijn.
  - Maak voor langdurige tests een aparte categorie.
- In het geval van C, C++: draai de unit tests ook onder valgrind!
  - Op deze manier wordt alle code die door de tests wordt uitgevoerd ook gecontroleerd op memory errors!

Test Frameworks

- Er bestaan al vele "test frameworks": raamwerken voor het schrijven van tests.
- Deze raamwerken bevatten functies om automatisch alle tests uit te voeren en fouten te rapporteren.
- Daarnaast worden de "test assertions" ook door het raamwerk gedefinieerd.

Test Frameworks (2)

- Test frameworks zijn vaak programmeertaalgebonden.
- We bekijken tijdens het hoorcollege "Boost Test" voor C++ codes en gebruiken dit ook in de huiswerkopgave.
- Voor Python, Java, enzovoort bestaan er weer andere test frameworks. Wel zijn deze op dezelfde ideeën gebaseerd.

Boost.Test

- Boost.Test is een unit test framework dat deel uitmaakt van de Boost libraries.
- Het kan automatisch een main functie genereren die alle tests uitvoert.
- Het bevat verschillende test assertions.
- Het idee is dat je naast je echte programma aparte "test executables" compileert waarin alle test code zit.

Boost.Test (2)

- We maken een apart bestand "mystacktest.cc" met daarin:

```
#include "mystack.h"
#define BOOST_TEST_MODULE MyStackTest
#include <boost/test/unit_test.hpp>

BOOST_AUTO_TEST_SUITE(mystack_test)

// hier komen de unit tests ...

BOOST_AUTO_TEST_SUITE_END()
```

Boost.Test (3)

- Een eerste test:

```
BOOST_AUTO_TEST_CASE(mystack_push)
{
  MyStack<int> stack;

  stack.push(10);
  BOOST_CHECK_EQUAL(stack.peek(), 10);
  BOOST_CHECK_EQUAL(stack.size(), 1UL);
  stack.push(20);
  BOOST_CHECK_EQUAL(stack.peek(), 20);
  BOOST_CHECK_EQUAL(stack.size(), 2UL);
  stack.push(30);
  BOOST_CHECK_EQUAL(stack.peek(), 30);
  BOOST_CHECK_EQUAL(stack.size(), 3UL);
}
```

Boost.Test (4)

- Test assertions:
  - `BOOST_CHECK_EQUAL(a, b)`
  - `BOOST_CHECK_NE(a, b)`
  - `BOOST_CHECK(expr)`
  - `BOOST_CHECK_THROW(expr, exception_type)`
- In plaats van `CHECK` mag je ook `REQUIRE` schrijven. In dit geval zal het programma direct stoppen als niet aan de assertie wordt voldaan.
- Voor een compleet overzicht: http://www.boost.org/doc/libs/1_42_0/libs/test/doc/html/utf/testing-tools/reference.html

Boost.Test (5)

- Om te compileren is het volgende belangrijk:

```
-DBOOST_TEST_DYN_LINK
-lboost_unit_test_framework
```

- Op deze manier wordt Boost Test automatisch dynamisch gelinkt.
- Dus bijvoorbeeld:

```
gcc -Wall -DBOOST_TEST_DYN_LINK \
  -o mystacktest mystacktest.cc \
  -lboost_unit_test_framework
```

Boost.Test (6)

- Het resultaat is een executable `mystacktest`.
- Deze kun je gewoon opstarten en draait automatisch alle gedefinieerde tests.
- Alle fouten worden gerapporteerd.

Code coverage in C++

- Om statistieken over code coverage van C++ te krijgen maken we gebruik van "gcov" en "lcov".
- Tijdens het draaien van alle tests wordt er bijgehouden welke regels code er worden uitgevoerd.
- Op die manier kunnen we dan afleiden welke regels code wel en niet door de tests zijn uitgevoerd. Dit geeft ons de "test coverage".

Code coverage (2)

- We moeten ons testprogramma met speciale compileropties compileren:

```
-g -O0 -coverage
```

- Wanneer we de resulterende executable uitvoeren, wordt er nu automatisch "coverage informatie" weggeschreven.

Code coverage (3)

- De informatie die nu wordt weggeschreven is niet eenvoudig te interpreteren.
- We zagen eerder in het college al een voorbeeld van HTML uitvoer. We gebruiken een aparte tool om dit te genereren: "lcov".
- Voorbeeld:

```
lcov --capture -i --directory . -o coverage.baseline
./mystacktest
lcov --capture --directory . --output-file coverage.out
lcov -a coverage.baseline -a coverage.out -o coverage.info
genhtml coverage.info --output-directory html
# in de html/ directory is nu de output te vinden
```

Testen voor memory errors

- Vooral voor C en C++ code is het belangrijk om ook te testen voor memory errors.
- Het idee is dat de unit tests een zeer hoge code coverage behalen.
- Dus als we de unit tests onder een memory debugger uitvoeren, testen we ook een zeer groot deel van de code voor memory errors.
  - Volautomatisch!
- Je kunt `mystacktest` gewoon draaien met behulp van bijvoorbeeld "valgrind".

Huiswerk

- **Opdracht**: gegeven een C++ implementatie van "MyStack", schrijf unit tests die alle aspecten van deze klasse testen.
- Op de website is de code voor MyStack te vinden, inclusief "beginnetje" voor Boost.Test.
- We leveren ook een Makefile mee om de unit test te compileren, te runnen en code coverage statistieken de genereren.
- Er zou nog een bugje in MyStack kunnen zitten ...
- **Deadline: 1 april**

Literatuur

- Refactoring: Improving the Design of Existing Code. Chapter 4: Building Tests. Fowler. 2000. Addison Wesley.
- Beautiful Code. Chapter 7: Beautiful Tests. Oram & Wilson. 2007. O'Reilly.

Volgende weken

- Volgende week: geen college.
  - Extra werkcollege?
- Over twee weken: Introduction to Java.