Testen

Aanvullende nota's inleiding software engineering

Filip Van Rysselberghe and Andy Zaidman

Academiejaar 2003-2004
Versie 1.0

1 Inleiding

Software schrijven houdt meer in dan code intikken in een editor en compileren. Software moet ook grondig getest worden: enkel zo kan je nagaan of de software die je hebt geschreven ook daadwerkelijk doet wat jij – of moeten we zeggen de klant – wil.
In het geval van kleine programma's kunnen deze tests vaak ad-hoc gebeuren, zonder dat er echt een strategie achter zit. Hoe groter een softwareproject wordt en hoe meer mensen er bij betrokken zijn, hoe moeilijker het wordt om met deze ad hoc tests het programma degelijk te testen. Er is dus nood aan een **test-strategie**.

> Program testing can be used to show the presence of bugs, but never to show their absence (Dijkstra).

2 Waarom testen?

2.1 Product-standpunt

Software maakt meer en meer deel uit van ons dagelijks leven. We vertrouwen erop, vaak voor niet-kritische handelingen, maar meer en meer ook voor extreem kritische zaken. Een bank bijvoorbeeld waarvan het softwaresysteem uitvalt voor meer dan 2 a 3 dagen is tegenwoordig zo goed als failliet. Niet alleen door de winsten die ze mislopen door het mislopen van transacties, maar zeker ook door een verlies aan reputatie.
De situatie van de bank is een goed voorbeeld van een situatie waarin het uitwerken van een geschikte test-strategie belangrijk is voor het *product*, namelijk het lopende programma. De eindgebruiker wil immers een stuk software dat voldoet aan een aantal eigenschappen:

- het programma moet **correct** zijn, m.a.w. het moet doen wat er door de opdrachtgever of bedenker van het programma is opgetekend in de specificatie
- de software moet **betrouwbaar** zijn, m.a.w. het programma mag niet crashen.
- de software moet tegen een stootje kunnen als de gebruiker er op de verkeerde manier mee omgaat. Het invullen van een numerieke waarde i.p.v. de de alfabetische waardie die het programma verwacht, moet op de juiste manier worden opgevangen. Deze laatste eigenschap valt onder het begrip **robuustheid**.

2.2 Proces-standpunt

Testen van software draagt ook bij tot een verbetering van het software proces. Met dit laatste bedoelen we de weg die het team van programmeurs aflegt om tot het afgewerkte product te komen. Stel je maar even voor dat je zelf een stuk software schrijft en dit zonder enige uitleg doorgeeft aan een van je collega's... Vaak kunnen de testen er voor zorgen dat die collega toch een goed zicht heeft op hoe er met jouw stuk code moet omgegaan worden.
Een iets andere situatie waarin het handig is dat jouw stuk code voorzien is van een test is als jij of een collega iets wil aanpassen aan de code. Door de bijgeleverde test uit te voeren kan je (collega) meteen zien of de aanpassingen het programma niet uit balans brengen...

3 Wat is testen?

> Testen is de activiteit van het uitvoeren van een (deel van een) programma met als doel het vinden van fouten.

Een goede test is een test die fouten vindt. In die zin is testen een **destructief** proces, je wil het programma immers kapot maken.

Testen is bovendien een volwaardige activiteit die deel uitmaakt van het ontwikkelingsproces: het is zelfs niet abnormaal dat testen 30 tot 40% van de totale ontwikkelingstijd in beslag neemt. Bovendien is het volume van de testcode vaak *groter* dan dat van de code die je wil testen.

4 Verschillende soorten testen

De manier waarop je test verschilt natuurlijk van het moment waarop je wil testen. Als je bijvoorbeeld een klein stukje wil testen, zal je test er natuurlijk anders uitzien dan wanneer je een heel groot stuk software wil testen. Daarom zijn er verschillende **test-technieken**:

- Een **unit test** test op die manier 1 (of een heel klein aantal) klassen.
- Een **integration test** gebruik je op het moment dat je 2 of meer kleine delen wil samenvoegen.
- Een **regression test** daarentegen moet op een *geautomatiseerde* manier het hele programma testen.

Een **test-strategie** is weten *wanneer* in het ontwikkelingsproces je het beste *welke* test-techniek toepast. Onderstaande secties proberen daar een antwoord op te bieden.

5 Unit testing

Unit testing kan je eigenlijk ook beschouwen als ”testen in het klein”. Je wil immers elke *unit* of module testen. Hiervoor bestaan 2 aanpakken, nl. **white-box testing** en **black-box testing**.

5.1 Black-box testing

Een stuk software testen als een black-box betekent dat je de software test zonder kennis over de exacte interne werking van de module. Testen worden opgesteld en uitgevoerd *enkel* door enkel terug te vallen op de specificatie en dus niet op de code. Black-box testen wordt ook wel *functioneel testen* genoemd.

Een voorbeeld hiervan zou een module `Stack` kunnen zijn. Wat betreft de implementatie weet je totaal niet of ze een enkel gelinkte lijst, een dubbel gelinkte lijst dan wel een array-constructie gebruiken. Je weet enkel wat de voornaamste operaties `push`, `pop`, `peek` en `isEmpty` doen. Op basis van deze kennis van het uitwendig gedrag van de module `Stack` schrijf je een klein programma – een unit test – die nakijkt of het gedrag dat je van een stak verwacht ook daadwerkelijk uitkomt als je deze implementatie gebruikt.

```
MODULE Stack;

    Stack* = POINTER TO StackDesc;
    StackDesc = RECORD
        (* velden *)
    END;

    PROCEDURE (this: Stack) push*(element: BasicStackElement);
        (* implementatie *)
    PROCEDURE (this: Stack) pop*(): BasicStackElement;
        (* implementatie *)
    PROCEDURE (this: Stack) peek*(): BasicStackElement;
        (* implementatie *)
    PROCEDURE (this: Stack) isEmpty*(): BOOLEAN;
        (* implementatie *)
```

```
BEGIN END Stack.

%%%%%

MODULE Stack_test;

IMPORT Stack, BasicStackElement;

    PROCEDURE (this: Stack_test) unitTest*;

        VAR
            s: Stack.Stack;
            a, b, temp: BasicStackElement;
    BEGIN
        NEW(s);
        NEW(a);
        NEW(b);
        s.push(a);
        s.push(b);
        if(~s.peek() = b)
            Out.println("Fout ...");
        temp := s.pop();
        if(~s.peek() = a)
            Out.println("Fout ...");
        temp := s.pop();
        if(~s.isEmpty)
            Out.println("Fout ...");
    END unitTest;

BEGIN
END Stack_test.
```

5.2 White-box testing

White-box testen wordt ook wel *structureel testen* genoemd: er wordt expliciet naar de interne structuur van het programma gekeken om de testen en de test-data op te stellen.
Er wordt bijvoorbeeld gekeken naar lusstructuren en conditionele structuren om te komen tot testen die rekening houden met randgevallen van de condities voor deze structuren. Een naam die je in deze context dan ook vaak hoort is *basic path testing*: alle mogelijke controlelijnen in een programma nagaan en daaruit de testen opbouwen.

In de praktijk wordt er voor unit testen een combinatie van zowel white-box als black-box technieken gebruikt om te komen tot de testen *en* de test-data voor een bepaalde module.

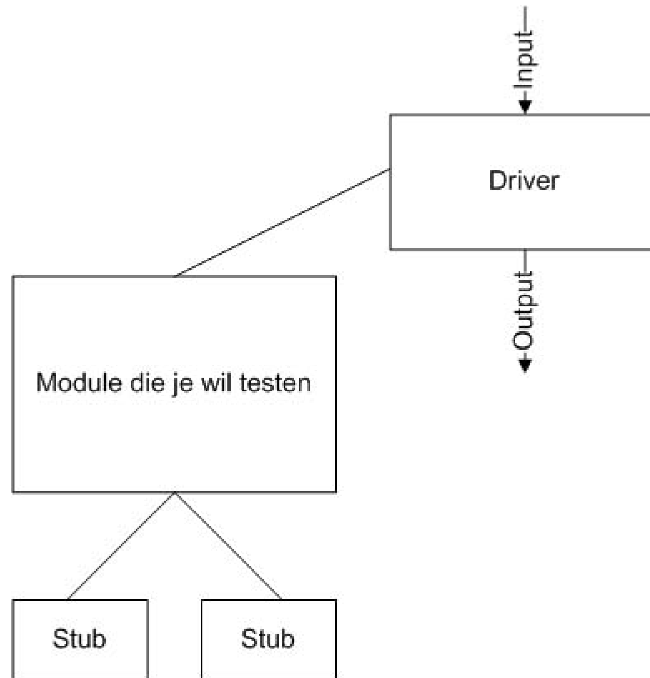
5.3 Hoe maak ik een unit test?

Een module op zich testen lijkt soms ingewikkelder dan het is... Om de module of klasse te testen schrijven we in elk geval een *main programma* of een **test-driver** (zie ook Figuur 1, m.a.w. een nieuw Oberon commando. Bijvoorbeeld:

```
Module_test.Test parameter1 parameter2~
```

Soms echter heeft de module die je geschreven hebt ook nog een aantal andere modules nodig om goed te kunnen werken. In bepaalde gevallen zijn deze modules nog niet beschikbaar (wegens nog niet af of niet

voldoende getest), maar willen we onze module toch al testen. In deze situatie kunnen we gebruik maken van zogenaamde **stubs**. Deze stub is een soort primitieve versie van de ”toekomstige”module en bevat dus weinig functionaliteit, maar wel net genoeg om het gedrag van de toekomstige module te simuleren.



Figuur 1: Overzicht van een unit test

5.4 Waarom belangrijk?

Unit testing is belangrijk omdat het je toelaat om kleine stukjes software te testen: je weet altijd dat eventuele fouten in het kleine stukje software zitten en je moet dus niet uren liggen zoeken naar waar de fout precies ontstaat.

6 Integration testing

Hoe goed de verschillende units of modules op zich ook getest zijn, vaak ontstaan er nog problemen als 2 of meer modules ineens moeten samenwerken. De oorzaken hiervan kunnen verschillend zijn:

- Vaak wordt software geschreven door meer dan 1 persoon. Vooraf worden meestal een aantal afspraken gemaakt over hoe de stukken software die iedere software ontwikkelaar van het team bouwt, moeten samenwerken. Hoe duidelijk deze afspraken ook zijn gezegd of opgeschreven, altijd wel onstaan er kleine misverstanden.
- Ook als jijzelf het hele ontwikkelingsteam vormt en je dus alle software zelf schrijft kunnen er soms vreemd en vooral onverwachte interacties zijn tussen 2 modules.
  Als voorbeeld kunnen we misschien de variabele parameter constructie gebruiken. In module A is een functie A gedefinieerd met een formele parameter x. Module B gebruikt deze, maar gaat er vanuit dat de actuele parameter x gewijzigd wordt, m.a.w. dat de waarde van de lokale variabele x gewijzig wordt. Echter, doordat je in module A vergeten bent de parameter x van de desbetreffende functie als VAR te definieren, zal je programma een heel ander gedrag vertonen dan je verwacht.

```
Module A:
    PROCEDURE (m: JustAClass) Init* (x: JustAnotherClass);
    BEGIN
        (* ... *)
    END Init;

Module B:
    PROCEDURE DoetVanallesMaarNietsGoed*;

        VAR
            a: JustAClass;
            x: JustAnotherClass;
    BEGIN
        NEW(a);
        NEW(x);
        x.Init(5);
        a.Init(x);
        (* Oops... nu werken we nog met oude waarde van x... *)
    END DoetVanallesMaarNietsGoed;
```

- ...

6.1 Waarom belangrijk?

Ook hier is een van de belangrijkste reden voor integration testing dat door in deze fase te testen je de zoektocht naar de bug zo beperkt mogelijk kan houden. Je weet immers heel goed in welke klassen of modules je de fout moet gaan zoeken.
Het is daarom een goed idee om **incrementeel** te werk te gaan, m.a.w. elke keer dat je 1 of 2 modules af hebt voer je een integration test uit. Een *big-bang aanpak* waarbij je alle aparte modules ineens gaat testen is geen goed idee omdat je op die manier veel minder controle hebt over de plaats waar de bug ontstaat en je dus veel meer tijd nodig hebt om de bug op te sporen!

7 Regression testing

7.1 Wat is een regression test?

Een regression test is een test die het ganse programma test. Bovendien is deze vorm van testen **volledig geautomatiseerd**. Hiermee bedoelen we dat door **1 commando** uit te voeren de gebruiker van de test weet of het programma goed werkt of dat er nog ergens een bug in zit.

Eigenlijk is een regression test een **verzameling van afzonderlijke tests**. Immers, als je het hele programma wil testen, dan is het vaak onmogelijk om 1 test te ontwikkelen die tot in alle uithoeken van je programma komt. We stellen dan ook dat een regession test bestaat uit een aantal **scenario's**.

Deze zogenaamde scenario's vormen een soort overzicht van mogelijke gebruiksmogelijkheden van de software. Dit houdt in dat er normale situaties moeten nagebootst worden, maar evengoed dat er abnormale situaties moeten gesimuleerd worden. Wat bedoelen we hier nu precies mee?

Normale situaties Kunnen omschreven worden als alle handelingen en alle inputs (zowel van de gebruiker, als van input, output als configuratie files die het programma gebruikt) die een normale werking van het programma als gevolg zouden moeten hebben.

Abnormale situaties Deze omschrijven we als foutieve input van de gebruiker (bv. een string ingeven als een reëel getal verwacht wordt) of foutieve input via configuratie en input files. Bovendien moet er voor een deel ook rekening worden gehouden met een abnormaal werkend systeem, bv. het aanmaken van een nieuwe outputfile die niet werkt...

Er is bovendien vaak een duidelijke link tussen de use-cases en de voorgestelde scenario's. Immers, use cases zijn generieke gebruikersscenario's van de software: ze zijn een kernachtige beschrijving van hoe de software *kan* gebruikt worden. Een bepaalde use-case kan zo aanleiding geven tot meer dan 1 gebruiksscenario van de software. Daarom moeten we er ook rekening mee houden dat 1 use-case aanleiding kan geven tot een of meerdere scenario's in de regression test!

Langs de andere kant kan je uit de unit en integration tests die je schrijft ook scenario's voor de regression test distilleren. Een goede test moet je immers niet weggooien, maar blijven gebruiken om fouten mee op te sporen! Zeker als je in staat bent om deze tests op een geautomatiseerde manier uit te voeren, bieden ze een uitstekende garantie om ongewilde neveneffecten van (kleine) veranderingen op te sporen.

OK, we weten nu dat een regression test bestaat uit een aantal scenario's, maar hoe bouwen we daarrond nu de regression test op? Aan de hand van Tabel 1 gaan we proberen uit te leggen hoe deze test werkt.

Scenario	Verwachte output	= of $\neq$	Verkregen output
1: een volledig correcte inputfile	Output_1	=	Output_1
2: een volledig correcte inputfile	Output_2	=	Output_2
3: ontbrekende XML sluitingstag	Foutmelding_1	=	Foutmelding_1
4: openings en sluitingstag komen niet overeen	Foutmelding_2	=	Foutmelding_2
5: XML file met onbekende tag	Foutmelding_3	$\neq$	Output_5

Tabel 1: Voorbeeld van een regression test die faalt

De regression test die staat afgebeeld in Tabel 1 bestaat uit 5 scenario's. Scenario's 1 en 2 bestaan uit correcte inputfiles en we verwachten voor de uitvoering van deze 2 scenario's dus gewone programma-output (aangeduid met Output_1 en Output_2). In de praktijk blijkt bij het uitvoeren van het programma dat we een identieke output krijgen.
Voor wat betreft scenario's 3 en 4 weten we dat de inputfiles enkele fouten bevatten, respectievelijk het ontbreken van een XML sluitingstag en het niet-overeenkomen van een een XML openings en sluitingstag. We verwachten dat het programma hiervoor een aantal foutmeldingen zal genereren tijdens de uitvoering en in de praktijk blijkt dit ook zo te zijn.
Tot op heden hebben we dus 4 succesvolle testen laten lopen en als de regression test hier zou eindigen dan zouden we spreken over een **geslaagde regression test**.

Nemen we er echter ook scenario 5 bij: in de XML file zit een XML tag die het programma niet kent. We verwachten dat het programma hiervoor ook een foutmelding genereert, maar tijdens de uitvoering van het programma blijkt dat het programma gewoon loopt en geen foutmelding genereert. Dit is een gedrag dat we niet hadden voorzien en dat ook onjuist is. Vandaar dat we in de tabel $\neq$ zetten. Doordat deze 5de test faalde spreken we nu ook over een **gefaalde regression test**.

7.2 ”Verbose output”

Een standaard regression test geeft als output:

```
"All tests passed"
```

of

```
"At least one test failed"
```

Het belangrijkste idee aan deze summiere output is dat je in 1 oogopslag moet kunnen zien of de aanpassingen die je aan je programma hebt aangebracht een impact hebben gehad op de correcte werking van je programma.

Niettemin zijn er situaties waarin je wil weten welke test(s) is/zijn gefaald. Hiervoor wordt een regression test voorzien van een ”verbose mode”. Dit betekent eigenlijk zoveel als dat je wat meer uitleg geeft over *welke* test faalt en eventueel nog wat meer uitleg (waarom de test faalt bijvoorbeeld). Bijvoorbeeld:

```
"At least one test failed"
    --> test 2 failed because of an unexpected symbol in XML-tag
        "<EXA\/MEN>"
```

7.3 Waarom belangrijk?

Deze vorm van testen is handig omwille van:

- Iemand kan de test laten runnen zonder grote voorkennis van het systeem: er moet immers maar één commando worden uitgevoerd om in één oogopslag te zien of het programma juist werkt of niet.
- Eenmaal je een basisversie van de software werkend hebt is het eenvoudig om te zien of bv. een bepaalde (functionele) aanpassing of een refactoringoperatie geen nadelig gevolg heeft gehad door de regression test te laten lopen.

8 Testing framework

8.1 Wat is een framework?

In programmeertermen is een **framework** een soort *geraamte*: het geeft vorm aan het stuk software dat je aan het schrijven bent. Met andere woorden: alle stukken software die je schrijft met behulp van een bepaalde framework zullen min of meer dezelfde structuur hebben. Bovendien bevat elk framework ook herbruikbare delen basisfunctionaliteit. Juist omdat een framework is uitgedacht voor een heel specifiek situatie, is de basisfunctionaliteit specifiek gericht op het probleem dat je probeert op te lossen.
Een aantal voordelen van frameworks kunnen zijn:

- Door een framework te gebruiken kan je dus veel tijd winnen omdat je een heleboel basisfunctionaliteit kan hergebruiken.
- Toepassingen die je schrijft d.m.v. je framework hebben allemaal dezelfde ruggegraat: hun structuur ziet er dan ook (bijna) hetzelfde uit. Het feit dat ze er hetzelfde uitzien maakt het makkelijker om ze te begrijpen (begrijp je er één, dan begrijp je ze allemaal).

8.2 Een test-framework

Doorheen de cursus hebben we reeds gezien dat er niet één magische – regression – test bestaat die ervoor zorgt dat je hele programma meteen getest is: een goede test bestaat uit een aantal testjes. In deze sectie bespreken we een framework voor unit-tests voor Oberon. We baseren ons hiervoor op de unittesten die zijn ontwikkeld voor de XML-parser die je voor je project mag gebruiken.

Een parser bekijkt en herkent de structuur van een tekstbestand. Sommige bestanden zijn (1) niet gestructuurd, andere zijn (2) slecht gestructureerd en hopelijk zijn de meesten (3) goed gestructureerd. Hoewel we als invoer voor ons programma goed gestructureerde XML-bestanden verwachten, zijn we natuurlijk nooit helemaal zeker dat de gebruikers van het programma zich daar wel aan houden. Dus zorgen we ervoor dat ons programma mooie foutboodschappen geeft bij slechte invoerbestanden. Wat we vooral niet willen is dat ons programma zomaar crasht bij een fout invoerbestand. Dit gaan we testen!

```
MODULE ParserTest;

     IMPORT Out, UnitTest, Parser, StringTypes;

     TYPE
          ParserTest* = POINTER TO ParserTestDesc;
          ParserTestDesc = RECORD (UnitTest.UnitTestData)
               testedParser: Parser.Parser;
          END;
```

Merk op dat we per klasse die we willen testen een nieuwe afgeleide maken van de oorspronkelijke unittest. In de afgeleide klasse stoppen we een object van het type dat we willen testen, in dit geval dus een object van het type Parser.

```
     PROCEDURE (this: ParserTest) Init*;

     BEGIN
          this.Init^;
          this.testedParser := NIL;
     END Init;
```

Deze methode zorgt ervoor dat alles goed geconfigureerd staat aan het begin van je test. Deze wordt dus slechts één keer opgeroepen aan het begin van je test.

```
     PROCEDURE (this: ParserTest) SetUp*(testCase: ARRAY OF CHAR);

     BEGIN
          this.SetUp^(testCase);
          NEW(this.testedParser);
          NEW(infoConsumer);
          this.testedParser.init(infoConsumer);
     END SetUp;
```

`SetUP` daarentegen wordt opgeroepen voor elk scenario dat je op het te testen opbject wil toepassen. De methode zorgt ervoor dat er terug een normale beginsituatie is voor de volgende test.

```
     PROCEDURE (this: ParserTest) TearDown*;

     BEGIN
        this.TearDown^();

        this.testedParser := NIL;
     END TearDown;
```

`TearDown` doet het tegenovergestelde van `SetUP` en zorgt ervoor dat alle variabelen die je voor een bepaald scenario gebruikt hebt, opnieuw in hun oorspronkelijke toestand staan.

```
     PROCEDURE MainSilent*;

     BEGIN
        Main(FALSE);
     END MainSilent;

     PROCEDURE MainVerbose*;

     BEGIN
        Main(TRUE);
```

```
END MainVerbose;

PROCEDURE Main* (verbose: BOOLEAN)

VAR
     test: ParserTest;
     testsPassed: BOOLEAN;

BEGIN
     (* TRUE omdat we veronderstellen dat alle tests zullen slagen *)
     testsPassed := TRUE;

     NEW(test);
     test.Init();
     test.verbose := verbose;

     (* eerste test *)
     test.SetUp("Parser - TestGoodFile");
     IF ~ test.testGoodFile() THEN
          testsPassed := FALSE;
     END;
     test.TearDown;

     (* tweede test *)
     test.SetUp("Parser - TestGoodFileWithNestedValues");
     IF ~ test.testGoodFileWithNestedValues() THEN
          testsPassed := FALSE;
     END;
     test.TearDown;
END Main;
```

De 3 bovenstaande procedures vormen samen eigenlijk de **test driver**: het is het uitvoerbaar commando waarmee je je tests opstart.
De individuele testjes zien er dan zo uit:

```
PROCEDURE (this: ParserTest) testGoodFile(): BOOLEAN;

VAR
     filename: StringTypes.SimpleTypeString;

BEGIN
     this.Run();
     filename := "GoodFile.txt";

     IF ~this.Should(this.testedParser.parse(filename) = NIL,
               "a perfect file should return NIL") THEN
          RETURN FALSE;
     END;

     RETURN TRUE;
END testGoodFile;
```

Zoals je ziet gebruikt bovenstaande methode de methode `Should`. Deze is overgeërfd van de basisklasse van ParserTest, meer bepaald van UnitTest. Deze procedure ziet er als volgt uit:

```
PROCEDURE (aTest: UnitTest) Should*(b: BOOLEAN;
```

```
                    msg: ARRAY OF CHAR): BOOLEAN;

BEGIN
    IF ~b THEN
        aTest^.Fail(msg);
    END;
    RETURN b;
END Should;
```

Should test de conditie `b` en beslist op basis daarvan of de foutboodschap die in `msg` zit moet uitgeschreven worden.
Merk ook op dat of de foutboodschap moet worden afgeprint ook afhankelijk is van de waarde van `verbose`. In dit voorbeeld is hier *geen* rekening mee gehouden.

8.3 Nuttige basisfunctionaliteit

`Should` is uiteraard niet de enige basisfunctionaliteit die in de klasse `UnitTest` zit. Er zijn ook methodes voorzien om files met elkaar te vergelijken. En dit kan handig zijn voor het vergelijken van de huidige output van je programma ten opzicht van de output die je verwacht.

Tot zover het voorbeeld van het test-framework. De volledige code van `UnitTest` en `ParserTest` zit mee in de file `ParserToolbox.Arc`.

9 Tips

Nu je de basis van testen onder de knie hebt, zit je natuurlijk te wachten op tips...

- Schrijf je testen voor je het programma zelf begint te schrijven. Hoewel dit niet zo heel makkelijk is, zet je jezelf er wel toe aan om heel goed na te denken over het stuk code dat je gaat schrijven. Daardoor alleen al ga je minder fouten schrijven.
- Test vanaf het begin, het voorkomt veel problemen die later de kop op kunnen steken.
- Test grondig, zorg er dus voor dat je elke module test. Kijk ook goed uit voor randgevallen (bv. past dit getal wel in het bereik van een integer?).
- Wanneer ben je klaar met testen? Eigenlijk nooit... maar, omdat we binnen budget en binnen de beschikbare tijd willen blijven, moeten we er ooit een punt achter zetten. Om ervoor te zorgen dat alle delen van de software getest zijn, wordt algemeen wel aangenomen dat er voor elke use-case minstens één test moet zijn.