Unit testing

Inleiding: wat en waarom?

Controleer steeds of je **business code** werkt zoals je verwacht door unit testen aan te maken en uit te voeren. Een unit test focust zich hoofdzakelijk op je *business code* en niet op aspecten van je applicatie die hier extern aan zijn, zoals bijvoorbeeld de interactie met de databank: dit is meer het domein van **integratietesten**.

Vaak is het geen slecht idee om de ontwikkeling van je applicatie te starten met je domeinlogica en een set unit testen: op die manier blijft de kern van je applicatie gezond. Delen (classes) van je applicatie die maken dat deze laag lijkt te werken maar die later een effectieve implementatie moeten krijgen, noemt men *stubs* of beter nog **mocks**.

Waarom spreken we precies van *unit* testen? Unit-testen is een methode om stukjes broncode (units) afzonderlijk te testen. Bij unit-testen zal voor iedere unit een of meer tests ontwikkeld worden. Hierbij worden dan verschillende test-cases doorlopen. In het ideale geval zijn alle testcases onafhankelijk van andere tests. Eventueel worden zogenaamde bijkomende mock-objecten gemaakt om de unittests gescheiden uit te kunnen voeren.

Unit-testen opstellen is typisch een taak voor de softwareontwikkelaar en/of het test-team. Dit in tegenstelling tot eindgebruikertests. Ook het doel is anders:

- met unit-testen controleert de ontwikkelaar of delen van de software goed werken of blijven werken gegeven bepaalde invoer (correct of foutief).
- met gebruikerstesten controleert men in samenwerking met de gebruiker (of alleen de gebruiker) of de software reageert op de manier zoals de gebruiker verwacht dat hij zou reageren.

Voordelen

- bij het **aanpassen van software**: unit-testen is gebaseerd op het gebruik van test-cases. Deze worden voorafgaand aan de test opgesteld, meestal zelfs tijdens het ontwikkelen van de software. De verschillende testcases vormen op deze manier een omschrijving van een goede werking van de unit. Wanneer in een later stadium aanpassingen of correcties uitgevoerd moeten worden op de unit, is betrekkelijk eenvoudig na te gaan voor de ontwikkelaar of de wijzigingen de unit in het gedrag niet hebben beïnvloed. De test die dan wordt uitgevoerd noemt men Regressietest.
- bij het **integreren van softwaremodules**: unit-testen gebeurt op afzonderlijke units, meestal door de ontwikkelaar. Als de ontwikkelaar tevreden is wordt de unit opgeleverd. De volgende stap is het testen van de geïntegreerde units in de Integratietests.
- bij het up-to-date houden van **documentatie**: documentatie bijhouden van software is een van de lastigste en vervelendste klusjes. Meestal wordt dit niet goed gedaan, wat resulteert in grote verschillen tussen de documentatie en de software. Unittests en testcase omschrijvingen zou men kunnen beschouwen als "live" documentatie. Iedere wijziging (groot of klein, ontwerpwijziging of bugfix) zal getest moeten worden. Wanneer men dit doet op basis van unittesten, dan zijn de testcases de meest recente beschrijvingen van de correcte werking van een unit en vormen zij dus de meest recente documentatie van de software.

Visual Studio bevat allerlei tools om het werken met unit testing te vereenvoudigen. Met *Live Unit Testing* is het zelfs mogelijk om te detecteren welke testen plots falen door je codewijzigingen omdat je testen op deze manier in de achtergrond uitgevoerd worden terwijl je aan het ontwikkelen bent. De tools die Visual Studio beschikbaar stelt, zullen in een latere walkthrough toegelicht worden.

Werken met unit tests brengt het meeste op indien je van bij aanvang van je project ermee start:

```
- je sleept minder fouten mee.
```

- denkt beter na over je code.

  Vaak is het bovendien onbegonnen werk om nadat een project is opgeleverd, nog met dezelfde granulariteit een set unit testen uit te werken (je code is er niet zo geschikt voor, wat refactoring en dus heel wat tijd impliceert; niemand wenst touwens na oplevering hier nog aanzienlijke tijd en resources aan te besteden).