Test Driven Development

Lorenz Verschingel

16 november 2014

Inhoudsopgave

1 JUnit

1.1 Voordelen van testen

De voordelen van testen zijn:

- Denken over wat de code moet doen.
- Design kan makkelijk mee evoluren.
- Minder tijd in de debugger bezig zijn.
- Korte feedback loop.
- Toont of refactoring code, die voordien werkte, gebroken heeft.
- Versimpeling van code:
  - Alleen code om de tests te doen slagen.
  - Kleine klasses met de focus op een ding.
  - Solide code.

Het resultaat is:

- De unit testen zijn simpel en dienen als documentatie.
- De code is verbeterd en er zijn minder bugs:
  - de code is simpel.
  - nieuwe code breekt de oude code niet.

1.2 Workflow

1. Klassendiagram
2. Schrijf testen
3. Voer de testen uit
4. Schrijf code:
   - Om tests te doen slagen.
   - Om code efficinter te maken.

Hier is het duidelijk dat men eerst de test schrijft en dan pas de code. Als dit niet gebeurt dan is er de neiging om de tests op de code te baseren en niet op de analyse.

1.3 JUnit

- Test Case wordt aangeduidt met *@Test* voor de methode te schrijven.
- Normale workflow:
  1. Maak een object van de te testen klasse.
  2. Roep de te testen methode op.
  3. Controlleer of de code correct is uitgevoerd.
- Door de annotatie van @Before wordt het stuk code in de methode voor iedere unit test uitgevoerd.

Bij JUnit moeten de testen in willekeurige volgorde uitgevoerd kunnen worden.
De te Testklasses kunnen allemaal samen uitgevoerd worden d.m.v. een Test Suite, deze kan m.b.v. JUnit aangemaakt worden.
JUnit voorziet ook geparameteriseerde tests. Deze kunnen eenvoudig opgeroepen worden door *@RunWith(Parameterized.class)*. Binnen de testklasse zelf moeten dan de methode getParameters voorzien worden met de annotatie *@Parameters*.

> *"Keep the bar green, to keep your code clean."*

1.4 Wat testen?

De volgende zaken moeten altijd getest worden:

- Grenswaarden uit de realiteit
- Niet toegelaten waarden uit de realiteit
- Niet toegelaten waarden door techniek (vb null)
- Verzamelingen
  - Normale verzameling
  - Verzameling met grenswaarden
  - Verzamelingen met 1 element
  - Lege verzamelingen
  - Null
- Strings
  - Normale waarde
  - Lege String
  - Null
  - Te lange waarde/Te korte waarde
  - Strings met vreemde tekens
- Exceptions

2 Mock

2.1 Waarom mock-objecten?

Bij het testen vormen de depencencies tussen de te testen klassen een probleem:

- Een van de klasses kan nog niet geschreven zijn.
- Een van de klasses is een UI en daardoor zou er interactie met een gebruiker nodig zijn.
- Moeilijk om te testen als een van de klasses excepties werpt.
- Testen met databank zijn vaak te traag.

Voor al deze problemen biedt een mockobject een oplossing. Een mockobject is dus een testtool.

2.2 Het mock-object

Mock-objecten zijn nepobjecten die het gedrag van het echte object nabootsen. Het mock-object wordt doot *dependency injection* in de echte code geplaatst.

2.3 Dependency Injection

Dependency injection is een geavanceerd ontwerppatroon uit de informatica. Dit ontwerp-patroon maakt het mogelijk om objecten losjes te koppelen. Hiermee wordt bedoelt dat de klasses data kunnen uitwisselen zonder deze relatie hard te koppelen.

2.4 Workflow

In het volgende deel zal steeds naar het mock-object gerefereerd worden als dummy.

1. De werkelijke implementatie, BImpl en de dummy, BDummy, implementeren eenzelfde interface I.
2. De dependeny wordt vastgelegd, in A, met een private variable van het I.
3. In A wordt:
    - Een constructor met een parameter van het type I voorzien.
    - Een set methode met een parameter van het type I voorzien.
4. Bij het uitvoeren wordt BImpl meegegeven.
5. Bij het testen wordt BDummy meegegeven.

Besluit

Mock-objecten zijn nep objecten. Ze worden gebruikt door echte code die getest wordt. Dit geeft tot gevolg dat er nooit business logic in de mock-objecten wordt geschreven.

3 Mockito

Mockito is een framework dat het testen met mock-objecten een stuk makkelijker maakt.