Unit testen met Rhino Mocks

TWEE HANDEN OP ÉÉN BUIK

Sinds Kent Beck in 2000 zijn meesterwerk 'Extreme Programming Explained' schreef, weet iedere ontwikkelaar dat gedegen testen van software noodzakelijk is. Diverse frameworks voor unit testing zijn sindsdien ontstaan. In dit artikel laten we zien dat één van die frameworks, mock testing, een krachtige aanvulling vormt op het 'traditionele' unit testen.

"We will write tests before we code, minute by minute." Het meesterwerk van Kent Beck resulteerde in een aanzienlijke verbetering in kwaliteit van onze software, maar smaakte naar meer. Ook de steeds complexere softwarearchitecturen (SOA, SaaS, SOAP, REST) hebben er toe bijgedragen dat de noodzaak is ontstaan voor nog krachtiger test-frameworks.
Met behulp van een separate test-class kunnen we nu het gedrag van een class en zijn services testen. De test-class legt per scenario de input en verwachte output vast. Deze vorm noemen we ook wel 'black box'-testen. Je bent niet zo zeer geïnteresseerd hoe het gebeurt, als het resultaat maar bij de input past.
Er zijn twee redenen waarom een 'white box'-aanpak relevant kan zijn. Ten eerste wil je het gedrag van een class expliciet controleren. Denk hierbij aan een multi-threaded-oplossing waarbij je heel nauwkeurig wilt vastleggen en controleren hoe het thread-management verloopt.
Ten tweede heb je bij het testen niet altijd invloed op en beschikking over de resources die de software gebruikt. Denk hierbij aan externe (web)services, database-resources of files en folders. Unit-tests moeten autonoom worden opgezet, onafhankelijk van andere componenten en externe resources. Het maakt unit-tests herhaalbaar en zij kunnen in fracties van secondes runnen. Dit bereik je door alle objecten die geen direct onderdeel uitmaken van de component te mocken.
In de volgende voorbeelden hebben we de praktische weg gekozen om aan te tonen dat unit- en mock-testen uitstekend bij elkaar passen. Je zult zien dat het een welkome aanvulling biedt op de dagelijkse uitdaging om kwalitatief goede software te realiseren.

Het voorbeeld

In dit artikel maken we gebruik van een class `Worker` die het `IWorker`-interface implementeert. In verschillende stappen zullen we de testcode voor deze class steeds verder uitbreiden.

```
public class Worker : IWorker
{
    public bool IsRunning { ... }
    public void Start() { ... }
    public void DoWork(IWorkItem workItem)
{ ... }
    public void Stop() { ... }
}
```

Om deze class te testen hebben we ook een class nodig die als argument aan de `DoWork`-method meegegeven kan worden, namelijk een class die het `IWorkItem`-interface implementeert.

CODEVOORBEELD 1. WORKER CLASS

```
public interface IWorkItem
{
    void DoWork(object[] arguments);
}
```

CODEVOORBEELD 2. IWORKITEM-INTERFACE

Je kunt hiervoor geen bestaande class gebruiken, want die hebben meestal afhankelijkheden naar andere classes en die classes weer naar andere classes, et cetera. Als je niet oplet, zit je niet alleen de `Worker`-class te testen, maar ook al die andere classes. Daarom gebruiken we een dummy-class `WorkItemDummyClass` die het `IWorkItem`-interface implementeert. Dat deze class verder niets doet, is niet van belang, we willen immers alleen de `Worker`-class testen (zie codevoorbeeld 3).

De implementatie van `WorkItemDummyClass` gaat eenvoudig. Je moet op zijn minst een property als `IsDoWorkCalled` hebben om te kunnen controleren of de `DoWork`-method wel aangeroepen is. Dat kan echt een heleboel werk betekenen als je veel classes en veel methods wilt testen.

```
[TestMethod]
public void TestDoWork()
{
    IWorker hardWorker = new Worker();
    hardWorker.Start();
    Assert.IsTrue(hardWorker.IsRunning,
"The worker is not running");

    WorkItemDummyClass workItemDummy = new
WorkItemDummyClass();

    hardWorker.DoWork(workItemDummy);

    Assert.IsTrue(workItemDummy.IsDoWork-
Called, "DoWork() is not called");

    hardWorker.Stop();
    Assert.IsFalse(hardWorker.IsRunning,
"The worker is still running");
}
```

CODEVOORBEELD 3. UNIT-TEST VOOR WORKER.

Mock-framework

Een mock-framework biedt hier uitkomst. Mock-objecten worden gebruikt om eigenschappen en gedrag van andere objecten te simuleren. Alle mock-frameworks kunnen op basis van een interface door middel van reflectie een proxy-object aanmaken, dat je vervolgens gebruikt om de class mee te testen.
Hierdoor zijn mock-objecten uitermate geschikt voor unit-testen. Bovendien dwingt het gebruik van mock-objecten je om interface-based te programmeren, wat de kwaliteit van je code verbetert. Grady Booch zegt in zijn boek Object Solutions:
"...all well structured object-oriented architectures have clearly-defined layers, with each layer providing some coherent set of services though a well-defined and controlled interface."

Mock-frameworks komen inmiddels in alle soorten en maten voor (TypeMock, EasyMock, Moq), allemaal met hun eigenaardigheden en aantrekkelijke kanten. Wij zijn erg gecharmeerd

Adv

van Rhino Mocks. Een paar redenen:

- een heel prettige syntax en daardoor eenvoudig te leren;
- zeer uitgebreide API waardoor je alles wat je maar wilt kunt mocken;
- al heel lang een stabiel product waar nog steeds nuttige uitbreidingen op komen.

Na het verder lezen van dit artikel deel je onze voorkeur wellicht.

Getting started

Om het RhinoMocks framework te kunnen gebruiken in je testcode moet je het volgende doen:

- Download the assembly van http://www.ayende.com/projects/rhino-mocks/downloads.aspx
- Neem in je testproject een reference op naar Rhino.Mocks.dll
- Zet boven in je test de volgende using-statements:

```
using Rhino.Mocks;
using Rhino.Mocks.Exceptions;
```

Voorbeeld met RhinoMocks

In codevoorbeeld 4 doen we dezelfde tests als in codevoorbeeld 3, maar gebruiken we de MockRepository uit het framework om een `IWorkItem`-implementatie te verkrijgen. Deze implementatie hoef je dus niet zelf uit te programmeren, maar genereert het RhinoMocks-framework run-time voor je. Hierdoor kun je je in de test focussen op de `Worker`-class en hoef je je niet bezig te houden met het `IWorkItem`-interface.

Opbouw van mock-sessie

Het mocken begint met het aanmaken van een `MockRepository`. Deze fungeert als een collectie van alle objecten die je in de test gaat mocken. De repository biedt verscheidene methodes om (op basis van een interface) mock-objecten te maken.
Als eerste (de recording-fase) geef je aan welke aanroepen er in de rest van je test verwacht worden. Daarna gaat het echte testen beginnen (de playback-fase) en als laatste wordt geverifieerd of aan alle verwachtingen is voldaan.

Record versus playback-state

De aangemaakte mock-objecten staan standaard in de recording-state. Dit betekent dat elke aanroep die je doet op zo'n object intern geregistreerd wordt. Anders gezegd: je definieert je verwachtingen (setting-expectations in mock-taal). Verderop in de code, bij het uitvoeren van de test, verwacht je dat alle aanroepen die je in de record-state hebt gedaan ook daadwerkelijk uitgevoerd gaan worden.

Met het statement `mockWorkItem.DoWork()` uit codevoorbeeld 4 wordt de verwachting gezet dat ooit een keer de `DoWork`-methode wordt aangeroepen op het `mockWorkItem`-object.
De recording-fase wordt afgesloten met `ReplayAll()`. Vanaf dit moment staan de mock-objecten in playback-state. De mock-objecten gedragen zich alsof ze echte implementaties van je interface zijn. De class waar het in deze test om gaat (de `Worker`) heeft geen idee dat de objecten waarmee hij werkt niet de echte implementaties zijn.
Aan het einde van de test volgt via `VerifyAll()` een controle of er aan alle verwachtingen is voldaan. Blijft een verwachting onvervuld, dan genereert het systeem een exception en faalt de test.

Strict en Dynamic mocks

De `MockRepository` heeft twee verschillende manieren voor het creëren van mock-objecten:

- `CreateMock<Type>()`
- `DynamicMock<Type>()`

Via `CreateMock` gecreëerde mock-objecten heten strict mocks. Alleen díe methodes die in de recording-fase zijn aangeroepen worden geaccepteerd. Bij het aanroepen van een method in de test die niet in de recording-fase is genoemd, volgt er een exception en faalt de test.

In codevoorbeeld 5 is het statement `mockWorkItem.DoWork()` niet opgenomen in de recording-fase, maar wordt het wel aangeroepen door `hardWorker.DoWork()`. Hierdoor faalt de test. In dit voorbeeld is deze omissie duidelijk zichtbaar, maar het zou ook kunnen zijn dat de `DoWork`-methode een andere methode op hetzelfde interface aanroept. Die aanroep moet je dan in je recording-fase ook hebben toegevoegd. Anders gezegd: je moet de verwachting hebben gezet dat deze methode wordt aangeroepen. Het gaat hier echt om white-box-testing omdat je precies moet weten hoe de class die je test in elkaar steekt.

Via `DynamicMock` gecreëerde Mock-objecten zijn mocks met dynamisch gedrag. De in de recording-state opgenomen methodes moeten uiteraard in de test worden aangeroepen. Maar tijdens de playback-fase aangeroepen andere methodes worden ook geaccepteerd.
In codevoorbeeld 6 ontbreekt het statement `mockWorkItem.DoWork()` in de recording-fase, maar wordt wel aangeroepen door `hardWorker.DoWork()`. De call wordt geaccepteerd en de test is geslaagd.

Gebruik van LastCall

Zoals al uitgelegd, zet je je verwachtingen door het aanroepen van methodes op je gemockte

```
 [TestMethod]
public void TestDoWorkUsingMocks()
{
      MockRepository mocks = new MockRepo-
sitory();
      IWorkItem mockWorkItem = mocks.
CreateMock<IWorkItem>();
      mockWorkItem.DoWork();

      mocks.ReplayAll();

      IWorker hardWorker = new Worker();
      hardWorker.Start();
    Assert.IsTrue(hardWorker.IsRunning,
"The worker is not running");

    hardWorker.DoWork(mockWorkItem);

    hardWorker.Stop();
    Assert.IsFalse(hardWorker.IsRunning,
"The worker is still running");

    mocks.VerifyAll();
}
```

CODEVOORBEELD 4. UNIT TEST REVISED

objecten. Het Rhino Mock-framework biedt een aantal classes die het mogelijk maken je verwachtingen nog verder te specificeren. Eén van de meest gebruikte is de `LastCall`-class.

Codevoorbeeld 7 geeft via `LastCall.IgnoreArguments()` aan dat je niet geïnteresseerd bent in de argumenten die aan `DoWork()` worden doorgegeven. Via `Repeat.Times(10)` geef je de verwachting aan dat de methode `DoWork()` tienmaal aangeroepen gaat worden.

Gebruik de Do

In de methodes die voor het mock-object zijn gegenereerd, zit geen code. Dat is niet altijd wat je wilt, bijvoorbeeld omdat je logging wilt kunnen doen van de doorgegeven parameters. Wens je toch dat er code wordt uitgevoerd in

```
[TestMethod]
public void TestStrictMock()
{
      MockRepository mocks = new MockRepo-
sitory();
      IWorkItem mockWorkItem = mocks.
CreateMock<IWorkItem>();

      mocks.ReplayAll();

      IWorker hardWorker = new Worker();
      hardWorker.Start();
      Assert.IsTrue(hardWorker.IsRunning,
"The worker is not running");

      hardWorker.DoWork(mockWorkItem);

      hardWorker.Stop();
    Assert.IsFalse(hardWorker.IsRunning,
"The worker is still running");

    mocks.VerifyAll();
}
```

CODEVOORBEELD 5. STRICT MOCK.

```
[TestMethod]
public void TestDynamicMock()
{
      MockRepository mocks = new MockRepo-
sitory();
      IWorkItem mockWorkItem = mocks.
DynamicMock<IWorkItem>();

      mocks.ReplayAll();

      IWorker hardWorker = new Worker();
    hardWorker.Start();
    Assert.IsTrue(hardWorker.IsRunning,
    "The worker is not running");

    hardWorker.DoWork(mockWorkItem);

    hardWorker.Stop();
    Assert.IsFalse(hardWorker.IsRunning,
"The worker is still running");

    mocks.VerifyAll();
}
```

CODEVOORBEELD 6. DYNAMIC MOCK

```
[TestMethod]
public void TestDoWorkWithArguments()
{
    MockRepository mocks = new MockReposi-
    tory();
    IWorkItem mockWorkItem = mocks.
    CreateMock<IWorkItem>();
    mockWorkItem.DoWork(0);
    LastCall.IgnoreArguments().Repeat.
    Times(10);

    mocks.ReplayAll();

    IWorker hardWorker = new Worker();
    hardWorker.Start();

    for (int index = 0; index < 10; ++in-
    dex)
    {
      hardWorker.DoWork(mockWorkItem, in-
      dex);
     }

    hardWorker.Stop();

    mocks.VerifyAll();
}
```

COVERVOORBEELD 7. GEBRUIK VAN LASTCALL

een methode, dan koppel je deze code aan je methode met `LastCall.Do()`.

In Codevoorbeeld 8 wordt de code `Console.WriteLine()`, etc. bij ieder aanroep van `DoWork()`-methode uitgevoerd. Het argument van de `Do()`-method is van het type `Delegate` en bevat de uit te voeren code. Let er wel op dat de method-signature van het `Delegate`-type overeen moet komen met de aan te roepen methode. Als jouw methode een string verwacht, moet de signature van de gebruikte Delegate ook een string bevatten. Daarvoor heb je definitie nodig in codevoorbeeld 9.

Het prettige van de anonymous delegate is dat je geen aparte private methode hoeft te definieren en vanuit de code ook lokale variabelen kunt gebruiken

Mock de database

In vrijwel elke applicatie heb je wel een keer componenten die een database nodig hebben voor hun data. Je wilt het gebruik van de database wel testen, maar niet de overhead van de database erbij (die misschien af en toe voor onderhoud uit de lucht is). Je wilt weten of jouw component op de juiste manier met je datalayer omgaat, zonder allerlei connection-strings, logins, drivers, et cetera nodig te hebben.
Een unit-test dient herhaalbaar te zijn. Dus voor je unit-test moet de gebruikte data iedere keer identiek zijn. In een test-database ondergaat de data vaak door alles en iedereen (waaronder andere tests) wijzigingen. Hierdoor

```
[TestMethod]
public void TestDoWorkWithDo()
{
    int numberOfCallsToDoWork = 0;
    MockRepository mocks = new MockReposi-
    tory();
    IWorkItem mockWorkItem = mocks.
    CreateMock<IWorkItem>();
    mockWorkItem.DoWork(0);
    LastCall.IgnoreArguments().Repeat.
    Times(10).Do(
      new DoWorkDelegate(
           delegate (int[] arguments)
        {
                    Console.
WriteLine("DoWork({0}) is called", argu-
ments[0]);
                    ++numberOfCallsToDo-
                    Work;
          }
      )
    );

    mocks.ReplayAll();

    IWorker hardWorker = new Worker();
    hardWorker.Start();

    for (int index = 0; index < 10; ++in-
    dex)
    {
      hardWorker.DoWork(mockWorkItem, in-
      dex);
    }

    hardWorker.Stop();

    mocks.VerifyAll();

    Assert.AreEqual(numberOfCallsToDoWork,
10);
}
```

CODEVOORBEELD 8. GEBRUIK VAN LASTCALL.DO()

```
private delegate void
DoWorkDelegate(params object[] arguments);
```

CODEVOORBEELD 9. DELEGATE VOOR DO()-METHOD

```
[TestMethod]
public void TestTransfer()
{
    MockRepository mocks = new MockReposi-
    tory();
    IDbConnection databaseConnection =
    mocks.DynamicMock<IDbConnection>();
    IDbCommand databaseCommand = mocks.
    DynamicMock<IDbCommand>();

    Expect.On(databaseConnection).
Call(databaseConnection.CreateCommand())
      .Return(databaseCommand).Repeat.
Times(5);
    Expect.On(databaseCommand).
Call(databaseCommand.ExecuteScalar()).
Return(1000.0);
    Expect.On(databaseCommand).
Call(databaseCommand.ExecuteScalar()).
Return(2000.0);
    Expect.On(databaseCommand).
Call(databaseCommand.ExecuteScalar()).
Return(900.0);
    Expect.On(databaseCommand).
Call(databaseCommand.ExecuteScalar()).
Return(2100.0);

    Expect.On(databaseCommand).
    Call(databaseCommand.ExecuteNonQue-
    ry())
      .Return(1).Repeat.Twice();

    mocks.ReplayAll();

    Bank bank = new
    Bank(databaseConnection);
    double balanceAccount1 = bank.GetBa-
    lance(12345);
    double balanceAccount2 = bank.GetBa-
    lance(67890);

    bank.Transfer(12345, 67890, 100);
    Assert.AreEqual(bank.GetBalan-
    ce(12345), balanceAccount1 - 100);
    Assert.AreEqual(bank.GetBalan-
    ce(67890), balanceAccount2 + 100);

    mocks.VerifyAll();
}
```

CODEVOORBEELD 10. GEMOCKTE DATABASE

is de data eigenlijk een ratjetoe, wat het erg lastig maakt om een eenvoudige en herhaalbare unit-test te ontwikkelen.
In Codevoorbeeld 10 zie je een `Bank`-class waarin we de methodes `GetBalance()` en `Transfer()` testen. De `Bank`-class maakt gebruik van de ADO.Net-interface `IDbConnection`. Door deze interface te mocken, worden we onafhankelijk van de echte data in de database.
De Bank-class maakt naast `IDbConnection` gebruik van `IDbCommand`'s. We moeten dan ook het `IDbCommand` mocken. Via `Expect.Call()` bepaal je welke data je wilt teruggeven. Je ziet dat `ExecuteScalar()` vier keer een waarde retourneert. Dit is omdat we in test `GetBalance()` viermaal aanroepen: twee keer vóór en twee keer na de `Transfer()`.
Het gaat om een echte white-box-test: je moet hier weten hoe de `Bank`-class gebruikmaakt

```
[TestMethod]
public void TestWorkerEvents()
{
    _logger.Debug("TestServiceEvents: started");

    MockRepository mocks = new MockRepository();
    IWorkItem mockWorkItem = mocks.CreateMock<IWorkItem>();
    mockWorkItem.DoWork(0);
    LastCall.IgnoreArguments().Repeat.Times(2).Do(
      new DoWorkDelegate(
        delegate(int[] arguments)
        {
          Console.WriteLine("DoWork({0}) is called", arguments[0]);
          int index = (int)arguments[0];
          if (index == 0)
          {
            throw new Exception("Generate an exception in DoWork()");
          }
        }
      )
    );


    IWorker eventWorker = new Worker();

    // Attach an inline handler to the different events event
    eventWorker.Started +=
      delegate(object sender, WorkerStartedEventArgs eventArgs)
      {
        _logger.Debug("MyService has started");
      };

    eventWorker.Stopped +=
      delegate(object sender, WorkerStoppedEventArgs eventArgs)
      {
        _logger.Debug("MyService has stopped");
      };

    eventWorker.ErrorOccurred +=
      delegate(object sender, ErrorEventArgs eventArgs)
      {
        _logger.Error("MyService generated an exception", eventArgs.Exception);
      };

    mocks.ReplayAll();

    eventWorker.Start();
    for (int index = 0; index < 2; ++index)
    {
      eventWorker.DoWork(mockWorkItem, index);
    }

    eventWorker.Stop();
    _logger.Debug("TestServiceEvents: done");
    mocks.VerifyAll();
}
```

CODEVOORBEELD 11. EVENT HANDLERS

van `IDbConnection` en `IDbCommand`. Hierdoor kan je test wel veel mock-code bevatten, maar als voordeel geldt dat de test onafhankelijk, herhaalbaar en zeer snel is.

```
private delegate void StartedEventHandlerD
elegate(object sender,

WorkerStartedEventArgs eventArgs);
private delegate void ErrorEventHandler
Delegate(object sender, ErrorEventArgs
eventArgs);
private delegate void StoppedEventHandlerD
elegate(object sender,

WorkerStoppedEventArgs eventArgs);
```

Test je event handlers

Events zijn cruciaal voor het opzetten van 'loosly-coupled' programmatuur. Je wilt testen dat geen events verloren gaan, dat de juiste events optreden en het op een juiste manier afhandelen van events. Rhino Mocks biedt de mogelijkheid events af te vangen (en er je eigen mock-code aan te hangen) en om events te genereren.

In codevoorbeeld 11 worden de `Started`-, `Stopped`- en `Error`-events opgevangen. Hier zijn ook weer `Delegate`-types voor gedefinieerd.

Tevens testen we dat wanneer er een exception optreedt in de `IWorkItem.DoWork()`, de `Worker` door kan lopen. De test is zo opgezet dat er alleen de eerste keer een Exception gegenereerd wordt (zie codevoorbeeld 12).

```
    if (index == 0)
    {
      throw new Exception("Generate an
exception in DoWork()");
    }
```

CODEVOORBEELD 12. EXCEPTION GENEREREN

```
[TestInitialize]
public void TestInitialize()
{
    _mocks = new MockRepository();
}

[TestCleanup]
public void TestCleanup()
{
    _mocks.VerifyAll();
}
<
```

CODEVOORBEELD 13. GEBRUIK TESTINITIALIZE/TEST-CLEANUP

Vergeet VerifyAll niet

Het is ons in de praktijk vaak overkomen dat we het `VerifyAll()`-statement in een test vergaten. Hierdoor vindt geen controle plaats of er aan alle verwachtingen voldaan is. Om dit te voorkomen, en om het gebruik van mocks eenvoudiger te maken, is het beter om je `MockRepository` een member-variabele te maken en initialisatie en verificatie in respectievelijk de `TestInitialize()`- en `TestCleanup()`-methodes te zetten. De mock-repository is beschikbaar in iedere test en `VerifyAll()` wordt nu altijd aan het einde van ieder test aangeroepen.

Hoe nu verder?

In dit artikel hebben we verschillende aspecten van unit-test en mocks de revue laten passeren. Toch zijn we geenszins volledig geweest. Ga ermee aan de slag en voel eigenhandig wat Rhino Mocks voor je kan betekenen. De code is te downloaden van http://www.microsoft.nl/netmagazine. .net

Referenties

Rhino Mocks, ontwikkeld door Oren Eini, http://www.ayende.com/projects/rhino-mocks.aspx

Kent Beck: Extreme programming explained, embrace change

Grady Booch: Object Solutions

www.mockobjects.com

www.ibm.com/developerworks/library/j-mocktest.html

Alex Harbers en **Richard de Zwart** zijn software engineer bij Luminis (www.luminis.nl). Heb je vragen of wil je contact opnemen met één van de auteurs stuur dan een e-mail naar alex.harbers@luminis.nl of richard.dezwart@luminis.nl.