

Mock-Objekte unter Microsoft .NET

# Unit-Tests für Fortgeschrittene

Dank Extreme Programming ist das Unit-Testen zu einer gängigen Disziplin geworden. Mock-Objekte stellen in diesem Zusammenhang eine Technik zum Optimieren eines testbaren Designs dar. Zusammen mit dem Open-Source-Testframework NUnit vereinfacht die Klassenbibliothek EasyMock.NET den Umgang mit Mock-Objekten wesentlich.

**T**estgetriebene Entwicklung und das damit zusammenhängende Unit-Testen gehören zu den wichtigsten Themen in der Entwicklerszene. Nicht zuletzt die unter Microsoft .NET verfügbaren Test-Frameworks wie das auf JUnit aus der Java-Welt basierende NUnit ([www.nunit.org](http://www.nunit.org)), oder csUnit ([\[nit.org\]\(http://www.csunit.org\)\) haben durch ihren professionellen Charakter dazu beigetragen.](http://www.csu-</a></p></div><div data-bbox=)

## Testgetriebene Entwicklung

Die testgetriebene Entwicklung in kleinen Schritten, der wesentliche Kern des so genannten Extreme Programming, führt zu einer deutlich verbesserten Qualität der Software. Des Weiteren profitiert das Design durch eine höhere Flexibilität.

Der Entwickler erstellt in der testgetriebenen Entwicklung zuerst einen Testfall in einem Unit-Testframework. Er ist dabei gezwungen, präzise zu definieren, was eine Methode leisten soll. Der Test schlägt zunächst fehl, weil die zu testende Funktionalität noch fehlt. Anschließend erstellt der Entwickler den eigentlichen Code bis zur fehlerfreien Ausführung des Testfalls.

Der Entwickler weiß mit diesem Verfahren sehr genau, wann die Methode fertiggestellt ist: sobald der zugehörige Testfall und die Gesamtheit aller Testfälle erfolgreich ausgeführt wurden.

## Theorie und Praxis

Dieses in der Theorie einfache Vorgehen der testgetriebenen Entwicklung bringt in der Praxis eine gewisse Komplexität mit sich. Abhängigkeiten zwischen Klassen, das Überschreiten von Systemgrenzen oder die Behandlung von Ausnahmesituationen sind Probleme, die im Unit-Test berücksichtigt werden müssen.

Ein Unit-Test prüft eine Funktionseinheit in Isolation, das heißt, es wird eine einzelne Methode oder eine einzelne Klasse getestet. In einem objektorientierten Programm werden Anforderungen jedoch durch das Zusammenspiel mehre-

## Listing I

### Die Klasse VertragGateway.

```
Imports DataAccessLayer
'enthält IDataAccessBroker

Public Class VertragGateway
'nimmt Referenz auf den Broker auf
Private m_broker As IDataAccessBroker

'Konstruktor mit Referenz auf den Broker
Public Sub New(ByVal broker As _
    IDataAccessBroker)
    Me.m_broker = broker
End Sub

'Leseoperation
Public Function Find(ByVal id _
    As Integer) As Vertrag

    Dim obj As Object = Me.m_broker._
        ReadObject(GetType(Vertrag), id)
    Return CType(obj, Vertrag)

End Function
End Class
```

rer Klassen gelöst. Schnell überschreiten die Abhängigkeiten zwischen zu testenden Klassen einen gewissen Umfang, so dass der Testaufwand überproportional ansteigt [1]. Spätestens dann, wenn für einen Unit-Test eine große Menge an Objekten initialisiert werden muss, stellt sich die Frage, ob der Aufwand dafür gerechtfertigt ist.

Überschreiten Testfälle die Systemgrenzen, zum Beispiel bei Zugriffen auf externe Ressourcen wie Datenbanken, Drucker oder externe Prozesse, leidet die Performance der Unit-Tests.

Die zu erstellende Software muss Ausnahmesituationen korrekt behandeln. So

## Auf einen Blick

### Autoren

**Dietmar Leibecke** ist geschäftsführender Gesellschafter der applied technologies

GmbH. Sein Schwerpunkt liegt auf den Themen Entwicklungsprozesse, Qualitätsmanagement, Architektur und Design von .NET-, C++- und J2EE-Anwendungen. Sie erreichen ihn unter der E-Mail-Adresse [dleibecke@appliedtechnologies.de](mailto:dleibecke@appliedtechnologies.de).



Dipl.-Wirt.Inf. **André Achtermeier** ist Software-Engineer bei der applied technologies GmbH. Er befasst sich hauptsächlich mit OOD unter .NET und J2EE.

Sie erreichen ihn unter der E-Mail-Adresse [aachtermeier@appliedtechnologies.de](mailto:aachtermeier@appliedtechnologies.de).

dotnetpro.code  
A0406Mock



**Sprachen** Visual Basic .NET

**Technik** EasyMock.NET V2.0 beta

**Voraussetzungen** NUnit V2.x

muss ein Kommunikationsbaustein im Falle eines nicht erreichbaren Partners oder Prozesses eine definierte Aktion ausführen und darf nicht einfach den Dienst einstellen. Das Gleiche gilt zum Beispiel auch für einen Druckertreiber, der nicht abstürzen darf, nur weil kein Drucker angeschlossen ist.

## Mock-Objekte

An dieser Stelle kommt die Technik der Mock-Objekte ins Spiel, die eine Lösung für diese Probleme verspricht. Erstmals wurde diese Technik während der Konferenz XP2000 vorgestellt [2], [3]. In der Testumgebung ersetzt ein Mock-Objekt ein „echtes“ Objekt, mit dem der zu testende Programmcode zusammenarbeiten soll. Dieser Code kann Methoden des Mock-Objekts aufrufen und Ergebnisse des Aufrufs verwerten. Gleichzeitig prüft das Mock-Objekt Argumente, die das zu testende Objekt verwendet hat.

Die Autoren Mackinnon/Freeman/Craig raten, Mock-Objekte in den folgenden Fällen einzusetzen:

- Das echte Objekt hat ein nicht-deterministisches Verhalten.
- Die Testumgebung für das echte Objekt ist sehr aufwändig aufzusetzen.
- Ausnahmesituationen sind mit dem echten Objekt nur sehr schwer zu

reproduzieren.

- Das echte Objekt ist langsam.
- Das echte Objekt hat (oder ist) ein User-Interface.
- Der Test muss das externe Objekt befragen, wie es benutzt wurde.
- Das echte Objekt existiert noch nicht.

Grundsätzlich ist bei der Verwendung von Mock-Objekten folgendes Vorgehen festzustellen:

- Das Mock-Objekt wird zunächst initialisiert und Erwartungen werden spezifiziert. Unter *Erwartung* versteht man beispielsweise die Aufrufreihenfolge von Methoden oder die in Methodenaufrufen verwendeten Argumente.
- Das echte Objekt wird mit dem Mock-Objekt getestet.
- Die Verwendung des Mock-Objekts wird geprüft.

## Beispiel Datenbankzugriff

In einer mehrschichtigen Architektur kapselt ein Data-Access-Layer den Zugriff auf die Datenbank. Der Data-Access-Layer im zugrunde liegenden Referenzprojekt stellt für den tatsächlichen Zugriff auf die Datenbank das Interface *IDataAccessBroker* mit der Operation *ReadObject* bereit. Als Argumente erwartet diese Opera-

tion die Typinformation des zu ladenden Objekts sowie den zugehörigen Schlüssel.

In einem beispielhaften Szenario soll eine Datenbank mit den Verträgen eines Energieversorgers angebunden werden. Der Zugriff auf das Interface *IDataAccessBroker* soll gemäß dem *Table Data Gateway*-Pattern [4] ausgeführt werden. Die Klasse *Vertrag* nimmt dabei die Vertragsinformationen auf.

Ein Unit-Test prüft eine Einheit in Isolation. Für die zu erstellende Methode *Find()* der Klasse *VertragGateway* muss der Unit-Test also prüfen, ob das Interface *IDataAccessBroker* zur Datenbank korrekt verwendet wird. Listing 1 zeigt den Code der Klasse *VertragGateway*.

Die Klasse *VertragGateway* nimmt über den Konstruktor eine Referenz auf das *IDataAccessBroker*-Interface entgegen und legt diese in der Member-Variablen *m\_broker* ab.

Die Leseoperation *Find()* ruft den Broker mit der Typinformation der Klasse *Vertrag* und der gewünschten ID auf.

Um diese Operation testen zu können, muss in der Testumgebung der Broker durch ein Mock-Objekt ersetzt werden.

## Ein handgestricktes Mock-Objekt

In einem ersten Schritt wird die Mock-Objekt-Technik mit einem handgestrickten

## Listing 2

### Das handgestrickte Mock-Objekt MockVertragGatewayBroker.

```
Imports NUnit.Framework 'NUnit laden
Imports BusinessLogicLayer 'beinhaltet Vertrag und VertragGateway
Imports DataAccessLayer 'beinhaltet IDataAccessBroker

Public Class MockVertragGatewayBroker
    Implements IDataAccessBroker

    'Member-Variablen der Klasse MockVertragGatewayBroker
    Private m_usageCorrect As Boolean = False
    Private m_typeObj As Type = Nothing
    Private m_id As Integer = 0

    'Konstruktor mit Erwartungen
    Public Sub New(ByVal typeObj As Type, ByVal id As Integer)
        'Parameter in Members ablegen
        Me.m_typeObj = typeObj
        Me.m_id = id
    End Sub

    'Die zu implementierende Operation ReadObject
    Public Function ReadObject(ByVal typeObj As Type, ByVal id _
        As Integer) As Object Implements IDataAccessBroker.ReadObject
        'Prüfungen ausführen
        Assertion.AssertEquals("Ungültiger Typ angefragt", _
            Me.m_typeObj, typeObj)
        Assertion.AssertEquals("Ungültige Id angefragt", Me.m_id, id)

        Me.m_usageCorrect = True

        Return Nothing
    End Function

    Public Sub WriteObject(ByVal obj As Object) Implements _
        IDataAccessBroker.WriteObject
        'Wird in diesem Fall nicht benötigt
    End Sub

    Public Function Verify() As Boolean
        Assertion.Assert("Broker nicht korrekt verwendet", _
            Me.m_usageCorrect)
        Return Me.m_usageCorrect
    End Function
End Class
```

### Listing 3

#### Der Unit-Test mit MockVertragGatewayBroker.

```
Imports NUnit.Framework
Imports BusinessLogicLayer
Imports DataAccessLayer

<TestFixture()> Public Class TestVertragGateway

    <Test()> Public Sub TestFind()

        Const VERTRAG_ID As Integer = 42

        'Erwartungen spezifizieren
        Dim broker As New MockVertragGatewayBroker(GetType(Vertrag), VERTRAG_ID)

        'Das echte Objekt mit einem Mock-Objekt initialisieren
        Dim gw As New VertragGateway(broker)

        'Die eigentliche Operation ausführen
        Dim v As Vertrag = gw.Find(VERTRAG_ID)

        'Verwendung des Brokers prüfen
        broker.Verify()

    End Sub

End Class
```

Mock-Objekt dargestellt. Listing 2 zeigt den Code der Klasse *MockVertragGatewayBroker*, die ein Mock-Objekt auf Basis des Interface *IDataAccessBroker* darstellt.

Die Mock-Klasse implementiert das *IDataAccessBroker*-Interface. Eine Instanz dieser Klasse kann somit also an allen Stellen verwendet werden, die eine Referenz auf ein *IDataAccessBroker*-Interface erwarten.

Der Konstruktor der Klasse nimmt die Erwartungen an das Mock-Objekt in Form eines Typobjekts und einer ID entgegen. Die Operation *Read()* prüft über die NUnit-Klasse *Assertion* die Gültigkeit der Eingangsparameter und setzt im Er-

folgsfall das Flag *m\_usageCorrect* auf *True*. Der entsprechende Unit-Test gestaltet sich wie folgt (siehe Listing 3).

Der Testfall *TestFind* deklariert zunächst die Konstante *VERTRAG\_ID*. Anschließend werden die Arbeitsschritte, die bei der Verwendung von Mock-Objekten eingehalten werden sollten, ausgeführt:

*Erwartungen spezifizieren:* Das Mock-Objekt wird instanziiert. Der Konstruktor nimmt dabei das erwartete Typobjekt und die ID des Vertragsobjekts entgegen.

*Das echte Objekt mit dem Mock-Objekt testen:* Der Testfall erzeugt eine In-

stanz der Klasse *VertragGateway*, die mit dem Mock-Objekt initialisiert wird. Der Aufruf der Operation *Find()* mit der Konstanten *VERTRAG\_ID* führt dazu, dass die Operation *ReadObject()* auf dem Mock-Objekt aufgerufen wird.

*Nutzung des Mock-Objekts prüfen:* Der Aufruf der Operation *Verify()* auf das Mock-Objekt prüft den korrekten Einsatz.

Diese Technik erlaubt es dem Entwickler, eine Klasse von innen zu testen. Dieser Unit-Test würde folgende Fehler erkennen:

- Aufruf des *IDataAccessBrokers* unter Angabe eines falschen Typs,
- Aufruf des *IDataAccessBrokers* unter Angabe einer ungültigen Objekt-ID,
- der *IDataAccessBroker* wird erst gar nicht aufgerufen.

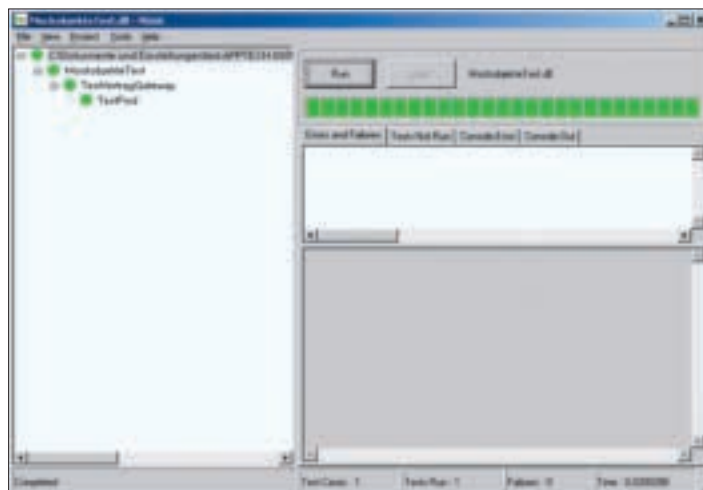
#### Nachteile „manueller“ Mock-Objekte

Mock-Objekte sind eine ausgezeichnete Technik, um Unit-Tests in isolierten Einheiten auszuführen. Das Design wird mit dieser Technik flexibler. Die Abhängigkeiten zwischen Klassen verringern sich, womit eines der Ziele objektorientierter Programmierung erreicht wird.

Auch wenn das Erstellen von Mock-Objekten einfach ist und Erfolge schnell sichtbar werden: Es ergeben sich kleinere Nachteile aus dem Einsatz manuell erstellter Mock-Objekte:

- Da sich die Mock-Objekte in separaten Klassen befinden, muss deren Code gelesen werden, um einen Unit-Test zu verstehen.
- Es kann mühsam sein, Mock-Objekte zu erstellen. Unter Umständen schleichen sich Fehler ein.
- Wird eine Methode zu einem Interface hinzugefügt, so ist das Mock-Objekt ebenfalls um diese Methode zu ergänzen.
- Wird der Name einer Methode verändert, ist dies in allen Mock-Objekten zu berücksichtigen.
- Wird eine Methode aus einem Interface gelöscht, so ist diese Methode in den Mock-Objekten ebenfalls zu löschen.

An dieser Stelle schafft das Open-Source-Produkt *EasyMock* Abhilfe. *EasyMock* ist ein in der Java-Welt bewährtes und einfach zu bedienendes Werkzeug



**Abbildung 1**  
Der erfolgreich ausgeführte Test mit NUnit.

## Listing 4

### Der Unit-Test mit EasyMock.NET.

```
Imports NUnit.Framework
Imports EasyMockNET

Imports BusinessLogicLayer
Imports DataAccessLayer

<TestFixture(>) Public Class TestVertragGatewayEasyMock

    <Test(>) Public Sub TestFind()
        Const VERTRAG_ID As Integer = 42

        'Control- und Mock-Objekte erzeugen
        Dim control As IMockControl = _
            EasyMock.ControlFor(GetType(IDataAccessBroker))
        Dim broker As IDataAccessBroker = _
            CType(control.GetMock(), IDataAccessBroker)

        'Erwartung spezifizieren
        broker.ReadObject(GetType(Vertrag), VERTRAG_ID)

        control.SetReturnValue(Nothing)

        'Mock-Objekt über den Controller aktivieren
        control.Activate()

        'Das echte Objekt mit einem Mock-Objekt initialisieren
        Dim gw As New VertragGateway(broker)

        'Die zu testende Operation ausführen
        Dim v As Vertrag = gw.Find(VERTRAG_ID)

        'Verwendung des Brokers prüfen (auf dem Controller!)
        control.Verify()

    End Sub

End Class
```

für die dynamische Erzeugung von Mock-Objekten [7]. Das Prinzip der Java-Lösung wurde mit dem Open-Source-Werkzeug EasyMock.NET auf die.NET-Welt übertragen [8].

### EasyMock.NET

Das Tool EasyMock.NET verfolgt einen dynamischen und hochflexiblen Ansatz. Ein so genanntes Mock-Control erstellt dabei zur Testausführung ein Mock-Objekt. Nach dem Erzeugen befindet sich dieses Objekt im Setup-Modus. In diesem Modus werden die Erwartungen spezifiziert. Nach dem Wechsel in den aktiven Modus verhält sich das Mock-Objekt wie zuvor programmtechnisch spezifiziert. Wird das Mock-Objekt in unerlaubter Art und Weise verwendet, wird umgehend ein Fehler signalisiert. Listing 4 zeigt die Testklasse nach Umstellung auf EasyMock.NET.

Im ersten Schritt wird zunächst ein Mock-Control für das Interface *IDataAccessBroker* über den Aufruf der Operation *EasyMock.ControlFor()* erzeugt. Diese Methode liefert als Ergebnis ein Objekt des Typs *IMockControl* zurück. Dieses fordert dann ein Mock-Objekt über den Aufruf *GetMock()* an.

Anschließend werden im zweiten Schritt die Erwartungen spezifiziert. Der Broker soll mit der Operation *ReadObject()* aufgerufen werden. Dabei werden als Argumente das Typobjekt der Klasse *Vertrag* und die Konstante *VERTRAG\_ID* erwartet. Da in diesem Test der Rückgabewert des

*ReadObject()*-Aufrufs keine Bedeutung hat, wird der Wert *Nothing* als Rückgabewert spezifiziert.

Der Aufruf der Operation *control.Activate()* versetzt das Mock-Objekt in den Ausführungsmodus. Jede fehlerhafte Verwendung des Mock-Objekts führt umgehend zu einer Fehlermeldung.

Anschließend wird die Funktion des echten Objekts ausgeführt und der Einsatz des Mock-Objekts über das Mock-Control verifiziert.

### Fazit

Das Erstellen von Unit-Tests ist eine Disziplin, die zu einer deutlich besseren Qualität des Designs und der Anwendung führt. Mock-Objekte stellen in diesem Zusammenhang eine wichtige unterstützende Technik dar. Sie ermöglichen es, echte Objekte durch eine Fälschung zu ersetzen. Der Entwickler erhält auf einfache Weise die Möglichkeit, sich auf den eigenen Code zu konzentrieren. Das Tool EasyMock.NET macht dabei die Entwicklung solcher Mock-Objekte deutlich einfacher:

- Der Code für einen Unit-Test wird an genau einer Stelle gepflegt.
- Das zeitaufwändige manuelle Erstellen von Mock-Objekten entfällt.
- Änderungen an Interfaces führen durch den dynamischen Ansatz von EasyMock.NET nicht zu Änderungen an Mock-Objekten.

Mock-Objekte sind jedoch kein All-

roundmittel, das in jedem Fall eingesetzt werden muss. Der Entwickler sollte das Design aus diesem Grund in folgenden Fällen refaktorisieren:

- Für den Test einer Klasse wird mehr als ein Mock-Objekt benötigt.
- Ein Mock-Objekt enthält eine so komplexe Logik, dass eigene Unit-Tests für das Mock-Objekt nötig sind.
- Ein Mock-Objekt benötigt für die Ausführung eines Testfalls weitere Mock-Objekte. |||||

- [1] Frank Westphal, Testgetriebene Entwicklung mit JUnit und FIT, dpunkt.verlag 2004, s. a. [www.frankwestphal.de/TestgetriebeneEntwicklung.html](http://www.frankwestphal.de/TestgetriebeneEntwicklung.html)
- [2] Tim Mackinnon, Steve Freeman und Philip Craig, XP 2000, [www.connextra.com/aboutUs/mockobjects.pdf](http://www.connextra.com/aboutUs/mockobjects.pdf)
- [3] Die Homepage der Mock Objects Community, [www.mockobjects.com](http://www.mockobjects.com)
- [4] Martin Fowler, Patterns Of Enterprise Application Architecture, Addison-Wesley 2003
- [5] Dietmar Leibecke, Bernd Gerwert, NUnit – Unit-Tests unter .NET, OBJEKTSpektrum November/Dezember 2002
- [6] Eric Gunnerson, Unit-Testing and Test-First-Development, <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dncscol/html/csharp03202003.asp>
- [7] Homepage des Java-Werkzeugs EasyMock, [www.easymock.org](http://www.easymock.org)
- [8] Homepage des .NET-Tools EasyMock.NET, [www.easymock.net](http://www.easymock.net)