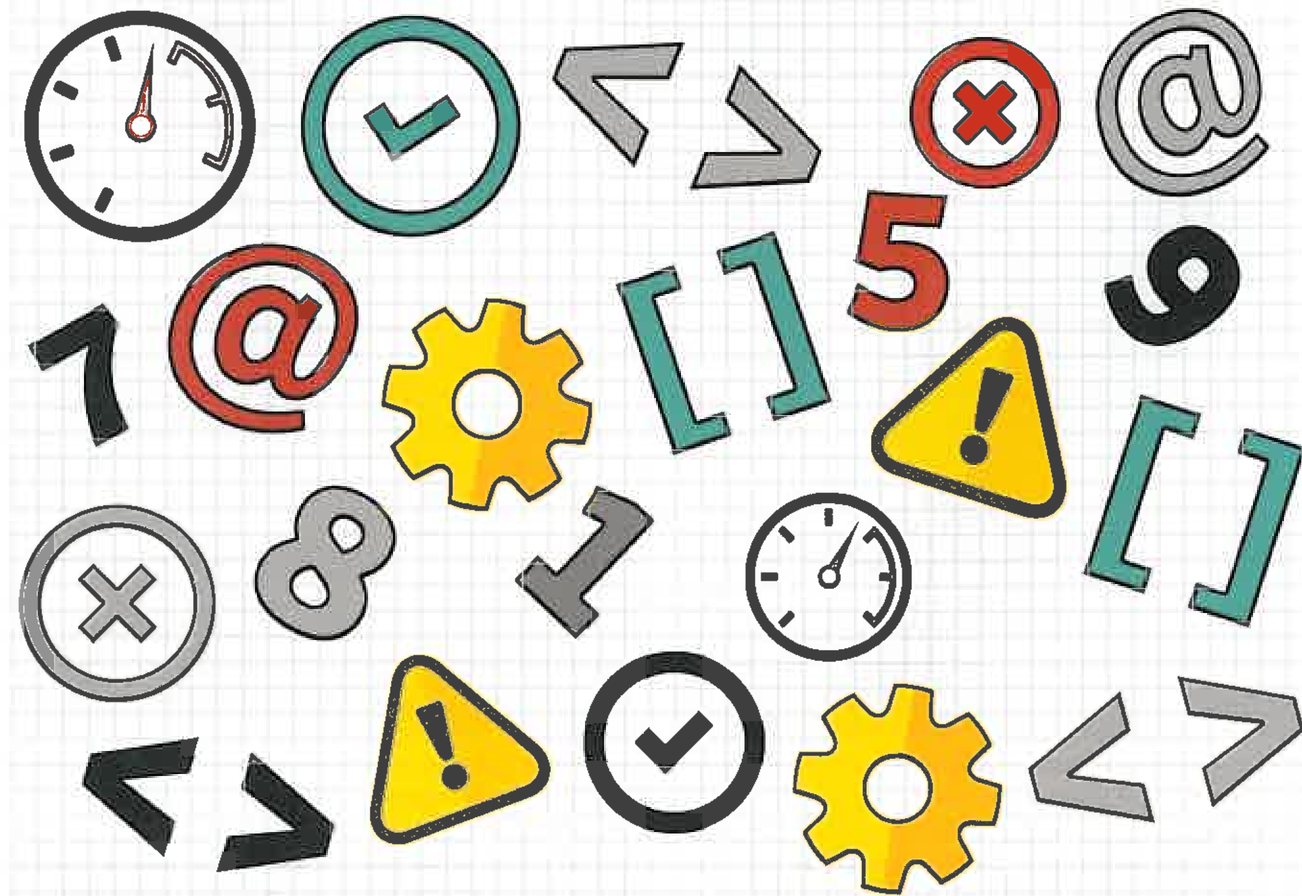


HALBAUTOMATISCHE GENERIERUNG VON UNIT- UND INTEGRATIONSTESTS

Testautomatisierung 2.0

Eine Kombination aus diversen Frameworks reduziert den Aufwand beim Testen.

Unit- und Integrationstests gehören zu den wichtigsten und am weitesten verbreiteten Testmethoden. Primäres Ziel von Unit-Tests ist, einen Teil der Software zu isolieren und zu testen. Unit-Tests erleichtern das Auffinden von Fehlern, die Pflege und die Verständlichkeit des Codes. Integrationstests sind eine Erweiterung von Unit-Tests. Dabei werden mehrere bereits getestete Aufgaben miteinander kombiniert und gemeinsam getestet.

Mit Unit- und Integrationstests können nicht nur Fehler schneller gefunden werden. Vielmehr lässt sich damit die Qualität über den gesamten Produktlebenszyklus einer Software gewährleisten.

Deren Qualität und Wartbarkeit hängen sowohl von einer sauber strukturierten Architektur als auch von einer guten Testabdeckung mit sinnvollen Unit- und Integrationstests ab. Hohe Testabdeckung ist wichtig, da durch Erweiterungen oder Architekturänderungen in bereits getesteten Teilen des Softwaresystems aufgrund sogenannter Nebeneffekte neue Fehler entstehen können. Daher zählt das wiederholte Testen zu den essenziellen Bereichen qualitativ hochwertiger Softwareentwicklung.

Um einen höheren Grad der Testabdeckung zu erreichen, ist aber viel Aufwand zu treiben. Aus diesem Grund gibt es große Bestrebungen, Testmethoden und die dazugehörigen Objekte automatisiert zu generieren.

Das CIT-Testtool

Die nötigen Unit-Tests wurden in der österreichischen COUNT IT Group wie auch in vielen anderen IT-Unternehmen immer per Hand geschrieben. Durch Änderungen der Anforderungen, des Source-Codes oder der Datenbank und dem in Softwareentwicklungsprojekten üblichen Zeitdruck litt in den heißen Phasen der Entwicklung oft die Testabdeckung. Ein großes Ziel war es daher, datenbankunabhängige Tests halbautomatisch, aber trotzdem qualitativ hochwertig erstellen zu können. Aus diesem Grund wurden in den letzten Jahren bestehende Testtools und -frameworks analysiert. Man entwickelte das COUNT IT Testtool (kurz CIT-Testtool).

Listing 1: Das Proxyobjekt

```
public interface ILocation
{
  int LocationID { get; set; }
  string Description { get; set; }
}

[TestMethod]
public void MockILocationTest()
{
  int id = 255;
  string description = "COUNT IT Group";

  Mock<ILocation> mockedLocation =
    new Mock<ILocation>();
  mockedLocation.Setup(location =>
    location.LocationID).Returns(id);
  mockedLocation.Setup(location =>
    location.Description).Returns(description);

  ILocation locationObjekt = mockedLocation.Object;

  Assert.AreEqual(id, locationObjekt.LocationID);
  Assert.AreEqual(description,
    locationObjekt.Description);
}
```

Dieses Werkzeug reduziert den Aufwand bei der Erstellung von Unit- und Integrationstests massiv, ohne dabei Auswirkungen auf die Testqualität zu haben. Damit können die Tests datenbank- und webserviceunabhängig erstellt werden, was ein isoliertes Testen der Programmlogik möglich macht.

Varianten des Testens von Softwaresystemen

Es gibt drei Varianten, ein Softwaresystem zu testen: Bei der ersten Variante führt ein Tester nach Änderungen im System einen kompletten manuellen Test durch. Da diese Tests vor jedem Release ausgeführt werden sollten, hat das bei großen Softwaresystemen einen sehr hohen Testaufwand zur Folge. Das ist nicht im Sinne der agilen Softwareentwicklung.

Die anderen beiden Varianten befassen sich mit dem Schreiben von automatisch ausführbaren Unit- und Integrationstests. Diese Tests werden entweder manuell oder automatisiert durch Tools erstellt, wobei das manuelle Schreiben von ausführbaren Tests sehr zeitaufwendig ist. Einerseits müssen die Tests wohlüberlegt sein, andererseits müssen alle möglichen (Fehler-)Fälle beachtet werden.

Im Gegensatz zur manuellen Erstellung wird für die automatisierte Generierung der Tests kaum Zeit benötigt. Die Erstellung geschieht mit wenigen Klicks und bietet ein breites Spektrum an möglichen Eingangsparametern für die zu testende Methode an. Im Gegenzug büßt man aber oft einen Teil der Testqualität ein.

Vorzüge eigens entwickelter Testtools

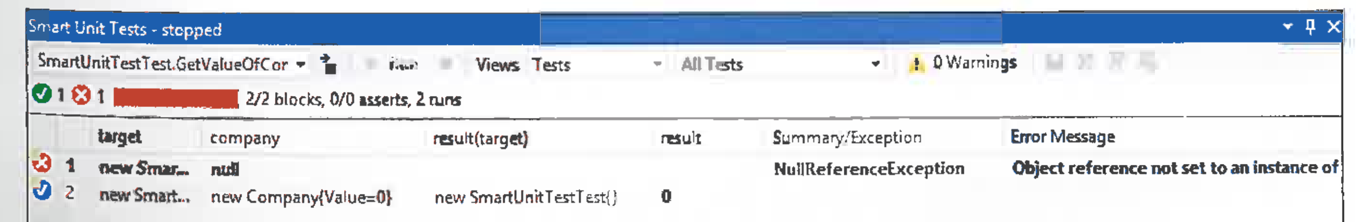
Abhilfe können eigene, auf die Architektur und Rahmenbedingungen angepasste Testtools schaffen. Selbst entwickelte Testtools bedeuten einen hohen initialen Entwicklungsaufwand, sind dafür aber später sehr effektiv bei der Erstellung von Unit- und Integrationstests. Außerdem lassen sie sich erweitern und anpassen, sodass auch komplexe Testfälle möglich werden. Der hohe Entwicklungsaufwand gleicht sich so durch die gesparte Zeit bei der Generierung von Unit- und Integrationstests aus.

Eine hohe Testabdeckung, ob durch manuell geschriebene oder generierte Tests, ist jedoch wertlos, wenn die Tests nicht laufend durchgeführt werden. Außerdem ist es erforderlich, die Testmethoden regelmäßig zu warten, um auf veränderte Anforderungen zu reagieren und die vorhandenen Tests immer auf dem aktuellen Stand zu halten. Vor allem bei Architekturänderungen oder Änderungen in der Datenstruktur können schlecht gewartete Testfälle zu einer Menge von Problemen führen.

Ein weiteres Problem stellen Methoden dar, die auf Datenbanken zugreifen, da sie sehr schwer isoliert zu testen sind. Es wird immer eine Datenbank im Hintergrund benötigt und die Daten müssen vor den Tests in der Datenbank zurückgesetzt werden, um einen für die Tests sauberen Datenstand zu gewährleisten.

Schwieriges Testen von Webservicemethoden

Webservicemethoden sind aus den gleichen Gründen genauso schwer zu testen. Oft werden deshalb Methoden, die Datenbankzugriffe und Webserviceaufrufe enthalten, nicht getestet. Genau diese Tests können aber in späteren Phasen des Softwareentwicklungsprozesses vor fatalen Fehlern schützen und sehr viel Zeit bei der Fehlersuche sparen. Das Ziel des CIT-Testtools ist es, die Tests so weit wie möglich von Datenbank und Webservice zu isolieren, um diese Probleme zu vermeiden. Um datenbank- und webserviceunabhängige ▶



Ergebnis einfacher Smart Unit Tests-Test (Bild 1)

Testfälle zu schreiben, werden einige Frameworks und Tools benötigt, die im Zusammenspiel isolierte Tests der übergeordneten Schichten ermöglichen:

- Mock-Framework (zum Beispiel Moq)
- Fakes-Framework
- PEX beziehungsweise Smart Unit Tests

Das Mocking-Framework Moq [1] kann über NuGet heruntergeladen und in Projekte eingebunden werden. Es ist ein minimalistisches, aber sehr mächtiges Werkzeug. Die Verwendung von Fakes [2] ist ab Visual Studio 2012 Premium Update 2 möglich. Für jede hinzugefügte Referenz lässt sich ein Fake erstellen, in dem dann Methoden überschrieben werden können.

PEX [3] ist ein Testgenerator, der Tests mit sinnvollen Werten für alle Eingangsparameter einer Methode generiert und die erstellten Tests durchführt. PEX ist bis Visual Studio Version 2010 enthalten. In die neueren Versionen wurde es nicht mehr aufgenommen. „Smart Unit Tests" ist der Nachfolger von PEX und Bestandteil von Visual Studio 2015.

Moq

Mock-Objekte sind Stellvertreter (Proxy-Objekte) für echte Objekte auf Basis eines Interface. Sie liefern statt Echtdaten vom Entwickler zuvor festgelegte Werte.

Um Tests unabhängig gestalten zu können, müssen zum Beispiel zuvor aus einer Datenbank geladene Objekte nachgebaut werden. Nur bedeutet das Nachbauen von großen, realen Datenstrukturen sehr viel Aufwand und ist fehleranfällig. Das Framework Moq erlaubt es, Objekte ohne Logik zu erstellen, die nur im Vorhinein definierte Daten enthalten und zurückliefern. Instanzen dieser Objekte können in Tests dafür genutzt werden, Methoden ohne Zugriff auf die Datenbank zu testen.

In Listing 1 wird ein generisches Mock-Objekt angelegt. Das Mock-Objekt wird mit einem Interface parametrisiert und liefert am Ende ein Objekt, das dieses Interface implementiert. Somit kann das Mock-Objekt für alle Methoden verwendet werden, die ein Objekt des Interfaces verlangen.
Mit der Methode *Setup* wird über eine Lambda-Expression

- eine Methode,
- ein Property,
- ein Event
- oder ein Callback

des Interface angegeben und anschließend mit der Methode *Returns* der Rückgabewert spezifiziert. Das Property *Object* liefert das fertige gemockte Objekt. Beispielsweise ergibt die Eigenschaft *LocationID* des Objekts aus Listing 1 nun immer den Wert 255.

Durch dieses Vorgehen können – angefangen von einfachen Objekten bis hin zu komplexen Baumstrukturen – nachgebildet werden, die normalerweise beim Zugriff auf die Datenbank oder durch eine Berechnungslogik entstehen. Ein Mocking-Framework ist besonders geeignet für

- schwer reproduzierbare Zustände (zum Beispiel Netzwerkfehler)
- langsame Methoden (Serviceaufrufe, Berechnungen)
- Objekte, die zur Testzeit nicht existieren

Vorteile, die das Framework Moq bietet:

- einfach zu nutzen
- stark typisiert (keine magischen Strings)
- Refactor-freundlich
- bei voller Funktionalität sehr minimalistisch

Fakes

Fakes ist ein von Microsoft zur Verfügung gestelltes Framework, mit dem es möglich ist, die Implementierung von Methoden zu überschreiben. Fakes löste das ältere Framework

Listing 2: Faked DateTime.Now

```
using Microsoft.QualityTools.Testing.Fakes;
using System.Fakes;

namespace CIT.DotNetPro.Fakes.Test
{
  [TestClass]
  public class FakesTest
  {
    [TestMethod]
    public void FakeDateTimeNowTest()
    {
      int year = 1960;
      using (ShimsContext.Create())
      {
        // DateTime.Now return always 01.01.1960
        ShimDateTime.NowGet = () =>
        {
          return new DateTime(year, 1, 1);
        };

        Assert.AreEqual(year, DateTime.Now.Year);
      }

      // DateTime.Now returns current Date
      Assert.AreEqual(2015, DateTime.Now.Year);
    }
  }
}
```

Listing 3: Einfacher Smart Unit Tests-Test

```
public double GetValueOfCompany(Company company)
{
  return company.CalculateValue();
}
```


Smart Unit Tests - stopped

SmartUnitTestTest.DistributeIncon · Views Tests · All Tests · 0 Warnings

5 ✔ 1 ✖ 17/17 blocks, 0/0 asserts, 6 runs

	target	locationCo...	message	result(target)	Summary/Exception	Error Message
1	new SmartUnitTestTest()	0	null	new SmartUnitTestTest()		
2	new SmartUnitTestTest()	1	null	new SmartUnitTestTest()		
3	new SmartUnitTestTest()	5	null	new SmartUnitTestTest()		
4	new SmartUnitTestTest()	6	null	new SmartUnitTestTest()		
5	new SmartUnitTestTest()	50	null		NotImplementedException	TODO
6	new SmartUnitTestTest()	25	null	new SmartUnitTestTest()		

Ergebnis komplexer Smart Unit Tests-Test (Bild 2)

Moles ab. Durch das Überschreiben kann der zu testende Teil der Logik isoliert werden, indem andere Bereiche der Anwendung (Methoden) durch Stubs oder Shims ersetzt werden. Fakes ist nur für die Visual-Studio-Versionen Premium und Ultimate verfügbar. In Listing 2 ist ein Beispiel für die Funktionalität von Fakes angeführt. Zuerst müssen für die Referenz (Assembly), in der sich die Klasse mit der zu überschreibenden Methode befindet, Fakes erstellt werden.

Im Solution Explorer kann im Kontextmenü der Referenz der Punkt *Add Fakes Assembly* ausgewählt werden. Visual Studio kompiliert anschließend eine mit Fakes versehene Version der Assembly. Zusätzlich wird eine Fakes-Datei erstellt, in der spezifiziert werden kann, welche Typen und Interfaces generiert werden sollen. Innerhalb des Shims-Kontexts können nun die Implementierungen von Methoden und Properties überschrieben werden. Diese Änderungen sind nur innerhalb des aktuellen Kontexts gültig. In Listing 2 ist zu sehen, dass sich der Wert von *DateTime.Now* innerhalb des Kontexts auf den vorher überschriebenen Wert ändert.

Der zweite Aufruf befindet sich außerhalb des Kontexts, greift auf die originale Implementierung zu und liefert deshalb das aktuelle Datum.

Das Fakes-Framework hat einige Vorteile, zum Beispiel

- ist es ein leichtgewichtiges Framework,
- besitzt es keine komplizierte Syntax und
- ist es gut geeignet, wenn mit Legacy-Code gearbeitet wird.

Das Framework enthält zwei verschiedene Arten von Konstrukten: Stubs und Shims.

Stubs werden bei Interfaces genutzt und können nur das Verhalten von öffentlichen Methoden verändern. Shims dagegen können das Verhalten jeder Methode verändern, wobei es egal ist, ob es sich um eine statische, eine öffentliche oder eine private Methode handelt. Shims sind langsamer, da sie zur Laufzeit den Code überschreiben, wohingegen Stubs diesen Mehraufwand nicht haben.

Gefälschte (faked) Methoden behalten ihren ursprünglichen Namen nur selten, da der Methodenname mit den Typen der Eingangsparameter erweitert wird. In Listing 2 ist zu sehen, dass das Property *DateTime.Now* nun den Zusatz *Get* enthält, da in Wirklichkeit der Getter überschrieben wird. Besitzt die gefakte Methode Parameter, werden deren Typnamen der Reihe nach an den Methodennamen angehängt. Der Methode wird ein Delegat zugewiesen, in dessen Klammern die Eingangsparameternamen zu definieren sind. Ist die gefälschte Methode nicht statisch, ist der erste Parameter das Objekt, auf dem die Methode aufgerufen wird.

PEX und Smart Unit Tests

PEX ist ein Tool von Microsoft, welches das White-Box-Testen unterstützt, indem es Unit-Tests automatisiert generiert und, wenn gewünscht, in ein Testprojekt exportiert. PEX generiert für alle Eingangsparameter eine Reihe von Eingabemöglichkeiten (zum Beispiel Extremwerte, Null etc.). Es hilft dem Tester insofern, als es die Eingangsparameter einer Methode mit diesen Werten befüllt und die Methode mit allen Varianten anschließend ausführt. ▶

Listing 4: Komplexer Smart Unit Tests-Test

```
public void DistributeIncomingMessageToLocation(
  int locationCode, string message)
{
  switch (locationCode)
  {
    case 0:
      RaiseIncomingMessageEvent(
        Location.management, message);
    break;
    case 5:
      RaiseIncomingMessageEvent(
        Location.development, message);
    break;
    case 25:
      RaiseIncomingMessageEvent(
        Location.acCOUNTing, message);
    break;
    case 50:
      throw new NotImplementedException("TODO");
    default:
      RaiseIncomingMessageEvent(
        Location.postOffice, message);
    break;
  }
}
```

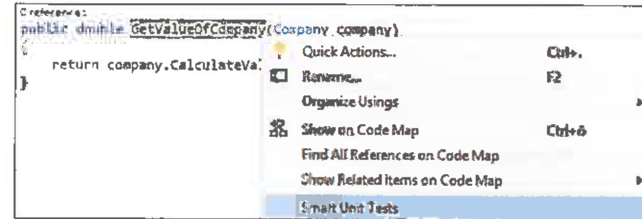
Der Smart Unit Tests-Prozess hat PEX in Visual Studio 2015 abgelöst. Es wird ähnlich wie bei PEX die ausgewählte Methode analysiert und basierend darauf werden Testdaten und eine Menge an Unit-Tests generiert.

Für jede ausgewählte Methode im Code werden wieder Eingangsparameter für die Methoden generiert und diese Tests anschließend ausgeführt. Zum Schluss wertet die Software alle Eingabewerte und das Verhalten der Methoden aus und zeigt sie an. Sollten Probleme auftreten, kennzeichnet die Software die einzelnen Tests mit einem roten Symbol, bei Erfolg mit einem grünen.

In Listing 3 wird ein einfacher Berechnungsprozess getestet. Der Methode *GetValueOfCompany* wird ein *Company*-Objekt übergeben. Dieses Objekt besitzt eine Methode zur Berechnung des Unternehmenswerts. Mit Smart Unit Tests werden Unit-Tests mit verschiedenen Eingangswerten erstellt. Wie in Bild 1 zu sehen ist, generiert der Smart Unit Tests-Prozess zwei Werte für den Eingangsparameter *company*. Im ersten Fall ist der Wert null und es wird eine Ausnahme beim Zugriff auf die Methode *GetValueOfCompany* geworfen. Mit dem roten Symbol wird der Test als fehlgeschlagen gekennzeichnet. In der Spalte für Fehlermeldungen erhält der Tester sofort die dementsprechenden Informationen über die fehlgeschlagenen Tests. Beim zweiten Test wurde automatisch ein neues Company-Objekt angelegt und die Methode darauf hin erfolgreich ausgeführt.

In Listing 4 wird der große Vorteil von Smart Unit Tests sichtbar. Der Smart Unit Tests-Prozess generiert nicht einfach zufällige Eingangswerte für eine Methode, sondern analysiert die Programmlogik und generiert dafür entsprechende Unit-Tests. Werden zum Beispiel Bedingungen (*if*, *switch* etc.) gefunden, wird für jede Bedingung mindestens ein Testfall generiert. In Listing 4 befinden sich fünf Bedingungen in Form eines *Switch-Case*-Konstrukts.

Bild 2 zeigt die sechs von Smart Unit Tests generierten Unit-Tests. Für jede Bedingung hat die Software einen Testfall generiert. Einer der Testfälle wurde als fehlgeschlagen gekennzeichnet, da mit dem generierten *locationCode* eine Ausnahme ausgelöst wurde.



Starten des Smart Unit Tests-Tool (Bild 3)

Smart Unit Tests-Dialog zeigt Ergebnisse

Smart Unit Tests können über das Kontextmenü der jeweiligen Methode unter dem Punkt *Smart Unit Tests* generiert werden (siehe Bild 3). Es öffnet sich der Smart Unit Tests-Dialog. In diesem Dialog werden die generierten Eingangsparameter und die Ergebnisse der Methode angezeigt. Wenn eine Ausnahme geworfen wird, gilt der Test als fehlgeschlagen. Über den Button *Speichern* können die parametrisierten Unit-Tests in einem Testprojekt gespeichert werden und stehen für die spätere Verwendung zur Verfügung.

Architektur

Das 3-Schichten-Modell ist eines der verbreitetsten architektonischen Modelle in der Softwareentwicklung. Die unterschiedlichen Schichten eines Softwaresystems werden logisch voneinander getrennt (siehe Bild 4, 3-Schichten-Modell). Die Software wird in Datenzugriff-, Logik- und Präsentationsschicht unterteilt.

Die Datenzugriffsschicht ist verantwortlich für das Laden und Speichern von Daten. Die Geschäftslogik beinhaltet Verarbeitungsmechanismen, in denen die Daten aus der Datenzugriffsschicht und der Präsentationsschicht validiert und weiterverarbeitet werden.

Die Präsentationsschicht ist für die Darstellung der Daten und die Benutzerinteraktion zuständig.

Bei sauberer Umsetzung ermöglicht das 3-Schichten-Modell hohe Skalierbarkeit und Flexibilität, verbessert die Wartbarkeit und spart so auf lange Sicht Zeit und Geld. Solange sich die Schnittstellen nicht ändern, können bei Einhaltung der Architektur ganze Schichten ausgetauscht werden (zum Beispiel Umstieg von Datenbank auf Webservices etc.).

Um in den oberen Schichten nicht von den Datenobjekten der Datenzugriffsschicht abhängig zu sein, werden sogenannte Datentransferobjekte (DTOs) verwendet.

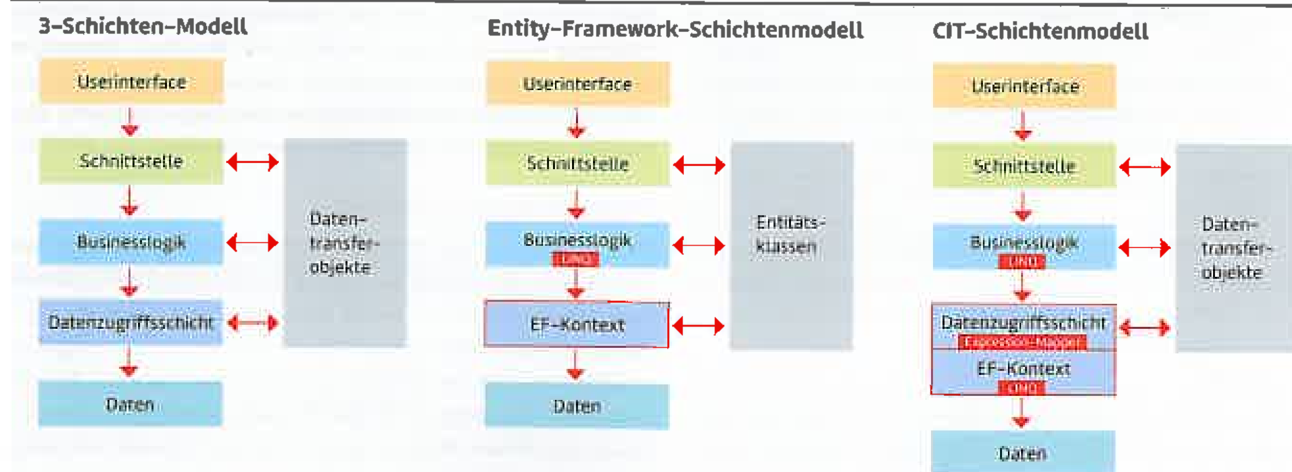
Bei Änderungen der Datenstrukturen oder der Datenbank müssen diese Änderungen jedoch mindestens bis zu den Datentransferobjekten nachgezogen werden und bedeuten somit einen gewissen Aufwand.

Mit dem Entity Framework (EF) von Microsoft wurde den Softwareentwicklern einiges an Arbeit abgenommen.

Das EF ist eine objektrelationale Zuordnung, die Entwicklern über domänenspezifische Objekte (Entitätsklassen) die Nutzung relationaler Daten ermöglicht. Ein Großteil der Datenzugriffsschicht wird vom EF übernommen (siehe Bild 4, Entity-Framework-Schichtenmodell) und muss daher nicht mehr selbst implementiert werden. Ein weiterer Vorteil ist, dass über Linq ein komfortables Laden der Daten ermöglicht wird und Datenzugriffsklassen zum Transport der Daten innerhalb der Datenzugriffsschicht angeboten werden.

Die Verwendung dieser Datenzugriffsklassen außerhalb der Datenzugriffsschicht führt allerdings zu einer Vermischung der Schichten und die lose Kopplung zwischen Geschäftslogik und Datenbank wird gefährdet. Mit den Vorteilen kommen also auch einige Nachteile daher. Das EF verleitet dazu, die Schichten zu vermischen. Der daraus resultierende Architekturbruch hebt die Vorteile auf, die das 3-Schichten-Modell mit sich bringt. Die generierten Datenzugriffsklassen implementieren außerdem keine Interfaces und machen das Mocken des Datenbankzugriffs unmöglich.

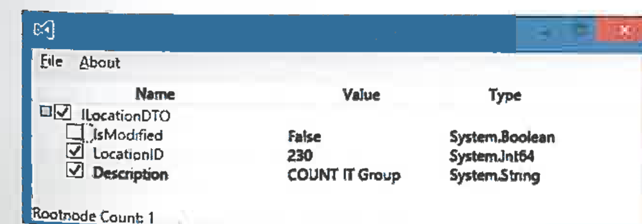
Architekturmuster (Bild 4)



Das CIT-Schichtenmodell

Im dritten Architekturmuster (siehe Bild 4, CIT-Schichtenmodell) wird eine zusätzliche Datenzugriffsschicht eingezogen. Sie ist ähnlich der Architektur im Artikel *Aufgeschichtet* [4]. Dieses Architekturkonzept erlaubt die Nutzung aller Vorteile, die das EF mitbringt, und sorgt für eine saubere Trennung der einzelnen Schichten. Außerdem werden in diesem Konzept die Daten nicht mehr mit den Datenzugriffsklassen aus der Datenschicht transportiert, sondern verwenden darauf aufgesetzte Datentransferobjekte (DTOs).

Die DTO-Klassen werden als partielle Klassen generiert und können sauber über eine weitere partielle Klasse getrennt von der Standardimplementierung erweitert werden. Bei Änderungen der Datenbank können die Standardimplementierungen der DTOs einfach neu generiert werden, die eigenen Erweiterungen bleiben unangetastet.



Auswahl der Properties und Fields für das gemockte Objekt (Bild 5)

Jede DTO-Klasse implementiert ein spezifisches Interface, welches dem CIT-Testtool erlaubt, die Klassen zu mocken.

Eine weitere Besonderheit dieses Konstrukts ist, dass in die Datenzugriffsschicht ein Expression-Mapper integriert ist. Dieser Mapper erlaubt, die Linq-Abfragen der Businesslogik von den DTO-Klassen auf die EF-Datenzugriffsklassen zu übersetzen, sodass sie direkt an die Datenzugriffsschicht übergeben werden können.

Mit dem CIT-Testtool können die Entwickler eine Methode auswählen und das Tool generiert in kürzester Zeit die Datenstrukturen für einen Unit-Test dazu, wobei die Daten über eine Benutzeroberfläche auf Basis der jeweiligen Datenquelle (Datenbank oder Webservice) festgelegt werden können.

Entwicklertests

Bei neu implementierten Komponenten werden oftmals Entwicklertests durchgeführt. Dafür werden Testdaten in die Datenbank gespeichert. Das CIT-Testtool greift auf die Datenbank zu und generiert damit gemockte Datenstrukturen mit den gewünschten Werten.

Die relevanten Properties und Felder kann der Softwareentwickler einfach auswählen (siehe Bild 5), anschließend wird daraus ein Mock-Objekt generiert.

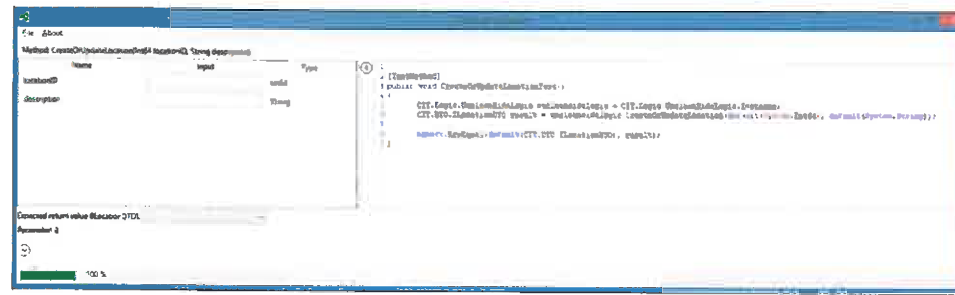
Das Ergebnis kann entweder als externe Methode in ein Testprojekt oder direkt in einen neuen Unit-Test übernommen werden. Durch diese Funktionalität wird dem Soft- ▶

SIGS DATACOM
Seminare
Bei uns lernen Sie von den Experten!
Seminar
Training
Schulung
Die Clean Code Developer Akademie oder
Wie können Teams Software von höherer Qualität entwickeln?
Von und mit den Experten: Ralf Westphal & Stefan Lieser
Die Seminare:
- **Clean Architecture**
  Agiler Softwareentwurf für mehr Effizenz
- **Clean Coding**
  Funktionaler Softwareentwurf für mehr Nachhaltigkeit

Diese aktuellen Seminare spiegeln die Erfahrungen der Trainer getreu dem Clean Code Developer Wert „Kontinuierliche Verbesserung“ und der Praktik „refactoring to deeper insight“ wieder!
Termine, Informationen und mehr unter:
www.sigs-datacom.de/seminare

Kontakt: SIGS DATACOM GmbH, Anja Keß
Lindlaustraße 2c, D-53842 Troisdorf
Tel.: +49 (0) 22 41 / 23 41-201
Fax: +49 (0) 22 41 / 23 41-199, Email: anja.kess@sigs-datacom.de

wareentwickler bereits ein großer Teil der Arbeit beim Erstellen eines Tests abgenommen. Der Vorteil bei der Erstellung einer externen Methode, die ein generiertes Mock-Objekt zurückgibt, ist, dass die selbe Methode durchaus in mehreren Testfällen genutzt werden kann.



CIT-Testtool (Bild 6)

Weitere Features

Wenn benötigt, kann das CIT-Testtool die Testfälle für den Softwareentwickler datenbank- und webserviceunabhängig gestalten. Das realisiert das CIT-Testtool, indem es alle Methoden, die auf die Datenbank oder einen Webservice zugreifen, mit Fakes überschreibt. Das gewünschte Ergebnis kann danach einfach aus den zuvor generierten Mock-Objekten ausgewählt werden. Durch dieses Zusammenspiel der einzelnen Komponenten erhält man in kurzer Zeit einen sehr umfangreichen und gut strukturierten Test.

Intern benutzt das CIT-Testtool einen der Erweiterungspunkte von Visual Studio, den sogenannten Visualizer. Ein Visualizer ist ein kleines Programm, das die Möglichkeit bietet, im Debugging-Modus von Visual Studio ein Objekt anzuzeigen. Es gibt bereits einige Standardvisualizer (Text, XML, JSON etc.). In diesem Fall wird vom CIT-Testtool ein DTO visualisiert, das die Basis für die späteren Mock-Objekte bildet.

Wichtig: Generierung von Unit-Tests

Eine der primären Anforderungen an das CIT-Testtool ist die Generierung von Unit-Tests, ohne den produktiven Code vorher verändern zu müssen. Diese Anforderung kann durch die Visualizer-Funktion erfüllt werden.

Das CIT-Testtool lässt sich neben den DTOs auch über den reflektierten Datentyp von Methoden (*MethodInfo*) aufrufen. Während des Debuggens kann über das Watch- oder das Quick-Watch-Fenster eine *MethodInfo* geladen und anschließend mit dem Tool visualisiert werden, um Tests dafür zu generieren.

Im oberen Abschnitt von Bild 6 wird die Methode angezeigt. Dabei erhält der Entwickler Informationen über den Methodennamen und die Typen der Eingangsparameter und deren Namen. Darunter werden die Eingangsparameter noch einmal extra mit allen Informationen dargestellt.

Über die Spalte *Input* können schließlich die Eingangsparameter festgelegt werden.

Texteditor mit Syntax-Highlighting

In der rechten Hälfte des CIT-Testtools befindet sich ein Texteditor mit Syntax-Highlighting. Dieser Texteditor ist über ein separates Projekt gelöst, das das Highlighting asynchron durchführt [5].

Sind die Eingangsparameter nicht festgelegt, wird der Standardwert des Typs mit der in .NET vorhandenen Funktion *default(T)* im Unit-Test übergeben (siehe Bild 6 rechts). Sobald ein Eingangsparameter festgelegt wird, aktualisiert sich der Texteditor automatisch.

Ist einer der Eingangsparameter ein DTO, wird in der gesamten Solution nach Methoden mit einem speziellen Attribut (MockTree-Attribut) gesucht. Stimmt der Typ des Rückgabewerts der Methode mit dem des Eingangsparameters überein, wird der Methodenname in einer Drop-down-Box als mögliche Auswahl angeboten.

Das Attribut *MockTree* wird zur Kennzeichnung von Methoden verwendet, die gemockte DTO-Objekte zurückgeben. Bei diesen Methoden handelt es sich um die weiter oben erwähnten generierten Methoden.

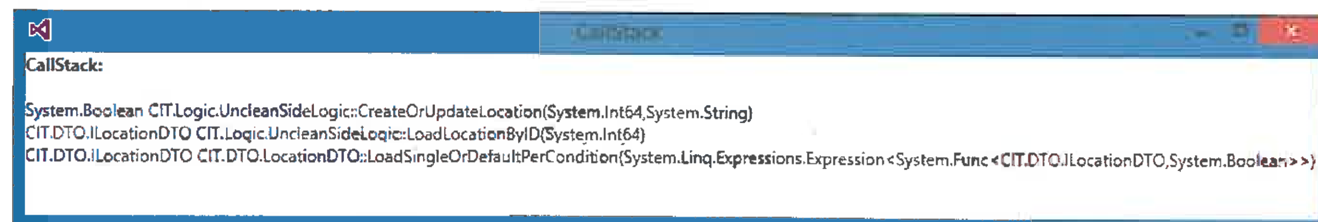
Dasselbe Prinzip gilt mit einer Besonderheit auch für den Rückgabewert. Ist der erwartete Rückgabewert eine DTO-Klasse, kann einerseits aus den gefundenen MockTree-Attribut-Methoden gewählt oder andererseits ein Mock-Objekt eigens für den Test generiert werden.

Dieses Objekt kann entweder erstellt werden, indem man den bereits generierten Test im Hintergrund ausführt und das Ergebnis des Tests verwendet, oder es können über einen eigenen Dialog die Schlüssel eines in der Datenquelle (Datenbank oder Webservice) vorhandenen Eintrags eingegeben werden, aus dem das Objekt generiert wird.

Dafür baut das CIT-Testtool eine Verbindung zur Datenquelle auf, lädt die Daten des Eintrags und verpackt diese in ein DTO. Dieses geladene Data Transfer Object wird als Baumstruktur dargestellt, wobei das Wurzelelement das DTO-Objekt selbst bildet. Die Äste darunter sind die Properties und Felder des Objekts (Bild 5).

Jedes ausgewählte Property wird mit dem dazugehörigen Wert später in das resultierende Mock-Objekt übertragen. Enthält ein Ast wieder ein DTO-Objekt oder eine Liste, lässt sich dieser Ast weiter auffächern, das Nachladen der Daten erfolgt automatisch und diese können ausgewählt werden. Auf diese Weise kann ein ganzer DTO-Baum oder eine Liste von DTOs für den Unit-Test erstellt werden.

Die so erstellten Mock-Objekte können wiederum entweder direkt in den Unit-Test generiert oder in eine Methode mit dem MockTree-Attribut ausgelagert werden.



CallStack (Bild 7)

Liste aller Webservice- und Datenbankzugriffe

Der untere Teil des Fensters listet alle Webservice- und Datenbankzugriffe auf. Die Methoden werden mit demselben Prinzip gefunden, mit dem auch ILSpy [6] arbeitet.

ILSpy dekompiliert .NET-Binärdateien zu lesbarem Code. Das wird durch die Bibliothek Cecil ermöglicht, die es erlaubt, existierende Assemblys zu dekompilieren, alle enthaltenen Typen zu betrachten, diese zu modifizieren und in modifizierter Form wieder zu speichern [7]. Außerdem erlaubt es Cecil, die Implementierung von Methoden einzusehen. Aufgrund des CIT-Schichtenmodells kann man erkennen, in welcher Assembly sich die Zugriffsmethoden befinden. Während des Durchlaufens der Methodenaufrufe wird ein Callstack (siehe Bild 7) aufgebaut, damit der Softwareentwickler die genaue Position kennt, an der die Zugriffsmethode aufgerufen wird.

In der Liste werden alle Methoden mit Datenbank- oder Webservicezugriff aufgeführt. Wie bei den normalen Logik-Methoden kann auch hier der Rückgabewert der Methoden festgelegt werden. Die Spalte „Parameter" zeigt die Parameterwerte für die Zugriffsmethode an. Diese werden im späteren Verlauf für das Fälschen der Methode verwendet. Die Werte für diese Spalte können allerdings nur ausgelesen werden, indem der Testfall durchgeführt wird. Listing 5 zeigt eine Methode, für die ein datenbankunabhängiger Test (Listing 6) erstellt werden soll.

Search Parameters

Caller	Method	Parameter	CallStack	Return Value	Return Type
LoadCustomerPerID	LoadCustomerPerID	Keys ◢ Parameters customerid = 1	CallStack	CIT.DotNetPro.Mo	CIT.DotNetPro.Domai
LoadCustomerPerID	LoadCustomerPerID	Keys ◢ Parameters customerid = 8	CallStack	CIT.DotNetPro.Mo	CIT.DotNetPro.Domai
LoadCustomerPerID	LoadCustomerPerID	Keys ◢ Parameters customerid = 200	CallStack	CIT.DotNetPro.Mo	CIT.DotNetPro.Domai
GetLocationIDPerCust	LoadLocationID	Keys ◢ Parameters customer Custome	CallStack	25	System.Int32

Zugriffsmethoden (Bild 8)

Das CIT-Testtool generiert einen Test für die *DotNetPro*-Methode. Um den Test datenbankunabhängig zu gestalten, werden für alle Methoden, die auf die Datenbank zugreifen, Fakes der Methoden verwendet.

Im ersten Schritt durchsucht das CIT-Testtool die zu testende Methode und alle aufgerufenen Methoden. Entdeckt es eine Methode, die auf eine Datenbank oder einen Webservice zugreift, nimmt es sie in eine Liste auf (siehe Bild 8).

Für die Haupttestmethode (in diesem Fall *DotNetPro*) müssen zuerst alle Eingabeparameter festgelegt werden. In der Methode *DotNetPro* befinden sich zwei unterschiedliche Zugriffsmethoden. Die Methode *LoadCustomerPerID* wird insgesamt dreimal im Code aufgerufen, während die Methode *GetLocationIDPerCustomer* nur einmal vorkommt (Listing 5). Die Liste der Zugriffsmethoden zeigt diese vier Zugriffe nun an. Anschließend können die Rückgabewerte der Zugriffsmethoden einzeln festgelegt werden.

Nach dem Klick auf *Search Parameters* generiert das Tool den Test und erweitert die Zugriffsmethoden mit Fakes, sodass die Eingangsparameter dieser Methoden an das CIT-Testtool übergeben werden können.

Wurde der Test durchgeführt, stellt das CIT-Testtool die Eingangsparameter der Zugriffsmethoden in der Oberfläche zur Verfügung. Anschließend wird ein datenbankunabhängiger Test generiert, indem für die Zugriffsmethoden wieder Fakes verwendet werden. In diesen gefakten Zugriffsmethoden werden entsprechende Bedingungen generiert, um das exakte Verhalten der Methode auch ohne Datenbank- und Webservicezugriff zu simulieren. Abschließend kann der neu generierte Unit-Test wieder in ein Testprojekt oder in die Zwischenablage gespeichert werden. Zusätzlich wird der generierte Test in einem Editor-Fenster angezeigt (Bild 6).

Fazit

Unit- und Integrationstests können schnell auf Fehler hinweisen, dazu müssen sie jedoch konsequent erstellt und ausgeführt werden. Den großen Aufwand für die Erstellung der Tests versucht man durch einen halbautomatischen Prozess zu verringern. Diverse Frameworks und Tools am Markt erleichtern das Erstellen von Datenstrukturen und Testfällen. Einige davon ermöglichen auch das Erstellen von datenbankunabhängigen Tests.

Durch eine Kombination diverser vorhandener Tools ist das CIT-Testtool entstanden, das die benötigte Zeit zur Erstellung von Tests massiv reduziert, ohne dabei die Qualität zu vermindern.

Genauso wichtig wie eine hohe Testabdeckung ist eine saubere Architektur. Es sollte das 3-Schichten-Modell (beziehungsweise eine Adaption davon) eingehalten werden, um die Software gut strukturiert und wartbar zu halten.

Sind die Schichten zu stark miteinander verwoben, ist ein sehr hoher Aufwand zu treiben, um eine Schicht abzuändern oder gar auszutauschen.

Aus diesem Grund hat die COUNT IT Group eine zusätzliche Schicht zum EF-Kontext eingezogen. Diese Schicht er- ▶

Listing 6: Gefälschter Test (Fake)

```
[TestMethod]
public void DotNetProTest()
{
  using (ShimsContext.Create())
  {
    CIT.DotNetPro.Domain.Fakes.ShimCustomerDTO
      .LoadCustomerPerIDInt32 =
    (System.Int32 customerid) =>
    {
      if (customerid == 1)
      {
        return CIT.DotNetPro.MockTrees
          .MockedCustomer_1();
      }
      else if (customerid == 8)
      {
        return CIT.DotNetPro.MockTrees
          .MockedCustomer_8();
      }
      else if (customerid == 200)
      {
        return CIT.DotNetPro.MockTrees
          .MockedCustomer_200();
      }
      else
      {
        return default(CIT.DTO.Domain
          .CustomerDTO);
      }
    };

    CIT.DotNetPro.Domain.Fakes.ShimProgram
      .GetLocationIDPerCustomerICustomerDTO =
    (CIT.DotNetPro.Domain.ICustomerDTO customer) =>
    {
      if (customer.CustomerID == 200)
      {
        return 25;
      }
      else
      {
        return default(System.Int32);
      }
    };

    CIT.DotNetPro.Domain.Program.DotNetPro();
  };
}
```

laubt eine saubere Implementierung des 3-Schichten-Modells, wodurch isolierte Tests einfach geschrieben werden können. Mit dem CIT-Testtool werden automatisierte datenbank- und webserviceunabhängige Tests inklusive der benötigten Datenstrukturen generiert.

Listing 5: Zu testende Methode

```
public static void DotNetPro()
{
  ICustomerDTO customer1 =
    CustomerDTO.LoadCustomerPerID(1);
  ICustomerDTO customer8 =
    CustomerDTO.LoadCustomerPerID(8);
  ICustomerDTO customer200 =
    CustomerDTO.LoadCustomerPerID(200);
  int locationID =
    GetLocationIDPerCustomer(customer200);
}

public static int GetLocationIDPerCustomer(
    ICustomerDTO customer)
{
  return DTOHelper.LoadLocationID(
    customer.CustomerID);
}
```

Smart Unit Tests bilden die ideale Ergänzung zum CIT-Testtool, da die generierten Whitebox-Tests dem Entwickler auf Knopfdruck ein erstes frühes Feedback zur Verfügung stellen. ■

[1] Mocking-Framework Moq, https://github.com/Moq/moq4

[2] Fakes VS Versionen, http://visualstudio.uservoice.com

[3] PEX, visualstudiomagazine.com

[4] Aufgeschichtet, Dr. Holger Schwichtenberg, dotnetpro 7/2014, S. 96ff, www.dotnetpro.de/A1407DataAccess

[5] Texteditor, syntaxhighlightbox.codeplex.com

[6] ILSpy, http://ilspy.net

[7] Mono Cecil, http://www.mono-project.com

Markus Szöky
ist Abteilungsleiter, .NET-Softwarearchitekt und agiler Projektleiter bei der COUNT IT Group aus dem Softwarepark Hagenberg, die maßgeschneiderte ERP-, DMS-, Individualsoftware auf Basis von .NET bietet. Sie erreichen ihn über **m.szoeky@countit.at** oder **www.countit.at.**

dnpCode A1506Testautomation