

Mario Rimann

Green Bar Feeling bei TYPO3-Extensions

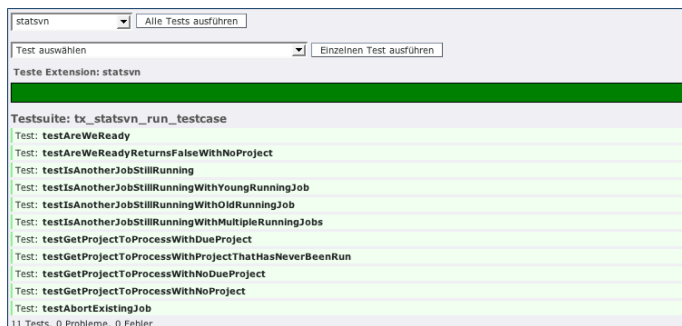
Sauberer Code durch Unit Testing

Prinzipiell sind TYPO3-Extensions auch „nur“ PHP-Code, der sich deshalb durch Unit Testing überprüfen lässt. Und doch gibt es gewisse Unterschiede zwischen einer eigenständigen Applikation und einer Extension für das TYPO3-Framework. Dieser Artikel zeigt, wie man Extensions effektiv testet und dabei mit den richtigen Tools viel Aufwand spart.

Nach der allgemein gehaltenen Einleitung zu Unit Testing (separater Artikel in diesem Heft) liegt der Fokus dieses Artikels nun speziell auf Unit Testing im Zusammenhang mit TYPO3. Das Testen von Extensions birgt besondere Schwierigkeiten: Extensions sind meist ohne TYPO3 nicht lauffähig. Damit eine Extension getestet werden kann, muss das Testumfeld die Funktionen des TYPO3-Frameworks so drapieren, dass die zu testenden Klassen darauf zugreifen können. Dazu kann beispielsweise die Extension „tx_phpunit“ [1] verwendet werden.

PHPUnit als Extension

Ein erster Anlauf, PHPUnit mit einer brauchbaren Oberfläche in die Extension „tx_t3unit“ zu kapseln, scheiterte vor einiger Zeit am Umfang des Projekts. Im Jahr 2007 begann der Däne Kasper Ligaard mit der Arbeit an seiner „tx_phpunit“ getauften Extension. Dabei baute er die bestehende Extension von Robert Lemke weiter aus und passte die Struktur an die Version 3.x. von PHPUnit an. In den vergangenen Monaten erschienen die neuen Versionen zeitweise im Wochentakt.



Das Backend-Modul zeigt übersichtlich den Verlauf der einzelnen Tests.

Die Extension bietet ein Backend-Modul, in dem Tests aller installierten Extensions insgesamt, pro Extension oder einzeln ausgeführt werden können. Momentan ist dieser Umfang allerdings nur ein Zwischenstand: Kasper Ligaards längerfristiges Ziel ist es, das gesamte TYPO3-Framework testbar zu machen. Da sich die Ausweitung der Testumgebung aber bisher schwieriger gestaltet als erwartet, wurde zunächst die Testbarkeit von Extensions umgesetzt. In Zukunft soll es möglich sein, die Tests nicht nur „live“ im Browser auszuführen, sondern auch auf einem separaten Server. Der Weg führt in Richtung Continuous Integration, nach Angaben des Entwicklers wird die Umsetzung aber noch auf sich warten lassen: Schwierigkeiten beim Testen von Extensions gebe es beispielsweise beim gesamten Frontend-Output, bei der TypoScript-Konfiguration und bei ausgehenden Mails. Die Frontend-Ausgabe kann bisher durch ein simuliertes Frontend getestet werden; TypoScript-Setup, ausgehende Mails und weitere Funktionen deckt das Testing-Framework in der Extension „tx_oelib“ ab.

Schritt für Schritt zur getesteten Extension

Um eine bestehende oder gerade neu angelegte Extension mit Unit Tests auszurüsten, sind nur wenige Schritte nötig. Obwohl hier die häufigsten Stolperfallen genannt werden, wird kein Entwickler um ein wenig „Trial & Error“ herumkommen.

Die zu testende Extension braucht ein Verzeichnis „tests“. Falls benötigt, kann darin auch gleich noch ein Unterverzeichnis „fixtures“ angelegt werden. Für jede zu überprüfende Klasse wird ein Test-Case als einzelne PHP-Datei in das erstellte „tests“-Verzeichnis gelegt. Diese Klassen werden später die einzelnen Test-Funktionen beinhalten.

Aufbau eines einfachen Test-Cases

```
class tx_extensionname_klassenname_testcase extends tx_phpunit_testcase {
protected function setUp() {
    // Für die Fixture – nur wenn benötigt.
}
protected function tearDown() {
    // Ebenfalls für die Fixture – nur wenn benötigt.
}
public function testSomeFunctionDoesSomething() {
    // Test Funktionalität
}
public function testSomeOtherFunctionDoesSomething() {
    // Test Funktionalität
}
}
```

Listing 1

Bei diesem Grundgerüst ist darauf zu achten, dass die Klasse sich von „tx_phpunit_testcase“ ableitet. Als Testfunktionen werden ausschließlich die Funktionen erkannt, deren Funktionsname mit „test“ beginnt und die als „public“ markiert sind. Der Aufruf des Backend-Moduls PHPUnit lässt nun bereits die Auswahl und den Durchlauf des Test-Case zu. Das ist natürlich erst dann sinnvoll, wenn die oben gezeigten Testmethoden auf vorhandenen Code, wie beispielsweise eine bestehende Klasse, angewendet werden.

Noch ein Framework

Falls die zu testende Klasse ausschließlich statisch aufrufbare Funktionen zur Verfügung stellt, ist das Testen relativ einfach, da während der Testdurchführung weder Objekte instantiiert werden noch Datensätze aus einer Datenbank beteiligt sind. Im Gegensatz dazu sind TYPO3-Extensions immer häufiger objektorientiert aufgebaut und greifen zum Großteil auf eine Datenbank zu.

Fixtures [2] bieten einen einfachen Weg, die benötigten Objekte zu instantiiieren, und ein kleines Testing-Framework für die benötigten Dummy-Datensätze in der Datenbank erleichtert die

Arbeit. Die Framework-Klasse ist in der Extension „tx_oelib“ zu finden und bietet unter anderem folgende Funktionalitäten:

- Erstellen, Modifizieren und Löschen von Dummy-Datensätzen
- Erstellen und Löschen von Relationen zwischen Datensätzen
- Ein- und Ausloggen von Dummy-Usern
- Automatisiertes Aufräumen aller Dummy-Datensätze
- Fake-Mailer zum Testen von ausgehenden E-Mails
- Konfigurations-Proxy zum Abrufen/Überschreiben von Einstellungen aus dem Extension-Manager
- Header-Proxy, um die Ausgabe von HTTP-Errorcodes zu testen
- Diverse Hilfsfunktionen

Diese Funktionen erleichtern bereits die Entwicklung von Tests, weitere Features, wie die Erstellung von Dummy-Dateien, sind bereits geplant.

Schreiten wir zur Tat

Das folgende Code-Beispiel zeigt einen realen Ausschnitt aus der „Seminar Manager“-Extension. Das Testobjekt ist die „Places“-Klasse, die einen Veranstaltungsort repräsentiert.

Zur Erzeugung einer Objektinstanz dieser Klasse ist ein Datensatz in der entsprechenden Tabelle nötig, der die Testdaten enthält. Vor dem Anlegen der Dummy-Datensätze bildet der Aufruf der Methode setUp() aber zunächst eine Instanz des Testing-Frameworks. Die Methode createRecord() legt dann einen Dummy-Datensatz an, mit dessen Daten das Testobjekt instanziiert wird und allen folgenden Testfunktionen als „\$this->fixture“ zur Verfügung steht.

Aufbaufunktion setUp()

```
private $fixture;
private $testingFramework;
public function setUp() {
    $this->testingFramework = new tx_oelib_testingFramework('tx_seminars');
    $suid = $this->testingFramework->createRecord(
        'tx_seminars_sites',
        array(
            'title' => 'TEST Place 1',
            'city' => 'Tokyo',
            'country' => 'JP'
        )
    );
    $this->fixture = new tx_seminars_place($suid);
}
```

Listing 2

Genau wie für den Aufbau der Testumgebung existiert auch eine Methode für deren Beendigung: Ziel der Funktion „tearDown()“ ist das Aufräumen nach dem Testen. Nach der Initialisierung des Aufräumvorgangs für das Testing-Framework zerstört die Methode unset() dann die erzeugten Objekte.

Aufräumfunktion tearDown()

```
public function tearDown() {
    $this->testingFramework->cleanup();
    unset($this->testingFramework);
    unset($this->fixture);
}
```

Listing 3

Nachdem das Grundgerüst nun aufgebaut wurde, fehlen in dem Beispiel noch einige Testfunktionen. Im folgenden Code ist zu erkennen, wie in den Tests auf die vorbereitete Fixture zugegriffen wird.

Zwei Test-Funktionen

```
public function testGetCity() {
    $this->assertEquals(
        'Tokyo',
        $this->fixture->getCity()
    );
}
public function testGetCountryIsoCode() {
    $this->assertEquals(
        'JP',
        $this->fixture->getCountryIsoCode()
    );
}
```

Listing 4

Bei nur zwei Tests wird der Vorteil der zentralen Erstellung von Fixtures nicht ganz so deutlich. Bei Klassen mit mehr als 100 Tests fällt es hingegen stark ins Gewicht, ob jeder einzelne Test selbst Dummy-Datensätze erstellen und Objekte instantiiieren muss oder ob diese Funktionen an zentraler Stelle geregelt werden.

TYPO3 und Unit Testing – eine Bestandsaufnahme

Bei den TYPO3-Extensions ist eine klare Bestandsaufnahme nicht möglich. Man kann aber davon ausgehen, dass nur die wenigsten Extensions mit Unit Tests „abgesichert“ sind. Bei den Entwicklerteams, die getestete Extensions veröffentlichen, können einzelne Extensions mehr als 500 Einzeltests beinhalten.

Den beiden Teamleitern von TYPO3 zufolge wird für den 4.x-Branch über die Einführung von Unit Tests nachgedacht – der einzige Hinderungsgrund ist der erwartete Initialaufwand bei der Einführung. In der ersten Ausgabe des T3N Magazins 2005 schrieb Robert Lemke in einem Artikel zu Unit Testing [3]: „Am besten verwendet man Unit Tests bereits bei der Entwicklung neuen Codes und nicht erst beim Refactoring.“ Der Satz bringt kurz und knapp auf den Punkt, dass nachträgliche Tests meistens mit erheblichem Aufwand einhergehen; wird von Anfang an jede Funktion getestet, hält sich der Zusatzaufwand dagegen in Grenzen.

Bei der Entwicklung von TYPO3 5.x und FLOW3 wurde von Beginn an konsequent auf Test Driven Development (TDD) gesetzt, dadurch erreichten die Entwickler eine hohe Code Coverage von über 70 Prozent. Laut Robert Lemke hat das Team ausschließlich positive Erfahrungen mit diesem Vorgehen gemacht und traf auf keine Akzeptanzprobleme bei den Entwicklern. Dafür dominierte das Green Bar Feeling, das die Entwicklung saubereren Codes und eine erhöhte Testfrequenz mit sich brachte.

Links und Literatur

 [Softlink 2054](#)

- [1] Extension zu PHPUnit: <http://typo3.org/extensions/repository/view/phpunit/>
- [2] Fixtures in Wikipedia: http://en.wikipedia.org/wiki/Test_fixture
- [3] Lemke, R. (2005): „Green Bar Feeling mit TYPO3“. T3N Magazin 01/2005, S. 37.

DER AUTOR



Mario Rimann wohnt in Dietikon bei Zürich. Hauptberuflich arbeitet er als System Engineer in einer Firma, die zwar viel mit PHP, aber noch nichts mit TYPO3 zu tun hat. Aus dem Hobby TYPO3 wurde eine Firma: Zusammen mit Sven Wächli gründete Mario die Screenteam GmbH. Neben der Umsetzung von Webprojekten mit TYPO3 spezialisierten sich beide auf Support und Extension-Entwicklung.