Wahlfach Softwarequalität
Dozent: Prof. Dr. Stephan Kleuker
WS 2010/2011

Abschlussaufgabe Variante C:

Evaluation des Software-Tools PHPUnit 3.5

David Sondermann

8. Februar 2011

Inhaltsverzeichnis

Name

PHPUnit 3.5

Homepage

https://github.com/sebastianbergmann/phpunit/
http://www.phpunit.de/manual/current/en/index.html

Lizenz

BSD-Lizenz

Untersuchte Version

PHPUnit 3.5.10 vom 19. Januar 2011

Letzter Untersuchungszeitpunkt

8. Februar 2011

Kurzbeschreibung

PHPUnit ist ein in PHP geschriebenes freies Framework zum Testen von PHP-Skripten, das besonders für automatisierte Tests einzelner Einheiten (Units, meist Klassen oder Methoden) geeignet ist. Es basiert auf dem xUnit-Konzept, welches auch für andere Programmiersprachen genutzt wird, wie zum Beispiel in JUnit für Java.

Fazit

PHPUnit ist mit Sicherheit ein lohnendes Werkzeug zur Qualitätssicherung von PHP-Projekten. Einsteiger werden in der Dokumentation sinnvoll an grundlegende Methoden und Konzepte von Software-Tests herangeführt. Erfahrenen Nutzern bietet PHPUnit komplexe Funktionen wie Benutzeroberflächen-Tests mit Selenium.

Meines Wissens existiert für PHP kein weiteres Tool für Software-Tests. In der Literatur wird immer wieder auf PHPUnit verwiesen. Ein Vergleich mit anderen Werkzeugen ist somit nicht möglich.

Einsatzgebiete

Mit PHPUnit lassen sich für PHP-Klassen, deren Quellcode vorliegt, Unit- und Whitebox-Tests schreiben. Die Entwicklung kann Test-Driven oder Behaviour-Driven unterstützt werden. PHPUnit wird auch im professionellen Umfeld eingesetzt, z.B. zur Qualitätssicherung von Facebook: http://sebastian-bergmann.de/archives/903-Thank-you,-Facebook!.html

Einsatzumgebungen

PHPUnit kann vollkommen eigenständig benutzt werden. Es bringt dazu ein Command Line Tool mit, über das die Tests ausgeführt werden können. Das Tool erstellt eigenständig Reports in verschiedenen Formaten (JUnit XML, TAP, DBUS, JSON).

Eine Einbindung ist Eclipse ist möglich. Eine Anleitung findet sich z.B. hier:
http://www.phphatesme.com/blog/tools/phpunit-mittels-pti-in-eclipse-einbinden/

Installation

Eine Installationsanleitung findet sich in der offiziellen Anleitung:
http://www.phpunit.de/manual/3.5/en/installation.html

Die Installation ist auf mehreren Wegen möglich. Da PHPUnit selbst in PHP geschrieben ist, wird es zur Benutzung einfach per include oder require eingebunden. Dazu müssen die PHPUnit-Dateien nur im globalen Include-Pfad oder im Projektverzeichnis liegen.

Unter einem paketbasierten Linux-System findet man PHPUnit oft direkt in den Standard-Repositories. Unter Ubuntu 10.04 wird zum Untersuchungszeitpunkt PHPUnit 3.4.5, das nicht mehr aktuell ist, mit folgendem Befehl installiert:

```
sudo aptitude install phpunit
```

Die aktuelle Version kann über die Bibliothek PEAR (PHP Extension and Application Repository) installiert werden. PEAR ist in Bezug auf PHP vergleichbar mit Standardbibliotheken wie die „Standard C Library“ für C, dem PyPi für Python oder dem Projekt CPAN für Perl. PEAR lässt sich unter Ubuntu wie folgt installieren:

```
sudo aptitude install php-pear
```

Danach müssen die Repositories von PHPUnit zu PEAR hinzugefügt werden:

```
pear channel-discover pear.phpunit.de
pear channel-discover components.ez.no
pear channel-discover pear.symfony-project.com
```

Anschließend kann PHPUnit mit folgendem Befehl installiert werden:

```
pear install phpunit/PHPUnit
```

Unter Ubuntu 10.04 war die Version vom PEAR Installer veraltet, so dass PHPUnit sich zunächst nicht installieren ließ:

```
phpunit/PHPUnit requires PEAR Installer (version >= 1.9.1), installed  version is 1.9.0
```

Dies konnte schnell behoben werden, da PEAR sich selbst aktualisieren kann:

```
pear upgrade-all
```

Danach lief die Installation von PHPUnit anstandslos durch.

Als dritte Möglichkeit, auch ohne PEAR und Paketverwaltung (z.B. Windows) kann man PHPUnit als Archiv herunterladen und einfach in das Projektverzeichnis entpacken. Die Download-Links finden sich auf der Projektseite. Hierbei muss man jedoch auf die automatische Installation des Command Line Tools verzichten, über das die Tests normalerweise ausgeführt werden.

Die Installation ist für einen erfahrenen PHP-Entwickler einfach möglich. Die PEAR-Bibliothek ist sehr bekannt, die nötigen Installationsschritte sind in der offiziellen Anleitung zu finden. Einzig die ggf. nötige Aktualisierung des PEAR-Frameworks stellte ein kleines Hindernis dar, welches aber durch klare Fehlermeldungen schnell erkannt werden kann. Die Installation aus einem Archiv ist immer möglich, selbst auf einem Webspace ohne Konsolenzugriff. Die Installation über die Paketverwaltung ist am Einfachsten, liefert aber eine veraltete Version.

Dokumentation

Die Dokumentation bietet einen guten Einstieg in die Notwendigkeit von Software-Tests, eigene manuelle Software-Tests in PHP und die Automatisierung mit PHPUnit. Vorkenntnisse in einem xUnit-Framework erleichtern das Verständnis, sind aber nicht zwingend notwendig.

Es gibt eine englische und japanische Sprachversion. Diese liegen weder als PDF-Datei noch Archiv vor, sondern sind online auf einzelnen HTML-Seiten abrufbar. Es gibt ein Stichwortverzeichnis mit allen Befehlen und Schlagwörtern mit Links auf die entsprechende Dokumentations-Seite.

Insgesamt macht die Dokumentation einen recht gelungenen Eindruck. Schwachpunkte sind einige undokumentierte oder nicht ausreichend beschriebene Funktionen. Mit einem kurzen Blick in den Quellcode von PHPUnit, der ebenfalls in PHP geschrieben ist, konnten jedoch alle Unklarheiten schnell beseitigt werden. Der Quellcode selbst ist gut dokumentiert.

Wartung der Projektseite

Das Projekt wird fortlaufend weiterentwickelt. Bis zum Untersuchungszeitpunkt gab es im Jahr 2011 bereits über 50 Commits. Für den aktuellen Branch 3.5 existiert ein ausführliches Changelog. Eine graphische Commit-Übersicht des Hauptentwicklers Sebastian Bergmann kann auf github eingesehen werden: https://github.com/sebastianbergmann

Mit PHP 5.0 (Juli 2004) sind Exceptions und eine Reflections API eingeführt worden. Mit PHP 5.1 (November 2005) die Datenbankabstraktionsschicht PDO. Diese Änderungen finden sich auch in PHPUnit wieder.

Die Reflection API wird von PHPUnit selbst genutzt, um die Tests auszuführen. Das Exceptions-Konzept wird ebenfalls von PHPUnit selbst genutzt. So werden z.B. PHP-Fehler in Exceptions umgewandelt. Ebenso können Exceptions in Testfällen überprüft werden. Datenbank-Tests werden über eine PDO-Verbindung realisiert.

Sebastian Bergmann beschäftigt sich auch mit Features, die noch nicht im offiziellen PHP enthalten sind, z.B. Traits (http://wiki.php.net/rfc/traits). In seinem Blog zeigt er ein kurzes Beispiel und neue Testfunktionen für Traits, die er bereits in Branch 3.6 eingepflegt hat: http://sebastian-bergmann.de/archives/906-Testing-Traits.html

Nutzergruppen und Support

Die Hauptanlaufstelle für PHPUnit ist github. Die Plattform bietet z.B. ein Git-Repository und einen Bugtracker.

Ferner ist die Homepage des Hauptentwicklers Sebastian Bergmann eine gute Informationsquelle, insbesondere sein Blog: http://sebastian-bergmann.de/plugin/tag/phpunit

Per Email ist der Entwickler über die Adresse sb@sebastian-bergmann.de zu erreichen. Eine offizielle FAQ, ein Forum oder eine Newsgroup gibt es nicht.

Intuitive Nutzbarkeit

Der Entwickler Sebastian Bergmann selbst sagt: „Lange war meine Antwort auf die Frage nach einer Dokumentation für PHPUnit, dass man eigentlich gar keine Dokumentation für PHPUnit brauche. Stattdessen solle man die Dokumentation von JUnit lesen und sich die Codebeispiele von Java™ und JUnit in PHP und PHPUnit »übersetzen«.“ Diese Einstellung habe er nach einer Anfrage von O'Reilly Deutschland noch einmal überdacht. Als Voraussetzung sollte ein Entwickler ferner objektorientierte Programmierung mit PHP 5 verstanden haben (vgl. [Ber06], S. 5).

Mit soliden PHP-Kenntnissen und Grundwissen eines xJunit-Frameworks gelingt die Einarbeitung tatsächlich sehr schnell. In der Dokumentation findet man schnell vertraute Elemente wie verschiedene Assert-Funktionen, Annotationen und Konzepte wie Fixtures und Data Provider.

Automatisierung

PHPUnit kann auf verschiedene Art und Weise automatisiert werden. Das Command Line Tool kann problemlos über Bash-Skripte angesteuert werden. Eine Ansteuerung über PHP selbst ist ebenfalls möglich, so dass über eigene Skripte Tests durchgeführt werden können. Eine Anbindung an Apache Ant ist über das Framework CruiseControl möglich. Ein Beispiel dazu findet sich in der Dokumentation zu PHPUnit 3.2: http://www.phpunit.de/manual/3.2/en/continuous-integration.html

Einführendes Beispiel

```php
<?php
require_once('PHPUnit/Autoload.php');

class ExampleTest extends PHPUnit_Framework_TestCase {
    public function testAssertEquals() {
        $this->assertEquals(42, 42, 'assertEqualsExample() fehlgeschlagen!');
    }

    /**
     * @test
     */
    public function assertEqualsExample() {
        $this->assertEquals(42, 23, 'testAssertEquals() fehlgeschlagen!');
    }

    /**
     * @depends assertEqualsExample
     */
    public function testDepends() {
        $this->fail('Dieser Test sollte übersprungen werden!');
    }

    /**
     * @expectedException InvalidArgumentException
     */
    public function testExceptionAnnotation() {
        $this->fail('Es wurde keine Exception geworfen!');
    }

    public function testException() {
        try {
                throw new InvalidArgumentException();
        } catch (InvalidArgumentException $expected) {
                return;
        }
        $this->fail('Es wurde keine Exception geworfen!');
    }
}
?>
```

Das Beispiel `exampleTest.php` zeigt eine simple Testklasse. Zu Beginn muss das PHPUnit-Framework eingebunden werden. Die Datei `PHPUnit/Autoload.php` sollte nach der Installation über PEAR automatisch im Include-Pfad liegen.

Eine Testklasse erbt immer von der Klasse `PHPUnit_Framework_TestCase`. Jeder Tests wird als Funktion implementiert, deren Name entweder mit `test` beginnen oder die Annotation `@test` enthalten muss.

Die Methode `assertEquals($expected, $actual, $message = '')` ist die gängigste Testmethode des Frameworks. Sie vergleicht den erwarteten und aktuellen Wert miteinander. Die übergebene Fehlermeldung ist optional.

Die Annotation `@depends` definiert die Abhängigkeit von Tests untereinander. Wird die Abhängigkeit

nicht erfüllt wird der Test übersprungen, damit der Fokus nicht vom ursprünglichen Problem abgelenkt wird (fehlgeschlagene Abhängigkeit).

Ausnahmen können über die Annotation `@expectedException` oder per `try` und `catch` überprüft werden. In diesem Zusammenhang findet die Funktion `fail($message)` Verwendung, die direkt einen Testfehler provoziert.

Gestartet wird der Test über das Command Line Tool `phpunit`, welches von PEAR mit installiert wird. Als Parameter wird der Name der Testklasse übergeben.

```
user@host:~/workspace$ phpunit Example/ExampleTest
PHPUnit 3.5.10 by Sebastian Bergmann.

.FSF.

Time: 0 seconds, Memory: 6.25Mb

There were 2 failures:

1) ExampleTest::assertEqualsExample
testAssertEquals() fehlgeschlagen!
Failed asserting that <integer:23> matches expected <integer:42>.

/home/user/workspace/Example/ExampleTest.php:13

2) ExampleTest::testExceptionAnnotation
Es wurde keine Exception geworfen!

/home/user/workspace/Example/ExampleTest.php:27

FAILURES!
Tests: 4, Assertions: 2, Failures: 2, Skipped: 1.
```

Die erste Zeile der Ausgabe gibt die Testergebnisse in Kurzform wieder. Die Abkürzungen stehen für folgende Ergebnisse:

.	Test erfolgreich	**F**	Test fehlgeschlagen	**S**	Test übersprungen
E	Fehler in Testfunktion	**I**	Test unvollständig oder nicht implementiert		

Nachfolgend werden Testdauer und Speicherverbrauch ausgegeben. Danach folgt eine genaue Auflistung aller Fehler. Es werden `TestKlasse::TestMethode`, die optionale eigene Fehlermeldung, die PHPUnit-Fehlermeldung und der genaue Fehlerort (Datei/Zeile) ausgegeben.

Zuletzt folgt eine Testzusammenfassung mit Anzahl der Tests, Überprüfungen, Fehler und übersprungenen Tests.

Alternativ kann ein Test auch in PHP ausgeführt werden, z.B. wenn das Command Line Tool nicht zur Verfügung steht. Hierzu wird ein Objekt der Klasse `PHPUnit_Framework_TestSuite` angelegt und mittels der Funktion `PHPUnit_TextUI_TestRunner::run()` gestartet. Die Ausgabe entspricht der des Command Line Tools.

```
<pre>
<?php
require_once('PHPUnit/Autoload.php');
require_once('ExampleTest.php');

$suite = new PHPUnit_Framework_TestSuite('ExampleTest');
```

```
PHPUnit_TextUI_TestRunner::run($suite);
?>
</pre>
```

Detaillierte Beschreibung

Assert-Funktionen

PHPUnit bietet eine Vielzahl von Assert-Funktionen, mit denen Testfälle erstellt werden können. Eine grobe Einteilung in Anwendungsgebiete könnte wie folgt aussehen:

Allgemeine Funktionen

`assertEmpty()`, `assertEquals()`, `assertFalse()`, `assertGreaterThan()`, `assertGreaterThanOrEqual()`, `assertInternalType()`, `assertLessThan()`, `assertLessThanOrEqual()`, `assertNull()`, `assertSame()`, `assertTrue()`, `assertType()`

Spezielle Funktionen für Arrays

`assertArrayHasKey()`, `assertContains()`, `assertContainsOnly()`

Spezielle Funktionen für Klassen/Objekte

`assertAttributeContains()`, `assertAttributeContainsOnly()`, `assertAttributeEmpty()`, `assertAttributeEquals()`, `assertAttributeGreaterThan()`, `assertAttributeGreaterThanOrEqual()`, `assertAttributeInstanceOf()`, `assertAttributeInternalType()`, `assertAttributeLessThan()`, `assertAttributeLessThanOrEqual()`, `assertAttributeSame()`, `assertAttributeType()`, `assertClassHasAttribute()`, `assertClassHasStaticAttribute()`, `assertInstanceOf()`, `assertObjectHasAttribute()`

Spezielle Funktionen für Strings

`assertRegExp()`, `assertStringMatchesFormat()`, `assertStringMatchesFormatFile()`, `assertStringEndsWith()`, `assertStringStartsWith()`

Spezielle Funktionen für HTML/XML

`assertTag()`, `assertXmlFileEqualsXmlFile()`, `assertXmlStringEqualsXmlFile()`, `assertXmlStringEqualsXmlString()`

Spezielle Funktionen für Dateien

`assertFileEquals()`, `assertFileExists()`, `assertStringEqualsFile()`

Undokumentierte Funktionen

`assertEqualXMLStructure()`, `assertSelectCount()`, `assertSelectEquals()`, `assertSelectRegExp()`

Komplexe Assert-Funktionen

`assertThat()`

Fast jede Funktion hat eine negierte Variante mit `Not` im Namen, z.B. `assertEquals()` und `assertNotEquals()`. Eine vollständige Auflistung, genaue Erklärung aller Funktionen und ihrer Parameter findet sich in der Dokumentation: http://www.phpunit.de/manual/current/en/writing-tests-for-phpunit.html#writing-tests-for-phpunit.assertions

Die Namen der Funktionen sind für einen PHP-Entwickler größtenteils selbsterklärend. So wird jeder Entwickler sofort darauf kommen, dass z.B. die Funktion `assertEmpty()` intern die PHP-Funktion `empty()` verwenden wird und welche Dinge somit als nicht mit einem Wert belegt gezählt werden (http://www.php.net/manual/de/function.empty.php).

Besonders interessant ist die Funktion `assertThat()`, mit der komplexe Assert-Funktionen erstellt werden können. Diese Funktion wertet Klassen des `PHPUnit_Framework_Constraint` aus. Eine vollständige Tabelle der Constraints findet sich in der Dokumentation: http://www.phpunit.de/manual/current/en/writing-tests-for-phpunit.html#writing-tests-for-phpunit.assertions.assertThat.tables.constraints

Das folgende Beispiel zeigt eine simple Klasse, deren Objekte verglichen werden.

```php
<?php
require_once('PHPUnit/Autoload.php');

class AssertObject {
    private $value;
    public function __construct($value) {
        $this->value = $value;
    }
}

class AssertThatTest extends PHPUnit_Framework_TestCase {
    public function testAssertThat1() {
        $original = new AssertObject('same same but different');
        $fake     = new AssertObject('same same but different');
        $this->assertThat($original, $this->identicalTo($fake));
    }

    public function testAssertThat2() {
        $original = new AssertObject('original');
        $fake     = new AssertObject('fake');
        $this->assertThat($original, $this->logicalNot($this->equalTo($fake)));
    }
}
?>
```

```
user@host:~/workspace$ phpunit Asserts/AssertThatTest
PHPUnit 3.5.10 by Sebastian Bergmann.

F.

Time: 0 seconds, Memory: 6.25Mb

There was 1 failure:

1) AssertThatTest::testAssertThat1
Failed asserting that two variables reference the same object.

/home/user/workspace/Asserts/AssertThatTest.php:15

FAILURES!
Tests: 2, Assertions: 2, Failures: 1.
```

Ablaufsteuerung / Test Fixtures

Für Software-Tests ist es wichtig, dass jede Testfunktion unter kontrollierten und immer wieder gleichen Bedingungen ausgeführt wird. So darf eine Testfunktion den Ablauf einer anderen nicht stören oder ungewollt beeinflussen. Erstellt man ein Objekt, das von verschiedenen Testfunktionen benutzt werden soll, so muss dieses vor jedem Test wieder in den Ausgangszustand versetzt werden.

Der Aufwand dies manuell zu gewährleisten wächst mit der Komplexität der Software stark an. PHPUnit bringt daher Funktionen mit, die vor und nach jedem Test automatisch ausgeführt werden. In diesen Funktionen können alle benötigten Objekte sauber erstellt und ggf. wieder aufgelöst werden (z.B. kritische Ressourcen wie geöffnete Dateien).

Die Funktionen `setUpBeforeClass()` und `tearDownAfterClass()` werden zu Beginn bzw. Ende einmalig ausgeführt. Hier können z.B. Datenbank-Verbindungen verwaltet werden, um die Testausführung zu beschleunigen. Die Funktionen `setUp()` und `tearDown()` werden vor bzw. nach jedem einzelnen Test ausgeführt. Hier können Objekte erzeugt werden, die sich für jeden Test in einem definierten Zustand befinden müssen.

Die Funktionen `assertPreConditions()` und `assertPostConditions()` werden zwischen `setUp()` bzw. `tearDown()` und jeder Testfunktion ausgeführt. Hier können Assert-Funktionen ausgeführt werden, die für alle Tests der Klasse durchgeführt werden sollen.

Die Funktion `onNotSuccessfulTest()` wird aufgerufen, wenn eine Testfunktion nicht erfolgreich ausgeführt wurde.

```php
<?php
require_once('PHPUnit/Autoload.php');

class FixtureTest extends PHPUnit_Framework_TestCase {
    public static function setUpBeforeClass() {
        echo __METHOD__."¥n";
    }

    protected function setUp() {
        echo __METHOD__."¥n";
    }

    protected function assertPreConditions() {
        echo __METHOD__."¥n";
    }

    public function testOne() {
        echo __METHOD__."¥n";
        $this->assertTrue(TRUE);
    }

    public function testTwo() {
        echo __METHOD__."¥n";
        $this->assertTrue(FALSE);
    }

    protected function assertPostConditions() {
        echo __METHOD__."¥n";
    }

    protected function tearDown() {
```

```
        echo __METHOD__."¥n";
    }

    public static function tearDownAfterClass() {
        echo __METHOD__."¥n";
    }

    protected function onNotSuccessfulTest(Exception $e) {
        echo __METHOD__."¥n";
        throw $e;
    }
}
?>
```

```
user@host:~/workspace$ phpunit Fixtures/FixtureTest
PHPUnit 3.5.10 by Sebastian Bergmann.

FixtureTest::setUpBeforeClass
FixtureTest::setUp
FixtureTest::assertPreConditions
FixtureTest::testOne
FixtureTest::assertPostConditions
FixtureTest::tearDown
.FixtureTest::setUp
FixtureTest::assertPreConditions
FixtureTest::testTwo
FixtureTest::tearDown
FixtureTest::onNotSuccessfulTest
FFixtureTest::tearDownAfterClass


Time: 1 second, Memory: 6.00Mb

There was 1 failure:

1) FixtureTest::testTwo
Failed asserting that <boolean:false> is true.

/home/user/workspace/Fixtures/FixtureTest.php:24

FAILURES!
Tests: 2, Assertions: 2, Failures: 1.
```

Logging

Zur Auswertung von Software-Tests sind Logs unverzichtbar. PHPUnit unterstützt verschiedene Typen von Log-Dateien. Als Software-Test wird folgend die Testklasse `FixtureTest` verwendet.

JUnit-XML

PHPUnit kann Log-Dateien im XML-Format von JUnit erzeugen. Diese können mit entsprechenden Tools ausgewertet werden. Als Parameter für das Command Line Tool muss hierzu `--log-junit` angegeben werden.

```
user@host:~/workspace$ phpunit --log-junit Logging/junit.xml Fixtures/FixtureTest
```

```
<?xml version="1.0" encoding="UTF-8"?>
<testsuites>
  <testsuite name="FixtureTest" file="/home/user/workspace/Fixtures/FixtureTest.php" tests="2"
assertions="2" failures="1" errors="0" time="0.003537">
    <testcase name="testOne" class="FixtureTest"
file="/home/user/workspace/Fixtures/FixtureTest.php" line="17" assertions="1" time="0.001999"/>
    <testcase name="testTwo" class="FixtureTest"
file="/home/user/workspace/Fixtures/FixtureTest.php" line="22" assertions="1" time="0.001538">
      <failure type="PHPUnit_Framework_ExpectationFailedException">FixtureTest::testTwo
Failed asserting that &lt;boolean:false&gt; is true.

/home/user/workspace/Fixtures/FixtureTest.php:24
</failure>
    </testcase>
  </testsuite>
</testsuites>
```

TAP (Test Anything Protocol)

Ebenfalls unterstützt wird TAP (http://testanything.org/wiki/index.php/Main_Page), ein simples Textprotokoll, das ursprünglich für PERL entwickelt wurde. Der Parameter für das Command Line Tool ist `--log-tap`.

```
user@host:~/workspace$ phpunit --log-tap Logging/tap.txt Fixtures/FixtureTest
```

```
TAP version 13
ok 1 - FixtureTest::testOne
not ok 2 - Failure: FixtureTest::testTwo
  ---
  message: 'Failed asserting that <boolean:false> is true.'
  severity: fail
  ...
1..2
```

JSON (JavaScript Object Notation)

Für Web-basierte Anwendungen ist JSON (http://www.json.org/) ein wichtiges Format. PHPUnit kann hierin ebenfalls Log-Dateien erstellen. Der Parameter hierfür lautet `–log-json`.

```
user@host:~/workspace$ phpunit --log-json Logging/log.json Fixtures/FixtureTest
```

```
{"event":"suiteStart","suite":"FixtureTest","tests":2}{"event":"testStart","suite":"FixtureTest
","test":"FixtureTest::testOne"}{"event":"test","suite":"FixtureTest","test":"FixtureTest::test
One","status":"pass","time":0.0020711421966553,"trace":[],"message":""}{"event":"testStart","su
ite":"FixtureTest","test":"FixtureTest::testTwo"}{"event":"test","suite":"FixtureTest","test":"
FixtureTest::testTwo","status":"fail","time":0.0018100738525391,"trace":[],"message":"Failed
asserting that <boolean:false> is true."}
```

Clover

Ebenfalls wird das XML-Format Clover (http://www.atlassian.com/software/clover/) unterstützt, das Informationen zur Code-Abdeckung enthält.

```
user@host:~/workspace$ phpunit --coverage-clover Logging/clover.xml Fixtures/FixtureTest
```

```
<?xml version="1.0" encoding="UTF-8"?>
<coverage generated="1297201679">
  <project timestamp="1297201679">
    <metrics files="0" loc="0" ncloc="0" classes="0" methods="0" coveredmethods="0"
conditionals="0" coveredconditionals="0" statements="0" coveredstatements="0" elements="0"
coveredelements="0"/>
  </project>
</coverage>
```

Für die Ermittlung der Code-Abdeckung wird die PHP-Erweiterung XDebug benötigt. Unter Ubuntu 10.04 kann diese einfach auf der Konsole installiert werden.

```
sudo aptitude install php5-xdebug
```

Eine weitere Konfiguration ist nicht notwendig.

Code-Abdeckung

Die Messung der Code-Abdeckung ist ein wichtiges Hilfsmittel zur Kontrolle der Software-Qualität. Mit diesem Konzept lassen sich ungenutzte Code-Abschnitte finden und die Vollständigkeit von Software-Tests ermitteln. PHPUnit kann nicht nur Log-Dateien im Clover XML-Format ausgeben, sondern direkt eine HTML-Ausgabe des Projekts erstellen, in der die Abdeckung inkl. Quellcode-Hervorhebung dargestellt wird.

Als Beispiel dient eine simple PHP-Klasse mit einem Konstruktor und zwei `get`-Funktionen.

```php
<?php
class SimpleClass {
	private $name = NULL;
	private $counter = 0;

	public function __construct($name) {
		$this->name = $name;
	}

	public function getName() {
		if (is_null($this->name)) {
			return 'no name';
		} else {
			return $this->name;
		}
	}

	public function getCounter() {
		return $counter;
	}
}
?>
```

Es wird nun eine PHPUnit-Test für diese Klasse geschrieben.

```php
<?php
require_once('PHPUnit/Autoload.php');
require_once('SimpleClass.php');

class SimpleClassTest extends PHPUnit_Framework_TestCase {
	public function testAssertEquals() {
		$simple = new SimpleClass('my name');
		$this->assertEquals('my name', $simple->getName());
	}
}
```

Die Testdurchführung auf der Konsole zeigt keine Fehler an.

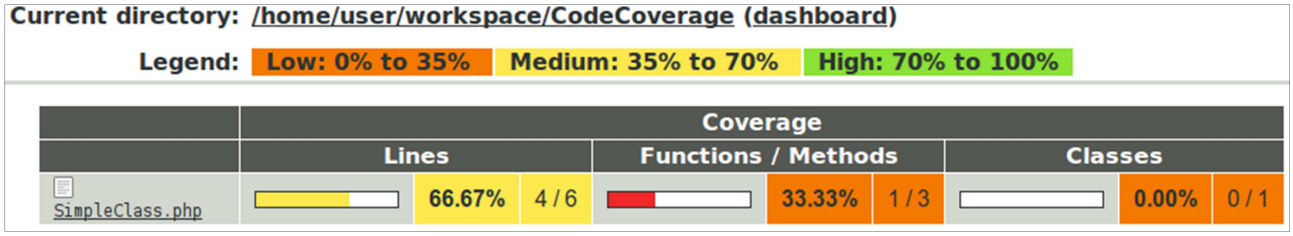
```
user@host:~/workspace$ phpunit CodeCoverage/SimpleClassTest
PHPUnit 3.5.10 by Sebastian Bergmann.

.

Time: 0 seconds, Memory: 5.75Mb
```

```
OK (1 test, 1 assertion)
```

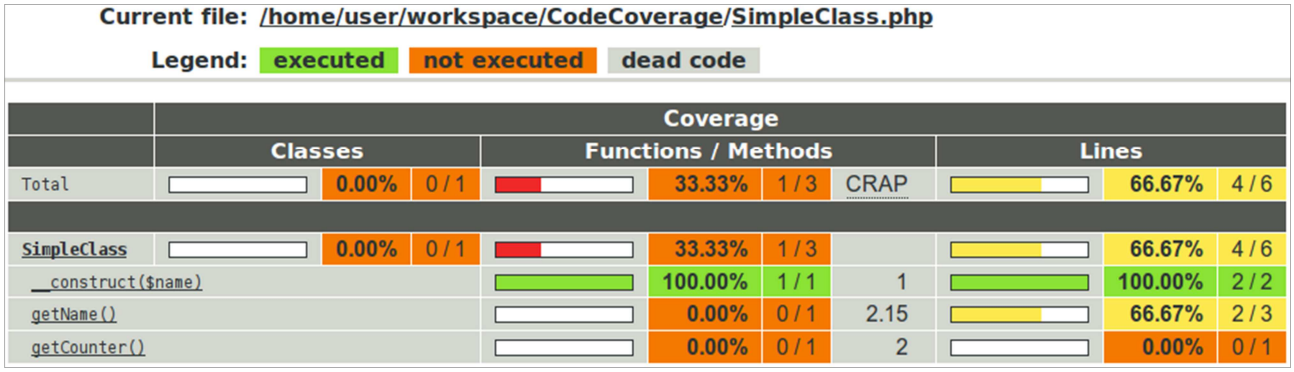
Dies allein ist jedoch wenig aussagekräftig, da wir nicht wissen, ob wir die Klasse vollständig getestet haben. Der Test wird daher erneut durchgeführt und das Ergebnis der Code-Abdeckung im HTML-Format gespeichert. Dazu wird der Parameter `--coverage-html` verwendet.

```
user@host:~/workspace$ phpunit --coverage-html CodeCoverage/Log CodeCoverage/SimpleClassTest
PHPUnit 3.5.10 by Sebastian Bergmann.

.

Time: 0 seconds, Memory: 7.00Mb

OK (1 test, 1 assertion)

Generating code coverage report, this may take a moment.
```

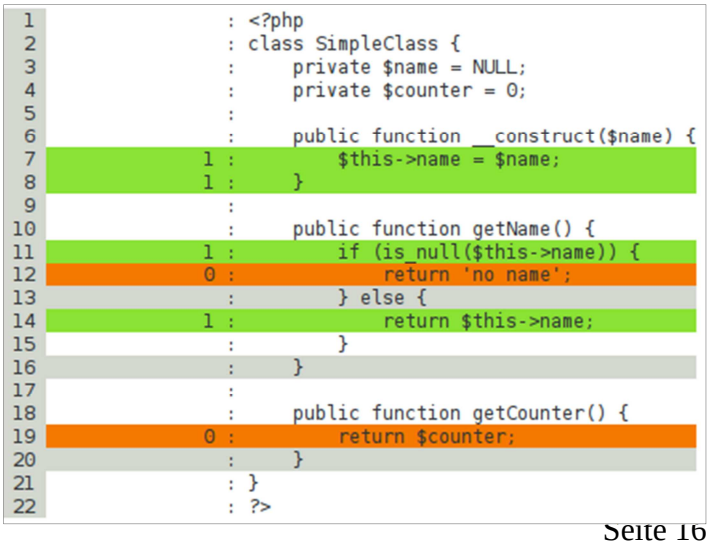
Im angegebenen Verzeichnis befinden sich nun die HTML-Auswertung der Code-Abdeckung. Die Index-Datei zeigt eine Übersicht über alle getesteten Klassen. Von der getesteten `SimpleClass` wurden nur 2/3 aller Code-Zeilen und 1/3 aller Funktionen vollständig ausgeführt.



Mit einem Klick auf den Dateinamen öffnet sich eine genaue Übersicht der Datei.



Auf dieser Seite wird auch der Quellcode der getesteten Klasse farblich markiert. Alle ausgeführten Zeilen sind grün hinterlegt, alle nicht ausgeführten Zeilen orange. So kann schnell erkannt werden, welche Funktionen, Schleifen oder `if-else`-Zweige nicht getestet oder überhaupt genutzt werden.



Test-Organisation

Ein Ziel von PHPUnit ist es einzelne Testfälle zu verschiedenen Tests zusammenstellen zu können. Eine solche Zusammenstellung von Testfällen nennt man Testsuite. Dies gilt bereits für einen einzelnen Test bzw. eine einzige Testklasse. Testsuiten können somit für einzelne Klassen, alle Klassen einer Komponenten oder das Gesamtsystem erstellt werden.

Testauswahl auf Dateiebene

Eine Möglichkeit der Test-Organisation ist die Auswahl von Tests auf Dateiebene. Normalerweise sind die PHP-Klassen eines Projekts auf entsprechende Unterordner verteilt. Diese Verzeichnis-Struktur sollten die Test-Klassen ebenfalls abbilden.

```
project
`--Organisation
   |--Letters
   |  |--A.php
   |  |--B.php
   |  |--C.php
   |  `--D.php
   `--Numbers
      |--Five.php
      |--Four.php
      |--One.php
      |--Three.php
      `--Two.php
```

```
phpunit
`--Organisation
   |--Letters
   |  |--ATest.php
   |  |--BTest.php
   |  |--CTest.php
   |  `--DTest.php
   `--Numbers
      |--FiveTest.php
      |--FourTest.php
      |--OneTest.php
      |--ThreeTest.php
      `--TwoTest.php
```

Eine Testauswahl kann durch Angabe eines Verzeichnisnamens als Parameter an das Command Line Tool getroffen werden. Das Tool wird dann nach allen `*Test.php`-Dateien suchen.

```
user@host:~/workspace$ phpunit Organisation
PHPUnit 3.5.10 by Sebastian Bergmann.

FourTest::testNumber
.OneTest::testNumber
.ThreeTest::testNumber
.FiveTest::testNumber
.TwoTest::testNumber
.CTest::testLetter
.ATest::testLetter
.DTest::testLetter
.BTest::testLetter
.

Time: 0 seconds, Memory: 6.75Mb

OK (10 tests, 10 assertions)
```

```
user@host:~/workspace$ phpunit Organisation/Numbers
PHPUnit 3.5.10 by Sebastian Bergmann.

FourTest::testNumber
.OneTest::testNumber
.ThreeTest::testNumber
.FiveTest::testNumber
```

```
.TwoTest::testNumber
.

Time: 0 seconds, Memory: 6.25Mb

OK (6 tests, 6 assertions)
```

```
user@host:~/workspace$ phpunit Organisation/Numbers/TwoTest
PHPUnit 3.5.10 by Sebastian Bergmann.

TwoTest::testNumber
.

Time: 0 seconds, Memory: 5.75Mb

OK (1 test, 1 assertion)
```

Mit dem Parameter `--filter` kann die Testauswahl auf einen Methodennamen beschränkt werden.

```
user@host:~/workspace$ phpunit --filter testLetter Organisation
PHPUnit 3.5.10 by Sebastian Bergmann.

CTest::testLetter
.ATest::testLetter
.DTest::testLetter
.BTest::testLetter
.

Time: 0 seconds, Memory: 6.75Mb

OK (4 tests, 4 assertions)
```

Allgemein hat dieses Vorgehen den Nachteil, dass die Reihenfolge der Tests nicht bestimmt werden kann. Dies kann in Anbetracht von Test-Abhängigkeiten zu Problemen führen.

Testsuiten mit XML-Dateien

PHPUnit unterstützt die Konfiguration über XML-Dateien, in denen auch Testsuiten definiert werden können. Eine vollständige Auflistung aller Konfigurations-Möglichkeiten findet sich in der Dokumentation: http://www.phpunit.de/manual/current/en/appendixes.configuration.html

Eine minimale Konfiguration enthält ein oder mehrere Verzeichnisnamen. Relative Pfade werden vom Speicherort der XML-Datei aufgelöst, nicht vom aktuellen Arbeitsverzeichnis aus.

```xml
<phpunit>
    <testsuites>
        <testsuite name="AllTests">
            <directory>Letters</directory>
            <directory>Numbers</directory>
        </testsuite>
    </testsuites>
</phpunit>
```

Die Konfigurationsdatei wird mit dem Parameter `-c` an das Command Line Tool übergeben.

```
user@host:~/workspace$ phpunit -c Organisation/AllTests.xml
PHPUnit 3.5.10 by Sebastian Bergmann.

CTest::testLetter
.ATest::testLetter
.DTest::testLetter
.BTest::testLetter
.FourTest::testNumber
.OneTest::testNumber
.ThreeTest::testNumber
.FiveTest::testNumber
.TwoTest::testNumber
.

Time: 1 second, Memory: 7.25Mb

OK (10 tests, 10 assertions)
```

Der Nachteil ist wiederum, dass die Reihenfolge der Tests nicht definiert ist. Dies kann durch eine genaue Angabe der gewünschten Test-Dateien gesteuert werden. Relative Pfade werden hier vom aktuellen Arbeitsverzeichnis aufgelöst, unabhängig vom Speicherort der XML-Datei.

```
<phpunit>
    <testsuites>
        <testsuite name="SortedTests">
            <file>Organisation/Letters/ATest.php</file>
            <file>Organisation/Letters/BTest.php</file>
            <file>Organisation/Letters/CTest.php</file>
            <file>Organisation/Letters/DTest.php</file>
            <file>Organisation/Numbers/OneTest.php</file>
            <file>Organisation/Numbers/TwoTest.php</file>
            <file>Organisation/Numbers/ThreeTest.php</file>
            <file>Organisation/Numbers/FourTest.php</file>
            <file>Organisation/Numbers/FiveTest.php</file>
        </testsuite>
    </testsuites>
</phpunit>
```

```
user@host:~/workspace$ phpunit -c Organisation/SortedTests.xml
PHPUnit 3.5.10 by Sebastian Bergmann.

ATest::testLetter
.BTest::testLetter
.CTest::testLetter
.DTest::testLetter
.OneTest::testNumber
.TwoTest::testNumber
.ThreeTest::testNumber
.FourTest::testNumber
.FiveTest::testNumber
.
```

```
Time: 0 seconds, Memory: 7.25Mb

OK (10 tests, 10 assertions)
```

Testsuiten mit PHP-Dateien

Mit der Klasse `PHPUnit_Framework_TestSuite` können Tests hierarchisch als Testsuiten organisiert werden.

In der obersten Klasse wird eine Testsuite definiert, in der alle untergeordneten Testsuiten eingehängt werden. Dazu müssen diese Testsuiten bekannt sein, also per `require()` eingebunden sein.

```php
<?php
require_once('PHPUnit/Autoload.php');
require_once('Letters/AllTests.php');
require_once('Numbers/AllTests.php');

class AllTests {
    public static function suite() {
        $suite = new PHPUnit_Framework_TestSuite('Organisation');
        $suite->addTest(Letters_AllTests::suite());
        $suite->addTest(Numbers_AllTests::suite());
        return $suite;
    }
}
?>
```

Die untergeordneten Test-Suiten enthalten die konkreten Testklassen oder weitere Testsuiten.

```php
<?php
require_once('PHPUnit/Autoload.php');
require_once('ATest.php');
require_once('BTest.php');
require_once('CTest.php');
require_once('DTest.php');

class Letters_AllTests {
    public static function suite() {
        $suite = new PHPUnit_Framework_TestSuite('Letters');
        $suite->addTestSuite('ATest');
        $suite->addTestSuite('BTest');
        $suite->addTestSuite('CTest');
        $suite->addTestSuite('DTest');
        return $suite;
    }
}
?>
```

```php
<?php
require_once('PHPUnit/Autoload.php');
require_once('OneTest.php');
require_once('TwoTest.php');
require_once('ThreeTest.php');
require_once('FourTest.php');
require_once('FiveTest.php');
```

```php
class Numbers_AllTests {
    public static function suite() {
        $suite = new PHPUnit_Framework_TestSuite('Numbers');
        $suite->addTestSuite('OneTest');
        $suite->addTestSuite('TwoTest');
        $suite->addTestSuite('ThreeTest');
        $suite->addTestSuite('FourTest');
        $suite->addTestSuite('FiveTest');
        return $suite;
    }
}
?>
```

Es kann nun jede Testsuite bzw. -klasse ausgeführt werden, wobei immer alle untergeordneten Tests ausgeführt werden. Die oberste Testsuite `Organisation/Alltests` enthält alle Tests.

```
user@host:~/workspace$ phpunit Organisation/AllTests
PHPUnit 3.5.10 by Sebastian Bergmann.

ATest::testLetter
.BTest::testLetter
.CTest::testLetter
.DTest::testLetter
.OneTest::testNumber
.TwoTest::testNumber
.ThreeTest::testNumber
.FourTest::testNumber
.FiveTest::testNumber
.

Time: 0 seconds, Memory: 6.75Mb

OK (10 tests, 10 assertions)
```

Die eingebundenen Testsuiten können auch einzeln aufgerufen werden, z.B. `Letters/AllTests`.

```
user@host:~/workspace$ phpunit Organisation/Letters/AllTests
PHPUnit 3.5.10 by Sebastian Bergmann.

ATest::testLetter
.BTest::testLetter
.CTest::testLetter
.DTest::testLetter
.

Time: 0 seconds, Memory: 6.00Mb

OK (4 tests, 4 assertions)
```

In der Ebene darunter wäre die einzelne Testklasse.

```
user@host:~/workspace$ phpunit Organisation/Letters/ATest
PHPUnit 3.5.10 by Sebastian Bergmann.
```

```
ATest::testLetter
.

Time: 0 seconds, Memory: 5.75Mb

OK (1 test, 1 assertion)
```

Eine weitere Reduzierung auf einen Methodennamen ist über den Parameter `--filter` möglich, der bereits vorgestellt wurde.

Insgesamt ist mit der Klasse `PHPUnit_Framework_TestSuite` jede beliebige Zusammenstellung von Tests und Testsuiten möglich, bei der zusätzlich die Reihenfolge der Tests bestimmt werden kann.

Testauswahl über Gruppen

Eine zusätzliche Möglichkeit zur Definition von Testsuiten bildet die Annotation `@group`. Hierfür muss allerdings der Quelltext der Testklassen angepasst werden. Mit der Annotation kann jeder einzelne Testfall zu einer oder mehreren Gruppen hinzugefügt werden.

```php
<?php
require_once('PHPUnit/Autoload.php');

class ATest extends PHPUnit_Framework_TestCase {
    /**
     * @group letters
     */
    public function testLetter() {
        echo __METHOD__."¥n";
        $this->assertTrue(TRUE);
    }
}
?>
```

```php
<?php
require_once('PHPUnit/Autoload.php');

class OneTest extends PHPUnit_Framework_TestCase {
    /**
     * @group numbers
     * @group odd
     */
    public function testNumber() {
        echo __METHOD__."¥n";
        $this->assertTrue(TRUE);
    }
}
?>
```

```php
<?php
require_once('PHPUnit/Autoload.php');

class TwoTest extends PHPUnit_Framework_TestCase {
    /**
     * @group numbers
```

```
     * @group even
     */
    public function testNumber() {
        echo __METHOD__."¥n";
        $this->assertTrue(TRUE);
    }
}
?>
```

Über die bereits vorgestellten Methoden (Command Line Tool, XML-Datei) können diese Gruppen nun selektiert oder ausgeschlossen werden, um Testsuiten zusammenzustellen.

Für das Command Line Tool stehen hierzu die Parameter `--group` und `--exclude-group` zur Verfügung. Mit dem Befehl `--list-groups` können alle verfügbaren Gruppen aufgelistet werden. Alle Tests ohne explizite Gruppenzuordnung werden als `__nogroup__` angezeigt. Nach diesem Namen kann jedoch nicht gefiltert werden.

```
user@host:~/workspace$ phpunit --list-groups .
PHPUnit 3.5.10 by Sebastian Bergmann.

Available test group(s):
 - __nogroup__
 - even
 - letters
 - numbers
 - odd
```

```
user@host:~/workspace$ phpunit --group letters .
PHPUnit 3.5.10 by Sebastian Bergmann.

CTest::testLetter
.ATest::testLetter
.DTest::testLetter
.BTest::testLetter
.

Time: 0 seconds, Memory: 9.00Mb

OK (4 tests, 4 assertions)
```

```
user@host:~/workspace$ phpunit --exclude-group even Organisation
PHPUnit 3.5.10 by Sebastian Bergmann.

OneTest::testNumber
.ThreeTest::testNumber
.FiveTest::testNumber
.CTest::testLetter
.ATest::testLetter
.DTest::testLetter
.BTest::testLetter
.

Time: 0 seconds, Memory: 6.75Mb
```

```
OK (7 tests, 7 assertions)
```

In einer XML-Datei können Gruppen mit den Tags `<include>` und `<exclude>` selektiert werden. Diese entsprechen den Parametern `--group` und `--exclude-group` des Command Line Tools.

```
<phpunit>
    <groups>
        <exclude>
            <group>odd</group>
        </exclude>
    </groups>
</phpunit>
```

```
user@host:~/workspace$ phpunit -c Organisation/GroupTests.xml Organisation
PHPUnit 3.5.10 by Sebastian Bergmann.

FourTest::testNumber
.TwoTest::testNumber
.CTest::testLetter
.ATest::testLetter
.DTest::testLetter
.BTest::testLetter
.

Time: 0 seconds, Memory: 7.25Mb

OK (6 tests, 6 assertions)
```

Globale Variablen

In PHP gehören globale Variablen trotz Objektorientierung zum Entwickler-Alltag. Anfragen an den Webserver werden z.B. in den Superglobals `$_GET`, `$_POST` bzw. `$_REQUEST` an das PHP-Skript übergeben. Eigene Variablen werden in `$GLOBALS` gespeichert. Diese Superglobals sind in allen Scopes definiert, müssen also nicht über das Schlüsselwort `global` bekannt gemacht werden (http://de2.php.net/manual/de/language.variables.superglobals.php).

Normalerweise hat man als Entwickler keinen Einfluss auf die Definition dieser Variablen (Benutzereingaben). Dies erschwert das Testen von Klassen, die auf Superglobals zugreifen. Zusätzlich können Veränderungen an globalen Variablen Auswirkungen auf andere Tests haben. Dazu zählen ebenfalls statische Variablen von Klassen.

PHPUnit lässt Tests normalerweise so ablaufen, dass Änderungen an Superglobals andere Tests nicht beeinflussen. Diese Isolation kann optional auf statische Variablen ausgeweitet werden.

```
<?php
require_once('PHPUnit/Autoload.php');

class SuperglobalsTest extends PHPUnit_Framework_TestCase {
    protected function setUp() {
        $_POST['para'] = 'you can test me';
    }

    public function testPOST() {
        $this->assertEquals('you can test me', $_POST['para']);
    }
```

```
    public function testGET1() {
        $_GET['para'] = 'forget me';
        $this->assertEquals('forget me', $_GET['para']);
    }

    /**
     * @depends testGET1
     */
    public function testGET2() {
        $this->assertNotEmpty($_GET['para']);
    }

    /**
     * @backupGlobals disabled
     */
    public function testREQUEST1() {
        $_REQUEST['para'] = 'do not forget this value';
        $this->assertEquals('do not forget this value', $_REQUEST['para']);
    }

    /**
     * @depends testREQUEST1
     */
    public function testREQUEST2() {
        $this->assertEquals('do not forget this value', $_REQUEST['para']);
    }
}
?>
```

Die Testausführung zeigt, dass auf Superglobals, die in `setUp()` definiert wurden, zugegriffen werden kann. Eine innerhalb des Tests gesetzte Superglobals-Variable hingegen existiert im nächsten Test nicht mehr. Dieses Verhalten kann mit der Annotation `@backupGlobals` gesteuert werden, so dass die Variable `$_REQUEST['para']` aus `testREQUEST1()` erhalten bleibt.

```
user@host:~/workspace$ phpunit Superglobals/SuperglobalsTest
PHPUnit 3.5.10 by Sebastian Bergmann.

..E..

Time: 0 seconds, Memory: 6.00Mb

There was 1 error:

1) SuperglobalsTest::testGET2
Undefined index: para

/home/user/workspace/Superglobals/SuperglobalsTest.php:22

FAILURES!
Tests: 5, Assertions: 4, Errors: 1.
```

Folgend ein Beispiel für eine Klasse mit einer statischen Variable.

```
<?php
```

```php
require_once('PHPUnit/Autoload.php');

class StaticTest {
    public static $msg = '';

    public function setMsg($msg) {
        StaticTest::$msg = $msg;
    }

    public function getMsg() {
        return StaticTest::$msg;
    }
}

class StaticAttributesTest extends PHPUnit_Framework_TestCase {
    protected $class;

    protected function setUp() {
        $this->class = new StaticTest();
    }

    public function testStatic1() {
        $this->class->setMsg('forget me not');
        $this->assertEquals('forget me not', $this->class->getMsg());
    }

    /**
     * @depends testStatic1
     */
    public function testStatic2() {
        $this->assertEquals('forget me not', $this->class->getMsg());
    }

    /**
     * @backupStaticAttributes enabled
     */
    public function testStatic3() {
        $this->class->setMsg('forget me');
        $this->assertEquals('forget me', $this->class->getMsg());
    }

    /**
     * @depends testStatic3
     */
    public function testStatic4() {
        $this->assertEquals('forget me', $this->class->getMsg());
    }
}
?>
```

Auch hier kann über die Annotation `@backupStaticAttributes` gesteuert werden, ob die Änderung einer statischen Variable gespeichert werden soll, oder nicht.

```
user@host:~/workspace$ phpunit Superglobals/StaticAttributesTest
PHPUnit 3.5.10 by Sebastian Bergmann.
```

```
...F

Time: 0 seconds, Memory: 6.00Mb

There was 1 failure:

1) StaticAttributesTest::testStatic4
Failed asserting that two strings are equal.
--- Expected
+++ Actual
@@ @@
-forget me
+forget me not

/home/user/workspace/Superglobals/StaticAttributesTest.php:47

FAILURES!
Tests: 4, Assertions: 4, Failures: 1.
```

Ausgabe-Validierung

Ein weiterer Schwerpunkt beim Testen von PHP-Anwendungen liegt auf der Validierung von Ausgaben per `print` oder `echo`. Hierzu nutzt PHPUnit die Ausgabepuffer-Funktionen von PHP (http://www.php.net/manual/de/ref.outcontrol.php).

Mit der Funktion `expectOutputString($string)` kann auf eine exakte Ausgabe geprüft werden. Die Funktion `expectOutputRegex($regex)` erlaubt einen Abgleich der Ausgabe mit einem regulären Ausdruck. Die Funktion `setOutputCallback($callback)` setzt eine Funktion, mit der die Ausgabe vor dem Abgleich bearbeitet werden kann.

```php
<?php
require_once('PHPUnit/Autoload.php');

class OutputTest extends PHPUnit_Extensions_OutputTestCase {
    public function testExpectOutputString() {
        $this->expectOutputString('this will match');
        echo 'this will not match';
    }

    public function testExpectOutputRegex() {
        $this->expectOutputRegex('/.*will this string match¥?.*/');
        echo '<html><body><h1>will this string match?</h1></body></html>';
    }

    public function testSetOutputCallback() {
        $this->assertTrue($this->setOutputCallback('OutputTest::stripTags'));
        $this->expectOutputString('will this string match?');
        echo '<html><body><h1>will this string match?</h1></body></html>';
    }

    public static function stripTags($input) {
        return strip_tags($input);
    }
}
?>
```

```
user@host:~/workspace$ phpunit OutputBuffering/OutputBufferingTest
PHPUnit 3.5.10 by Sebastian Bergmann.

F..

Time: 0 seconds, Memory: 6.25Mb

There was 1 failure:

1) OutputTest::testExpectOutputString
Failed asserting that two strings are equal.
--- Expected
+++ Actual
@@ @@
-this will match
+this will not match
```

```
FAILURES!
Tests: 3, Assertions: 4, Failures: 1.
```

Data Provider

Für einen Testfall kann es nötig sein eine Funktion mit unterschiedlichen Variablen zu überprüfen. Für diesen Zweck gibt es das Konzept des Data Providers. Hierbei liefert eine Funktion mehrere Datensätze, die in einer anderen Funktion nacheinander verarbeitet werden.

Der Lieferant (Data Provider) ist eine öffentliche Funktion der Testklasse, die ein mehrdimensionales Array mit Daten zurück geben muss. Die verarbeitende Testfunktion wird mit der Annotation `@dataProvider` versehen, die den Namen des Data Providers angibt. Dieser Funktionsname kann frei gewählt werden.

```
<?php
require_once('PHPUnit/Autoload.php');

class DataProviderTest extends PHPUnit_Framework_TestCase {
    /**
     * @dataProvider dataProvider
     */
    public function testDataProvider($m1, $m2, $produkt) {
        $this->assertEquals($produkt, $m1 * $m2, "Multiplikation: $m1 * $m2,  Erwartet:
$produkt, Erhalten: ".($m1 * $m2));
    }

    public function dataProvider() {
        return array(
                array(0, 1, 0),
                array(1, 2, 2),
                array(2, 3, 6),
                array(3, 4, 8)
        );
    }
}
?>
```

Das Beispiel zeigt einen simplen Data Provider, der ein mehrdimensionales Array zurück gibt. Jeder Datensatz des Arrays wird als eigener Testfall von der Funktion `testDataProvider` ausgeführt. Jedes Element eines Datensatzes wird als eigene Variable an die Testfunktion übergeben. Die Testdurchführung zeigt drei erfolgreiche Ausführungen und einen Fehler.

```
user@host:~/workspace$ phpunit DataProvider/DataProvider
PHPUnit 3.5.10 by Sebastian Bergmann.

...F

Time: 0 seconds, Memory: 6.00Mb

There was 1 failure:

1) DataProviderTest::testDataProvider with data set #3 (3, 4, 8)
Multiplikation: 3 * 4,  Erwartet: 8, Erhalten: 12
Failed asserting that <integer:12> matches expected <integer:8>.
```

```
/home/user/workspace/DataProvider/DataProvider.php:9

FAILURES!
Tests: 4, Assertions: 4, Failures: 1.
```

Als Data Provider kann ferner ein Objekt zurückgegeben werden, dass das Interface `Iterator` implementiert (http://php.net/manual/de/class.iterator.php).

```php
<?php
require_once('PHPUnit/Autoload.php');

class TestIterator implements Iterator {
    private $keys = array();
    private $values = array();
    public function __construct($keys, $values) {
        if (is_array($keys) && is_array($values)
                    && count($keys) == count($values)) {
                $this->keys = $keys;
                $this->values = $values;
        }
    }
    public function rewind() {
        reset($this->keys);
        reset($this->values);
    }
    public function current() {
        return array(current($this->keys), strrev(current($this->values)));
    }
    public function key() {
        return key($this->keys);
    }
    public function next() {
        return array(next($this->keys), strrev(next($this->values)));
    }
    public function valid() {
        return (current($this->keys) !== FALSE);
    }
}

class DataProviderIteratorTest extends PHPUnit_Framework_TestCase {
    /**
     * @dataProvider dataProviderIterator
     */
    public function testDataProviderIterator($key, $value) {
        $this->assertEquals($value, $key, "Erwartet: $value, Erhalten: $key");
    }

    public function dataProviderIterator() {
        $keys   = array('foo', 'bar', 'foobar');
        $values = array('oof', 'rab', 'zaboof');
        return new TestIterator($keys, $values);
    }
}
?>
```

Die Klasse `TestIterator` benötigt zur Instantiierung zwei gleich große Arrays als Parameter und gibt diese beim Durchlaufen als Wertepaar zurück. Dabei wird der zweite Wert rückwärts ausgegeben (siehe http://www.php.net/manual/de/function.strrev.php).

Der Data Provider erstellt ein Objekt dieser Klasse mit entsprechenden Wertepaaren und gibt das resultierende Objekt zurück. Die Testfunktion vergleicht die Strings auf Gleichheit.

```
user@host:~/workspace$ phpunit DataProvider/DataProviderIterator
PHPUnit 3.5.10 by Sebastian Bergmann.

..F

Time: 0 seconds, Memory: 6.00Mb

There was 1 failure:

1) DataProviderIteratorTest::testDataProviderIterator with data set #2 ('foobar', 'foobaz')
Erwartet: foobaz, Erhalten: foobar
Failed asserting that two strings are equal.
--- Expected
+++ Actual
@@ @@
-foobaz
+foobar

/home/user/workspace/DataProvider/DataProviderIterator.php:36

FAILURES!
Tests: 3, Assertions: 3, Failures: 1.
```

Die Testausgabe zeigt den zu erwartenden Fehler im letzten Datensatz.

Test von Benutzeroberflächen mit Selenium

Mit PHPUnit können Tests von HTML-basierten Benutzeroberflächen durchgeführt werden. Hierzu wird das bekannte Framework Selenium verwendet. Eine komplette Einarbeitung in das Tool Selenium kann im Rahmen dieser Untersuchung nicht geleistet werden Folgend daher nur ein kurzes Beispiel zur Einführung.

Zu Beginn muss der Selenium-Server auf einem geeigneten Host gestartet werden. Dieser braucht eine grafische Oberfläche und einen Webbrowser wie Firefox, Safari oder den Internet Explorer. Das Server-Modul liegt als JAR-Archiv vor und kann somit auf verschiedenen Betriebssystemen gestartet werden.

```
user@host:~/workspace/Selenium$ java -jar selenium-server-standalone-2.0b1.jar
10:52:38.013 INFO - Java: Sun Microsystems Inc. 19.0-b09
10:52:38.014 INFO - OS: Linux 2.6.35-25-generic-tuxonice amd64
10:52:38.018 INFO - v2.0 [b1], with Core v2.0 [b1]
10:52:38.133 INFO - RemoteWebDriver instances should connect to: http://127.0.0.1:4444/wd/hub
10:52:38.134 INFO - Version Jetty/5.1.x
10:52:38.134 INFO - Started HttpContext[/selenium-server/driver,/selenium-server/driver]
10:52:38.135 INFO - Started HttpContext[/selenium-server,/selenium-server]
10:52:38.135 INFO - Started HttpContext[/,/]
10:52:38.145 INFO - Started org.openqa.jetty.jetty.servlet.ServletHandler@78dc6a77
10:52:38.146 INFO - Started HttpContext[/wd,/wd]
10:52:38.149 INFO - Started SocketListener on 0.0.0.0:4444
10:52:38.149 INFO - Started org.openqa.jetty.jetty.Server@75d9fd51
```

Der Server lauscht nun auf dem lokalen Host und nimmt Verbindungen von Selenium-Clients entgegen. Einen solchen Client kann man als Testfall in PHPUnit erstellen. Die Testklasse erbt dazu von der Klasse `PHPUnit_Extensions_SeleniumTestCase`.

Es werde Pfade gesetzt, unter dem der Selenium-Server Screenshots speichern soll und per Webbrowser anzeigen kann. In der Funktion `setUp()` muss ein Browser angegeben werden, mit dem die Benutzeroberfläche getestet werden soll. Der Test prüft auf den Seitentitel der aufgerufenen Seite.

```php
<?php
require_once('PHPUnit/Autoload.php');

class WebTest extends PHPUnit_Extensions_SeleniumTestCase {
    protected $captureScreenshotOnFailure = TRUE;
    protected $screenshotPath = './Screenshots';
    protected $screenshotUrl = 'http://localhost/phpunit/Selenium/Screenshots';

    protected function setUp() {
        $this->screenshotPath = __DIR__.'/Screenshots';
        $this->setBrowser('*firefox');
        $this->setBrowserUrl('http://www.google.de/');
    }

    public function testTitle() {
        $this->open('http://www.google.de/');
        $this->assertTitle('Goggel');
    }
}
?>
```

```
user@host:~/workspace$ phpunit Selenium/WebTest
PHPUnit 3.5.10 by Sebastian Bergmann.

F

Time: 7 seconds, Memory: 7.00Mb

There was 1 failure:

1) WebTest::testTitle
Current URL: http://www.google.de/
Screenshot: http://localhost/phpunit/Selenium/Screenshots/bc2c503347b3ea4d8d3a1a7627ff3877.png

Failed asserting that <string:Google> matches PCRE pattern "/Goggel/".

/home/user/workspace/Selenium/WebTest.php:17
/home/user/workspace/Selenium/WebTest.php:17

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
```

Der Test schlägt fehl. Selenium gibt die URL und den Link zum Screenshot zurück. PHPUnit zeigt die genaue Fehlerursache auf. In der Konsole des Selenium-Servers werden zusätzliche Ausgaben zum Testablauf angezeigt.

Mit Selenium sind ferner Steuerungen der Benutzeroberfläche möglich, so dass auch komplexe PHP-Anwendungen vollständig getestet werden können. Zum Erstellen der einzelnen Tests gibt es verschiedene IDEs und Tools, z.B. ein Plugin für Firefox, mit dem Eingaben aufgezeichnet werden können. Hieraus können dann Testfälle für PHPUnit geschrieben werden.

Literatur

[Ber10] S. Bergmann & S. Priebsch, **Softwarequalität in PHP-Projekten**, Hanser, München, 2010

Dieses Buch, dessen Mitautor der Punisch-Hauptentwickler Sebastian Bergmann ist, wird als Standardwerk zur Qualitätssicherung (von PHP) benannt. Neben Grundlagenwissen zum Testen von PHP-basierter Software werden Fallstudien bekannter Firmen und Projekte vorgestellt (TYPO3, Digg, studiVZ, swoodoo). PHPUnit wird hier als Werkzeug vorgestellt.

[Ber06] S. Bergmann, **PHPUnit kurz und gut**, O'Reilly, Köln, 2006

[Ber05] S. Bergmann, **PHPUnit Pocket Guide**, O'Reilly, Sebastopol, 2005

Diese Bücher von Sebastian Bergmann sind mittlerweile veraltet. Beide Bücher werden als freie Bücher unter der Creative Commons-Lizenz beworben, die stets frei im Netz verfügbar sein sollen. Die angegebene URL ist jedoch offline. Eine Text-Referenz findet sich zuletzt in der Dokumentation zu PHPUnit 2.3. Die Bücher scheinen damit von der Online-Dokumentation abgelöst worden zu sein.

[Mer10] D. Merkel, **Expert PHP 5 Tools**, Packt Publishing, Birmingham, 2010

Dieses Buch beschäftigt sich mit verschiedenen PHP-Tools (Coding Guidelines, phpDocumentor, Eclipse PDT, Subversion, Xdebug, PHP Frameworks, PHPUnit, Deploying Applications, UML, Continuous Integration). Der Anteil von Softwaretests und PHPUnit macht ungefähr ein Zehntel des Buches aus.