

Automatische Softwaretests

7. Juni 2004

Note légale

Dieser Beitrag ist lizenziert unter der GNU Free Documentation License.

Zusammenfassung

Software, die nicht das tut, was sie tun soll, ist nutzlos. Eines der wichtigsten Kriterien für die Qualität von Software ist daher die Fehlerfreiheit. Eine bewährte Methode um Fehler in Software zu finden sind systematische Tests. Ein Ansatz der sich besonders gut für Freie Software eignet, sind automatische Tests, da für regelmäßige und umfangreiche Manuelle Tests bei den meisten freien Projekten die Ressourcen fehlen.

In den letzten Jahren sind diverse Variationen automatischer Tests unter den Schlagworten "Test Driven Development" oder "Unit Tests" insbesondere von Befürwortern des "Extreme Programming" bekannt gemacht worden. Die Idee, Tests zu automatisieren, ist zwar schon deutlich älter und wird in manchen freien Software Projekten schon seit Jahren eingesetzt, der Rummel um Extreme Programming hat jedoch automatisierte Tests mehr Entwicklern bekannt gemacht und immer mehr Projekte fangen an sie zu verwenden.

Automatische Tests, sind, wenn sie richtig gehandhabt werden, aus mehreren Gründen interessant. Sie können praktisch auf Knopfdruck ausgeführt werden und der Entwickler kann das Resultat auf einen Blick beurteilen. Damit verschwindet die Mühsal von manuellen Tests. Die Tests können sehr einfach auch nach kleinen Änderungen durchgeführt werden, so daß Fehler oft sehr früh entdeckt werden. Änderungen an der Software, insbesondere Refactoring, werden dadurch erleichtert.

Wenn man dann noch die Tests gleichzeitig mit der Software entwickelt – "test a little, code a little" wird auch das Erstellen der Testsuite selbst recht einfach. Ausserdem wird die Software von vornherein so entworfen, daß automatische Tests einfach zu schreiben sind. Das führt in der Regel auch zu gut modularisiertem und wartbarem Code.

Der Vortrag wird theoretische Aspekte von automatischen Tests vorstellen und ihre praktische Umsetzung bei Freier Software untersuchen und mit Beispielen erläutern.

Im theoretischen Teil werden verschiedene Arten von Tests wie etwa Unittests, funktionale Tests oder Regressionstests vorgestellt und Wege besprochen, wie diese Test in den Entwicklungsprozess eingebunden werden können. Des weiteren werden einige Grenzen von automatisierten Tests aufgezeigt.

Im praktischen Teil wird anhand von einigen Projekten, an denen der Vortragende selbst beteiligt ist, wie etwa dem Geodatenbetrachter Thuban und dem Vektorzeichenprogramm Skencil, demonstriert, wie man konkret Testfälle schreiben kann. Da sowohl Thuban als auch Skencil hauptsächlich in Python geschrieben sind ist dieser Teil im wesentlichen eine Einführung in das unittest Modul von Python.

1 Einführung

Software, die nicht das tut, was sie tun soll, ist nutzlos. Eines der wichtigsten Kriterien für die Qualität von Software ist daher die Fehlerfreiheit. Eine bewährte Methode um Fehler in Software zu finden sind systematische Tests. Ein Ansatz der sich besonders gut für Freie Software eignet, sind automatische Tests, da für regelmäßige und umfangreiche manuelle Tests bei den meisten freien Projekten die Ressourcen fehlen, und selbst wenn man systematische manuelle Tests verwendet, sind automatische Tests eine gute Ergänzung.

In den letzten Jahren sind diverse Variationen automatischer Tests unter den Schlagworten "Test Driven Development" oder "Unit Tests" insbesondere von Befürwortern des "Extreme Programming" bekannt gemacht worden. Die Idee, Tests zu automatisieren, ist zwar schon deutlich älter und wird in manchen freien Software Projekten schon seit Jahren eingesetzt, Extreme Programming hat jedoch automatisierte Tests mehr Entwicklern bekannt gemacht und immer mehr Projekte fangen an, sie zu verwenden.

Automatische Tests, sind, wenn sie richtig gehandhabt werden, aus mehreren Gründen interessant. Sie können praktisch auf Knopfdruck ausgeführt werden und der Entwickler kann das Resultat auf einen Blick

beurteilen. Damit verschwindet die Mühsal von manuellen Tests. Die Tests können sehr einfach auch nach kleinen Änderungen durchgeführt werden, so dass Fehler oft sehr früh entdeckt werden. Änderungen an der Software, insbesondere Refactoring, werden dadurch erleichtert.

Wenn man dann noch die Tests gleichzeitig mit der Software entwickelt – "test a little, code a little" wird auch das Erstellen der Testsuite selbst recht einfach. Außerdem wird die Software von vornherein so entworfen, dass automatische Tests einfach zu schreiben sind. Das führt in der Regel auch zu gut modularisiertem und wartbarem Code.

Nicht zuletzt sind Tests auch eine Form der Dokumentation. Die Tests sind Beispiele dafür, wie der getestete Code eingesetzt werden kann. Sie bilden damit einen Teil der Kommunikation zwischen den Entwicklern, da sie zeigen, was sich die ursprünglichen Autoren des Codes während des Schreibens gedacht haben. Dies sind Informationen, die ohne Tests nur selten festgehalten werden.

Im folgen wollen wir theoretische Aspekte von automatischen Tests betrachten und mit Beispielen erläutern. Die Verwendung von automatischen Tests wie sie hier beschrieben wird ist im wesentlichen das "Test Driven Development". Die Beispiele sind in Python geschrieben und bilden eine Einführung in Pythons Testmodule.

2 Aufbau einer Testsuite

Betrachten wir zunächst den Aufbau einer Suite von automatischen Tests. Das Wort *Aufbau* ist hier doppeldeutig, denn es könnte sich einerseits um die Struktur der Testsuite handeln aber andererseits auch um den Prozess wie die Testsuite entsteht. In diesem Abschnitt wird letztlich beides behandelt, denn in der Praxis ist beides miteinander verknüpft.

Damit eine Testsuite aufgebaut wird, müssen alle beteiligten Entwickler die Testsuite verwenden und bei ihrer Pflege und weiterentwicklung mithelfen. Damit die Testsuite den Entwicklern auch nützt, muss sie einfach zu verwenden sein, ihre Beziehung zum zu testenden Code muss von den Entwicklern verstanden werden und ihr Aufbau - in beiden Bedeutungen - muss dies unterstützen.

2.1 Organisation des Quelltexts

Die Testsuite ist ein Teil des Quelltextes. Der testende Code und der getestete Code sind in gewissem Umfang komplementär. Die Struktur der Testsuite hat Einfluss auf die Struktur des getesteten Codes und umgekehrt. Durch die relativ enge Verzahnung von Testsuite und Code ist die Testsuite jeweils auf eine bestimmte Version zugeschnitten, muss also gemeinsam mit dem zu testenden Code in einem Versionskontrollsystem verwaltet werden.

Damit die Entwickler eine Testsuite häufig genug verwenden, muss es leicht sein sie aufzurufen und Erfolg oder Misserfolg eines Testlaufs muss auf einen Blick erkennbar sein.

In der Regel wird die Testsuite über ein einfaches Shellkommando aufgerufen. Das kann ein speziell Target im Makefile sein (etwa "make check") oder ein separates Script. Es ist sinnvoll wenn eine Möglichkeit besteht, zur Fehlersuche gezielt einzelne Tests ausführen zu können. Bei sehr umfangreichen Testsuites kann es nützlich sein nur eine gewisse Untermenge auszuführen um Zeit zu sparen. Dabei sollte man natürlich nicht vergessen, regelmäßig die gesamte Testsuite auszuführen.

Wie man den Quelltext der Testsuite organisiert, ist bei verschiedenen Projekten unterschiedlich. Bei manchen gibt es parallel zum eigentlichen Quelltextbaum eine separates Verzeichnis mit der Testsuite. Bei anderen hat jede Teilkomponente ihre eigene Suite. Man kann auch beides kombinieren, und Komponenten spezifische Tests mit der jeweiligen Komponente verwalten und zusätzlich noch Tests des Gesamtsystems in einem separaten Verzeichnis haben.

2.2 Einbindung in den Entwicklungsprozess

Die wichtigste Regel beim Einsatz von automatischen Tests ist:

Nach jeder Änderung muss die Testsuite erfolgreich durchlaufen werden.

Wenn Tests schiefehen, liegt der Grund dafür höchstwahrscheinlich in den gerade vorgenommenen Änderungen, so dass es leicht sein sollte, die Ursachen zu bestimmen und das Problem zu beheben.

Aus dem obigen Grundsatz folgt im übrigen, dass vor dem Einchecken ins Versionskontrollsystem die Testsuite erfolgreich ausgeführt worden sein muss.

Die Regel für den Ausbau der Testsuite ist:

Zu jedem neuen Feature gehören neue Tests

Wenn man während der Entwicklung eines neuen Features gleich auch entsprechende Tests schreibt, sorgt man dafür, dass die Testsuite diese auch gleich mit abdeckt. Außerdem kann man vielen Fällen die manuellen Tests sparen, da man von Anfang an einen passenden automatischen Test hat. Man muss nur darauf achten, nach jeder Änderung die Testsuite laufen zu lassen.

Es ist sinnvoll, die Tests vor dem neuen Code zu schreiben, da man so gezwungen ist, sich von vornherein Gedanken über die Testbarkeit der neuen Features zu machen. Dies hat gewöhnlich auch positive Auswirkungen auf das Design, wie wir unten noch sehen werden. Ferner vermeidet man so, dass das Schreiben der Tests als notwendige Pflichtübung gesehen wird, wie es leicht passieren kann, wenn die Tests nachträglich geschrieben werden und es so aussehen kann als schreibe man die Tests lediglich, um dem Prozess genüge zu tun und nicht, um die Qualität der Software zu sichern.

Leider kann auch eine Testsuite die Software nicht vollständig fehlerfrei machen. Daher gilt:

Zu jedem neuen Bugfix gehört ein Test

Ein Bug bedeutet, dass die Testsuite noch nicht vollständig genug ist. Nachdem man den Bug analysiert hat, sollte man zuerst einen Test hinzufügen, der den Fehler reproduziert und bei Vorhandensein des Bugs fehlschlägt. Zur Kontrolle der Funktionsfähigkeit des Tests sollte man die Testsuite einmal ablaufen lassen, um zu verifizieren, dass der Test funktioniert, also das Vorhandensein des Bugs erkennt. Dann behebt man den Fehler und führt die gesamte Testsuite aus, um zu prüfen, dass der Bug tatsächlich behoben wurde und auch alle anderen Tests noch erfolgreich sind.

2.3 Design

Wie beim Aufbau, so müssen wir auch beim Design sowohl das der Testsuite als auch das der zu testenden Software berücksichtigen. Beide Designs beeinflussen einander. Die zu testende Software muss so aufgebaut sein, dass automatische Tests leicht möglich sind, während die Testsuite so gestaltet sein sollte dass die einzelnen unabhängig voneinander sind und jeweils einen genau definierten Zweck haben.

Die Unabhängigkeit der einzelnen Tests dient dazu einen einzelnen Test allein ausführen zu können und dazu dass man die Funktionsweise eines Tests verstehen kann, ohne die anderen Tests zu kennen. Da bei einem Testlauf aber ein große Zahl von Tests ausgeführt wird, muss die zu testende Software so gestaltet sein, dass unabhängige Tests von Teilkomponenten möglich sind. Insbesondere globale Variablen oder Singletons können dies erschweren.

Manche Teile der zu testenden Software sind nur schwer automatisch zu testen. Dies trifft häufig zum Beispiel auf Datenbankanbindungen und grafische Benutzeroberflächen zu. In beiden Fällen kann man sich dadurch helfen, dass man die Programmlogik von der Datenbankschnittstelle bzw. der grafischen Oberfläche weitestgehend trennt.

Bei grafischen Benutzeroberflächen bietet sich zum Beispiel eine Model-View-Controller Architektur an. Das Model ist sowieso unabhängig von der eigentlichen Oberfläche und gegebenenfalls lassen sich auch einige Teile von View und Controller von der eigentlichen Benutzeroberfläche abstrahieren. Diese Teile sind dann einem automatischen Test leichter zugänglich. Außerdem ist Model-View-Controller generell ein sinnvoller Ansatz bei GUI-Programmen und sorgt für verständlicheren und wartbareren Code.

Während man bei der Benutzeroberfläche das Problem einfach dadurch löst, dass man die Schicht zwischen testbarem Code und GUI-Bibliothek so dünn und einfach wie möglich macht, ist das bei Datenbankanbindungen nicht ganz so einfach. Der zu Testende code hängt ja von der Datenbank ab und wird ganz ohne die Schnittstelle nicht testbar sein.

In einem solchen Fall kann man die Schnittstelle etwas abstrahieren, so dass man nicht eine konkrete Schnittstelle fest verdrahtet, sondern eine beliebige von außen vorgebbare Anbindung verwendet. Im produktiven Einsatz verwendet man die echte Datenbankanbindung und in den Tests verwendet man eine *Mock*-Datenbankanbindung, also eine Ersatzschnittstelle, die nur so tut, als greife sie auf eine Datenbank zu. Diese Trennung führt auch zu klarerem Code, da die Programmlogik deutlich vom Datenaustausch mit der Datenbank getrennt wird.

Solche Mock-Objekte haben sowohl Vorteile als auch Nachteile. Der Hauptvorteil ist, dass man mit ihnen in einem Test die volle Kontrolle über die Datenbankschnittstelle hat und auch leicht Situationen simulieren kann, die ohne sie nur schwer herstellbar wären. Es besteht allerdings die Gefahr, dass die Mock-Objekte sich eben doch nicht genau so verhalten wie die echten Objekte und damit der zu testende Code zwar in den Tests

korrekt zu sein scheint, sich in seltenen realen Situationen aber doch nicht richtig verhält, obwohl man meint, einen entsprechenden Test zu haben.

2.4 Testklassifikation

Wenden wir uns nun den eigentlichen Tests zu. Man kann einen Testfall nach verschiedenen Kriterien klassifizieren:

- Granularität: Betrachtet der Test eine einzelne Methode für sich genommen, eine ganze Komponente des Systems, etwa ein Dokumentobjekt mit einigem Inhalt, oder das System als Ganzes?
- Kopplungsgrad: Beruht der Test nur auf der Spezifikation der Öffentlichen Schnittstellen eines Objekts oder wird Wissen über die Interna verwendet?

Anhand der Granularität werden üblicherweise *Unittests*, *Integrationstests* und *Funktionstests* unterschieden und anhand des Kopplungsgrads *Black-Box Tests* und *White-Box Tests*.

Black-Box Tests testen eine Komponente nur anhand der Spezifikation ihrer Schnittstelle. White-Box Tests hingegen verwenden auch Wissen über Implementationsdetails. White-Box Tests können zum Beispiel sinnvoll sein, um sicherzugehen, dass alle Zweige in einer Funktion getestet werden.

Unittests testen die kleinste Code-Einheit für sich in Isolation, also zum Beispiel eine einzelne Funktion oder Klasseninstanz. Wenn das zu testende Objekt normalerweise mit anderen interagiert, kann es sinnvoll sein, Mock-Objekte als diese anderen Objekte zu verwenden, um die Isolation zu erhöhen.

Bei Integrationstests werden mehrere interagierende Komponenten zusammen getestet und bei Funktionstest schließlich das System als Ganzes. Funktionstests sind gewöhnlich Black-Box Tests.

Der tatsächliche Sprachgebrauch dieser Begriffe variiert etwas. Zum Beispiel werden häufig alle Tests in einer Testsuite als Unittests bezeichnet, auch wenn sie vielleicht eher Integrationstests oder Funktionstests sind.

2.5 Einführen einer Testsuite

Am einfachsten gestaltet sich die Einführung einer Testsuite bei ganz neuen Projekten. Wenn man gleich zu Anfang darauf achtet, für jedes neue Feature auch einen entsprechenden Test zu haben, erreicht man von Anfang an eine sehr gute Abdeckung des Codes.

Wenn die Architektur des Codes zu Anfang noch nicht feststeht ist es sinnvoll zuerst sehr grobkörnige Tests zu schreiben, also Funktionstests. Die Grundregel, dass man für jede neue Funktionalität auch neue Tests braucht, sollte man dabei aber nicht aus den Augen verlieren. Später, wenn sich das Design der Programms verfestigt, kann man dann feinkörnigere Tests hinzufügen.

Bei bestehenden Projekten, die noch keine Testsuite haben, gestaltet sich die Einführung schwieriger. Zunächst gilt es, ein psychologisches Problem zu überwinden. Die Entwickler müssen davon überzeugt werden, überhaupt damit anzufangen, eine Testsuite aufzubauen. Da eine neu eingeführte Testsuite zuerst nur einen sehr kleinen Teil der Software abdecken wird, kann es schwer sein, den Beteiligten ihren Nutzen zu vermitteln.

Dieses psychologische Problem stellt sich bei neuen Projekten zwar auch, gerade bei Freier Software sind bei neuen Projekten für gewöhnlich zunächst nur sehr wenige Programmierer beteiligt, so dass nicht sehr viele Leute zu überzeugen sind, und wenn die Testsuite von vornherein gut gepflegt ist, ist es leichter, Neulingen ihren Wert zu demonstrieren. Wenn Programmierer neu zu einem Projekt hinzukommen tragen sie ihren Code meist zuerst nur in Form von Patches bei. Das bietet die Möglichkeit nur solche Patches zu akzeptieren, die entsprechende Änderungen an der Testsuite beinhalten.

Wenn man die Entwickler von der Nützlichkeit automatischer Tests überzeugt hat, ist bei bestehenden Projekten das nächste Problem, wie man die Testsuite aufbaut. Am besten ist es, zunächst zumindest die oben aufgeführten Grundsätze anzuwenden, und für neue Features und Bugfixes entsprechende Tests zu schreiben. Dabei kann man dann die Gelegenheit nutzen und für das Modul, an dem man Änderungen vornehmen möchte zuerst ein paar Tests zu schreiben, die die korrekte Implementation der bereits vorhandenen Features testen. Gegebenenfalls bietet es sich auch an, den Code zu Refaktorisieren, um ihn besser testbar zu machen. Da testbarer Code meist auch ein besseres Design hat, als schlechter testbarer Code ergibt sich hier auch gleich noch ein weiteres Argument, um Entwickler zu überzeugen.

3 Beispiele

Nach der Theorie wollen wir und nun anhand von Beispielen genauer ansehen, wie man Tests schreibt. Die Beispiele sind alle in Python geschrieben. Daher werden hier einige Grundkenntnisse in dieser Sprache vorausgesetzt.

3.1 Pythons Modul `unittest`

Angenommen, wir brauchen eine Funktion, die einen Zeichenkette mit durch Leerzeichen, Tabulatoren und anderen Zwischenräumen getrennten ganzen Zahlen in eine Liste mit Integerobjekten umwandelt. Also soll etwa aus `'1 2 3'` die Liste `[1, 2, 3]` werden.

Im Sinne von Test Driven Development sollten wir uns erst einmal überlegen, welche Tests wir brauchen, und diese dann implementieren. Für die Implementation der Tests verwenden wir das Modul `unittest` aus Pythons Standardbibliothek, das im wesentlichen eine Python Variante des Java Testframeworks JUnit ist. Das Grundgerüst des Testmoduls, nennen wir es `test_parselist.py` sieht so aus:

```
import unittest
import parselist

class TestParseIntList(unittest.TestCase):

    def test_multiple(self):
        """Test parse_int_list with several ints"""
        self.assertEqual(parselist.parse_int_list("1 2 3"),
                        [1, 2, 3])

if __name__ == "__main__":
    unittest.main()
```

Die zu testende Funktion ist `parse_int_list` im Modul `parselist`. Wir importieren das Modul am Anfang.

Bei Tests, die mit dem `unittest` Modul implementiert sind, sind die einzelnen Testfälle mit Hilfe der Klasse `TestCase` implementiert. Und zwar leiten wir eine eigene Klasse von `TestCase` ab, in der wir unsere Tests als Methoden, deren Namen mit `test` anfangen, implementieren.

Die If-Anweisung am Ende sorgt dafür, dass `unittest.main()` aufgerufen wird, wenn `test_parselist` als Hauptprogramm des Pythoninterpreters ausgeführt wird, jedoch nicht, wenn es als normales Modul importiert wird. Dies ist nützlich wenn die Testsuite wächst und nicht mehr auf ein Modul beschränkt ist. Dann wird man in der Regel ein anderes Script haben, das alle Testmodule importiert und ihre Tests ausführt.

Die Funktion `unittest.main()` benutzt Pythons Möglichkeiten zur Introspektion, um zu bestimmen, welche von `TestCase` abgeleiteten Klassen es im Hauptmodul gibt und welche Testmethoden diese haben. Für jede der Testmethoden wird die Klasse dann einmal instanziiert und dann mit Hilfe einiger Infrastruktur in `unittest` ausgeführt.

Bisher ist lediglich ein einziger Testfall implementiert, um die Struktur zu demonstrieren. Wir haben uns aber noch nicht überlegt, welche möglichen Variationen im Eingabeformat wir unterstützen müssen. In der Praxis ist das häufig durch externe Spezifikationen wie RFCs vorgegeben, wenn man ein neues Format definiert hat man hier natürlich mehr Freiheiten.

Nehmen wir für unser Beispiel also an, dass die Zahlen ganze Zahlen in dezimalschreibweise sind und die Zahlen durch alle Arten von Leerzeichen getrennt werden dürfen, die in ASCII vorkommen, darunter Leerzeichen, Tabulatoren oder Newline. Der String darf leer sein oder nur aus Leerzeichen bestehen, dann soll auch entsprechend eine leere Liste zurückgegeben werden. Leerzeichen am Anfang oder Ende des Strings werden ignoriert, und zwischen zwei Zahlen müssen ein oder mehrere Leerzeichen sein. In Testmethoden ausgedrückt könnte das etwa so aussehen:

```

def test_multiple(self):
    """Test parse_int_list with several ints"""
    self.assertEqual(parselist.parse_int_list("1 2 3"),
                     [1, 2, 3])

def test_single(self):
    """Test parse_int_list with one ints"""
    self.assertEqual(parselist.parse_int_list("42"),
                     [42])

def test_empty(self):
    """Test parse_int_list with empty string"""
    self.assertEqual(parselist.parse_int_list(""),
                     [])

def test_whitespace(self):
    """Test parse_int_list with various amounts of whitespace"""
    self.assertEqual(parselist.parse_int_list(" 1 2\t 4"),
                     [1, 2, 4])

    self.assertEqual(parselist.parse_int_list("1\t2\n4\n"),
                     [1, 2, 4])

```

Alle Testfälle verwenden die Methode `assertEquals`, ihre beiden Parameter vergleicht und falls sie verschieden sind, eine Exception wirft, so dass der Test als fehlgeschlagen angesehen wird.

Bevor wir die Testsuite sinnvoll ausführen können, müssen wir erst noch das zu testende Modul `parselist.py` soweit implementieren, dass unsere Funktion ausgeführt werden kann, aber nicht soweit, dass sie schon die eigentliche Aufgabe erfüllt:

```

def parse_int_list(text):
    pass

```

Diese Funktion tut offensichtlich noch nichts. Sie gibt lediglich `None` zurück, wie alle Funktionen in Python, die keinen expliziten Rückgabewert haben.

Jetzt können wir unsere Testsuite ausführen. Wir erhalten folgendes (leicht gekürztes) Ergebnis:

```

$ python test_parselist.py
FFFF
=====
FAIL: Test parse_int_list with empty string
-----
Traceback (most recent call last):
  File "test_parselist.py", line 19, in test_empty
    []
  File "/usr/lib/python2.1/unittest.py", line 273, in failUnlessEqual

```

```

    raise self.failureException, (msg or '%s != %s' % (first, second))
AssertionError: None != []
=====
FAIL: Test parse_int_list with several ints
-----
Traceback (most recent call last):
[...]
AssertionError: None != [1, 2, 3]
=====
FAIL: Test parse_int_list with one ints
-----
Traceback (most recent call last):
[...]
AssertionError: None != [42]
=====
FAIL: Test parse_int_list with various amounts of whitespace
-----
Traceback (most recent call last):
[...]
AssertionError: None != [1, 2, 4]
-----
Ran 4 tests in 0.001s

FAILED (failures=4)

```

Wie man sieht, sind alle Test fehlgeschlagen. Während des Testlaufs wird für nach jedem einzelnen Testfall ein Zeichen ausgegeben, das Erfolg (".") oder Misserfolg ("F") anzeigt. Wenn ein unerwarteter Fehler auftritt, wird Ääusgegeben. In unserem Fall war die Ausgabe immer "F". Nach durchlauf der aller Tests wird für jeden fehlgeschlagenen Test der entsprechende Traceback ausgegeben, und ganz zum Schluss eine Zusammenfassung.

Dass die Tests zunächst fehlschlagen ist gewollt. Denn so können wir demonstrieren, dass die Tests die fehlerhafte Implementation bemerken. Nachdem dies demonstriert ist, können wir die Funktion tatsächlich implementieren:

```

def parse_int_list(text):
    return [int(num) for num in text.split()]

```

Jetzt sind die Tests erfolgreich:

```

$ python test_parselist.py

```

```

....

```

```

-----
Ran 4 tests in 0.000s

```

```

OK

```

Bisher haben wir nur gültige Eingaben für unsere Funktion in den Testfällen verwendet. Es gibt aber meist auch ungültige Eingabewerte, die als solche erkannt werden müssen. In unserem Fall wären das zum Beispiel

Strings, bei denen statt der Ziffern Buchstaben verwendet werden oder Gleitkommazahlen statt Integer verwendet werden. In solchen Fällen sollte die Funktion eine Exception werfen. Die jetzige Implementation tut das bereits, da `int` die Exception `ValueError` wirft, falls das Argument nicht in einen Integer umgewandelt werden kann. Dies ist erst einmal akzeptabel und wir sollten dieses Verhalten testen.

Für die Tests verwenden wir die `assertRaises` Methode, die als erstes Argument die erwartete Exception-Klasse erhält, als zweites die aufzurufende Funktion und in restlichen Argumenten die Parameter, mit denen die Funktion aufgerufen werden soll. `assertRaises` ruft dann die Funktion mit den gegebenen Argumenten auf, und testet, ob tatsächlich die Exception auftritt. Die Testfälle sehen dann etwa so aus:

```
def test_non_int(self):
    """Test parse_int_list with non-integers"""
    self.assertRaises(ValueError, parselist.parse_int_list, "1.0")

def test_non_number(self):
    """Test parse_int_list with non-numbers"""
    self.assertRaises(ValueError, parselist.parse_int_list, "1 abc")
```

Zum Abschluss diese kurzen Einführung in `unittest` noch ein paar fortgeschrittenere Features. Die `main` Funktion erlaubt es, gezielt einzelne Tests auszuführen, in dem man ihren Namen an der Kommandozeile angibt. Der Name setzt sich in unserem Fall aus Klassenname und Methodename zusammen. Um die Methode `test_empty` auszuführen könnte man den Namen `TestParseIntList.test_empty` verwenden:

```
$ python test_parselist.py TestParseIntList.test_empty
```

```
.
```

```
-----
Ran 1 tests in 0.000s
```

```
OK
```

Das hier in diesem Abschnitt gewählte Beispiel ist sehr einfach gehalten, damit wir uns auf die Tests selbst konzentrieren können. Daher braucht man nur einen kleinen Teil der Infrastruktur, die `unittest` bietet. In komplexeren Programmen kommt es zum Beispiel vor, dass man eine umfangreichere Datenstruktur herstellen muss damit man sie testen kann. Meist möchte man auch mehrere verschiedene Tests mit dieser Struktur durchführen. Für diesen Zweck gibt es die Methoden `setUp`, die immer vor der Testmethode aufgerufen wird und `tearDown`, die immer nachher aufgerufen wird, unabhängig davon, ob der Test erfolgreich war oder nicht. Beide Methoden kann man in abgeleiteten Klassen überschreiben und in `setUp` eine Datenstruktur aufbauen und sie in `tearDown` wieder sauber abbauen.

3.2 Pythons Modul `doctest`

Eine alternative zum `unittest` Modul ist das `doctest` Modul. Es wurde ursprünglich geschrieben, um Beispiele in Docstrings zu prüfen, um sicherzustellen, dass sie das Verhalten der Funktion richtig beschreiben. Es lässt sich aber auch umgekehrt einsetzen, um zu prüfen, ob die Funktion richtig funktioniert.

Nehmen wir wieder das Beispiel aus dem vorigen Abschnitt. Die Funktion zusammen mit einem sehr einfachen Docstring könnte so aussehen:

```

def parse_int_list(text):
    """Parse einen String mit einer Liste von ganzen Zahlen.

    Der Eingabestring sollte durch Whitespace getrennte ganze Zahlen in
    dezimalschreibweise enthalten. Der Rückgabewert ist eine Liste mit
    entsprechenden Integerobjekten. Beispiel:
    >>> from parselist import parse_int_list
    >>> parse_int_list('1 2 3')
    [1, 2, 3]
    """
    return [int(num) for num in text.split()]

```

Das Beispiel im Code ist einfach eine Kopie einer Interaktiven Session im Pythoninterpreter, so wie ein Benutzer es von Hand tun könnte, wenn er die Funktion ausprobieren möchte. Analog zu `test_parselist.py` können wir ein `doctest_parselist.py` schreiben:

```

import doctest
import parselist

doctest.testmod(parselist)

```

Die `testmod` Funktion im `doctest` Modul bekommt das zu testende Modul übergeben und sucht selbständig nach docstrings und testet die darin enthaltenen Beispiele. Standardmäßig wird nichts ausgegeben, wenn man dieses Script aufruft, falls alles in Ordnung ist. Mit der `-v` Option wird angezeigt, was geprüft wird:

```

$ python doctest_parselist.py -v
Running parselist.__doc__
0 of 0 examples failed in parselist.__doc__
Running parselist.parse_int_list.__doc__
Trying: from parselist import parse_int_list
Expecting: nothing
ok
Trying: parse_int_list('1 2 3')
Expecting: [1, 2, 3]
ok
0 of 2 examples failed in parselist.parse_int_list.__doc__
1 items had no tests:
  parselist
1 items passed all tests:
  2 tests in parselist.parse_int_list
2 tests in 2 items.
2 passed and 0 failed.
Test passed.

```

Wie man sieht war der Test erfolgreich. Wir könnten jetzt alle Tests aus dem vorigen Abschnitt auch hier in den Docstring aufnehmen, zur Demonstration soll es hier aber reichen, noch eine der Exceptions hinzuzufügen:

```

def parse_int_list(text):
    """Parse einen String mit einer Liste von ganzen Zahlen.

    Der Eingabestring sollte durch Whitespace getrennte ganze Zahlen in
    dezimalschreibweise enthalten. Der Rückgabewert ist eine Liste mit
    entsprechenden Integerobjekten. Beispiel:
    >>> from parselist import parse_int_list
    >>> parse_int_list('1 2 3')
    [1, 2, 3]

    Wenn nicht alle Elemente Zahlen sind, gibt es eine ValueError
    Exception:
    >>> parse_int_list("2 abc")
    Traceback (most recent call last):
      File "parselist.py", line 15, in parse_int_list
        return [int(num) for num in text.split()]
    ValueError: invalid literal for int(): abc
    """
    return [int(num) for num in text.split()]

```

Bei Exceptions vergleicht `doctest` nur die letzte Zeile. Das ist sinnvoll, da sich die Details des Tracebacks leicht ändern können und diese ja auch nicht Teil der Schnittstelle sind.

Doctests bieten eine interessante Alternative zu `unittest` basierten Tests, da sie Erläuterungen und Testcode in relativ natürlicher Weise verbinden können. Anders als `unittest` bietet `doctests` jedoch keine ausgefeilte Infrastruktur für große Testsuites. Seit Python 2.3 lassen sich `doctest` basierte Tests aber relativ einfach in das Framework des `unittest` Moduls einbinden.

4 Zusammenfassung

Der Artikel sollte deutlich gemacht haben, dass automatische Tests die Softwareentwicklung vereinfachen können, die Qualität des Codes und der Dokumentation verbessern können. Dies ist gerade auch bei Freier Software interessant, da die Reputation der Entwickler stärker von der Qualität ihrer Arbeit abhängt als bei proprietärer Software.

Der Einsatz einer Testsuite ist ein Zeichen von Professionalität in der Softwareentwicklung. Es wäre schön, wenn dieser Artikel dazu beiträgt, dass die Freie Software professioneller wird.