Hochschule für angewandte Wissenschaften Augsburg

Fakultät für Informatik

# Bachelorarbeit

**Python Test-Tools für Test-driven development im Vergleich**

**zur Erlangung des akademischen Grades**
**Bachelor of Science**

**Thema:** Python Test-Tools für Test-driven development im Vergleich

**Autor:** Maximilian Konter
maximilian.konter@hs-augsburg.de
MatNr. 951004

**Version vom:** 29. April 2019

**1. Betreuer:** Dipl.-Inf. (FH), Dipl.-De Erich Seifert, MA
**2. BetreuerIn:** Prof. Dr. X

Diese Arbeit befasst sich mit den aktuellen Tools zum testen von Python Software im Aspekt Test-driven development.

Dabei wird auf verschiedene Test-Tools eingegangen um diese dann zu vergleichen um so einen Leitfaden für das richtige Test-Tool zu bieten. Der Vergleich behandelt die Anwendbarkeit, Effizienz, Komplexität sowie die Erweiterbarkeit der Tools im Bezug auf Test-driven development.

Am Ende dieser Arbeit wird über die allgemeine Anwendbarkeit von Test-driven development diskutiert.

# Inhaltsverzeichnis

# Listingverzeichnis

# Glossar

**annotation** Eine Annotation ist das gleiche wie ein decorator, mit ausnahme, dass eine Annotation über jedem Wert, jeder Funktion und jeder Klasse stehen kann. Eine Annotation wird mit hilfe von @{name} über das zu annotierende Objekt geschrieben. Diese dienen dazu gewisse aktionen aus zu führen oder dem Programm etwas mit zu teilen, wie zum Beispiel, dass eine Methode veraltet ist. 32, 33

**bug** Ein Bug(zu deutsch Käfer) ist ein Fehler in einem Programm oder in einer Software. 17

**Command-line interface** Eine Benutzerschnitstelle für das Terminal. 9

**commit** Ein commit ist die Sammlung von änderungen an Dateien, welche mithilfe eines VCS verwaltet werden. 17

**contextmanager** Ein Contextmanager in Python ist ein Objekt, dass einen Kontext bietet, in dem gearbeitet werden kann. Dieser Kontext wird mit with contextmanager as c geöffnet und schließt sich selbst beim verlassen des Kontextes. Dadurch stehen dem entwickler der nutzbare Kontext als Variable c zur Verfügung. 19, 22

**decorator** Ein Decorator ist eine Funktion die eine andere Funktion oder Methode umschließt, wodurch der Code im Decorator vor und/oder nach der Funktion oder Methode ausgeführt werden kann. Ein Decorator wird einer anderen Funktion mit @{decorator−funktion} über der Funktions- oder Methoden-Definition übergeben. Das adjektiv für einen Decorator ist allerdings nicht dekorieren, sondern annotieren. 20, 22, 23

**docstring** Ein Docstring ist ein Block von Text der sich zwischen jeweils drei Anführungsstrichen befindet. Mit diesem Text wird ein Objekt in Python dokumentiert. 16

**fixture** Beschreibt die Präperation von Tests mit festen (fix) Werten, so wird zum Beispiel vor jedem Test eine Variable gesetzt. Dies dient der übersichtlichkeit, da dies dann nicht in jedem Test vorgenommen werden muss und um varianz zwischen Tests und Test läufen zu vermeiden. 11, 13–17, 19–21

**fuzz-testing** Beschreibt eine Art von Test, bei dem das zu testende Modul oder Programm mit zufälligen Werten aufgerufen wird. Dabei soll jede mögliche anwendung des Moduls oder Programms dargestellt werden. 11, 12, 19

**mock** Ein Mock (zu Detusch: Attrappe) ist ein Objekt, dass von dem zu mockenden Objekt abstammt. Das heißt es besitzt alle eigenschaften des originals, jedoch fügt es noch weitere Methoden hizu. Diese ermöglichen das prüfen, welche weiteren Methoden und Funktionen mit welchen Parametern ausgeführt wurden. 4, 11, 12, 14, 16–18, 20–22, 24–28, 31

**mock** Ein Stub (zu Deutsch: Stumpf) ist, ähnlich wie ein Mock, eine Attrappe. Jedoch werden hier nicht weitere Methoden hinzugefügt, sondern jediglich die vorhandenen Methoden werden ersetzt. Dabei wird allerdings keine Logik implementiert, sondern es wird ein Wert gesetzt den die Methode zurück gibt. 22–24, 26–30

**Paket-Manager** Ein Paket-Manager ist ein Tool zum verwalten von externen Python bibliotheken, mitdessen Hilfe Pakete (bibliotheken) installiert und versioniert werden können. 20

**StackTrace** Der StackTrace ist ein Protokoll der aufgerufenen Funktionen und Methoden die ineinander verschachtelt sind, bis hin zu der tiefsten Funkion/Methode die einen Fehler geworfen hat. 15

**Versions Control System** Ein VCS ist ein Dienst, der es seinen Nutzern ermöglicht änderungen an Dateien in einer Chronik zu speichern um später darauf zu zu greifen. Dies dient auch der verteilung von Daten über mehrere Systeme sowie sicherung der Daten vor verlust. 9

# Abkürzungsverzeichnis

| | |
|---|---|
| CLI | Command-line interface |
| GNU | GNU is not Unix |
| GPL | GNU General Public License |
| GUI | Graphical User Interface |
| LGPL | GNU Lesser General Public License |
| STDLIB | Python Standard Bibliothek |
| TDD | Test-driven development |
| VCS | Versions Control System |

# 1 Einleitung

TDD wird in der heutigen Softwareentwicklung immer verbreiteter und beliebter. Die Ansprüche an Software sind in den letzten Jahren immer weiter gestiegen. Dies liegt vor allem an der Reichweite, die Software heute hat. So besitzen nach Schobelt, 2017 im Jahr 2018 bereits circa 66% aller Menschen ein Smartphone. Im Arbeitsleben ist ein PC meist gar nicht mehr weg zu denken. Doch mit den steigenden Nutzerzahlen steigen auch die Anforderungen, welche die Nutzer an die Software stellen. Somit wird die Anzahl der gefundenen Bugs dementsprechend größer.

Im weltweiten Markt gibt es viele große Unternehmen, die gegenseitig um die Nutzer kämpfen. Selbstverständlich präferieren die Nutzer denjenigen Anbieter, welcher die bessere Software bietet. Dies kann sich heute jedoch stetig ändern. Mit der steigenden Anzahl an Bugs, die gefunden werden, steigt auch die Anzahl der Nutzer, die von diesen Bugs betroffen sind. Diese Bugs sollen natürlich schnellstmöglich gefixt werden um so zu verhindern, dass die Nutzer die Software wechseln.

So schwer es ist seine Nutzer zu halten, umso schwerer ist es, zum Start einer Software Nutzer zu akquirieren. Es gibt bereits Software, die ähnliche Services anbieten. So ist es noch schwerer dem Markt beizutreten. Die Anforderungen werden durch die bereits am Mark tätigen Firmen gesetzt, damit sollten Fehler, die bereits gelöst wurden nicht mehr auftauchen.

Für Unternehmen sind diese Anforderungen meist schwer zu meistern, weshalb Software meist mit Fehlern released wird, um diese dann von den Nutzern aufdecken zu lassen und zu fixen.

Sowohl als Entwickler als auch als Arbeitgeber muss man sich bei der Wahl der Programmiersprache Gedanken darüber ob und wie einfach eine Sprache für Test-driven development ein zu setzen ist. Der wichtigste Aspekt bei diesem Prozess ist die Auswahl und die Qualität der von der Sprache zur Verfügung gestellten Tools.

## 1.1 Die Programmiersprache Python

In diesem Kapitel wird die Programmiersprache Python beschrieben, um einen genauen Überblick über die Sprache zu bekommen mit dem diese Arbeit besser verstanden werden kann.

## 1.2 Test-driven development

In diesem Kapitel wird das Thema testen und Test-driven development behandelt um ein Grundlegendes Verständnis von TDD zu schaffen mit welchem diese Arbeit besser zu verstehen ist.

# 2 Python Test-Tools

Dieses Kapitel befasst sich mit den von der STDLIB bereitgestellten Test-Tools sowie denen aus externen Paketen. Diese werden unter 2.1 und 2.2 zusammengefasst, wobei diese unterteilt sind in unit-testing -, mock-testing - und fuzz-testing Tools.

Die unit-testing Tools sind Tools die Funktionalität zum testen bereitstellen. Jedoch wird für TDD weit mehr als nur Tests benötigt, aus diesem Grund werden mock-testing - und fuzz-testing Tools zusätzlich behandelt. Dabei soll aber zwischen einer reinen Erweiterung eines Tools und der Erweiterung von allen Tools unterschieden werden. Ist zum Beispiel ein Tool nur zusammen mit einem anderen Tool, das die Funktionalität bereit stellt Tests aus zu führen, so ist dieses Tool als Erweiterung zu sehen. Bietet ein Tool allerdings Code zum Erweitern von verschiedenen Test Tools zur Verfügung, so wird es hier zur Analyse verwendet. Diese Unterscheidung wird verwendet um Duplikationen in den einzelnen vergleichen zu vermeiden, des weiteren gibt es Tools mit x Erweiterungen, diese alle zu vergleichen würde den Rahmen dieser Arbeit bei weitem sprengen. Sollten sich für ein Tool besonders interessante Erweiterungen finden, so werden diese in der Analyse des jeweiligen Tools erwähnt und verlinkt.

Jedes Tool wird anhand folgender Aspekte untersucht:

- Anwendbarkeit:
  Bietet das Tool alles, um TDD betreiben zu können? (Fixtures und Mocks) Mit wie vielen Paketen muss das Tool betrieben werden? (Mehr Abhängigkeiten führen zu mehr externe Entwicklern auf die man sich verlassen muss.)
  - Bei mock-testing- und fuzz-testing-testing-Tools wird hier auf die Features die das jeweilige Tool bietet geprüft.
- Effizienz:
  Wie viel lässt sich mit diesem Tool möglichst einfach und schnell erreichen? Ist besonders viel Vorarbeit notwendig um die Tests auf zu setzen oder kann sofort

mit dem Schreiben der Tests begonnen werden?
Genauso stellt sich die Frage, wie Effizient der Entwickler die Tests auswerten kann (Nur bei Tools mit test-runner).

- Komplexität:
  Wie komplex ist das Tool? Das heißt, wie viel Funktionalität bietet das Tool dem Entwickler von Haus aus, aber auch wie schwer ist es einen Code zu schreiben oder wie schnell wird ein Code unübersichtlich, da das Tool viel Code abseits der Tests benötigt.

- Erweiterbarkeit:
  Wie leicht lässt sich das Tool mit anderen Tools erweitern? Gibt es vielleicht Erweiterungen der Community für dieses Tool, die sehr hilfreich sind?

  – Dieser Punkt wird bei mock-testing - und fuzz-testing-testing Tools ignoriert, da diese selbst Erweiterungen darstellen.

Zum Vergleich der einzelnen unit-testing Tools untereinander werden sie auf den in Listing 1 abgebildeten Code angewandt, da sich diese Arbeit auch mit mock-testing - und fuzz-testing-testing Tools beschäftigt wurden auch für diese Code geschrieben der zu testen ist. Der Code für die mock-testing Tools befindet sich in Listing 2 und der für fuzz-testing-testing in Listing 3. Dadurch lassen sich die unterschiedlichen Anforderungen zwischen den Arten der Tools besser vergleichen.

Das Modul für unit-testing enthält eine selbst geschriebene Implementierung der Funktion pow(), welche eine Zahl a mit einer Zahl b quadriert. Um die Komplexität zu erhöhen wurde eine in-memory Datenbank implementiert, welche Items enthält. Jedes Item hat eine ID (id), einen Namen (name), einen Lager Platz (storage_location) und eine Anzahl der vorrätigen Items amount. Jedes Item besitzt zudem eine Methode do_something() welche eine externe Funktion/Methode aufruft, die jedoch noch nicht existiert do_something_which_does_not_exist().

Für die mock-testing Tools wurde eine Klasse geschrieben, welche Funktionen besitzt die in ihrem Namen ausdrücken welche Methode sie ausführen, jedoch macht nicht jede Methode das was sie soll. Die mock-testing Tools sollen hier die Methoden so mocken, dass sie das jeweilige ausführen. Selbstverständlich ist realer Code viel komplexer als in diesem Listing (2) dargestellt, jedoch reicht dieser Code aus um viele Tools an Ihre grenzen zu bringen. So muss in call_internal_functin_n_times überprüft werden ob sie n mal aufgerufen wurde, auch call_helper_help wird nicht ohne weiteres funktionieren, da die Klasse Helper noch nicht implementiert wurde (Was in TDD sehr oft der Fall sein wird).

Auch die Methode return_false_filepath bringt das ein oder andere Tool zum schwitzen, da eine externe Methode aus der STDLIB ersetzt werden muss, wobei dies nicht global geschehen darf.

TODO fuzzetsting

Verfügt ein Tool über keinen test-runner, so wird der von der STDLIB gestellte runner unittest verwendet.

## 2.1 Tools der Standard Bibliothek

Die Standard Bibliothek von Python bietet zwei verschiedene Test-Tools(Muthukadan et al., 2011). Zum einen ist dies unittest [1] und zum anderen doctest [2] . Diese beiden Tools reichen sind im ihrem Umfang bereits so vielseitig dass, es einfach ist eine hohe Test-Abdeckung eines Programms oder eine Bibliothek zu erreichen.

Beide Tools zählen zu den „Unit Testing Tools“(Muthukadan et al., 2011) - auf Deutsch Modul Test-Tools - mit deren Hilfe die einzelnen Module eines Programms getestet werden können. In einem Programm oder einer Bibliothek wären dies die einzelnen Funktionen und Methoden.

### 2.1.1 unittest

Das von JUnit inspirierte (Brandl, van Rossum, Heimes, Peterson, Pitrou et al., 2007) Tool unittest, ist bestand der Python Standardbibliothek und bietet seit jeher seinen Nutzern ein umfangreiches Repertoire an Funktionen zum testen von Python Code. Die Funktionen von unittest lassen sich unter folgenden Punkten beschreiben:

- Fixture, zum präparieren der Tests.
- Test Fälle, zum gliedern einzelner Tests.
- Testumgebungen, zum gliedern von zusammengehörigen Tests.
- Test runner, zum ausführen von Testumgebungen oder Test Fällen.

Mithilfe der genannten Punkte ist es dem Entwickler möglich eine Stabile Test Umgebung auf zu bauen. Jedoch bietet unittest alleine nicht alles um TDD betreiben zu können.

[1] http://pyunit.sourceforge.net/pyunit.html
[2] https://docs.python.org/3/library/doctest.html

Bei TDD werden zuerst die Tests und dann die Funktionalitäten geschrieben, daher muss es möglich sein andere Module(units) zu mocken auf denen ein Test basiert. Durch die Fixtures ist es bereits möglich den Test oder die Tests so vor zu bereiten, dass diese funktionieren, jedoch bieten Mocks einfachere und schnellere Möglichkeiten Funktionen, Methoden, Klassen usw. zu imitieren.

Jedoch gibt es in der STDLIB eine Erweiterung zu unittest mit dem Namen `unittest.mock` welche unter eben diesem importiert werden kann um die mocking Funktionalität zu bekommen. Diese Erweiterung, auch als submodule bezeichnet ist Teil der STDLIB seit Python 3.3 wie in PEP417 von Foord et al., 2012 definiert wird.

Das Tool bietet des weiteren einen CLI, mit welchem es dem Benutzer möglich ist seine Tests gebündelt aus zu führen und aus zu werten. Mit dem CLI ist es auch auch möglich automatisch Tests in einem Ordner zu „entdecken“(discover) und aus zu führen. Dadurch ist es sehr leicht neue Tests in ein bestehendes Test System ein zu führen und diese ohne Veränderungen am bestehenden System aus zu führen.

Im Aspekt Anwendbarkeit bietet `unittest` alles um als Tool für TDD in Frage zu kommen, jedoch nur unter Einbezug der Erweiterung `unittest.mock`.

Mit Hilfe von unittest lässt sich sehr einfach und schnell ein Test schreiben, so würde das der Code aus Listing 4 bereits unsere selbst geschriebene quadrat-Funktion aus Listing 1 testen, einmal mit unserem selbst berechneten Wert und einmal gegen den wert der quadrat-Funktion aus der STDLIB.

Mit `self.assertEqual` wird überprüft ob der erste Wert dem zweiten Wert gleicht. Im ersten `assert` wird auf einen im Kopf ausgerechneten Wert geprüft und im zweiten wird die von Python gegebene Methode zum überprüfen verwendet.

Die basis-Funktionalität von unittest ist schnell verstanden und setzt sich eigentlich nur aus `self.assert{irgendwas}(...)` zusammen. Hat man die richtige assert Funktion gefunden lässt sich eigentlich jede unabhängige Funktion testen.

Möchte man allerdings fortgeschrittenere Tests schreiben so muss man sich der Dokumentation bedienen, welche unter https://docs.python.org/3/library/unittest.html zu finden ist. Würde man die Seite als PDF herunterladen so wären dies 58 Seiten Fließtext. Möchte man nun zum Beispiel vor den Tests etwas vorbereiten oder präparieren so lässt sich mit `setUp()` und `tearDown()` dies realisieren, diese zwei Methoden überschreiben die Methoden aus `unittest.TestCase` und werden vor jeder Funktion ausgeführt. Das Gleiche gibt es auch für den Test Fall, bei dem `setUp()` und `tearDown()` allerdings nur beim eintritt

der Klasse und beim austritt ausgeführt werden. Diese Methoden sind die sogenannten Fixtures.

Die folgenden Listings 5 und 6 zeigen wie ein Erfolgreicher Test und wie ein Misslungener Test aussehen.

In Beiden Listings ist gut zu erkennen ob und was schief gegangen ist bei einem Test. Der erfolgreiche Test zeigt dem Nutzer sofort wie viele Test in wie vielen Sekunden gelaufen sind, und ist dies erwünscht kann der Nutzer mit −−verbose sich noch mehr Informationen anzeigen lassen.

Bei misslungenen Tests ist im Output immer der StackTrace abgebildet um so den Fehler bis zur Wurzel zurück verfolgen zu können. Auch der Fehler selbst wird in den Output geschrieben, wie in Zeile 8 in Listing 6 zu erkennen ist. Am Ende wird auch noch einmal angezeigt wie viele Tests schief gelaufen sind.

Die Effizient Tests aus zu werten ist also sehr hoch, jedoch wird der Output bei Mehreren Fehlern und langem StackTrace sehr schnell sehr unübersichtlich für das Terminal. Ein Farbiger Output würde hier Abhilfe schaffen.

Um die Komplexität von unittest dar zu stellen wurde mithilfe des in Listing 1 definierten Codes ein Fortgeschrittener Test geschrieben, welcher die Basis Features von unittest umfasst. Dieser Code befindet sich in Listing 7.

Unittest verfügt über einige interessante Erweiterungen, welche das Tools mit neuen Funktionalitäten auffrischen. Die folgende Auflistung zeigt ein paar dieser Erweiterungen.

- nose [3] (veraltet)
- nose2 [4]
- twisted [5]
- testtools [6]

### 2.1.2 doctest

„Das doctest Modul sucht nach Textstücken, die wie interaktive Python-Sitzungen aussehen, und führt diese Sitzungen dann aus, um sicherzustellen, dass sie genau wie gezeigt funktionieren.“ (Brandl, van Rossum, Heimes, Peterson, Melotti et al., 2007). Diese

[3] https://github.com/nose-devs/nose
[4] https://github.com/nose-devs/nose2
[5] https://github.com/twisted/twisted
[6] https://launchpad.net/testtools

Textstücke müssen sich in Kommentaren befinden, da sie sonst von Python als Code interpretiert werden.

Wie unittest, ist auch doctest in der STDLIB, wodurch keine externen Abhängigkeiten geladen werden müssen.

Doctest bietet dem Nutzer eine Möglichkeit mithilfe von einem Test den Code zu dokumentieren. Da doctest lediglich Code ausführt wie in einer interaktiven Python-Sitzung, ist auch nur das möglich aus zu führen was im Code geschrieben ist. Test Fälle oder gar Mocks sind hierbei nicht möglich, Fixtures hingegen sind teilweise realisierbar in Form von Code der vor dem Test ausgeführt wird.

Doctest selbst nutzt keine Assert Funktionen oder Methoden, stattdessen wird der Output eines Befehls überprüft. Dabei spielt der Typ auch eine Rolle, so ergibt eine Funktion return_3(s=None, i=None) bei s=True eine '3' und bei i=True eine 3. Dieses Beispiel ist in Listing 8 zu sehen, um einen Output zu erzeugen wurde noch ein fehlerhafter Test hinzu gefügt. Dieser Output ist in Listing 9 zu sehen.

Möchte der Entwickler einen etwas größeren Test schreiben, so kann er sich einer Funktionalität von doctest bedienen, die es Ihm ermöglicht Tests in eine externe Datei zu verlagern. Dabei wird die Datei als ein Docstring behandelt, wodurch sie keine """ benötigt. Um jedoch Code in diese Datei ausführen zu können muss das zu testende Modul importiert werden.

Durch das schreiben der Tests in den Docstrings werden die Module in denen Tests geschrieben wurden schnell sehr unübersichtlich und lang, wenn der Entwickler große Tests schreibt. Zwar wird durch externe Textdateien Abhilfe geschaffen, dennoch sind doctests wenn sie länger werden schwer zu lesen und nach zu vollziehen.

Die Komplexität als auch die Anforderungen für doctest sind sehr gering. Das testen geschieht mithilfe der interaktiven Python-Sitzung, welche jedem Python Entwickler bekannt ist. Die Tests selbst sind aufrufe der geschriebenen Funktionen und Methoden und ein Abgleich des Outputs mit dem Erwarteten Wert.

Das Listing 11 zeigt den Code aus 1 mit Docstrings versehen. Dieser Test wurde mithilfe der main Funktion von Python ausgeführt. Der in Listing 12 ausgelagerte Test, wurde mithilfe des CLIs von doctest ausgeführt. Da beide Test keine Fehler werfen existiert auch kein Output für diese Tests.

Die einzige Möglichkeit hier einen Output zu bekommen wäre mit -v im CLI oder verbose=True im Funktionsaufruf. Die dort dargestellten Informationen sind lediglich welche Kommandos ausgeführt wurden, was erwartet wurde und das, dass erwartete eingetroffen ist. Ein kleines Beispiel ist in Listing 10 zu sehen.

Da doctest selbst keine Test Fälle unterstützt besitzt unittest eine Integration für doctest, diese ermöglicht es Tests aus Kommentaren sowie Textdateien in unittest zu integrieren und zu gliedern.

Da sich mit Doctest leider Funktionen nicht präparieren lassen, ist dieses Test Modul für die alleinige Anwendung in TDD nicht nutzbar. Das Tool bietet keine Möglichkeiten Objekte zu mocken wodurch Tests an nicht implementieren Methoden und Funktionen scheitern. Auch Fixtures sind nur bedingt realisierbar und benötigen viel Code der dupliziert werden muss, da dieser vor und nach jedem Test geschrieben werden muss.

## 2.2 Tools abseits der Standard Bibliothek

Abseits der Standard Bibliothek gibt es einige Tools deren Nutzung von Vorteil gegenüber der STDLIB ist. Diese werden unter diesem Punkt aufgeführt.

Auf https://wiki.python.org/moin/PythonTestingToolsTaxonomy werden viele externe Tools gelistet, jedoch scheinen viele inaktiv zu sein da ihre commits teilweise mehr als ein Jahr zurück liegen. Dies ist weder für eine Bibliothek noch für ein Tool ein gutes Zeichen, da sich die Anforderungen stetig ändern und niemals alle Bugs gefixt sind.

Manche der dort aufgelisteten Tools sind bereits oder werden gerade in andere Tools integriert. Dies kann zum einem sein, da ein Tool eine Erweiterung für ein anderes war und die Entwickler die Änderungen angenommen haben und zum anderen um die Tools zu verbessern und mehr Entwickler zur Verfügung zu haben. Eventuell sind auch andere Gründe dafür verantwortlich, jedoch war dies der Grund bei zum Beispiel Testify von Yelp [7] .

Da sowohl Software als auch Programmiersprachen sich stetig weiter entwickeln werden in dieser Arbeit nur jene Tools behandelt die sich diesen Entwicklungen Anpassen, sei dies durch das unterstützen der aktuellsten Version von Python als auch durch neue Innovationen, sowie Bug-fixes. Aus diesem Grund werden Tools deren letzter Commit weiter als ein Jahr zurück liegt hier nicht behandelt.

Lässt man zusätzlich die Erweiterungen von Tool zunächst außen vor, so bleiben nach Muthukadan et al., 2011 folgende Modul Test-Tools zur Verfügung:

[7] https://github.com/Yelp/Testify/

- pytest [8]

Tools wie reahl.tofu[9] oder zope.testing[10] sind zwar mehr oder weniger aktiv, da sie beide auf die neusten Python Versionen patchen, jedoch bieten sie sonst keinen Mehrwert in ihren Updates. reahl.tofu[9] selbst ist auch nur eine Erweiterung für bestehende Test Tools, wie zum Beispiel pytest, weshalb es hier nicht behandelt wird. So wird zope.testing[10] auch nicht behandelt, da dieses Tool wie bereits beschrieben nicht aktiv weiter entwickelt wird und es eher dafür gemacht wurde zope [11] Applikationen zu testen und nicht Python Applikationen.

Der Test-Runner nose [12] , der eine Erweiterung zu unittest bietet ist nicht mehr in Entwicklung, dennoch haben sich ein paar Liebhaber des Tools zusammengeschlossen und nose2[13] geschrieben. Da nose2[13] wie sein Vorfahre eine Erweiterung zu unittest darstellt wird es hier nicht analysiert.

Auch sehr bekannt ist testtools [14] , welches jedoch als Erweiterung zu unittest gesehen werden muss. Jedoch findet dort auch keine wirkliche weiter Entwicklung statt.

Die Quelle aus dem Offiziellen Python Wiki beschreibt sonst keine weiteren Tools. Auch die suche auf einschlägigen Suchmaschinen liefert sonst keine relevanten Tools.

Als weitere Kategorie werden Mock-Tools geführt. Auch wenn fast alle unittest-Tools integriertes mocking haben, so lässt sich mit diesen Tools meist mehr erreichen. Es wurden die gleichen Filter-Kriterien verwendet wie bei den Test-Tools oben.

- stubble [15]
- mocktest [16]
- flexmock [17]
- python-doublex [18]

[8] https://github.com/pytest-dev/pytest/
[9] https://www.reahl.org/docs/4.0/devtools/tofu.d.html
[10] https://pypi.org/project/zope.testing/
[11] http://www.zope.org/en/latest/
[12] https://pypi.org/project/nose/1.3.7/
[13] https://pypi.org/project/nose2/
[14] https://pypi.org/project/testtools/
[15] https://www.reahl.org/docs/4.0/devtools/stubble.d.html
[16] https://github.com/timbertson/mocktest/tree/master
[17] https://github.com/bkabrda/flexmock
[18] https://bitbucket.org/DavidVilla/python-doublex

- python-aspectlib [19]

Auch hier finden sich sonst keine weiteren Tools die Open-source sind.

Als letzte Kategorie werden hier die Fuzz-testing Tools behandelt, da diese eine gute Möglichkeit bieten Code ausgiebig zu testen. Das wohl umfangreichste und nach den obigen Kriterien einzige Tool ist hypothesis [20] .

### 2.2.1 pytest

Das am 04. August 2009 in Version 1.0.0 veröffentlichte Tool pytest (auch py.test genannt) ist ein sehr umfangreiches und weit entwickeltes Tool. Seit 2009 wird das Tool stets weiter entwickelt und vorangetrieben, wodurch es eine Menge an Features gewonnen hat. Die Basis Features von pytest sind folgende:

- Simple `assert` statements.

  Kein `self.assert`

  Error Überprüfung mit Contextmanager

- Informativer Output in Farbe

  Der gesamte Output kann angepasst werden

- Feature reiche Fixtures

  Vordefinierte Fixtures von pytest

  Geteilte Fixtures unter Tests

  Globale Fixtures zwischen Modulen

  Parametrisierung von Test als Fixtures

- Überprüfung von `stdout` und `stderr`

- Gliedern von Test in Fällen

  ... durch Markierung

  ... durch Nodes (Auswahl der Modul Abhängigkeit)

  ... durch String Abgleich der Funktions-Namen

[19] https://github.com/ionelmc/python-aspectlib
[20] https://github.com/HypothesisWorks/hypothesis

Da pytest nicht in der STDLIB ist, muss es mit einem Paket-Manager installiert werden. Dabei werden für pytest 4.4.0 die in Listing 13 gezeigten Abhängigkeiten installiert. Demnach benötigt pytest sechs externe Abhängigkeiten zusätzlich zu sich selbst.

Wie anhand der Basis Features erkennbar ist, bietet pytest einiges um Tests zu schreiben und aus zu führen. So ist mit den gebotenen Fixtures bereits eine Voraussetzung mehr als erfüllt, da pytest viele verschiedene Arten bietet Fixtures zu benutzen. Fixtures werden in pytest allerdings nicht mit setUp und tearDown geschrieben, sondern werden mithilfe eines decorators markiert und den Test Funktionen als Parameter übergeben. Um die setUp und tearDown funktionalität zu bekommen muss lediglich das keyword yield verwendet werden, die Fixture wird dann den Code bis zum yield ausführen und nach der Funktion den Rest nach yield ausführen.

Diese lassen sich durch zusätzliche Parameter weiter anpassen. Diese Funktionen sind in Kapitel fünf nach Krekel und pytest-dev team, 2019 beschrieben.

Des weiteren bietet pytest auch mocking, so lässt sich beispielsweise mit monkeypatch.setattr() der return Wert einer Funktion ersetzen. Dies wird in pytest automatisch am ende der Funktion rückgängig gemacht, wodurch der Entwickler sich mehr auf das eigentliche Testen konzentrieren kann.

Auch hier bietet pytest weitere Möglichkeiten Mocks zu verwenden, dazu hat Krekel und pytest-dev team, 2019 in Kapitel sieben einige Worte geschrieben.

Ein weiteres sehr praktisches Feature von pytest ist die Gliederung in Fälle. Dies ist bei pytest sehr fein einstellbar, so lässt sich wie in den Basis Features bereits beschrieben ein Test mit einer oder mehreren Markierungen versehen wodurch eine Gliederung nach Markierung entsteht. Auch durch das selektieren bestimmten Wörtern lassen sich Tests nach ihrem Namen gliedern, so entsteht einerseits eine Gliederung zur Ausführung der Tests und andererseits eine Gliederung für die Entwickler die sie selbst im Code sehen können. Als letzte alternative lassen sich Tests anhand ihrer Module ausführen, dies geschieht durch die Angabe der Module. So würde test_datei.py::TestKlasse::test_methode den Test test_methode der Klasse TestKlasse in der Datei test_datei.py ausführen.

Diese Features werden in Kapitel sechs von Krekel und pytest-dev team, 2019 beschrieben.

Selbstverständlich bietet pytest noch weitere Features jedoch sind diese nicht zwangsläufig notwendig um TDD zu betreiben und sind mehr ein nice to have Feature als

wirklich benötigt. Für eine Vollständige Auflistung und Erklärung aller Features kann jederzeit unter https://docs.pytest.org/en/latest/ die aktuellste Version der Dokumentation abgefragt werden.

Demnach bietet pytest alles um TDD anwenden zu können und die sieben zusätzlichen Abhängigkeiten die ein Entwickler bei der Nutzung von pytest eingeht sollten keinen Entwickler davon abhalten pytest und seine Features zu genießen.

Da pytest, wie bereits erwähnt keine neuen assert Methoden hinzufügt lässt sich sehr schnell und einfach ein Test schreiben. Selbst die Nutzung von Fixtures ist in pytest sehr einfach, da lediglich eine Funktion geschrieben werden muss die mit `@pytest.fixture` markiert wurde und der Test Funktion oder Methode als Parameter übergeben wird. Den Rest erledigt pytest selbst im Hintergrund. Genauso einfach gestaltet sich die Nutzung von Mocks.

Die Effizienz, die bei der Nutzung von pytest entsteht ist demnach sehr hoch. Denn der Entwickler muss nicht wirklich etwas neues dazu lernen um Tests zu schreiben oder zu präparieren. Genauso leicht kann ein Entwickler den Output von pytest auswerten, da dieser erstens, in Farbe ist, was die Lesbarkeit deutlich erhöht, zweitens sehr gut gegliedert ist und Wichtige Strukturen klar darstellt und drittens nach den Wünschen der Entwickler sich gestalten und verbessern lässt.

Durch die eben genannten Features im Bezug auf das schreiben von Tests mit assert, sowie Fixtures und Mocks lässt sich sagen das pytest sehr viel Funktionalität bietet wobei es trotzdem Struktur im Code bietet und komplexe Features einfach ermöglicht. Demnach kann ein Entwickler mit wenig Code viel Testing Funktionalität erstellen.

Sollten dem Entwickler die Features von pytest nicht ausreichen, so findet man unter http://plugincompat.herokuapp.com/ eine Liste von 618 (Stand: 2. April 2019) Erweiterungen für `pytest 4.3.0`. pytest selbst kann allerdings auch als Erweiterung zu unittest genutzt werden, indem ein Entwickler pytest zum ausführen der unittests verwendet. Dadurch bietet sich dem Entwickler ein verbesserter Output des Test Ergebnisses.

Im Listing 14 und 15 befindet sich der Code und der Output zum Test von dem in Listing 1 definierten Modul.

### 2.2.2 Mocking Tools

Um einen bessere Gliederung zur Verfügung zu stellen, werden die Mocking Tools in diesem Kapitel gegliedert.

Wie in Python Test-Tools bereits beschrieben, werden hier Tools analysiert, die eine Erweiterung für Test Tools zur Verfügung stellen im Bezug auf verbessertes mocking.

#### 2.2.2.1 stubble

Stubble zählt zwar zu den mocking Tools, jedoch werden hier Mocks gegeben. So lässt sich ein Objekt durch ein selbst geschriebenes Objekt ersetzen.

Angenommen man hat ein Objekt `class Original` welches eine Abhängigkeit zu dem zu testenden Objekt darstellt, so lässt sich mit stubble ein Objekt `class Fake` erstellen welches einen decorator besitzt `@stubclass(Original)`. Dadurch wird überprüft, dass alle Methoden aus `class Fake` der Signatur derer aus `class Original` entsprechen. Wenn `class Fake` von `class Original` erbt, werden nur die Funktionen überschrieben die in `class Fake` definiert wurden.

Um allerdings kleine Helfer zu `class Fake` hinzu zu fügen, müssen diese mit `@exempt` annotiert werden. Ist das objekt von `class Original` bereits initialisiert, so lässt es sich an `class Fake` übergeben, wenn diese von `reahl.stubble.Delegate` erbt, so lässt sich zur Laufzeit ein Objekt durch einen Mock ersetzen. Als letztes Hauptfeature wird die Möglichkeit geboten Mocks mit setuptools zu verwenden.

Die Hauptfeatures von stubble sind jedoch die vorgefertigten Mocks. So werden dem Entwickler folgende Mocks geboten:

- SystemOutStub [21]

  Ein Stub, der den Standard Out ersetzt

- CallMonitor [22]

  Ein Stub, zum überprüfen welche Methoden wann, wie und wie oft aufgerufen wurde

Diese Mocks lassen sich als contextmanagers einsetzen, wie auch ein weiteres Feature von stubble. `replaced` lässt den Entwickler eine Methode oder Funktion durch eine andere ersetzen, solange der contextmanagers aktiv ist.

[21] https://www.reahl.org/docs/4.0/devtools/stubble.d.html#systemoutstub
[22] https://www.reahl.org/docs/4.0/devtools/stubble.d.html#callmonitor

Des weiteren bietet stubble einige Experimentelle Features die, nach Vosloo, Sparks und Nagel, 2018 eher weniger für den praktischen Einsatz geeignet sind, aber dennoch interessante Features darstellen. So ist es möglich `class Fake` von `reahl.stubble:Impostor` erben zu lassen, wodurch jede Instanz von `class Fake` eine Instanz von `class Original` wäre, wie in Listing 16 zu erkennen ist. Ein weiteres experimentelles Feature ist das ersetzen eines bereits bestehenden Objekts (Delegation). Dies wurde bereits in den Hauptfeatures beschrieben, da es als sehr praktisch an zu sehen ist. Lediglich die Instanz Variablen eines Objektes machen hierbei Probleme, da diese nicht mit einem Mock ersetzt werden können. So werden auch Objekt variablen die von Originalen Methoden gesetzt oder verändert werden nicht im Mock gesetzt. Aus diesem Grund wird Delegation als Experimentelles Features gesehen, da ein Bug, der durch dieses Feature entstanden ist schwer zu finden ist.

Zusammenfassend ist stubble ein stabiles Tool, das dem Entwickler diverse Möglichkeiten an die Hand gibt Objekte vor ihrer Instanziierung als auch danach durch einen Mock zu ersetzen.

Im Bezug auf die Anwendbarkeit für TDD ist stubble demnach ein Tool das durchaus in Betracht gezogen werden kann, auch die Abhängigkeiten halten sich mit einer (six [23] ) in grenzen.

Die Effizienz mit der ein Entwickler stubble einsetzen kann ist hoch, denn abgesehen von einem decorator benötigt der Entwickler nichts weiter um Objekte zu ersetzen.

Lediglich die Komplexität von stubble ist nicht sehr hoch, zwar wird dem Entwickler alles geboten um ein Objekt zu ersetzen, jedoch aber auch nicht mehr. Zusätzlich bietet stubble mit seinen vorgefertigten Mocks eine gute schnelle Möglichkeit den Stdout zu ersetzen oder ein Objekt zu Monitoren. Um jedoch selbst ein Objekt zu ersetzen wird vergleichsweise viel Code benötigt, da nicht einfach nur der Return Wert neu gesetzt wird sondern die Methode neu geschrieben werden muss. Dies kann allerdings bei Komplexeren Anforderungen wiederum zu Vorteil werden weshalb sich das ganze wiederum ausgleicht.

Der Code für den Vergleichstest von stubble ist in Listing 17 zu finden. Mit stubble alleine lässt sich der in Listing 2 definierte Code nicht zu 100% optimal testen. So ist die nicht existierende Methode `Helper.help( self )` nicht einfach zu ersetzen, lediglich wenn man in der Mock Klasse die Methode `help( self )` schreibt und sie mit `@exempt` annotiert. Jedoch verliert man dadurch die Funktionalität von stubble, welche überprüft ob die Signatur der Funktionen übereinstimmen. Alternativ hätte man auch einfach eine Klasse nehmen können die ohne `@stubclass` auskommt. Ansonsten Zeigt der Code wie ein recht

[23] https://github.com/benjaminp/six

simples Tool viel erreichen kann, wenn auch mit viel Code abseits der Tests. Dennoch ging das überschreiben von os.path.abspath(path) sehr einfach durch reahl.stubble.replaced. Sehr einfach ging auch das abfangen des STDOUTs mithilfe der vorgefertigten Mocks.

#### 2.2.2.2 mocktest

mocktest ist nach Cuthbertson, 2018 ein mocking Tool, das von rspec[24] inspiriert wurde, dabei soll mocktest kein Port von rspec[24] sein, sondern eine kleine leichtere Version in Python. Da sich zum mocktest derzeit in der Version 0.7.3 (Stand 16. April 2019) befindet sind noch nicht alle Features vollkommen entwickelt und ausgereift, dennoch lässt sich das Tool bereits Produktiv einsetzen, da das sehr Nützliche Feature sohould_receive() von rspec[24] für mocktest implementiert wurde.

Das Hauptfeature von mocktest ist die Test Isolation, die verhindert das mocks Objekte außerhalb eines Test Falles verändern und so andere Tests beeinflussen. Demnach werden Tests immer innerhalb einer mocktest.MockTransaction ausgeführt, sofern diese ein oder mehrere Objekte mocken sollen.

Innerhalb dieses Kontextes stehen dem Entwickler verschiedene Möglichkeiten zur Verfügung Tests aus zu führen. So lässt sich Beispielsweise überprüfen ob eine Funkion oder Methode aufgerufen wurde, dabei spiel es keine Rolle ob es sich hierbei um ein Globales Objekt handelt oder ein Lokales. Des weiteren lassen sich auch Mocks nutzen, mit deren Hilfe Funktionen und Methoden präpariert werden können, wie auch beim überprüfen des Aufrufs spielt es hier keine Rolle ob es sich um ein Lokales oder Globales Objekt handelt. Die Präparation lässt sich nach bedarf auch anpassen, so kann der Entwickler beispielsweise festlegen, dass nur bei einem bestimmten Aufruf mit bestimmten Parametern das Mock Objekt aufgerufen wird. Andernfalls wird das Originale Objekt genutzt.

Mocktest nutzt unittest als Basis für seine Testumgebung, mocktest.TestCase erbt von unittest.TestCase, wodurch die gesamt Funktionalität von unittest gegeben wird. mocktest.TestCase führt dabei in seiner setUp() und tearDown() methode Code aus, der benötigt wird um mocktest Nutzen zu können, möchte man jedoch nicht unittest verwenden, so ist es möglich mocktest.MockTransaction als Kontextmanager mit with MockTransaction: zu nutzen. Zusätzlich wird mit mocktest unittest um eine assert Methode erweitert, assertRaises() verfügt über verbesserte Funktionalität zum überprüfen auf Exceptions.

Zum überprüfen von Mocks kann entweder when(obj) oder expect(obj) verwendet werden. Der unterschied hier ist lediglich, dass when(obj) nicht überprüft ob eine Methode aufgeru-

[24] http://rspec.info/

fen wurde oder nicht. Legt man hingegen Beispielsweise fest dass, expect(os).system dann wird ein Fehler geworfen, wenn os.system nicht aufgerufen wurde, was bei when(os).system nicht der Fall ist. Möchte man später überprüfen wie oft und mit was ein Objekt aufgerufen wurde, ist es möglich received_calls ab zu prüfen, dies ist eine Liste von allen getätigten aufrufen und Ihren Parametern. Hier ist eine Liste von allen weiteren Features von mocktest:

- Ergebnisvariation mit .and_return(erg1, erg2, ergX), alternativ ist .then_return() die gleiche Methode.
- Mit .at_least(n), .at_most(n)), .between(a, b) und .exact(n) lässt sich festlegen wie viele aufrufe erfolgt sein müssen.
  - Als alias wird .never() und .once() geboten für nie und einmal aufgerufen.
- Durch .then_call(func) wird die Funktion oder Methode func ausgeführt anstatt der eigentlichen.
- Properties lassen sich mit .with_children(**kwargs) setzen, wobei mock('mein_mock').with_children(x=1) den Wert x für mein_mock gleich 1 setzt.
- Das gleiche ist mit .with_method(**kwargs) möglich, nur für Methoden anstatt Properties.

Um TDD zu betreiben bietet mocktest alles was ein Entwickler zum mocken benötigt. So ist es möglich bestehende Objekte gänzlich oder Teilweise zu ersetzen, oder neue Objekte zu erstellen die, die Signatur eines anderen Objektes haben. Dabei kann selbstverständlich geprüft werden wie und wie oft etwas aufgerufen wurde. Auch Globale Objekte lassen sich für den Zeitraum eines Tests ersetzen, dadurch ist alles an Funktionalität geboten, was ein mock-testing Tool bieten muss.

Bezüglich der Effizienz ist mocktest äußerst gut, da quasi kein Vorwissen von Nöten ist um Objekte zu mocken. Die Vorarbeit die geleistet werden muss hält sich je nach Anwendungsfall sehr in grenzen oder ist fast nicht existent. Nutzt man die von mocktest zur Verfügung gestellte Klasse mocktest.TestCase für einen Testfall, so ist bereits alles fertig und direkt einsetzbar. Selbstverständlich kann mocktest.TestCase nach den Bedürfnissen um Funktionalität erweitert werden, jedoch sollte sich dies als sehr leicht gestalten. Entscheidet man sich gegen unittest, so ist mit dem Kontextmanager mocktest.MockTransaction schnell eine Mock Umgebung aufgesetzt und Nutzbar.

mocktest lässt den Entwickler Code schreiben der sich wie gesprochen liest. Durch die Methoden when(obj) und expect(obj) lässt sich ein Mock erstellen der mit Methoden angepasst werden kann, die wenn man sie aneinander reiht wie beschrieben lesen. Dies ist zum einen durch die verschiedenen Aliase möglich, wie zum Beispiel .once() welches .exact(1) ausführt. auch .then_return() ist sehr leicht zu lesen und vereinfacht das verstehen des Codes enorm. Abseits der eigentlichen Funktionalität die mit mocktest gesetzt wird fällt quasi kein Code an der geschrieben werden muss.

Um auch hier eine Analyse erstellen zu können, wurde der Code aus Listing 2 mit mocktest getestet. Dieser Code ist in Listing 18 zu finden. Der Code dazu ist recht einfach zu implementieren gewesen. Da mocktest keine setUp() Methode benötigt um zu funktionieren spart man sich hier bereits viel Code. Durch expect() und when() ist es möglich jedes beliebige Objekt zu nehmen und zu ersetzen.Dabei gab es keine Probleme hinsichtlich der implementierung der Mocks oder Mocks.

#### 2.2.2.3 flexmock

flexmock ist ein weiteres Tool, dass von der Ruby Community inspiriert wurde. Dabei diente das gleichnamige Tool flexmock [25] als Inspiration. „[...]. Es ist jedoch nicht das Ziel von Python flexmock, ein Klon der Ruby-Version zu sein. Stattdessen liegt der Schwerpunkt auf der vollständigen Unterstützung beim Testen von Python-Programmen und der möglichst unauffälligen Erstellung von gefälschten Objekten.“ [26] (Kabrda und Sheremetyev, 2019).

Die Features von flexmock sind denen von mocktest sehr ähnlich. So bildet flexmock seine Tests, mocks und mocks wie ausgesprochen dar, ein Feature aus der Ruby Welt, dass anscheinend sehr viel anklang findet bei Python Entwicklern.

flexmock bietet dem Nutzer einiges an Funktionalität auch wenn es sich erst in der Version 0.10.4 (Stand 22. April 2019) befindet. So lässt sich ein Mock für Klassen, Module und Objekte mithilfe von flexmock() einrichten. Damit lässt sich alles was ein Entwickler zum erfolgreichen durchlaufen seiner Tests brauch einstellen. Auch mocking ist kein Problem mit flexmock. Mit der gleichen Funktion, mit der auch Mocks erstellt werden lässt sich auch ein Mock abbilden. So kann ein Mock auch ein Mock sein und umgekehrt. Dadurch ist es dem Entwickler möglich zu überprüfen wie oft etwas aufgerufen wurde, welche Parameter verwendet wurden, welcher Rückgabe wert zu erwarten ist und/oder welche Exception zu erwarten ist. Zusätzlich ist es möglich originale Funktionen eines

[25] https://github.com/jimweirich/flexmock
[26] Übersetzt aus dem Englischen

Objektes zu mocken, dadurch enthält der Entwickler die Möglichkeit zu überprüfen ob eine unveränderte Funktion oder Methode bestimmten Anforderungen Entspricht, dabei ist das Interface das gleiche wie bei einem Mock. flexmock bietet des weiteren den gefälschten Objekten neue Methoden hinzu zu fügen, wodurch nicht existierende Funktionen abgebildet werden können oder Objekte in ihrem Funktionsumfang temporär erweitert werden können.

Ein besonders interessantes Feature ist, dass überprüfen des Ergebnisses nach Typ. Dabei wird .and_return() nur der Typ mitgegeben, der erwartet wird. So würde .and_return((int, str , None) jedes Tuple zulassen, dass als ersten Wert einen Int hat, als zweiten einen String und als drittes None, dabei kann selbstverständlich auf jede beliebige Instanz überprüft werden, auch auf eigene Klassen. Dies ist allerdings nur möglich bei .should_call() nicht bei .should_receive().

Das letzte in der von Kabrda und Sheremetyev, 2019 erwähnte Feature ist das ersetzen von Klassen noch vor ihrer Instanziierung. Dabei gibt es mehrere Ansätze die allerdings auch verschiedene Ausgänge haben, der erste ist auf Modul Level, dabei wird im Modul die Klasse mit einem Objekt ersetzt, dass entweder ein flexmock sein kann oder ein Beliebig anderes. Der Nachteil dieser Methode ist, dass es eventuell zu Problemen führen kann, dass die Klasse durch eine Funktion ersetzt wurde. Die alternative ist, .new_instance(obj) zu verwenden, welches beim erstellen das Objekt zurück gibt, welches .new_instnce() übergeben wurde oder man ersetzt die __new__() Methode der Klasse mit .should_receive('__new__').and_return(obj) was im Endeffekt das gleiche ist, nur ausgeschrieben.

Die Anwendbarkeit von flexmock ist sehr gut, da es zunächst keine weiteren Abhängigkeiten installiert außer sich selbst und mit allen Test-runnern kompatibel ist. Im Bezug auf TDD lässt flexmock nichts zu wünschen übrig, dem Entwickler werden allerhand Funktionalität geboten Mocks oder Mocks zu erstellen und zu verwenden. Dabei lässt sich alles überprüfen von der Aufruf Anzahl bis zu den Parametern die verwendet wurden.

Im Aspekt Effizienz ist flexmock ein Parade-Beispiel, einmal flexmock() aufgerufen kann bereits los gelegt werden, dabei muss der Entwickler selbst wenig von mocking verstehen um die Funktionalität nutzen zu können, da sich der Code wie gesprochen ließt (Sofern der Entwickler englisch sprechen kann). Das gleiche gilt für die Mocks und fake Objekte.

Das gleiche gilt für die Komplexität von flexmock, an Funktionalität bietet flexmock alles was man sich als Entwickler wünschen kann und bietet dabei auch noch ein Interface, dass es ermöglicht übersichtlichen Code zu schreiben. Dies liegt vor allem an der deskriptiven Programmierung die von flexmock geboten wird. Unübersichtlich wird dadurch der Code

nicht mehr als er im laufe der Zeit sowieso werden würde, eher noch hilft flexmock dabei die Übersicht länger zu wahren.

Der Code zu Listing 2 wurde mithilfe des Codes aus Listing 19 getestet, dabei ergaben sich die eben aufgelisteten Aspekte von flexmock. flexmock überzeugt dabei mit der Einfachheit mit der es aufgesetzt werden muss. In der setUp() wird global im Modul my_package.my_mock_module die Klasse NotMocked ersetzt. Dadurch ist ex möglich in jedem Test Fall eine Annahme auf zu stellen, die dann auf alle Objekte der Klasse NotMocked zutreffen. Der aufwand diesen Code zu schreiben war demnach sehr gering. **EXAMPLE funktioniert noch nicht, siehe github**

#### 2.2.2.4 python-doublex

Mit python−doublex wird dem Entwickler ein Tool an die Hand gelegt, dessen Funktionalität sich voll und ganz um doubles dreht. Ein double ist je nach Anwendung ein Mock, ein Mock oder ein Spy(Spion), der eine Klasse oder ein Objekt imitiert.

python−doublex bietet drei Verschiedene Interfaces (vier zählt man die Unterklasse von Spy dazu), Stub, Spy (und ProxySpy) und Mock. Da die Dokumentation hier sehr schöne und treffende Beschreibungen nimmt, werden diese nun hier zitiert.

- „Stubs sagen dir was du hören willst.“[27] [28]
- „Spies erinnern sich an alles was Ihnen passiert.“[27] [28]
- „Proxy spies leiten aufrufe an Ihre originale Instanz weiter.“[27] [28]
- „Mock erzwingt das vordefinierte Skript.“[27] [28]

Mit diesen Beschreibungen der einzelnen Interfaces kann ein Entwickler bereits entscheiden welches von Ihnen für Ihn in Frage kommt, ohne den Code dazu kennen zu müssen. Um dennoch einen erweiterten Einblick zu schaffen werden die einzelnen Funktionen hier noch ein mal genauer beschrieben. Wichtig ist hierbei zu beachten, dass python−doublex lediglich doubles, also Duplikate anbietet, die das gleiche Interface wie eine Klasse habe, jedoch keine Instanz dieser Klasse sind. Dadurch bleiben manche Möglichkeiten dem Entwickler verwehrt, wie zum Beispiel das Nutzen einer implementierten Methode aus der original Klasse. Die Duplikate sollen als Ersatz gelten für nicht implementierte Methoden, deren Funktionalität und deswegen der Rückgabe Wert bekannt ist.

[27] Alises, 2014
[28] Übersetzt aus dem Englischen

Das Stub Interface ist, wie der Name bereits vermuten lässt, ein Mock mit dem es dem Entwickler möglich ist Rückgabe Wert von Funktionen zu setzen oder ähnliche Aktionen fest zu definieren. Dazu ist es dem Entwickler möglich zwischen verschiedenen Parametern zu unterscheiden oder gar auf alle aufrufe zu prüfen. Das Interface bietet auch die Möglichkeit Parameter von einem Objekt zu ändern um den ausgang von Methoden zu verändern. Das Interface unterscheiden zwischen einem Stub und einem „free“ Stub. Ersterer nimmt eine Klasse als Parameter und ersetzt diese, zweiterer nimmt keine Parameter und fungiert als generelles Objekt. Dabei ist der wichtigste unterschied, dass der normale Stub nur Methoden ab ändern kann die, die originale Klasse auch besitzt, der „free“ Stub ist in dieser Hinsicht ungebunden und kann alles sein was der Entwickler Ihm zuweist, also ist er frei.

Als zweites Interface wird von Alises, 2014 in der Dokumentation das Spy Interface erwähnt. Dieses bietet dem Nutzer die Möglichkeit Klassen zu überwachen, dabei legt der Entwickler fest welche Methode wie oft und mit welchen Parametern aufgerufen werden muss. Werden die Erwartungen nicht getroffen wird eine Exception geworfen. Das überprüfen der Parameter ist dabei möglich mithilfe des exakten wertes, einer Regex oder mit dem von python−doublex definierten Schlüsselwort ANY_ARG, welches, wie der Name sagt, jedes Argument validiert. Zum Spy gibt es auch wieder einen „free“ Spy, welcher von keiner Klasse abhängig ist. Dadurch ist es möglich jede beliebige Methode auf diesem aus zu führen ohne dass dies zu einer Exception führen würde, danach kann dann überprüft werden mit welchen Parametern und wie oft der Spy aufgerufen wurde.

Zusätzlich zum Spy gibt es noch einen ProxySpy, welcher das zu überwachende Objekt aufruft und nicht ersetzt. Aus diesem Grund erhält der ProxySpy ein Objekt als Parameter und kein Klasseninterface. Sämtliche Methoden werden auf das originale Objekt ausgeführt wodurch keine Veränderung an diesem Vorgenommen werden kann, in Form von einem Mock.

Das letzte Interface, ist das Mock Interface. Mit diesem lässt sich vor dem Test festlegen welche Methode wann und wie aufgerufen werden muss, damit der Test erfolgreich wird. Die Reihenfolge spielt dabei eine Rolle, außer man nutzt any_order_verify(). Das Mock Objekt lässt sich dabei wie jedes andere Objekt verändern und wie ein Stub präparieren. Auch der „free“ Mock ist wie bei den anderen Interfaces verfügbar und bietet die gleichen Möglichkeiten der freien Gestaltung des Objekts.

Zusammenfassen lässt sich sagen, das jedes Interface die Möglichkeit bietet eine Klasse zu verändern ausgenommen, der ProxySpy der nur auf einer Instanz arbeiten kann. Zu jedem Interface ist ein freies Interface verfügbar, welches mit beliebigen Methoden

aufgerufen werden kann ohne Fehler zu werfen. Zum Abschluss lässt sich auch noch sagen, dass jedes Interface, die Möglichkeit bietet stubbing zu betreiben, wobei nur das Stub Interface sonst keine weitere Funktionalität bietet.

python−doublex bietet dem Entwickler viel, aber nicht alles für effizientes TDD. Wie der Name des Tools bereits vermuten lässt handelt es sich um Duplikate von Objekten, deren Funktionalität aber vom Entwickler festgelegt werden muss. So würde eine Klasse mit einer Methode return_input(input) die den übergebenen Parameter zurück gibt standardmäßig None zurück geben, solange nichts anderes im Duplikat festgelegt wurde. Lediglich die Signartur des Aufrufs wird überprüft, so würde stub.reuturn_input() eine Exception werfen. Zwar lässt sich das standardverhalten von python−doublex verändern, jedoch kann man hier nur einen festen Wert setzen der für alle nicht ersetzten Methoden gilt. Auch wenn man Globale Objekte oder Module ersetzen möchte ist dies nicht möglich. Hinzu kommt das, dass Tool drei Abhängigkeiten benötigt und selbst nicht in stabilem Zustand ist. Währen der Analyse sind zwei DeprecationWarnings aufgetaucht, wovon eine bereits mit Python 3.0 ausgelaufen ist. Dies weißt auf eine nicht sehr aktive Entwicklung des Tools hin. Selbstverständlich wurde die Fehler auf GitHub gemeldet, sodass man sich darum kümmern kann. (Stand 23. April 2019)

Auch die Effizienz lässt etwas zu wünschen übrig. Dadurch, dass das Duplikat nicht die originalen Methoden aufruft, sofern verfügbar, muss jede Methode mit einem Return Wert überschrieben werden. Dies mag für manche Tests vollkommen ausreichen, macht aber die Entwicklung mit TDD sehr schwer, da Tests von Zeit zu Zeit ausgeführt werden müssen und dort die Teil Implementierung selbst verständlich genutzt werden soll. Hat man allerdings eine Externe Klasse auf die kein Einfluss ist kann dieses Tool durchaus nützlich werden.

An Komplexität fehlt es python−doublex etwas, so kann der Mock lediglich festlegen welche Funktion ausgeführt, welcher Rückgabewert zurück gegeben oder welche Exception geworfen wird. Das Spy Interface hingegen ist leicht zu nutzen und bietet fast alles was ein Entwickler brauchen kann um zu überprüfen ob und wie etwas aufgerufen wurde. Jedoch scheitert es hier auch an der Implementieren von echten Objekten, so fungiert der ProxySpy zwar als Spy auf einem Objekt, jedoch kann er nur verzeichnen welche Methode auf Ihm aufgerufen wurde. So würde spy.call_other() nicht obj.other() registrieren, wodurch es nicht möglich ist zu überprüfen ob other() nun aufgerufen wurde oder nicht. Das gleiche gilt mit Mock, lediglich Methoden die auf dem Mock-Objekt aufgerufen werden werden registriert. Aus diesem Grund ist es auch hier nicht optimal nutzbar für TDD.

Auch hier wurde der Code aus Listing 2 mit python−doublex getestet. Der Code dazu befindet sich in Listing 20. Wie dort zu erkennen ist, konnte nicht jeder Test ohne Probleme validiert werden. So war es zwar möglich die ersten drei Tests erfolgreich zu validieren, jedoch fragt man sich dort ob es wirklich etwas bringt ein Objekt zu testen welches einfach das zurück gibt was man erwartet. Beim Test, der eine interne Methode aufrufen soll kommt das Spy Interface an seine grenzen, da es nicht überprüfen kann ob die interne Methode aufgerufen wurde. Der Error wurde in einem Kommentar in der Teste Methode festgehalten. Das Testen ob Helper.help() aufgerufen wurde, war teilweise erfolgreich, da es dank des Duplikates ein Objekt mit dem Richtigen Interface bekommen hat, dennoch ist es leider mit python−doublex nicht möglich vor zu täuschen eine andere Klasse zu sein. Auch hier wurde der Fehler in einem Kommentar festgehalten. Zuletzt sollte getestet werden ob das os Modul ausgetauscht werden kann, jedoch ist dies mit python−doublex leider nicht möglich. Der Fehler dazu wurde in einem Kommentar innerhalb der Funktion festgehalten, sowie eine mögliche, wenn auch umständliche Lösung. Die würde das os Objekt duplizieren und der Methode als Parameter übergeben, was im Praktischen Einsatz aber nicht gemacht werden sollte.

### 2.2.3 Fuzz-testing Tools

Zusätzlich zu den hier behandelten Test runnern und den mocking Tools werden zusätzlich noch Fuzz-testing Tools behandelt. Unter Fuzz-testing versteht man das Testen eines Programms oder einem Modul mit zufälligen Werten. Diese Werte können komplett zufällig sein, aber auch einem gewissen Format entsprechen oder einem Typ.

Fuzz-testing funktioniert dabei anders als die gewohnten Methoden des Testens. Der Normale Flow ist, dass Test Daten präpariert werden, dann werden Tests mit diesen Daten aus geführt und danach validiert, dass das Ergebnis richtig ist. Mit Fuzz-testing sieht das ganze etwas anders aus, zuerst wird festgelegt, welchem Schema die Daten entsprechen müssen (Beispiel: Es wird eine IP erwartet, [1−9]{1,3}.[1−9]{1,3}.[1−9]{1,3}.[1−9]{1,3}), danach wird mit den Daten der Test ausgeführt und danach validiert (MacIver, 2019).

Der unterschied ist nun, das der Entwickler selbst sich nicht Daten ausdenken muss. Im genannten Beispiel müsste der Entwickler sich verschiedene IP Adressen ausdenken, die entweder richtig oder Falsch sind und danach diese überprüfen. Fuzz-testing übernimmt das ausdenken der Daten und sorgt dafür, dass diese der Spezifikation entsprechen. So kann eine viel größere Anzahl an Tests mit erheblich weniger aufwand auf einem Modul oder Programm ausgeführt werden.

Das ist aber noch nicht alles, schlägt ein Test fehl, mit Daten die per Spezifikation richtig sind, so wird der Wert der Daten für später gespeichert und sofern möglich wird versucht mit ähnlichen Daten dieser Fehler zu erreichen. Dadurch hat ein Entwickler am Ende der Tests Daten die fehlgeschlagen sind. Mit diesen kann er dann die Weiteren Tests befüllen um so zu verhindern, dass diese Daten zu Fehlern führen. Im Normalfall übernimmt dies allerdings das Tool, wodurch der Entwickler sich nur darum kümmern muss, dass die Tests erfolgreich verlaufen.

Von Zeit zu Zeit entsteht dadurch ein Katalog an Daten für verschiedene Tests, der vorgibt bei welchen Daten ein Test fehl geschlagen ist. Dadurch wird verhindert, dass beim ändern des Codes dieser Fehler wieder auftritt. Führt man dies über den gesamten Entwicklungsprozess hinweg durch, so erhält man am Ende ein sehr ausgiebig getestetes Programm.

Diese Methode des Testens ist auch „property based testing“, also Eigenschafts- basierte Tests genannt. In dieser Arbeit, wird allerdings weiterhin der Englische Begriff Fuzz-testing genutzt.

#### 2.2.3.1 hypothesis

Das größte und am weitesten verbreitete Tool für Fuzz-testing mit Python ist hypothesis. Recherchen im Internet weisen auf kein weiteres aktuelles Tool hin, dass Open Source ist, demnach ist hypothesis das einzige Tool welches zum Fuzz-testing mit Python derzeit genutzt werden kann (Python 3.7).

Bevor man anfangen kann mit hypothesis zu testen, muss man verstehen wie hypothesis funktioniert. Möchte man Daten haben die einer Spezifikation entsprechen, benötigt man hypothesis. strategies . In hypothesis sind Strategien Spezifikationen für die Generierung von Daten. Wäre eine Spezifikation also, dass die Daten Integer sind, dann würde man hypothesis. strategies . integers () verwenden. Diese Strategie würde verschiedene Daten die Integer sind ausgeben. Möchte man nur Werte zwischen A und B, so wäre die Strategie . integers (min_value=A, max_value=B). Dies kann man mit beliebig vielen Daten Typen und Spezifikationen durchführen, dabei verfügt hypothesis über so viele verschiedene Spezifikationen, dass es schwer sein wird alle zu kennen. Die Basics lassen sich allerdings in der Dokumentation finden [29] .

Ist die Spezifikation bekannt und die Strategie gefunden werden Tests mithilfe einer Annotation der Test Funktion übergeben, diese Annotation nennt sich hypothesis.given().

[29] https://hypothesis.readthedocs.io/en/latest/data.html

Ein kleines Beispiel dazu kann in Listing 21 gefunden werden. .given() unterstützt dabei die *args als auch die **kwargs Übergabe der Strategien.

Hat man nun einen Fehler gefunden und möchte diesen für alle weiteren Tests testen, so kann man mit hypothesis.example() den Test annotieren und die Werte bei denen der Test fehl geschlagen ist dort übergeben. Möchte der Entwickler etwas mehr Information darüber weshalb ein Test fehl geschlagen ist, so ist es Ihm möglich mit hypothesis.note() vor einer Assertion einen Text aus zu geben, der Ihm diese Informationen gibt. note() wird allerdings nur aus gegeben, wenn der Test fehl schlägt.

Möchte man ein paar Statistiken darüber was hypothesis gemacht hat und man nutzt pytest als Test runner, so kann man mit −−hypothesis−show−statistics sich die Statistiken zu den Tests anschauen. Dabei werden folgende Informationen zu jeder Test Funktion gezeigt:

- Gesamte Anzahl der Durchläufe mit der Anzahl der fehlgeschlagenen und der Anzahl der invaliden Tests.
- Die ungefähre Laufzeit des Tests.
- Wie viel Prozent davon für die Datengenerierung aufgewendet wurde.
- Wieso der Test beendet wurde.
- Alle aufgetretenen Events von hypothesis

Ein Event kann allerdings auch vom Entwickler genau so wie note() genutzt werden, mit dem Unterschied, dass ein Event immer zu einem Output führt wenn die Zeile im Code ausgeführt wird (Sofern pytest und −−hypothesis−show−statistics genutzt werden).

Mit diesem Wissen lässt sich hypothesis bereits für die einfachsten Dinge Nutzen, hat man allerdings etwas Spezifischere Anforderungen, so muss etwas tiefer in die Materie eingestiegen werden. Angenommen, eine Funktion soll mit einem Integer getestet werden, dieser muss aber durch sich selbst teilbar sein. Dies könnte mit einem if Statement geprüft werden, würde aber die Anzahl der relevanten Tests erheblich senken. Standessen ist es möglich auf Strategien filter an zu wenden. Filter müssen dabei Funktionen sein, die einen Wert filtern und Ihn zurück geben. Für das eben genannte Beispiel würde die Annotation dann so aussehen: @given(integers(). filter (lambda i: i % i == 0)).

Gibt hypothesis einem Test Daten die nicht gefiltert werden können oder sollen, so ist es möglich mit hypothesis.assume() eine Annahme auf zu stellen. Ist die Annahme falsch, so wird dieser Test übersprungen. Allerdings kann dies zu Fällen führen bei denen hypothesis

keine validen Daten finden kann, was zu einer Exception und zum fehlschlagen des Tests führt.

hypothesis verfügt noch über weit mehr Funktionalität als hier genannt werden kann, wie zum Beispiel das verketten von Strategien. Die Dokumentation zu allem, was hypothesis ohne Erweiterungen erschaffen kann, kann online [30] eingesehen werden.

# 3 Zusammenfassung

Hier werden in kurzen Punkten die Vor- und Nachteile der einzelnen Tools zusammengefasst, um dem Leser einen schnellen Überblick zur Verfügung zu stellen.

# 4 Vergleich der Tools

Die in Kapitel 3 aufgeführten Vor- und Nachteile werden hier untereinander verglichen. Wodurch eine Ausführung entstehen soll welches Tool am besten für Welchen Zweck geeignet ist.

# 5 Kombinierung von Tools

Sofern dies möglich ist werden hier mögliche Kombinationen von Tools diskutiert.

# 6 Diskussion: Test-driven development in der Praxis

Hier wird die Einleitung der Diskussion stehen.

## 6.1 Stärken von Test-driven development

Hier soll eine Diskussion der Schwächen von TDD entstehen.

## 6.2 Schwächen von Test-driven development

Dieses Kapitel soll die Schwächen von TDD Diskutieren, dabei wird in Vermeidbare Schwächen beschrieben welche dieser Schwächen umgangen werden können und in Unumgängliche Schwächen werden jene Schwächen die nicht vermeidbar sind analysiert.

[30] **https://hypothesis.readthedocs.io/en/latest/data.html**.

### 6.2.1 Vermeidbare Schwächen

Vermeidbare Schwächen

### 6.2.2 Unumgängliche Schwächen

Unumgängliche Schwächen

## 6.3 Wirtschaftlichen Aspekte von Test-driven development

Dieses Kapitel befasst sich mit den möglichen Wirtschaftlichen folgen, die die Anwendung von TDD mit sich bringt.

## 6.4 Zusammenfassung

Hier wird die gesamte Diskussion zusammengefasst um danach im Fazit eine Fundierte Meinung wieder zu geben.

# 7 Fazit

FAZIT

# 8 Nachwort

Nachwort

# Literaturverzeichnis

Alises, D. V. (2014). doublex - doublex 1.8.1 documentation. Website. Online erhältlich unter https://python-doublex.readthedocs.io/en/latest/; abgerufen am 23. April 2019.

Brandl, G., van Rossum, G., Heimes, C., Peterson, B., Melotti, E., Murray, R., ... Miss Islington (bot). (2007). *doctest — Test interactive Python examples*. Website. Online erhältlich unter https://github.com/python/cpython/blob/3.7/Doc/library/doctest.rst oder https://docs.python.org/3/library/doctest.html; abgerufen am 25. März 2019.

Brandl, G., van Rossum, G., Heimes, C., Peterson, B., Pitrou, A., Dickinson, M., ... Palard, J. (2007). *unittest — Unit testing framework*. Website. Online erhältlich unter https://github.com/python/cpython/blob/3.7/Doc/library/unittest.rst oder https://docs.python.org/3/library/unittest.html; abgerufen am 25. März 2019.

Cuthbertson, T. (2018). About mocktest - mocktest 0.7.2 documentation. Website. Online erhältlich unter http://gfxmonk.net/dist/doc/mocktest/doc/; abgerufen am 16. April 2019.

Foord, M., Melotti, E., Coghlan, N., Storchaka, S., Peterson, B., Huang, H. & Wijaya, M. (2012). PEP 417 – Including mock in the Standard Library. Website. Online erhältlich unter https://github.com/python/peps/blob/master/pep-0417.txt oder https://www.python.org/dev/peps/pep-0417/; abgerufen am 26. Januar 2019.

Kabrda, S. & Sheremetyev, H. (2019). flexmock - Testing Library – flexmock 0.10.3 documentation. Website. Online erhältlich unter https://flexmock.readthedocs.io/en/0.10.3/; abgerufen am 22. April 2019.

Krekel, H. & pytest-dev team. (2019). pytest Documentation - Release 4.4. PDF. Online erhältlich unter https://media.readthedocs.org/pdf/pytest/4.4.0/pytest.pdf; abgerufen am 2. April 2019.

MacIver, D. R. (2019). Welcome to Hypothesis! — Hypothesis 4.18.0 documentation. Website. Online erhältlich unter https://hypothesis.readthedocs.io/en/latest/; abgerufen am 29. April 2019.

Muthukadan, B., jtatum, CPE0014bf07ffd2-CM001ac30d4aca, 71-35-143-156, gjb1002, little-black-box, ... Barnes, S. (2011). PythonTestingToolsTaxonomy. Website. Online erhältlich unter https://wiki.python.org/moin/PythonTestingToolsTaxonomy; abgerufen am 14. März 2019.

Schobelt, F. (2017). Weltweite Smartphone-Verbreitung steigt 2018 auf 66 Prozent. Online erhältlich unter https://www.wuv.de/digital/weltweite_smartphone_verbreitung_steigt_2018_auf_66_prozent; abgerufen am 25. Januar 2019.

Vosloo, I., Sparks, C. & Nagel, P. (2018). Stubble – A collection of tools for writing stubs in unit tests (reahl.stubble). Website. Online erhältlich unter https://www.reahl.org/docs/4.0/devtools/stubble.d.html; abgerufen am 15. April 2019.

# Fußnotenverzeichnis

# Listings

```python
"""my_module.py"""
from sqlalchemy import (
    Column,
    Integer,
    String,
    create_engine,
)
from sqlalchemy.orm import sessionmaker
from sqlalchemy.ext.declarative import declarative_base

engine = create_engine('sqlite:///:memory:')
Session = sessionmaker(bind=engine)
session = Session()
Base = declarative_base()

do_something_which_does_not_exist = None


def my_pow(a, b):
    result = a
    for _ in range(1, b):
        result = result * a

    return result


class Item(Base):
    __tablename__ = 'items'

    id_ = Column('id', Integer, primary_key=True)
    name = Column(String(64))
    storage_location = Column(Integer)
    amount = Column(Integer)

    def do_something(self):
        return do_something_which_does_not_exist(self)

    def __repr__(self):
        return f'item<{self.id_}, {self.name}, {self.storage_location}, {self.amount}>'


Base.metadata.create_all(engine)
```

Listing 1: Basis Modul zum testen

```python
import os


class Helper:
    """Diese Klasse simuliert eine Klasse die noch nicht geschrieben wurde"""
    pass
```

```
7
8
9  class NotMocked:
10     """Originale Klasse.
11     Die Methodennamen geben an was sie eigentlich machen sollten
12     """
13     def print_foo(self):
14         print("I should print foo, instead I print this!")
15
16     def return_42(self):
17         return 21
18
19     def raise_error(self):
20         return
21
22     def _internal_function(self):
23         return
24
25     def call_internal_function(self):
26         self._internal_function()
27
28     def call_internal_function_n_times(self, n):
29         for _ in range(0, n):
30             self._internal_function()
31
32     def call_helper_help(self, h: Helper):
33         assert isinstance(h, Helper)
34         assert h.help() is True
35
36     def return_false_filepath(self):
37         """Diese Methode soll /foo/bar/baz.py zurueckgeben"""
38         return os.path.abspath(__file__)
```

Listing 2: Basis Modul zum testen von mocks

```
1 pass
```

Listing 3: Basis Modul zum testen von fuzz-testing

```
1 import unittest
2
3 from my_package.my_module import my_pow
4
5
6 class TestMyModule(unittest.TestCase):
7     def test_my_pow(self):
8         self.assertEqual(my_pow(2, 2), 4)
9         self.assertEqual(my_pow(4, 8), pow(4, 8))
10
11
12 if __name__ == '__main__':
13     unittest.main()
```

Listing 4: unittest einfaches Beispiel

```
1 .
2 ----------------------------------------------------------------------
3 Ran 1 test in 0.000s
4
5 OK
```

Listing 5: unittest einfaches Beispiel: Output erfolgreich

```
1 F
2 ======================================================================
3 FAIL: test_my_pow (unittest_.example.TestMyModule)
4 ----------------------------------------------------------------------
5 Traceback (most recent call last):
6   File "unittest_/example.py", line 9, in test_my_pow
7     self.assertEqual(my_pow(4, 7), pow(4, 8))
8 AssertionError: 16384 != 65536
9
10 ----------------------------------------------------------------------
11 Ran 1 test in 0.000s
12
13 FAILED (failures=1)
```

Listing 6: unittest einfaches Beispiel: Output misslungen

```
import unittest
import unittest.mock as mock

from my_package.my_module import (
    Item,
    session
)

class TestMyDatabase(unittest.TestCase):
    def setUp(self):
        self.item = Item(id_=1, name='name', storage_location=7, amount=3)

    def test_creation(self):
        session.add(self.item)
        session.commit()

        self.assertIs(len(session.new), 0)
        self.assertEqual(session.query(Item).first(), self.item)

    @mock.patch('my_package.my_module.do_something_which_does_not_exist')
    def test_external_function(self, mock_do_something_which_does_not_exist):
        mock_do_something_which_does_not_exist.return_value = 42


        self.assertIs(self.item.do_something(), 42)
        mock_do_something_which_does_not_exist.assert_called_with(self.item)

    def tearDown(self):
        session.close()

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

Listing 7: unittest my_module

```
import random

def return_3(s=None, i=None):
    """
    >>> return_3(s=True)
    '3'

    >>> return_3(i=True)
    3

    >>> return_3(i=True)
    '4'
    """
    if s:
        return '3'
    elif i:
        return 3


if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

Listing 8: doctest unterscheidet Typen

```
**********************************************************************
File "example.py", line 14, in __main__.return_3
Failed example:
    return_3(i=True)
Expected:
    '4'
Got:
    3
**********************************************************************
1 items had failures:
   1 of   3 in __main__.return_3
***Test Failed*** 1 failures.
```

Listing 9: doctest unterscheidet Typen: Output

```
Trying:
    from advanced import *
Expecting nothing
ok
Trying:
    my_pow(2, 2)
Expecting:
    4
ok
```

Listing 10: doctest verbose Output

```
def my_pow(a, b):
    """
    >>> my_pow(2, 2)
    4
    >>> my_pow(4, 8)
    65536
    """
    result = a
    for _ in range(1, b):
        result = result * a

    return result


class Item(Base):
    """
    Da Doctest keine Fixtures unterstuetzt muss hier der setUp Code stehen:
    >>> Base.metadata.create_all(engine)
    >>> item = Item(id_=1, name='name', storage_location=1, amount=1)
    >>> session.add(item)
    >>> session.commit()


    Check ob das item commited wurde
    >>> len(session.new)
    0

    Check ob das item in der Datenbank ist.
    Das item wird mit seiner __repr__() Methode representiert, da die
    Werte bekannt sind kann ueberprueft werden ob diese uebereinstimmen.
    >>> session.query(Item).first()
    item<1, name, 1, 1>

    do_something existiert nicht, da mit Doctest kein Mock erstellt werden
    kann wird auf die Exception ueberprueft.
    >>> item.do_something()
    Traceback (most recent call last):
        ...
    TypeError: 'NoneType' object is not callable

    Hier wird der tearDown Code ausgefuehrt
    >>> session.close()
    """
    __tablename__ = 'items'

    id_ = Column('id', Integer, primary_key=True)
    name = Column(String(64))
    storage_location = Column(Integer)
    amount = Column(Integer)

    def do_something(self):
        return do_something_which_does_not_exist(self)
```

```
53
54      def __repr__(self):
55          return f'item<{self.id_}, {self.name}, {self.storage_location}, {self.amount}>'
56
57  if __name__ == '__main__':
58      import doctest
59      # Fuehrt die internen Tests aus
60      doctest.testmod()
61      # Fuehrt die externen Tests aus
62      doctest.testfile('advanced.txt')
```

Listing 11: doctest: my_module

```
1  In dieser Datei stehen alle Docstrings ausgelagert
2
3  Um my_module zu testen muss alles importiert werden.
4  >>> from my_module import *
5  >>> my_pow(2, 2)
6  4
7  >>> my_pow(4, 8)
8  65536
9
10
11 Da Doctest keine Fixtures unterstuetzt muss hier der setUp Code stehen:
12 >>> Base.metadata.create_all(engine)
13 >>> item = Item(id_=1, name='name', storage_location=1, amount=1)
14 >>> session.add(item)
15 >>> session.commit()
16
17
18 Check ob das item commited wurde
19 >>> len(session.new)
20 0
21
22 Check ob das item in der Datenbank ist.
23 Das item wird mit seiner __repr__() Methode representiert, da die
24 Werte bekannt sind kann ueberprueft werden ob diese uebereinstimmen.
25 >>> session.query(Item).first()
26 item<1, name, 1, 1>
27
28 do_something existiert nicht, da mit Doctest kein Mock erstellt werden kann
29 wird auf die Exceptioni ueberprueft.
30 >>> item.do_something()
31 Traceback (most recent call last):
32     ...
33 TypeError: 'NoneType' object is not callable
34
35 Hier wird der tearDown Code ausgefuehrt
36 >>> session.close()
```

Listing 12: doctest my_module: Textdatei

```
atomicwrites==1.3.0
attrs==19.1.0
more-itertools==7.0.0
pluggy==0.9.0
py==1.8.0
pytest==4.4.0
six==1.12.0
```

Listing 13: pytest Abhängigkeiten

```
import pytest

from my_package.my_module import (
    Item,
    session,
)

class TestMyDatabase:
    @pytest.fixture
    def setUp_tearDown(self):
        # setUp
        self.item = Item(id_=1, name='name', storage_location=7, amount=3)
        yield self.item
        # tearDown
        session.close()

    def test_creation(self, setUp_tearDown):
        session.add(self.item)
        session.commit()

        assert len(session.new) is 0, 'Lenght is not 0, session is dirty'
        assert session.query(Item).first() is self.item, 'Item is not in databse'

    def test_external_function(self, setUp_tearDown, monkeypatch):
        def mock_return(something):
            return 42

        monkeypatch.setattr(Item, 'do_something', mock_return)
        assert self.item.do_something() is 42, 'Do something wasn\'t patched'


if __name__ == '__main__':
    pytest.main()
```

Listing 14: pytest my_module

```
============================= test session starts ==============================
platform linux -- Python 3.7.3, pytest-4.4.0, py-1.8.0, pluggy-0.9.0
rootdir: /this/is/my/secret/path
collected 2 items

pytest_my_module.py ..                                                   [100%]

```

```
8 ============================== 2 passed in 0.11 seconds ==============================
```

Listing 15: pytest my_module: Output

```
1 from reahl.stubble import Impostor, stubclass
2
3
4 class Original:
5     pass
6
7
8 @stubclass(Original)
9 class Fake(Impostor):
10     pass
11
12
13 fake = Fake()
14 assert isinstance(fake, Original)  # True
```

Listing 16: stubble: Impostor

```
1 import unittest
2 from reahl.stubble import (
3         exempt,
4         replaced,
5         stubclass,
6 )
7 from reahl.stubble.intercept import (
8     CallMonitor,
9     SystemOutStub,
10 )
11
12 from my_package.my_mock_module import (
13     NotMocked,
14     Helper,
15 )
16
17 @stubclass(NotMocked)
18 class Mocked(NotMocked):
19     def print_foo(self):
20         print("foo")
21
22     def return_42(self):
23         return 42
24
25     def raise_error(self):
26         raise BaseException("success")
27
28
29 @stubclass(Helper)
30 class MockedHelper(Helper):
31     @exempt
32     def help(self):
```

```
33          """Loest das Problem zwar, aber wenn die Funktion existiert wird nicht
34          die signatur ueberprueft.
35          """
36          return True
37
38
39  class TestStubble(unittest.TestCase):
40      def setUp(self):
41          self.mock = Mocked()
42
43      def test_print_foo(self):
44          with SystemOutStub() as monitor:
45              self.mock.print_foo()
46
47          assert monitor.captured_output == 'foo\n'
48
49      def test_return_42(self):
50          self.assertEqual(self.mock.return_42(), 42)
51
52      def test_raise_error(self):
53          self.assertRaises(BaseException, self.mock.raise_error)
54
55      def test_call_internal_function(self):
56          with CallMonitor(self.mock._internal_function) as monitor:
57              self.mock.call_internal_function()
58
59          self.assertIs(monitor.times_called, 1)
60          self.assertEqual(monitor.calls[0].args, ())
61          self.assertEqual(monitor.calls[0].kwargs, {})
62          self.assertIs(monitor.calls[0].return_value, None)
63
64      def test_call_internal_function_n_times(self):
65          with CallMonitor(self.mock._internal_function) as monitor:
66              self.mock.call_internal_function_n_times(4)
67
68          self.assertIs(monitor.times_called, 4)
69          self.assertEqual(monitor.calls[0].args, ())
70          self.assertEqual(monitor.calls[0].kwargs, {})
71          self.assertIs(monitor.calls[0].return_value, None)
72
73      def test_call_helper_help(self):
74          helper = MockedHelper()
75          with CallMonitor(helper.help) as monitor:
76              self.mock.call_helper_help(helper)
77
78          self.assertIs(monitor.times_called, 1)
79          self.assertEqual(monitor.calls[0].args, ())
80          self.assertEqual(monitor.calls[0].kwargs, {})
81          self.assertIs(monitor.calls[0].return_value, True)
82
83      def fake_os_path_abspath(self, path):
84          return "/foo/bar/baz.py"
```

```
85
86      def test_return_false_filepath(self):
87          from my_package.my_mock_module import os as my_os
88          with replaced(my_os.path.abspath, self.fake_os_path_abspath, on=my_os.path):
89              self.assertEqual("/foo/bar/baz.py", self.mock.return_false_filepath())
90          self.assertNotEqual("/foo/bar/baz.py", self.mock.return_false_filepath())
91
92
93  if __name__ == '__main__':
94      unittest.main()
```

Listing 17: stubble my_mock_module lastline

```
1  import sys
2  from mocktest import *
3
4  from my_package.my_mock_module import (
5      NotMocked,
6      Helper,
7  )
8
9
10 class TestMocktest(TestCase):
11     def setUp(self):
12         self.not_mocked = NotMocked()
13
14     def test_print_foo(self):
15         expect(self.not_mocked).print_foo().then_return(lambda: print('foo'))
16         self.not_mocked.print_foo()
17
18         self.assertEqual(sys.stdout.getvalue(), 'foo\n')
19
20     def test_return_42(self):
21         when(self.not_mocked).return_42().then_return(42)
22         self.assertEqual(self.not_mocked.return_42(), 42)
23
24
25     def test_raise_error(self):
26         def raise_error():
27             raise BaseException('success')
28
29         modify(self.not_mocked).raise_error = raise_error
30         self.assertRaises(BaseException, self.not_mocked.raise_error,
31                           message='success', matching=r'.*')
32
33     def test_call_internal_function(self):
34         expect(self.not_mocked)._internal_function.once()
35         self.not_mocked.call_internal_function()
36
37     def test_call_internal_function_n_times(self):
38         expect(self.not_mocked)._internal_function.thrice()
39         self.not_mocked.call_internal_function_n_times(3)
40
```

```
41      def test_call_helper_help(self):
42          helper = Helper()
43          when(helper).help.then_return(True).once()
44          self.not_mocked.call_helper_help(helper)
45
46      def test_return_false_filepath(self):
47          import os
48          when(os.path).abspath.then_return('/foo/bar/baz.py')
49          self.assertEqual('/foo/bar/baz.py', self.not_mocked.return_false_filepath())
50
51
52  if __name__ == '__main__':
53      import unittest
54      unittest.main(buffer=True)
```

Listing 18: mocktest my_mock_module

```
1  import sys
2  import unittest
3  import flexmock
4
5  from my_package.my_mock_module import (
6      NotMocked,
7      Helper,
8  )
9
10
11 class TestFlexmock(unittest.TestCase):
12     def setUp(self):
13         self.mock = flexmock(NotMocked)
14         self.not_mocked = NotMocked()
15
16     def test_print_foo(self):
17         self.mock.should_receive('print_foo').replace_with(lambda: print('foo'))
18         self.not_mocked.print_foo()  # print_foo wurde ersetzt
19
20         self.assertEquals(sys.stdout.getvalue(),'foo\n')
21
22     def test_return_42(self):
23         self.mock.should_receive('return_42').and_return(42)
24         self.assertIs(42, self.not_mocked.return_42())  # returned jetzt immer 42
25
26     def test_raise_error(self):
27         self.mock.should_receive('raise_error').and_raise(BaseException, 'success')
28         self.assertRaises(BaseException, self.not_mocked.raise_error)
29
30     def test_call_internal_function(self):
31         self.mock.should_call('_internal_function').once()
32         self.not_mocked.call_internal_function()
33
34     def test_call_internal_function_n_times(self):
35         self.mock.should_call('_internal_function').times(3)
36         self.not_mocked.call_internal_function_n_times(3)
```

```
37
38      def test_call_helper_help(self):
39          helper_mock = flexmock(Helper())
40          helper_mock.should_receive('help').replace_with(lambda: True)
41          #helper_mock.should_call('help').once()  # Check ob help aufgerufen wurde
42
43          self.not_mocked.call_helper_help(helper_mock)
44
45      def test_return_false_filepath(self):
46          import os
47          new_os = flexmock(os)
48          new_os.should_receive('path.abspath').and_return('/foo/bar/baz.py')
49          self.assertEqual(self.not_mocked.return_false_filepath(), '/foo/bar/baz.py')
50
51
52  if __name__ == '__main__':
53      unittest.main(buffer=True)
```

Listing 19: flexmock my_mock_module

```
1  import sys
2  import unittest
3
4  from doublex import (
5      Stub,
6      Spy,
7      ProxySpy,
8      assert_that,
9      is_,
10     called,
11 )
12
13
14 from my_package.my_mock_module import (
15     NotMocked,
16     Helper,
17 )
18
19
20 class TestFlexmock(unittest.TestCase):
21     def setUp(self):
22         pass
23
24     def test_print_foo(self):
25         with Stub(NotMocked) as stub:
26             stub.print_foo().delegates(lambda: print('foo'))
27         stub.print_foo()  # Das Duplikat kann nun print_foo()
28         self.assertEquals(sys.stdout.getvalue(),'foo\n')
29
30     def test_return_42(self):
31         with Stub(NotMocked) as stub:
32             stub.return_42().returns(42)
33
```

```
34          assert_that(stub.return_42(), is_(42))
35
36      def test_raise_error(self):
37          with Stub(NotMocked) as stub:
38              stub.raise_error().raises(BaseException('success'))
39
40          self.assertRaises(BaseException, stub.raise_error)
41
42      def test_call_internal_function(self):
43          spy = ProxySpy(NotMocked())
44
45          spy.call_internal_function()
46          assert_that(spy._internal_function, called())
47          '''
48          Das ist mit doublex nicht moeglich, der Error ist folgender:
49
50          Expected: these calls:
51                  NotMocked._internal_function(ANY_ARG)
52              but: calls that actually ocurred were:
53                  NotMocked.call_internal_function()
54          '''
55
56
57      def test_call_internal_function_n_times(self):
58          pass
59          '''
60          Hier kommt der gleiche Fehler wie bei test_call_internal_function
61          '''
62
63      def test_call_helper_help(self):
64          with Stub() as helper:
65              helper.help().returns(True)
66
67          self.assertTrue(NotMocked().call_helper_help(helper))
68          '''
69          Dieser Test wird auch fehlschlagen, da es nicht moeglich ist vor zu
70          taeuschen eine Klasse zu sein. Ein doublex Objekt ist immer ein
71          Duplikat aber niemals die Klasse selbst.
72          '''
73
74      def test_return_false_filepath(self):
75          with Stub(NotMocked) as stub:
76              stub.return_false_filepath().returns('/foo/bar/baz.py')
77
78          assert_that(stub.return_false_filepath(), is_('/foo/bar/baz.py'))
79          '''
80          Es ist mit doublex leider nicht moeglich globale Objekte oder Module
81          zu ersetzen. Lediglich ein Duplikat koennte als parameter uebergeben
82          werden.
83          '''
84
85
```

```
86 if __name__ == '__main__':
87     unittest.main(buffer=True)
```

Listing 20: python-doublex my_mock_module

```
1 from hypothesis import given
2 from hypothesis.strategies import integers
3
4
5 @given(integers())
6 def test_int(i):
7     assert (i + i) == (i * 2)
8
9
10 if __name__ == '__main__':
11     test_int()
```

Listing 21: hypothesis @given() Beispiel

## Eidesstattliche Erklärung

# Eidesstattliche Erklärung zur Abschlussarbeit

Hiermit versichere ich, die eingereichte Abschlussarbeit selbständig verfasst und keine andere als die von mir angegebenen Quellen und Hilfsmittel benutzt zu haben. Wörtlich oder inhaltlich verwendete Quellen wurden entsprechend den anerkannten Regeln wissenschaftlichen Arbeitens zitiert. Ich erkläre weiterhin, dass die vorliegende Arbeit noch nicht anderweitig als Abschlussarbeit eingereicht wurde.

Das Merkblatt zum Täuschungsverbot im Prüfungsverfahren der Hochschule Augsburg habe ich gelesen und zur Kenntnis genommen. Ich versichere, dass die von mir abgegebene Arbeit keinerlei Plagiate, Texte oder Bilder umfasst, die durch von mir beauftragte Dritte erstellt wurden.

$Unterschrift:$ $Ort, Datum:$