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 (maximilian.konter@pm.me) MatNr. 951004
Version vom:	12. August 2019
1. Betreuer:	Dipl.-Inf. (FH), Dipl.-De Erich Seifert, MA
2. BetreuerIn:	Prof. Dr.-Ing. Alexandra Teynor

Abstrakt

Da die Anforderungen an Software im Laufe der Zeit stark angestiegen sind, haben sich einige Technologien entwickelt, die dazu verwendet werden diese Anforderungen zu gewährleisten. Diese Arbeit befasst sich mit Test-driven development im Bezug zu den aktuellen Test-Tools der Programmiersprache Python und analysiert diese auf ihre Tauglichkeit im Bezug zum Test-driven development.

Dabei werden zuerst drei verschiedenen Arten von Tools untersucht, die unit-testing-, mock-testing- und fuzz-testing Tools. Diese werden anhand ihrer Anwendbarkeit, Effizient, Komplexität und Erweiterbarkeit beschrieben und analysiert, um mit Hilfe eines Vergleichs derer einen Leitfaden für das am besten für Test-driven development geeignete Tool zu bieten. Das daraus resultierende Ergebnis soll aufzeigen, welche Toolkombinationen die beste Wahl anhand der vorgegebenen Kriterien darstellen.

Die dabei entstandenen Ergebnisse zeigen auf, dass nicht alle Tools optimal für Test-driven development geeignet sind. Schlussendlich ergaben sich vier Tools, welche gemeinsam eingesetzt, das optimale Toolset für die allgemeine Anwendung bieten. Dabei ist jedoch zu beachten, dass dies immer vom Anwendungsfall, sowie den Anforderungen an die Software und an die Tools abhängig ist. Dabei ergab sich, dass pytest zusammen mit mocktest sowie doctest und hypothesis gemeinsam eine sehr gute Wahl darstellen.

Inhaltsverzeichnis

Einleitung

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 im Jahr 2018 bereits circa 66% aller Menschen ein Smartphone (Schobelt, 2017) und im Arbeitsleben ist ein PC meist nicht mehr weg zu denken. Die dadurch benötigte Software muss bestimmte Anforderungen, welche die Nutzer an diese stellen, gewährleisten und sicherstellen. Mit steigenden Nutzerzahlen, steigen oft auch die Anforderungen an eine Software, sowie die gefundenen Bugs in dieser.
Im weltweiten Markt gibt es viele große Unternehmen, die gegenseitig um ihre Nutzer kämpfen. Die Anwender präferieren meist denjenigen Anbieter, welcher die bessere Software bietet. An dieser Stelle ist der Software-Architekt einer jeden Anwendung gefragt, welche Technologie eingesetzt wird um die qualitativen Anforderungen an diese zu gewährleisten. Eine dieser Technologien ist das Test-driven development (TDD), welches sich durch eine hohe resultierende Testabdeckung ihren Namen gemacht hat. Bei dieser Methode werden zuerst die Tests und dann der Code geschrieben, um so später sicher stellen zu können, dass die Software alle Anforderungen erfüllt und keine Bugs enthält.
Gerade zum Release einer neuen Software ist es für Unternehmen schwer, Nutzer zu akquirieren. Andere Anwendungen, die bereits auf dem Markt sind, haben schon viele ihrer Bugs gefixed und verfügen über eine durch den Anwender ausgiebig getestete Software. Um eine neue Applikation auf diesen Markt zu bringen, ist es erforderlich, dass diese zum Start ohne Fehler läuft. Dies kann mit Test-driven development sicher gestellt werden.
Python wird heutzutage von vielen Entwicklern immer häufiger eingesetzt und ist Mittlerweile die am zweit meisten beliebte und am vierthäufigsten genutzte Sprache weltweit (Stack Exchange Inc, 2019).
Aufgrund der zunehmenden Beliebtheit, sowie meiner eigenen Vorliebe für die Programmiersprache Python (auch Pythonista genannt) habe ich mich dazu entschieden meine Bachelorarbeit dieser zu widmen. Da ich beginnend mit dem Praxissemester und danach als Werkstudent in einer Firma an der Entwicklung der hausinternen Test-Umgebung mitgewirkt habe, wurde ich immer interessierter an dem Thema Testen. Durch die Beschäftigung mit Tests in meiner Freizeit bin ich auf das Test-driven development gestoßen und war sofort überzeugt, dass dies eine Technologie ist, die ich anwenden möchte. Doch leider war die Auswahl der Tools, welche für Test-driven development in

Frage kamen so hoch, das ich Angst hatte etwas verpassen zu können, da ich ein Tool einem anderen vorzog. Beim Analysieren der Tools viel mir auf, dass viele der Tools veraltet oder ungeeignet waren, was einen Einsatz für Test-driven development ausschloss. Die übrig gebliebenen Tools waren so umfangreich, dass ich das Projekt erst einmal zur Seite legte.
Die Bachelorarbeit erschien mir als der passende Rahmen für die intellektuelle und praktische Beschäftigung mit dieser Thematik. Zum einen konnte ich so meine Privaten Pläne diesbezüglich verwirklichen und zum anderen erschien mir das Thema relevant für meine späteren Beruflichen Perspektiven.
Das Ziel dieser Arbeit ist es, verschiedene aktuelle Test-Tools der Programmiersprache Python auf ihre Tauglichkeit für den Einsatz von Test-driven development zu untersuchen und zu vergleichen. Dabei wird zuerst jedes Tool und seine Funktionalitäten beschrieben. Im Anschluss wird ein jedes auf die verschiedene Anforderungen untersucht, welche es erfüllen muss um für Test-driven development in Frage zu kommen. Dies geschieht anhand eines erstellten Codebeispiels, welches gewisse Funktionalität liefern sollte. Dieses Beispiel wird mithilfe der verschiedenen Tools auf die gleiche Art getestet. Die dabei entstehenden Komplikationen, sowie der benötigte Aufwand zeigen auf, welche Tools für die Analyse benötigt werden.
Die Arbeit soll aufzeigen welche Kombination der verschiedenen Tools die optimale Verbindung bieten um Test-driven development durch zu führen. Dabei erwarte ich mir tiefere Einblicke in das Test-driven development zu erhalten, um heraus zu finden ob diese Technologie für mich und für meine weiteren Arbeiten relevant ist.
Dazu wird in Kapitel 1 zunächst geklärt, welche Eigenschaften die Programmiersprache Python hat, sowie die Definition von Test-driven development. Die genutzte Methodik wird in Kapitel 2 beschrieben und anschließend in Kapitel 3 angewendet. Die Analyse ist in vier Unterpunkte gegliedert, in Punkt 3.1 werden zunächst die Tools der Standardbibliothek analysiert. Diese sind unittest und doctest. Zusammengefasst unter Punkt 3.2 sind die Tools, welche nicht in der Standardbibliothek zu finden sind. Diese setzen sich aus pytest, stubble, mocktest, flexmock, doublex und hypothesis. Darauffolgend werden unter Punkt 3.3 die analysierten Tool verglichen um unter Punkt 3.4 mögliche Kombinationen dieser auf zu Zeigen. Bevor in Kapitel 5 das Fazit gezogen wird, werden die Ergebnisse der Arbeit in Kapitel 4 diskutiert.

1 Definition

1.1 Die Programmiersprache Python

Python ist eine dynamisch typisierte Programmiersprache, sie ist Open Source unter der Python Software Foundation License (PSFL[1]). Eine dynamisch typisierte Programmiersprache bestimmt zur Laufzeit, welchem Typ eine Variable angehört. Dies ist möglich, da Python eine interpretierte und keine kompilierte Sprache ist. Interpretierte Sprachen werden, zur Laufzeit, in Maschinencode umgewandelt, während kompilierte Sprachen davor umgewandelt werden. Beides hat seine Vor- und Nachteile auf diese hier aber nicht weiter eingegangen werden sollen, da dies nicht die Aufgabe dieser Arbeit ist.
Python unterstützt verschiedene Programmier-Paradigmen, wie die funktionale Programmierung oder die objektorientierte Programmierung. Dadurch, dass Python interpretiert ist, wird die Sprache auch oft als Skript Sprache genutzt.
Das Hauptmerkmal der Sprache ist die Syntax, die dabei verwendet werden muss. Bei Python werden keine geschweiften Klammern genutzt, stattdessen werden Einrückungen in Form von vier Leerzeichen verwendet. Dadurch entsteht ein leicht zu lesender Code, der gerade für Anfänger besser zu verstehen ist.

1.2 Test-driven development

Das Test-driven development (TDD) wurde durch Kent Beck, einem Amerikanischen Softwareingenieur und Autor bekannt. Dieser beschreibt in seinem Buch „Test Driven Development: By Example“ wie bei der Anwendung von TDD vorgegangen werden soll. Demnach sieht der Entwicklungsprozess nach Beck, 2002 folgendermaßen aus (siehe Abbildung 1):

Am Anfang sollte geklärt sein, welche Funktionen erfüllt werden müssen. Anhand dieser beginnt dann die Anwendung von TDD. Dazu wird der folgende Zyklus beschrieben, der sich so lange wiederholt, bis alle Features implementiert und getestet wurden.
Der Erste Schritt (siehe Abbildung 1 #1) dazu ist das Schreiben eines Tests, der neue Funktionalität oder Verbesserung zu einer bestehenden Unit hinzufügt. Dieser Test sollte möglichst kurz und aussagekräftig sein. Dazu muss der Entwickler die Spezifikationen und Anforderungen des Features genau verstehen um den Test wirklich effektiv schreiben zu können. Ein Entwickler, der die Anforderung nicht ganz versteht, ist nicht in der

[1] https://docs.python.org/3/license.html

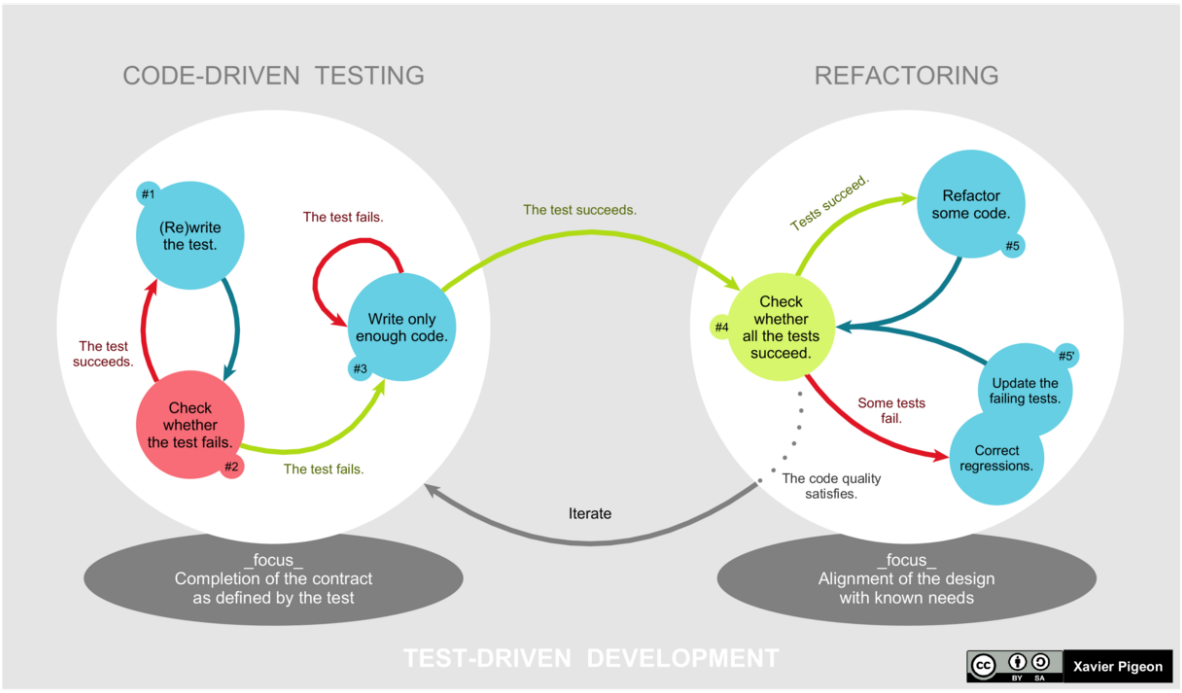

Abbildung 1: TDD Zyklus

Lage einen Test zu schreiben, welcher alle Bereiche abdeckt. Dieser Entwickler muss allerdings nicht zwangsläufig später auch die Funktionalität implementieren. Der Test sollte demnach so umfassend und genau sein, dass jeder Entwickler die dazugehörige Logik implementieren kann.
Als nächstes (siehe Abbildung 1 #2) werden alle Tests die bereits bestehen, sowie der Neue durchgeführt um zu sehen, ob der neue Test fehlschlägt. Dies bestätigt unter anderem auch, dass der neue Test die anderen Tests nicht beeinflusst und auch ohne neuen Code, der Funktionalität implementiert, nicht bestanden wurde. Dadurch kann der Entwickler ausschließen, dass ein Test immer korrekt ist, wodurch das Vertrauen des Entwicklers in den Test gesteigert wird.
Der nächste Schritt (siehe Abbildung 1 #3) ist die Implementierung des Codes, der getestet werden soll. Dabei ist unwichtig, wie der Code geschrieben wird. Das bedeutet, dass es erlaubt ist „schmutzigen“ (Code der nicht den gefordertem Standard entspricht) Code zu produzieren, welcher so nicht akzeptiert werden würde. Die Hauptsache dabei ist, dass der Test bestanden wird. Die Codequalität spielt hier keine Rolle, da sie in Punkt Fünf überarbeitet wird. Der Entwickler darf in diesem Prozess auch keinen weiteren Code, der nicht notwendig ist um den Test zu bestehen, hinzufügen. Dies verhindert, dass Code geschrieben wird, der keine zugehörigen Tests hat.

Ist die gesamte Logik implementiert, werden im vorletzten Schritt (siehe Abbildung 1 #4) alle Tests durchgeführt und überprüft, ob diese erfolgreich waren. Sollte dies nicht der Fall sein, muss die Implementierung korrigiert werden und/oder eventuell der Test angepasst werden (siehe Abbildung 1 #5').
Sind alle Tests bestanden wird im finalen Schritt (siehe Abbildung 1 #5) der neue Code aufgeräumt. Dieser Punkt ist sehr wichtig, da in Punkt Drei der Code nur funktionieren muss. Dadurch kann es sein, dass hier Code entstanden ist, der nicht den qualitativen Anforderungen entspricht. Sämtliche Objekte sollten einen aussagekräftigen Namen erhalten und der Code sollte, sofern dies nötig ist an einen Ort verlegt werden, der seiner logischen Aufgabe entspricht. Duplikationen müssen entfernt werden und nach jeder Aufräumaktion sollten die Tests noch einmal ausgeführt durchlaufen, um zu verifizieren, dass alles noch funktioniert.

Dieser Zyklus wird nun von vorne so lange wiederholt, bis jedes Feature implementiert ist und die Software als fertig gestellt gilt. Die dafür nötigen Schritte sollten möglichst klein sein, um eine höhere Testabdeckung zu erreichen.

2 Methodik zur Analyse von Python Testtools

Die zu analysierenden Tools, werden zwischen Standardbibliothek (STDLIB) und externen Tools unterschieden. Zusätzlich findet eine Gliederung in unit-,mock- und fuzz-testing Tools statt.
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 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, welches die Funktionalität bereit stellt Tests aus zu führen, so ist dieses Tool als Erweiterung zu sehen und wird nicht analysiert. Bietet ein Tool allerdings Funktionen zum Erweitern von verschiedenen Test Tools, so wird es hier zur Analyse verwendet. Diese Unterscheidung wird genutzt um Duplikationen in den einzelnen Vergleichen zu vermeiden und verhindert, dass Tools mit sehr vielen Erweiterungen den Rahmen dieser Arbeit 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? Bei unit-testing Tools beinhaltet dies Fixtures und Mocks während es bei den mock- und fuzz-testing Tools um die verschiedenen Features geht. Sowohl das Mocken als auch das Stuben sind für das TDD von großer Bedeutung. Einerseits ist es wichtig nach zu verfolgen, welches Modul wann und mit welchen Parametern aufgerufen wurde. Andererseits ist es genau so wichtig Tests ab zu schotten. Dies geschieht mithilfe der Stubs, welche es ermöglichen externe Abhängigkeiten zu ersetzten um Fehler aus anderen Quellen aus zu schließen (Percival, 2014). Zusätzlich wird in diesem Aspekt auch die benötigten Abhängigkeiten eines Pakets untersucht, um dieses betreiben zu können. Mehr Abhängigkeiten führen dazu, dass mehr Entwicklern vertraut werden muss, dass diese Ihre Pakete aktuell und fehlerfrei halten. Der zweite Aspekt ist die Effizienz mit der sich ein Tool einsetzen lässt. Dabei wird darauf geachtet, wie viel Aufwand benötigt wird, das Tool ein zu setzen. Im Besonderen wird hier auf die Vorarbeit, die benötigt wird um das Tool verwenden zu können, geachtet. Des weiteren wird in diesem Aspekt bei den unit-testing Tools die Effizienz, mit der Tests ausgewertet werden können, analysiert. Als dritter Aspekt wird die Komplexität des Tools bewertet. Dabei wird diese nicht zwangsläufig als negativ gesehen. Hat ein Tool viel Funktionalität neben seiner Basis Features, so hat es in diesem Bezug eine hohe Komplexität, die positiv zu sehen ist. Hingegen wird viel oder unübersichtlicher Code den das Tool produziert als negativer Komplex gesehen. Zuletzt wird bei den unit-testing Tools auf die Erweiterbarkeit geachtet. Diese umfasst Erweiterungen der Entwickler als auch der Community. Bei den mock- und fuzz-testing Tools, wird dieser Aspekt aus gelassen, da es sich bei diesen Tools vorwiegend um Erweiterungen zu den unit-testing Tools handelt.

Zum Vergleich der einzelnen unit-testing Tools untereinander werden diese auf den in Listing 1 abgebildeten Code angewendet. Da sich diese Arbeit auch mit mock-testing Tools beschäftigt, wurde auch für diese ein Modul geschrieben, welches zu testen ist. Der Code dazu befindet sich in Listing 2. Durch die Anwendung des jeweiligen Tools auf diesen Code, ist es möglich die Aspekte dieser untereinander zu vergleichen.
Das Modul aus Listing 1 für die unit-testing Tools 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 eine Items Tabelle enthält. Jedes Item hat eine ID (id), einen Namen (name), einen Lagerplatz (storage_location) und eine Anzahl der vorrätigen Items amount. Dazu besitzt jedes Item

eine Methode do_something() welche eine externe Funktion/Methode, die jedoch noch nicht existiert do_something_which_does_not_exist(), aufruft.
Für die mock-testing Tools wurde eine Klasse geschrieben, welche Funktionen besitzt, durch ihren Namen ausdrücken, welche Funktionalität sie eigentlich haben sollten. Um jedoch testen zu können, ob das jeweilige Tool diese Methode stuben kann, gibt nicht jede dieser Methoden das gewünschte Ergebnis zurück. Das Tool soll dafür sorgen, dass das gewünschte Ergebnis erfolgt. Selbstverständlich ist realer Code viel komplexer als in diesem Listing (2) dargestellt, jedoch reicht dieser Code aus, um viele Tools an die Grenzen Ihrer Funktionalität 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 Methode der Klasse Helper noch nicht implementiert wurde. Auch die Methode return_false_filepath bringt das ein oder andere Tool an die grenzen seiner Funktionalität, da eine externe Methode aus der STDLIB ersetzt werden muss, wobei dies nicht global geschehen darf.
Für die Fuzz-testing Tools werden keine Beispiele geschrieben, da diese meist auf komplexen Beispielen basieren um Sinn zu ergeben. Stattdessen werden im jeweiligen Kapitel kleinere Beispiele genannt, wie das Tool ein zu setzen ist oder es wird auf Beispiele aus der jeweiligen Dokumentation verwiesen.
Verfügt ein Tool über keinen Test-runner, so wird der von der STDLIB gestellte Runner unittest verwendet.

3 Python Test-Tool Analyse

In diesem Kapitel werden die einzelnen Tools anhand der in Kapitel 2 beschriebenen Methodik analysiert und verglichen. Am Ende des Kapitel wird für jede Kategorie (unit-, mock- und fuzz-testing) eine Empfehlung anhand der jeweiligen Analyse gegeben, welches Tool die beste Eignung für die Anwendung von TDD hat.

3.1 Tools der Standard Bibliothek

Die Standard Bibliothek von Python bietet zwei verschiedene Test-Tools (Muthukadan et al., 2011). Zum einen unittest[2] und zum anderen doctest[3]. Diese beiden Tools sind in ihrem Umfang bereits so vielseitig, dass es für einen Entwickler unproblematisch ist, eine

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

hohe Testabdeckung eines Programms oder einer Bibliothek zu erreichen.

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

3.1.1 unittest

Das von JUnit inspirierte (Brandl, van Rossum, Heimes, Peterson, Pitrou et al., 2007) Tool unittest, ist Bestandteil der Python Standardbibliothek und bietet seit jeher seinen Nutzern ein umfangreiches Repertoire an Funktionen zum Testen von Python Code. Die Funktionalität von unittest lässt sich mit folgenden Punkten beschreiben:Fixture, zum Präparieren der Tests, Testfälle zum Gliedern einzelner Tests, Testumgebungen zum Gliedern von zusammengehörigen Tests und Test runner zum Ausführen von Testumgebungen oder Testfällen. Diese Funktionalität von unittest ermöglicht es Anwendern eine komplette Testumgebung zu erstellen und zu nutzen.

Dazu wird die Testumgebung, eines der Hauptfeatures, von unittest verwendet, mit der es möglich ist, Tests in logische Klassen zu gliedern. Es besteht die Möglichkeit ohne die Testumgebung unittest einzusetzen, jedoch ermöglichen diese eine strukturierte Arbeitsumgebung, sowie die Verwendung von Fixtures.
Jede Testumgebung verfügt über Testfälle, welche einen Test repräsentieren. Ist eine setUp() und/oder tearDown() Methode für eine Testumgebung definiert, so werden diese vor und nach jedem Test ausgeführt. Alternativ kann dies aber auch nur bei einer Initialisierung der Testumgebung und beim Verlassen dieser geschehen. Allerdings sind die Tests danach nicht mehr untereinander unabhängig, weshalb sie dann „Integration Tests“ genannt werden.
Testfälle bestehen aus den jeweiligen Tests, die mit Hilfe der verschiedenen assert Methoden von unittest geprüft werden. Eine assert Methode stellt eine Behauptung auf, die getroffen werden muss, damit ein Test als bestanden angesehen wird. Die von unittest bereit gestellten Methoden unterscheiden sich ein wenig von der assert Methode aus der STDLIB. Eine Überprüfung auf True würde mit der aus der STDLIB stammenden Funktion so aussehen: assert xy is True, während unittest eine Methode bereit stellt, mit der es etwas

kürzer und leichter zu lesen ist, assertTrue(xy)). unittest stellt noch viele weitere dieser Methoden zur Verfügung. Die Dokumentation verweist auf diese [4].
Ein kleines Beispiel zu den assert Methoden ist in Listing 3 zu finden. Dabei wurde die Funktion my_pow() aus my_module (Listing 1) getestet. Dabei ist zu erkennen, wie problemlos es möglich ist, eine Funktion auf einen Wert zu prüfen. Dies geschieht einerseits anhand eines selbst errechneten Wertes und andererseits anhand der Quadratfunktion aus der STDLIB.
Um mit TDD Units unabhängig testen zu können, müssen abhängige Units mit einem Mock oder einem Stub ersetzt werden. 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.
Allerdings 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 definiert wurde (Foord et al., 2012).
unittest bietet des Weiteren ein CLI, mit welchem es dem Benutzer möglich ist, seine Tests gebündelt auszuführen und auszuwerten. Mit dem CLI ist es auch möglich, automatisch Tests in einem Ordner zu „entdecken“(engl.: discover) und auszuführen. Dadurch ist es sehr leicht, neue Tests in ein bestehendes Test System einzuführen und diese ohne Veränderungen am bestehenden System aus zu führen.
Die folgenden Listings 4 und 5 zeigen wie ein erfolgreicher und ein misslungener Test aussehen können. Beide Outputs stammen aus dem Test der my_pow() Funktion aus Listing 1. In beiden Listings ist gut zu erkennen, ob und was bei einem Test fehlgeschlagen ist. Der erfolgreiche Test zeigt dem Nutzer sofort wie viele Test in wie vielen Sekunden gelaufen sind. Falls gewünscht 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 5 zu erkennen ist. Am Ende wird auch noch einmal angezeigt, wie viele Tests fehlgeschlagen sind.
Um einen Vergleich zwischen den unit-testing Tools zu schaffen, wurde die in Listing 1 definierte Klasse Item mit unittest getestet. Der Code dazu ist in Listing 6 zu finden. Die Tests wurden in einer Testumgebung gegliedert und verfügen über eine setUp() und tearDown() Fixture. Die setUp() Methode startet eine Datenbank session auf der gearbeitet werden kann, sowie ein Objekt self.item, mit welchem in den jeweiligen Tests gearbeitet

[4] https://docs.python.org/3/library/unittest.html

wird. Die tearDown() Methode sorgt dafür, dass die Datenbank nach jedem Test geschlossen wird, um sicher zu stellen, dass ungenutzte Datenbank Sessions offen sind. Im ersten Test wird überprüft, ob ein der Datenbank hinzugefügtes Objekt auch wirklich in dieser ist. Im zweiten Test wird überprüft, ob es möglich ist, eine nicht existierende externe Funktion zu ersetzen, sodass die Tests erfolgreich verlaufen.

Im Aspekt Anwendbarkeit bietet unittest alles um als Tool für TDD in Frage zu kommen. Dies kann jedoch nur unter Einbezug der Erweiterung unittest.mock. Da sich das Tool in der STDLIB befindet, sind keine weiteren Pakete zur Verwendung von unittest nötig.
Die Effizienz Tests aus zu werten ist hoch. Ein Entwickler kann mit ein paar Zeilen Code eine Testumgebung, die Fixtures besitzt, sowie verschiedene Testfälle, die von ihnen abhängig sind, aufsetzen. Der Code, der vor den eigentlichen Tests geschrieben werden muss, hält sich also in Grenzen. Die Effizienz mit der ein Entwickler Tests auswerten kann ist hingegen nicht optimal. Bei mehreren Fehlern und langem StackTrace wird der Output schnell für das Terminal unübersichtlich. Ein farbiger Output würde hier ein wenig Abhilfe schaffen.
unittest bietet einiges an Funktionalität zum Schreiben von Tests. Durch die verschiedenen assert Methoden, ist es dem Entwickler möglich, übersichtliche und lesbare Tests zu schreiben. Die gebotene Funktionalität sollte für die meisten Nutzer, abgesehen von ein paar Randbedingungen, bei denen spezielle Anforderungen gestellt sind, ausreichend sein. unittest verfügt über einige interessante Erweiterungen, welche das Tools mit neuen Funktionalitäten auffrischen. Die folgende Auflistung zeigt ein paar der bekannteren Erweiterungen: nose[5](veraltet), nose2[6], twisted[7] und testtools[8].

3.1.2 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.“[9] (Brandl, van Rossum, Heimes, Peterson, Melotti et al., 2007). Diese Textstücke müssen sich in Kommentaren befinden, da sie sonst von Python als Code interpretiert werden.

[5]https://github.com/nose-devs/nose
[6]https://github.com/nose-devs/nose2
[7]https://github.com/twisted/twisted
[8]https://launchpad.net/testtools
[9]Übersetzt aus dem Englischen

doctest bietet dem Nutzer eine Möglichkeit mit Hilfe von einem Test den Code zu dokumentieren. Da doctest lediglich Code, wie in einer interaktiven Python-Sitzung ausführt, werden die Möglichkeiten der Ausführung auf das, im Code geschrieben beschränkt. Testfälle oder gar Mocks sind hierbei nur bedingt realisierbar. Fixtures hingegen sind in Form von Code, der vor dem Test ausgeführt wird, teilweise realisierbar.
doctest selbst nutzt keine assert Funktionen oder -Methoden, stattdessen wird der Output eines Befehls überprüft. 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 7 zu sehen. Um einen Output zu erzeugen wurde noch ein fehlerhafter Test hinzu gefügt. Dieser Output ist in Listing 8 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 """ (drei Interpunktionszeichen) 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 längere Tests geschrieben werden, schnell unübersichtlich und lang. Zwar wird durch externe Textdateien Abhilfe geschaffen, dennoch sind zu lange und komplexe doctests schwer zu lesen und nach zu vollziehen.

Das Listing 10 zeigt den Code aus Listing 1, versehen mit Docstrings. Dieser Test wurde mit Hilfe der main Funktion von Python ausgeführt. Der in Listing 11 ausgelagerte Test wurde mit Hilfe des CLIs von doctest ausgeführt. Da beide Tests 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 zeigen lediglich, welche Kommandos ausgeführt wurden, was erwartet wurde und dass, das Erwartete eingetroffen ist. Ein kleines Beispiel ist in Listing 9 zu sehen.
Da doctest selbst keine Testfälle unterstützt, besitzt unittest eine Integration für doctest. Diese ermöglicht es Tests aus Kommentaren, sowie Textdateien in unittest zu integrieren und zu gliedern. Wie unittest, ist auch doctest in der STDLIB, wodurch keine externen Abhängigkeiten geladen werden müssen. Für die Analyse wird diese Integration allerdings nicht in Betracht gezogen, da doctest als Modul selbst analysiert werden soll und nicht die Funktionalität, die von unittest geboten wird.

Im Bezug auf die Anwendbarkeit von doctest lässt sich sagen, dass nicht alles zur Verfügung steht um TDD zu betreiben. Tests können nicht vor der eigentlichen Funktionalität geschrieben werden. Fixtures und Mocks werden nicht unterstützt.
Die Anforderungen für doctest sind sehr gering. Das Testen geschieht mit Hilfe der interaktiven Python-Sitzung, welche jedem Pythonentwickler bekannt ist. Die Tests selbst sind Aufrufe der geschriebenen Funktionen und Methoden, sowie ein Abgleich des Outputs mit dem erwarteten Wert. Sollten allerdings Fixtures gewünscht sein, so ist dies nur mit Vorarbeit möglich, da diese jedes mal an der benötigten Stelle geschrieben werden müssen. Der Output von doctest gleicht dem von unittest und verfügt über dieselben Schwächen. Bei vielen Fehlern und langem StackTrace wird der Output im Terminal schwer lesbar. Die Effizienz mit der doctest genutzt werden kann, ist demnach mittelmäßig. Zum einen ist es einfach Tests zu schreiben und zum anderen ist viel Code nötig. Funktionalität, wie das mocken, fehlen.
doctest selbst bietet wenig Komplexität von sich aus. Stattdessen ist die Komplexität, welche möglich ist, vom Entwickler abhängig, da dieser die Tests komplett selber schreiben muss. Je nach zu testender Funktion/Methode kann dies, abhängig vom Testaufwand schnell unübersichtlich werden, wenn viel Code abseits des eigentlichen Tests benötigt wird. doctest stellt dem Entwickler lediglich ein paar Möglichkeiten Optionen zu aktivieren, die den Test beeinflussen können. Diese sind in der Dokumentation unter dem Punkt Option Flags zu finden[10].
Als Erweiterung besteht lediglich die Integration in unittest[11] sowie pytest[12].

Da sich mit doctest Funktionen nicht präparieren lassen, ist dieses Testmodul für die alleinige Anwendung in TDD nicht nutzbar. Das Tool bietet keine Möglichkeiten Objekte zu mocken, wodurch Tests an nicht implementierbaren 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.

3.2 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

[10] https://docs.python.org/3.7/library/doctest.html#option-flags
[11] Brandl, van Rossum, Heimes, Peterson, Melotti et al., 2007.
[12] Krekel und pytest-dev team, 2019.

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 Fehler behoben und Features implementiert sein können.
Manche der dort aufgelisteten Tools sind bereits oder werden gerade in andere Tools integriert. Dies ist nötig, da ein Tool eine Erweiterung für ein anderes war und die Entwickler die Änderungen angenommen haben oder um die Tools zu verbessern und mehr Entwickler zur Verfügung zu haben. Selbstverständlich können auch andere Gründe dafür verantwortlich sein.
Da sowohl Software als auch Programmiersprachen sich stetig weiter entwickeln, werden in dieser Arbeit nur jene Tools behandelt, die sich diesen Entwicklungen anpassen. Diese Anpassung zeigt sich durch die Unterstützung der aktuellsten Version von Python oder 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 bleibt lediglich pytest[13] übrig (Muthukadan et al., 2011). Tools, wie reahl.tofu[14] oder zope.testing[15], 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[14] selbst ist auch nur eine Erweiterung für bestehende Test Tools, wie zum Beispiel lstinlinepytest, weshalb es hier nicht behandelt wird. So wird zope.testing[15] auch nicht behandelt, da dieses Tool wie bereits beschrieben, nicht aktiv weiter entwickelt wird, da es eher, zope[16] Applikationen und nicht Python Applikationen testet.
Der Test-Runner nose[17], der eine Erweiterung zu unittest bietet, ist nicht mehr in Entwicklung, dennoch haben sich ein paar Enthusiasten des Tools zusammengeschlossen und nose2[18] geschrieben. Da nose2[18], wie sein Vorgänger eine Erweiterung zu unittest darstellt, wird es hier nicht analysiert.
Auch sehr bekannt ist testtools[19], welches jedoch auch als Erweiterung zu unittest gesehen werden muss. Hier findet ebenfalls keine weitere Entwicklung statt.

Als weitere Kategorie werden Mock-Tools geführt. Auch wenn fast alle unittest-Tools

[13]https://github.com/pytest-dev/pytest/
[14] https://www.reahl.org/docs/4.0/devtools/tofu.d.html
[15] https://pypi.org/project/zope.testing/
[16]http://www.zope.org/en/latest/
[17]https://pypi.org/project/nose/1.3.7/
[18] https://pypi.org/project/nose2/
[19]https://pypi.org/project/testtools/

integriertes mocking haben, lässt sich mit diesen Tools meist mehr erreichen. Es wurden die gleichen Filterkriterien verwendet wie bei den unit-testing Tools (Muthukadan et al., 2011). Die demnach relevanten Tools sind: stubble[20], mocktest[21], flexmock[22], python-doublex[23] und python-aspectlib[24].

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[25] (Muthukadan et al., 2011).

3.2.1 pytest

Das am 04. August 2009 in Version 1.0.0[26] 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 anstatt self.assert, Error Überprüfung mit Contextmanager, informativer Output in Farbe, der angepasst werden kann, Feature reiche Fixtures sowie vordefinierte Fixtures von pytest selbst, die unter den verschiedenen Tests geteilt werden können oder global definiert für mehrere Module verfügbar gemacht werden können, sowie die Parametrisierung dieser, die Überprüfung von stdout und stderr, das stuben von Objekten und zuletzt die Gliederung der Testfälle durch Markieren, Nodes (Auswahl der Modul Abhängigkeit) oder anhand der Funktionsnamen (String Abgleich) (Krekel und pytest-dev team, 2019).
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 für TDD erfüllt, da pytest viele verschiedene Arten bietet, diese zu benutzen. Fixtures werden in pytest allerdings nicht mit setUp und tearDown geschrieben, sondern werden mit Hilfe 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.

[20] https://www.reahl.org/docs/4.0/devtools/stubble.d.html
[21] https://github.com/timbertson/mocktest/tree/master
[22] https://github.com/bkabrda/flexmock
[23] https://bitbucket.org/DavidVilla/python-doublex
[24] https://github.com/ionelmc/python-aspectlib
[25] https://github.com/HypothesisWorks/hypothesis
[26] https://github.com/pytest-dev/pytest/releases/tag/1.0.0

Die Fixture wird dann den Code bis zum yield fortsetzen und nach der Funktion den Rest nach yield ausführen. Ein Beispiel dazu ist in Listing 12 zu finden, der dazugehörige Output in Listing 13. Diese lassen sich durch zusätzliche Parameter weiter anpassen. Die dazugehörigen Dokumentation ist in Kapitel Fünf der Dokumentation beschrieben (Krekel und pytest-dev team, 2019).
Des Weiteren bietet pytest auch stubing. So lässt sich beispielsweise mit monkeypatch.setattr() der Rückgabewert 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 und weniger um das Aufräumen nach dem Testen. Auch hier bietet pytest weitere Möglichkeiten Stubs zu verwenden. Die dazugehörige Dokumentation ist in Kapitel Sieben zu finden (Krekel und pytest-dev team, 2019).
Ein weiteres Feature von pytest ist die Gliederung in Test-Fälle. Diese lassen sich vom Entwickler mit vielen Einstellungsmöglichkeiten verfeinern. So lässt sich, wie in den Basis Features bereits beschrieben, ein Test mit einer oder mehreren Markierungen versehen, wodurch eine Gliederung nach Markierung entsteht. Durch die Selektion bestimmter Wörter 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 sind in Kapitel Sechs notiert (Krekel und pytest-dev team, 2019).
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 ein wirklich benötigtes. Für eine vollständige Auflistung und Erklärung aller Features kann jederzeit unter https://docs.pytest.org/en/latest/ die aktuellste Version der Dokumentation abgefragt werden.
Das Testen des in Listing 1 definierten Moduls gestaltete sich mit pytest ohne Probleme (vgl. Listing 14, sowie 15). Da beim Testen keine Probleme gefunden wurden, wird hier nicht weiter auf den Test eingegangen.

Im Aspekt der Anwendbarkeit bietet pytest fast alles, um TDD effektiv anwenden zu können, da sowohl Stubs, als auch Fixtures ohne Erweiterungen möglich sind und dazu noch im Umfang einiges zusätzlich zur Basisfunktionalität bieten. Lediglich das Mocken von Objekten ist im Basisumfang nicht enthalten. Da pytest nicht in der STDLIB ist,

muss es aus externen Quellen installiert werden. Dabei werden für pytest 4.4.0 die in Listing 16 gezeigten Abhängigkeiten installiert. Demnach benötigt pytest sechs externe Abhängigkeiten.
Da pytest keine neuen assert Methoden hinzufügt, lässt sich ein Test ohne großen Zeitaufwand und Komplikationen schreiben. Mit pytest ist es dem Entwickler möglich mit einem geringen Aufwand Fixtures und Mocks zu nutzen.
Die Effizienz, mit der pytest verwendet werden kann, ist hoch, denn der Entwickler muss keine neuen Konstrukte lernen um Tests zu schreiben. Die Auswertung der Tests erfolgt auf der Konsole in farbigen Output und lesbarer Gliederung, wie in Listing 15 zu sehen ist. Demnach ist es dem Entwickler möglich, ohne großen Zeitaufwand die Ergebnisse der Test effizient aus zu werten.
Die Features von pytest sind äußerst umfassend und bieten dem Nutzer nahezu alles an Funktionalitäten, die gewünscht sein könnten. Durch die unkomplizierte Verwendung von pytest ist es den Entwicklern möglich, einen übersichtlichen und gut lesbaren Code zu schreiben. Dieser wird auch mit steigender Komplexität, dank der Strukturen von pytest nicht weniger lesbar.
Sollten dem Nutzer die Features von pytest nicht genügen, so lässt sich unter http://plugincompat.herokuapp.com/ eine Liste von 683 (Stand: 19. Juli 2019) Erweiterungen für pytest 5.0.0 finden. pytest selbst kann allerdings auch als Erweiterung zu unittest genutzt werden, indem es als Test-runner für diese verwendet wird. Dadurch ist es dem Entwickler möglich, den verbesserten Output von pytest zu verwenden. Zusätzlich bietet pytest eine sichtbare Gliederung des Outputs, wodurch dieser zusätzlich lesbarer wird.

3.2.2 Mocking Tools

Um eine bessere Übersicht zur Verfügung zu stellen, werden die Mocking Tools in diesem Kapitel gegliedert.
Wie in Python Test-Tool Analyse bereits beschrieben, werden hier Tools analysiert, die eine Erweiterung für Test Tools im Bezug auf verbessertes mocking zur Verfügung stellen.

3.2.2.1 stubble

stubble zählt zwar zu den mocking Tools, jedoch werden hier nur Stubs als Funktionalität gegeben. Dennoch ist es möglich, mithilfe eines der vorgefertigten Stubs, mocking Funktionalität zu erhalten.

Angenommen ein Objekt class Original hat eine Abhängigkeit zu dem zu testenden Objekt. Mit stubble lässt sich ein Objekt class Fake erstellen, welches einen decorator besitzt @stubclass(Original). Dadurch stellt stubble sicher, dass alle Methoden aus class Fake der Signatur derer aus class Original entsprechen. Zusätzlich kann class Fake von class Original erben, wodurch nur die Funktionen überschrieben werden, die in class Fake definiert wurden.
Um Helfermethoden zu class Fake hinzu zu fügen, müssen diese mit @exempt annotiert werden. Ist das Objekt von class Original bereits initialisiert, so lässt sich dieses an class Fake übergeben, wenn diese von reahl.stubble.Delegate erbt. Dadurch ist es möglich zur Laufzeit ein Objekt durch einen Stub ersetzen. Als weiteres Feature wird die Möglichkeit geboten, Stubs mit setuptools zu verwenden um Installationen zu stubben.
Die Hauptfeatures von stubble sind jedoch die vorgefertigten Stubs. Dem Entwickler stehen dabei folgende Stubs zur Verfügung: Zum einen der SystemOutStub[27], der den Standard Out ersetzt und zum anderen der CallMonitor[28], der dazu dient zu überprüfen, welche Methoden wann, mit welchen Parametern und wie häufig aufgerufen wurden. Diese Stubs lassen sich als Contextmanagers einsetzen, wodurch es leichter wird, Code mit diesen zu schreiben. Ein weiterer Contextmanagers ist replaced. Dieser lässt den Entwickler eine Methode oder Funktion durch eine andere ersetzen, wodurch solange der Contextmanager aktiv ist, die neue Methode/Funktion genutzt wird.
Des Weiteren bietet stubble einige experimentelle Features, die jedoch eher weniger für den praktischen Einsatz geeignet sind (Vosloo, Sparks und Nagel, 2018), 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 ist. Das Beispiel dazu ist in Listing 17 zu finden.
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 Stub ersetzt werden können. So werden auch Objektvariablen, die von originalen Methoden gesetzt oder verändert werden, nicht im Stub gesetzt. Aus diesem Grund wird Delegation als experimentelles Features gesehen, da ein Bug, der durch dieses Feature entstanden ist, schwer zu finden ist (Vosloo et al., 2018).
Mit stubble alleine lässt sich dieser in Listing 2 definierte Code nicht zu 100% optimal testen (vgl. Listing 18). So ist die nicht existierende Methode Helper.help(self) nicht ohne

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

Probleme zu ersetzen. Dies gelingt, lediglich, wenn in der Stub Klasse die Methode help(self) definiert und sie mit @exempt annotiert wird. Jedoch verliert man dadurch die Funktionalität von stubble, welche überprüft ob die Signatur der Funktionen übereinstimmen. Alternativ wäre es möglich eine Klasse zu nehmen, die ohne die @stubclass auskommt. Ansonsten zeigt der Code, wie ein recht simples Tool viel erreichen kann, wenn auch mit viel Code abseits der Tests. Dabei ließ sich os.path.abspath(path) ohne Komplikationen mit Hilfe von reahl.stubble.replaced ersetzen. Das Abfangen des STDOUTs mit Hilfe der vorgefertigten Stubs erwies sich als unkompliziert.

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 Abhängigkeit (six[29]) 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 geringer. Zwar wird dem Entwickler alles geboten um ein Objekt zu ersetzen, jedoch aber auch nicht mehr. Zusätzlich bietet stubble mit seinen vorgefertigten Stubs eine gute schnelle Möglichkeit den stdout zu ersetzen oder ein Objekt zu überwachen. Um jedoch selbst ein Objekt zu auszutauschen, wird vergleichsweise viel Code benötigt, da nicht nur der Rückgabewert neu gesetzt wird, sondern die Methode neu geschrieben werden muss. Dies kann allerdings bei komplexeren Anforderungen wiederum zum Vorteil werden. Durch die Annotationen ist der Code, der geschrieben wird, sehr übersichtlich und gut gegliedert.

3.2.2.2 mocktest

mocktest ist ein mocking Tool, das von rspec[30] inspiriert wurde. Dabei soll mocktest kein Port von rspec[30] sein, sondern eine kleine leichtere Version in Python (Cuthbertson, 2018). Da sich mocktest derzeit in der Version 0.7.3 (Stand 19. Juli 2019) befindet, sind noch nicht alle Features vollkommen entwickelt und ausgereift. Dennoch lässt sich das Tool bereits produktiv einsetzen, da die Basisfunktionalität aus rspec[30] bereits implementiert wurde (Cuthbertson, 2018).

Das Hauptfeature von mocktest ist die Testisolation. Diese verhindert, dass Mockobjekte außerhalb eines Testfalles weiterhin Veränderungen vornehmen und so andere Tests beein-

[29] https://github.com/benjaminp/six
[30] http://rspec.info/

flussen. Demnach werden Tests immer innerhalb einer mocktest.MockTransaction ausgeführt, sofern diese einen oder mehrere Mocks nutzen.
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 ist es nebensächlich, ob es sich hierbei um ein globales oder lokales Objekt handelt. Des Weiteren lassen sich auch Stubs nutzen, mit deren Hilfe Funktionen und Methoden präpariert werden können. Wie auch beim Überprüfen des Aufrufs ist es ebenfalls irrelevant, 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 gesamte Funktionalität von unittest geerbt wird.
mocktest.TestCase führt dabei in seiner setUp() und tearDown() Methode Code aus, der benötigt wird, um mocktest nutzen zu können. Zusätzlich wird mit mocktest unittest um eine assert Methode erweitert. assertRaises() verfügt über eine verbesserte Funktionalität für die Überprüfung auf Exceptions. Ist es jedoch erwünscht unittest nicht zu verwenden, so ist es möglich mocktest.MockTransaction als Kontextmanager mit with MockTransaction: zu nutzen. MockTransaction.__enter__() und MockTransaction.__exit__() sind die Methoden, die in setUp() und tearDown() automatisch aufgerufen werden, weshalb es möglich ist, mocktest mit jedem Test-runner zu verwenden, der Fixtures unterstützt.

Für die Überprüfung von Mocks kann entweder when(obj) oder expect(obj) verwendet werden. Der Unterschied hier ist lediglich, dass when(obj) nicht überprüft, ob eine Methode aufgerufen wurde oder nicht. expect(obj) hingegen überprüft ob die Methode aufgerufen wurde. So würde beispielsweise expect(os).system einen Fehler werfen, wenn os.system nicht aufgerufen wurde. Bei when(os).system wäre dies nicht der Fall. Ist es zu einem späteren Zeitpunk gewünscht zu überprüfen, wie oft und mit welchen Parametern ein Objekt aufgerufen wurde, ist es möglich received_calls ab zu prüfen. Diese Variable ist eine Liste von allen getätigten Aufrufen und ihren Parametern.

Folgende Features werden von mocktest unterstützt: Rückgabewert Variation der Stubs mit .and_return(erg1, erg2,ergX) oder alternativ .then_return(), Festlegen der erwarteten Aufrufe einer Funktion/Methode mit .at_least(n), .at_most(n), .between(a, b) und .exact(n),

Stuben von Methoden/Funktionen mit einer anderen Funktion/Methode mit Hilfe von .then_call(ersatz_func), Setzen von Klassenvariablen mit .with_children(**kwargs) und das Setzen von neuen Klassenmethoden mit Hilfe von .with_method(**kwargs) (Cuthbertson, 2018).

Diese Features wurden anhand des in Listing 2 definierten Codes, mit mocktest getestet (vgl. Listing 19). Bei der Implementierung der Tests gab es keine Komplikationen mit dem Tool. Da mocktest keine setUp() Methode benötigt um zu funktionieren, können hier ein paar Zeilen Code gespart werden. Durch expect() und when() ist es möglich jedes beliebige Objekt zu nehmen und zu ersetzen. Da die Implementierung keine Komplikationen aufwies wird hier auf eine weitere Analyse verzichtet.

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, welche die Signatur eines anderen Objektes haben. Auch globale Objekte lassen sich für den Zeitraum eines Tests ersetzen. Dadurch ist alles an Funktionalität geboten, was ein mock-testing Tool bieten muss. Dabei wird keine weitere Abhängigkeit zu mocktest selbst benötigt.
Bezüglich der Effizienz kann mocktest problemlos eingesetzt werden, da quasi kein Vorwissen von Nöten ist um Objekte erfolgreich zu mocken. Die Vorarbeit, die geleistet werden muss, hält sich je nach Anwendungsfall in Grenzen oder ist so gering, dass sie kein Hindernis darstellt. Wird die von mocktest zur Verfügung gestellte Klasse mocktest.TestCase für einen Testfall genutzt, so ist bereits alles fertig und direkt einsetzbar. Selbstverständlich kann mocktest.TestCase je nach Bedürfnis um Funktionalität erweitert werden, jedoch ist dies als unkompliziert zu betrachten.
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, sich wie ein Satz lesen. Dies ist zum einen durch die verschiedenen Aliase möglich, wie zum Beispiel .once() welches .exact(1) ausführt, als auch mit .then_return(). Abseits der eigentlichen Funktionalität, die mit mocktest genutzt wird, fällt kein Code an, der geschrieben werden muss um Tests ausführbar zu machen.

3.2.2.3 flexmock

flexmock ist ein weiteres Tool, dass von der Ruby Community inspiriert wurde. Dabei

diente das gleichnamige Tool flexmock[31] 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.“[32] (Kabrda und Sheremetyev, 2019). Die Features von flexmock sind denen von mocktest sehr ähnlich. So bildet flexmock seine Tests, Stubs und Mocks wie ausgesprochen dar.

flexmock bietet dem Nutzer einiges an Funktionalität, auch wenn es sich erst in der Version 0.10.4 (Stand 19. Juli 2019) befindet. So lässt sich bereits ein Stub für Klassen, Module und Objekte mithilfe von flexmock() einrichten. Damit ist es möglich alles Notwendige für ein erfolgreiches Durchlaufen der Tests zu erstellen.
Auch Mocking ist kein Problem mit flexmock. Mit der gleichen Funktion, mit der Stubs erstellt werden, lassen sich Mocks erstellen. Dadurch ist es möglich, einen Stub als einen Mock und umgekehrt zu verwenden. Daraus resultierend besteht für den Entwickler die Möglichkeit, zu überprüfen wie oft etwas aufgerufen wurde, welche Parameter verwendet wurden, welcher Rückgabe Wert zu erwarten ist und/oder welche Exception zu geworfen werden soll.
Zusätzlich ist es möglich, originale Funktionen eines Objektes zu mocken. Dadurch enthält der Entwickler die Möglichkeit zu überprüfen, ob eine unveränderte Funktion oder Methode bestimmten Anforderungen entspricht. Das Interface dafür ist das Selbe, wie bei einem Stub. flexmock bietet die Möglichkeit einen leeren Mock zu erstellen und diesem eine beliebige Methoden hinzu zu fügen. Dies ist jedoch nicht bei Objekten möglich, deren Methoden bereits fest stehen. Deswegen kann eine neue Methode zu einem gemockten Objekt nicht hinzugefügt werden.

Ein besonders interessantes Feature ist die Überprüfung 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, das als ersten Wert einen int hat, als zweiten einen String und als drittes None. Dabei kann selbstverständlich auf jede beliebige Instanz überprüft werden, so auch auf eigene Klassen. Dies ist allerdings nur bei .should_call() und bei .should_receive() nicht möglich.
Das letzte in der Dokumentation erwähnte Feature ist die Ersetzung von Klassen noch vor ihrer Instanziierung (Kabrda und Sheremetyev, 2019). Dabei gibt es mehrere Ansätze,

[31] https://github.com/jimweirich/flexmock
[32] Übersetzt aus dem Englischen

die verschiedene Ausgänge haben. Der erste Ansatz ist auf Modul Level. Dabei wird im Modul die Klasse mit einem Objekt, das entweder ein flexmock oder ein beliebig anderes sein kann, ersetzt. Nachteil dieser Methode ist, dass durch das Ersetzen der Klasse durch eine Funktion, Probleme herbeigeführt werden. Um dem entgegen zu wirken, kann alternativ .new_instance(obj) verwendet werden, welches beim Erstellen das Objekt zurück gibt, welches .new_instnce() übergeben wurde. Ein weiterer Lösungsweg könnte sein, die __new__() Methode der Klasse mit .should_receive('__new__').and_return(obj) zu ersetzen, was im Endeffekt das gleiche ist, nur verständlich ausgeschrieben.

Der Code zu Listing 2 wurde mit Hilfe des Codes von flexmock getestet. Dabei ergaben sich die eben aufgelisteten Eigenschaften (vgl. Listing 20). flexmock überzeugt dabei mit seiner Einfachheit des Aufsetzens. In der setUp() wird global im Modul my_package.my_mock_module die Klasse NotMocked ersetzt. Dadurch besteht die Möglich in jedem Testfall eine Annahme zu tätigen, die auf alle Objekte der Klasse NotMocked zutreffen. Der Aufwand diesen Code zu schreiben war demnach sehr gering. Lediglich das Erstellen einer Methode, die im originalen Objekt nicht existiert, war unmöglich. Aus diesem Grund ist der Code in test_call_helper_help() nicht korrekt und wirft eine Exception. Diese kann in Listing 21 eingesehen werden. Das Problem wurde mit Hilfe des Erstellers des Tools versucht zu lösen, jedoch konnte bis zur Abgabe dieser Arbeit keine funktionierende Lösung erarbeitet werden.

Die Anwendbarkeit von flexmock ist ausgezeichnet, da es zunächst keine weiteren Abhängigkeiten, außer sich selbst, installiert und mit allen Test-runnern kompatibel ist. Im Bezug auf TDD ist flexmock sehr zu empfehlen. Dem Entwickler werden zahlreiche Funktionalität geboten, Stubs oder Mocks zu erstellen und zu verwenden. Dabei lässt sich von der Aufrufanzahl bis zu den verwendeten Parametern, alles überprüfen. Lediglich das Setzen einer nicht implementieren Methode war mit flexmock nicht möglich.
Im Aspekt Effizienz ist flexmock ein Parade-Beispiel für den benötigten Aufwand, Tests zu implementieren. Der Entwickler muss selbst wenig von mocking, verstehen um die Funktionalität von flexmock nutzen zu können, da sich der Code wie eine Aussage liest. Das Gleiche gilt natürlich auch für die Stubs.
Genauso ist die Komplexität von flexmock ausgezeichnet. flexmock bieten für den Entwickler zahlreiche sinnvolle Aspekte und darüber hinaus ein Interface, welches das Schreiben von übersichtlichen Codes ermöglicht. Dies liegt vor allem an der deskriptiven Programmie-

rung, die von flexmock geboten wird. Auch die Unübersichtlichkeit des Codes hält sich dabei in Grenzen.

3.2.2.4 doublex

Mit 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 Stub oder ein Spy(Spion), der eine Klasse oder ein Objekt imitiert (Alises, 2014).

doublex bietet drei verschiedene Interfaces (vier zählt man die Unterklasse von Spy dazu), Stub, Spy (ProxySpy) und Mock. Da die Dokumentation hier sehr schöne und treffende Beschreibungen nimmt, werden diese nun zitiert. „Stubs sagen dir, was du hören willst.“[33] [34]. „Spies erinnern sich an alles was Ihnen passiert.“[33] [34]. „Proxy spies leiten Aufrufe an ihre originale Instanz weiter.“[33] [34]. „Mock erzwingt das vordefinierte Skript.“[33] [34].
Mit diesen Beschreibungen der einzelnen Interfaces kann ein Entwickler bereits entscheiden, welches der Objekte relevant für die zu erledigende Arbeit ist, ohne den Code kennen zu müssen. Um dennoch einen erweiterten Einblick zu schaffen, werden die einzelnen Funktionen nun genauer beschrieben.
Wichtig hierbei ist zu beachten, dass doublex lediglich doubles, also Duplikate anbietet, die das gleiche Interface wie eine Klasse haben, jedoch keine Instanz dieser Klasse sind. Dadurch bleiben manche Möglichkeiten, wie zum Beispiel das Nutzen einer implementierten Methode aus der original Klasse, dem Entwickler verwehrt. Die Duplikate sollen als Ersatz für Objekte gelten, die in einer bestimmten Umgebung ausgeführt werden müssen oder Änderungen vornehmen, die während eines Tests unerwünscht sind.
Das Stub Interface ist ein Stub mit dem es dem Entwickler möglich ist, Rückgabewerte von Funktionen zu setzen oder ähnliche Aktionen zu definieren. Dabei ist es dem Entwickler möglich, zwischen verschiedenen Parametern zu unterscheiden oder gar auf alle Aufrufe zu prüfen.
Das Interface bietet die Möglichkeit Parameter von einem Objekt zu modifizieren um den Ausgang von Methoden zu verändern. Hierbei wird zwischen einem Stub und einem „free“ Stub unterschieden. 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 abändern kann, welche die originale

[33] Übersetzt aus dem Englischen
[34] Alises, 2014

Klasse auch besitzt. Der „free“ Stub hingegen ist in dieser Hinsicht ungebunden und kann alles sein, was der Entwickler programmiert.
Als zweites Interface wird in der Dokumentation das Spy Interface erwähnt (Alises, 2014). 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. Mit Hilfe des Überprüfen der Parameter und mit Hilfe des exakten Wertes ist es möglich, einer Regex oder mit dem von doublex definierten Schlüsselwort ANY_ARG zu erstellen, welches, wie der Name sagt, jedes Argument validiert.
Zusätzlich gibt es einen „free“ Spy, welcher von keiner Klasse abhängig ist. Hierdurch ist es möglich, jede beliebige Methode auf diesem aus zu führen, ohne dass dies zu einer Exception führen würde.
Zusätzlich zum Spy und „free“ 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.
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 ist. Die Reihenfolge spielt dabei eine wichtige Rolle, außer es wird any_order_verify() genutzt. Das Mockobjekt lässt sich, 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.
Zusammenfassend lässt sich sagen, dass 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 verfügbar, welches mit beliebigen Methoden aufgerufen werden kann ohne Fehler zu werfen. Abschließend lässt sich sagen, dass jedes Interface, die Möglichkeit bietet, stubing zu betreiben, wobei nur das Stubinterface sonst keine weitere Funktionalität bietet.

Die Features von doublex wurden auf den in Listing 2 definierten Code angewandt (vgl. Listing 22). 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 ist es fraglich, ob dieser Code wirklich etwas bringt, da die getesteten Objekte einfach nur zurück geben, was erwartet wird und dabei nicht einmal das originale Objekt sind.

Bei dem 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. Stadtessen wird eine Exception geworfen, welche in einem Kommentar in der Testmethode festgehalten wurde.
Das Testen, ob Helper.help() aufgerufen wurde, war teilweise erfolgreich, da es Dank des Duplikates ein Objekt mit dem richtigen Interface bekommen hat. Es konnte jedoch nicht vortäuschen, die gewünschte Klasse zu sein. Auch hier wurde der Fehler in einem Kommentar festgehalten.
Zuletzt sollte getestet werden, ob das os Modul ausgetauscht werden kann. Dies ist mit doublex jedoch nicht möglich. Der Fehler dazu wurde in einem Kommentar innerhalb der Funktion festgehalten, sowie eine mögliche, wenn auch umständliche Lösung. Diese würde das os Objekt duplizieren und der Methode als Parameter übergeben, was in der Praxis nicht gemacht werden sollte, da dazu das Interface der Methode geändert werden müsste.

doublex bietet dem Entwickler viel, aber nicht alles Notwendige für effizientes TDD. Wie der Name des Tools verrät, 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 Signatur des Aufrufs wird überprüft. Deswegen würde stub.reuturn_input() eine Exception werfen.
Zwar lässt sich das Standardverhalten von doublex verändern, jedoch kann nur ein fester Wert gesetzt werden, der für alle nicht ersetzten Methoden gilt. Globale Objekte oder Module lassen sich ebenso nicht ersetzen. Hinzu kommt, dass das Tool drei Abhängigkeiten benötigt und selbst noch nicht in stabilem Zustand ist. Während der Analyse sind zwei DeprecationWarnings aufgetaucht, wovon eine bereits mit Python 3.0 ausgelaufen ist. Diese weisen auf eine nicht sehr aktive Entwicklung des Tools hin.
Selbstverständlich wurden die Fehler auf GitHub gemeldet, sodass der Entwickler sich in Zukunft darum kümmern kann (Stand 23. April 2019). Jedoch wurden diese bis zum Abschluss dieser Arbeit nicht gefixed (Stand 19. Juli 2019).
Auch die Effizienz ist nicht optimal. Dadurch, dass das Duplikat nicht die originalen Methoden aufruft, muss jede Methode mit einem Returnwert 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 original Implementierung ab und zu genutzt werden soll. Hat man allerdings eine externe unbeeinflussbare Klasse kann dieses Tool durchaus nützlich werden.

An Komplexität fehlt es doublex ein wenig. So kann der Stub 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 Implementierung von echten Objekten. So fungiert der ProxySpy zwar als Spy auf einem Objekt, jedoch kann er nur verzeichnen, welche Methode auf ihm aufgerufen wurde.
spy.call_other() würde nicht obj.other() registrieren, wodurch es unmöglich ist zu überprüfen, ob other() nun aufgerufen wurde oder nicht. Das gleiche gilt für Mock. Lediglich Methoden, die auf dem Mock-Objekt aufgerufen werden, werden registriert. Aus diesem Grund ist es auch hier nicht optimal nutzbar für TDD.

3.2.3 Fuzz-testing Tools

Zusätzlich zu den hier behandelten Test runnern und den mocking Tools werden 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 oder einem Typ entsprechen.
Fuzz-testing funktioniert dabei anders als die gewohnten Methoden des Testens. Normalerweise werden im Test Daten präpariert, die im Test ausgeführt und danach validiert werden, sodass 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 schlussendlich validiert (MacIver, 2019).
Der Unterschied ist, dass der Entwickler sich selbst keine 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.
Schlägt ein Test mit Daten, die per Spezifikation richtig sind, fehl, so wird der Wert der Daten für später gespeichert und, sofern möglich, versucht mit ähnlichen Daten dieser Fehler zu erreichen. Dadurch hat ein Entwickler am Ende der Tests Daten, die fehlgeschlagen sind. Mit diesen kann er 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 bei der Änderung 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 wird auch „property based testing“, also Eigenschaftsbasierte Tests genannt. In dieser Arbeit, wird allerdings weiterhin der englische Begriff Fuzztesting genutzt.

3.2.3.1 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, welches Open Source ist, demnach ist hypothesis das einzige Tool welches zum Fuzz-testing mit Python derzeit genutzt werden kann (Python 3.7).
Bevor angefangen werden kann, mit hypothesis zu testen, ist es notwendig zu verstehen, wie hypothesis funktioniert. Sind Daten gewünscht, die einer Spezifikation entsprechen, so wird hypothesis. strategies benötigt. In hypothesis sind Strategien Spezifikationen für die Generierung von Daten. Wäre eine Spezifikation also, dass die Daten Integer sind, so würde hypothesis. strategies . integers () verwendet werden. Diese Strategie würde verschiedene Daten die Integer sind ausgeben. Sind Werte zwischen A und B erwünscht, so wäre die Strategie . integers (min_value=A, max_value=B). Dies kann mit beliebig vielen Daten Typen und Spezifikationen durchgeführt werden. Dabei verfügt hypothesis über so viele verschiedene Spezifikationen, dass es schwer sein wird, alle zu kennen. Die Basics lassen sich allerdings in der Dokumentation [35] finden.
Ist die Spezifikation bekannt und die Strategie gefunden, werden Tests mithilfe einer Annotation der Test Funktion übergeben. Diese Annotation nennt sich hypothesis.given(). Ein kleines Beispiel dazu kann in Listing 23 gefunden werden. .given() unterstützt dabei die ∗args als auch die ∗∗kwargs Übergabe der Strategien.
Ist ein Fehler gefunden worden und es ist erwünscht, diesen für alle weiteren Tests zu testen, so kann mit hypothesis.example() der Test annotieren werden und die Werte, bei denen der Test fehl geschlagen ist, übergeben. Alternativ übernimmt dies hypothesis durch das Speichern des Wertes im Cache. Allerdings kann dieser verloren gehen oder durch

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

Umbenennen einer Funktion invalide werden. Um zu verhindern, dass der Cache von Entwickler zu Entwickler unterschiedlich ist, lässt sich der Cache in der VCS einbinden. Dazu sollte alles in .hypothesis/examples/ hinzu gefügt werden. Alle anderen Ordner in .hypothesis sind nicht dafür ausgelegt in das VCS integriert zu werden, da sie zu Mergekonflikten führen können.
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.
Ist es erwünscht ein paar Statistiken darüber zu erhalten, was hypothesis gemacht hat und pytest wird als Test runner verwendet, so kann ist es möglich mit −−hypothesis−show−statistics, sich die Statistiken zu den Tests anzuschauen. Dabei werden folgende Informationen zu jeder Test Funktion gezeigt: Die 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, weshalb der Test beendet wurde sowie 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. Bestehen etwas Spezifischere Anforderungen, so muss etwas tiefer in die Materie eingestiegen werden. Angenommen, eine Funktion soll mit einem Integer getestet werden, der durch sich selbst teilbar sein muss. 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 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[36] eingesehen werden.
Auch wenn die Standard Bibliothek von hypothesis bereits viele Strategien enthält, so gibt es dennoch einiges, was mithilfe von Erweiterungen dazu gewonnen werden kann. So werden von hypothesis selbst First-party Erweiterungen zur Verfügung gestellt. Dies dient vor allem dazu die Abhängigkeiten von der Standard Bibliothek so gering wie möglich zu halten (eine zusätzliche Abhängigkeit) und zum anderen um Kompatibilitätsprobleme zu verhindern. Aus diesem Grund können diese zusätzlichen Erweiterungen extra installiert werden, dazu finden sich in der Dokumentation drei Seiten, eine Generelle[37], eine für Django[38] und eine für Wissenschaftliche Module[39]. Zusätzlich zu den First-party Erweiterungen, gibt es auch noch Erweiterungen der Community, eine kleine Liste dazu kann in der Dokumentation[40] gefunden werden oder durch das Explizite suchen einer Strategie im Internet.
Ein kleines Beispiel einer Anwendung von hypothesis war bereits in Listing 23 zu sehen, für weitere Beispiele kann in der Dokumentation unter „Quick start guide“[41] und unter „Some more examples“[42] nachgesehen werden.
Da hypothesis das derzeit einzige aktiv entwickelte Open Source Tool ist, müssen Entwickler, die fuzz-testing betreiben wollen, auf dieses Tool zurück greifen. Für das Testen von Funktionen und Methoden reicht dies aus. Selbstverständlich kommt es hier stets auf den Anwendungsfall an, denn nicht jede Funktion oder Methode lässt sich mit zufälligen Werten wirklich effektiv testen. Jedoch bietet hypothesis dort, wo es sinnvoll ist, alles um ausgiebig zu testen. Sollten die in der Standard Bibliothek enthaltenen Strategien nicht reichen, so bietet hypothesis mit seinen First-party Erweiterungen, Erweiterungen die zu 100% mit hypothesis funktionieren. Dadurch ist die Standardbibliothek besonders schlank. hypothesis selbst lässt den Entwickler sehr schnell einsteigen, da nicht sonderlich viel Vorarbeit notwendig ist um hypothesis an zu wenden. Allerdings kommt mit den steigenden Anforderungen an die Daten auch die steigende Vorarbeit, die nötig ist um fuzz-testing-testing zu betreiben.

[36] https://hypothesis.readthedocs.io/en/latest/data.html
[37] https://hypothesis.readthedocs.io/en/latest/extras.html
[38] https://hypothesis.readthedocs.io/en/latest/django.html
[39] https://hypothesis.readthedocs.io/en/latest/numpy.html
[40] https://hypothesis.readthedocs.io/en/latest/strategies.html
[41] https://hypothesis.readthedocs.io/en/latest/quickstart.html
[42] https://hypothesis.readthedocs.io/en/latest/examples.html

Die Komplexität von hypothesis ist allerdings sehr hoch. So viel wie hypothesis bietet, so komplex kann es sein, dieses an zu wenden. Sind spezielle Anforderungen abseits der Standardwerte, wie int oder str benötigt, ist mehr Arbeit und Verständnis notwendig. Dies kann dazu führen, dass Entwickler etwas Zeit investieren müssen, um die Spezifikationen auf eine Strategie an zu wenden. Je nach Komplexität der erforderlichen Daten ist es also leicht bis schwer hypothesis zu verwenden. Bei komplexeren Anforderungen kann es dazu führen, dass der geschriebene Code unübersichtlich wird, wenn er nicht richtig dokumentiert wurde.
Auch wenn hypothesis als Erweiterung in dieser Arbeit behandelt wird, so bietet es Erweiterungen für sich selbst. Dies liegt daran, dass den Entwicklern nicht standardmäßig alle Funktionalität an die Hand gibt, die vermutlich gar nicht verwendet wird. Und auch wenn hypothesis über sehr viele Strategien verfügt, so hat die Community dennoch weitere Strategien entwickelt die zusätzlich genutzt werden können.

3.3 Vergleich der Tools

Die in Kapitel 3.1 und 3.2 analysierten Tools, werden anhand der Ergebnisse aus der Anwendung des jeweiligen Tools in diesem Kapitel verglichen. Der Vergleich wird in unit- und mock Tools gegliedert. Die fuzz-testing Tools werden hier nicht verglichen, da nur ein Tool analysiert wurde. Am Ende dieses Kapitels wird für die jeweilige Kategorie (unit- und mock-testing Tools) eine Empfehlung ausgesprochen, die anhand der Ergebnisse des Vergleichs festgelegt wird. Anhand dieser Empfehlungen werden in Kapitel 3.4 verschiedene Möglichkeiten aufgezeigt, die Tools untereinander zu kombinieren um mehr Funktionalität für eine mögliche Anwendung zu bekommen.

3.3.1 Unit-testing Tools

Die unit-testing Tools belaufen sich auf drei Tools, unittest, doctest und pytest. Die ersten beiden Tools sind beide in der STDLIB, während pytest mit sechs weiteren Paketen betrieben werden muss. Während unittest und pytest sich sehr ähnlich sind, ist doctest vollkommen anders im Bezug auf Anwendung und gegebener Funktionalität. doctest unterstützt keine Testfälle oder Testumgebungen, stattdessen wird im docstring der jeweiligen Funktion oder Methode der Test geschrieben. Zwar ist es möglich diesen in eine externe Datei aus zu lagern um so einen zentralen Punkt zu schaffen, an dem sich alle Tests befinden (Testumgebung/ Testfälle), jedoch ist hier die Datei das Konstrukt, dass die Gliederung ermöglicht. Sowohl unittest als auch pytest bieten hier die Möglichkeiten,

Tests in Testumgebungen (Klassen) und Testfällen (Funktionen/ Methoden) zu gliedern. pytest bietet zusätzlich auch noch die Möglichkeit diese Tests zu markieren um eine noch feinere Gliederung zu schaffen oder um zwischen Testumgebungen und Testfälle zu kombinieren. Im Fall der Gliederung und Verwaltung von Tests sind hier sowohl pytest als auch unittest, doctest überlegen. Lässt man die unittest Integration für doctest außen vor, so ist unittest in diesem Fall der Gewinner. Wird allerdings die Anwendung der unittest Integration von doctest in Betracht gezogen, so sind unittest und doctest auf einem Level, was Gliederung und Verwaltung von Tests betrifft.
Ein Feature, das eine Gliederung voraussetzt, sind die Fixtures. Da doctest keine Gliederung bietet, sind Fixtures für dieses Tool nicht als Feature integriert. Es ist zwar möglich Fixtures manuell zu schreiben, jedoch ist der Aufwand, der dafür betrieben werden muss, zu hoch, als dass es sich rentieren würde. Die unittest Integration ermöglicht es zwar Fixtures zu nutzen, jedoch wird diese Integration außen vor gelassen. unittest und pytest hingegen unterstützen Testfixtures mit ihren setUp() und tearDown() Methoden. Beide Tools bieten die Möglichkeit die Fixtures sowohl für jedem Testfall, jede Testumgebung oder jedes Modul aus zu führen. pytest bietet zusätzlich noch weitere Funktionalität, sowie fertige Fixtures die dem Entwickler bei der Nutzung zur Verfügung stehen. Diese wurden in Kapitel 3.2.1 beschrieben und sind in der Dokumentation[43] zu finden. Im Bezug auf Testfixtures ist also pytest das Tool mit den meisten Features, gefolgt von unittest. doctest scheidet bei dieser Kategorie vollkommen aus, da die Funktionalität gänzlich fehlt.
Als nächstes integriertes Feature der unit-testing Tools sind Mocks und Stubs zu betrachten. Um für TDD in Frage zu kommen sollte ein unit-testing Tool Stubs und Mocks unterstützen. Eine ausführliche Erklärung dazu ist in Kapitel 3 zu finden. Wie in Kapitel 3.1.2 aus der Analyse von doctest hervorgeht, gibt es weder die Möglichkeit einen Mock noch einen Stub auf zu setzen, weshalb doctest in dieser Kategorie als unbrauchbar gesehen werden kann. unittest als Modul selbst, besitzt keine mocking Funktionalität, jedoch ist es möglich, dass Submodul unittest.mock dafür zu verwenden. unittest.mock ist zwar eine Erweiterung in Form eines Submoduls, weshalb es zu unittest dazu gehört und in diesem Vergleich beachtet wird. Damit unterstützt unittest sowohl das Mocken, als auch das Stuben. pytest hingegen verfügt in seiner Basisausführung lediglich über die Möglichkeit des Stubens, nicht jedoch über die Möglichkeit ein Objekt zu mocken. In diesem Falls ist also unittest der klare Gewinner, gefolgt von pytest.

[43] https://docs.pytest.org/en/latest/fixture.html

Im Bezug auf die Effizienz der unit-testing Tools lässt sich folgendes feststellen. doctest ist zwar für jeden Entwickler einfach und verständlich anwendbar, jedoch ist die Vorarbeit um Tests auf zu setzen im Vergleich zu unittest und pytest höher. Sowohl unittest als auch pytest benötigen das gleiche Maß an Vorarbeit um Tests auf zu setzen, wobei der Aufwand das jeweilige Tool ein zu setzen bei unittest etwas höher ist, da es so viele verschiedene assert Methoden zu merken gibt. Die Auswertung der Tests erfolgt bei allen drei unit-testing Tools auf dem Terminal. doctest und unittest haben beide einen Output der voneinander fast nicht zu trennen ist. Beide Tools haben den Nachteil gegenüber pytest, dass ihr Output nicht farbig, sondern monochrom ist. Aus diesem Grund ist pytest, unittest und doctest im Bezug auf effiziente Anwendung vor zu ziehen, wobei unittest noch etwas besser als doctest ist.

Der Aspekt der Komplexität ist in zwei Unterkategorien gegliedert. So wird zum Einen die Fülle an Funktionen, die geboten werden und Komplexität liefern, und andererseits die Menge an Code, die zu schreiben ist und die Anwendung und Wartung Komplexer gestaltet, betrachtet. Im Falle der ersten Kategorie, erweist sich doctest als nicht all zu komplex. Fast die Gesamte gebotene Funktionalität bezieht sich auf die Ausführung, jedoch nicht das Schreiben der Tests, welches alleine durch die Fähigkeit Python zu schreiben bestimmt wird. unittest und pytest hingegen bieten dem Entwickler einiges an Funktionalität um Tests zu schreiben und aus zu führen. Auch hier kann unittest nur mithalten, wenn das Submodul unittest.mock mit in die Wertung aufgenommen wird, da dieses weitere Funktionalität bietet. unittest und pytest haben viele gemeinsame Features, wovon lediglich das mocking in pytest nicht ohne Erweiterungen verfügbar ist. Ansonsten bietet pytest zusätzliche Funktionalität um Tests feiner ein zu stellen oder um Randbedingungen ab zu decken. In dieser Kategorie ist pytest also etwas besser als unittest. Beide Tools sind doctest bei Weitem im Bezug auf die gebotenen Funktionalitäten überlegen.
In der zweiten Kategorie, der Komplexität im Bezug auf Codelesbarkeit und -umfang, erweisen sich unittest und pytest als gleichauf. Beide Tools haben einen ähnlichen Aufbau, mit den der Code geschrieben wird (vgl. Listing 6 und 14). Die Lesbarkeit ist bei beiden Tools gleich gut und die Menge an Code, die benötigt wird, gleicht sich auch. Hier ist es dem Entwickler überlassen zu entscheiden, welches Tool den schöneren Code schreibt. doctest hingegen produziert viel Code, der mit zunehmender Funktionalität an Lesbarkeit verliert.Im Bezug auf die Komplexität der Tools ist pytest also knapp vor unittest, gefolgt

von doctest.

Als letztes ist die Erweiterbarkeit der unit-testing Tools zu vergleichen. Beginnend mit doctest, lässt sich sagen, dass es zwei sehr gute Erweiterungen gibt, die das Tool erheblich verbessern. Diese Erweiterungen sind die Integration in unittest und pytest. Erstere wird von der Python Sprache selbst geboten und ist in unittest verfügbar, zweite wird von pytest geboten und ist auch im Basis Paket enthalten. Abgesehen von diesen zwei gibt es noch weitere Integrationen in andere Toolsets, jedoch wurden diese hier wie in Kapitel 2 beschrieben, nicht behandelt. Das Modul unittest verfügt über einige Erweiterungen welche neue Funktionalität hinzu fügen oder alte Funktionalität verbessern. Das Tool mit den meisten Erweiterungen ist pytest, welches derzeit 683 (Stand: 19. Juli 2019[44]) Erweiterungen bietet, wovon die meisten sehr nützliche Funktionalität bieten.

Im Vergleich der unit-testing Tools ergibt sich also ein klarer Sieger: pytest. Das Tool ist in fast allen Analysierten Aspekten die beste Wahl für den Einsatz mit TDD. Jedoch bietet unittest für Entwickler, die keine externen Abhängigkeiten möchten eine Alternative, die durchaus mit pytest mithalten kann.

3.3.2 Mock-testing Tools

Die analysierten mock-testing Tools, sind stubble, mocktest, flexmock und doublex. Alle vier bieten verschiedene Möglichkeiten Mocks und/oder Stubs ein zu setzen. Da es sich hier um Tools handelt, deren Funktionalität sich um das Mocken von Objekten dreht, wird der erste analysierte Aspekt, die Anwendbarkeit, auf die gebotene Auswahl an Funktionalität im Bezug auf Mocken und Stuben betrachtet.
Das einzige Tool, das bei der Anwendung auf den zu testenden Code keine Probleme mit sich brachte, ist mocktest. Das Tool bietet im Aspekt der Anwendbarkeit sämtliche Features um effektiv Mocks und Stubs ein zu setzen. mocktest bietet seinem Anwender eine deskriptive Schreibweise, mit der es problemlos möglich ist Tests zu schreiben. Des Weiteren wird eine Testisolation geboten, mit deren Hilfe es möglich ist, Tests besser voneinander ab zu schotten. Zusätzlich benötigt das Tool keine weiteren Abhängigkeiten zu sich selbst. Direkt danach wäre stubble das nächstbeste Tool im Aspekt der Anwendbarkeit. So bietet das Tool zwar alles um es für TDD zu verwenden, jedoch ist das Stuben einer nicht existierenden Methode nur unter Einschränkungen möglich. Die Hauptfeatures von stubble sind dabei die vorgefertigten Stubs, welche nützliche Features für den Entwickler

[44]http://plugincompat.herokuapp.com/

bieten. Um einen Stub zu erstellen muss die zu stubende Klasse mit stubble neu geschrieben werden. Zusätzlich benötigt stubble eine weitere Abhängigkeit zu sich selbst. An dritter Stelle ist das Tool flexmock welches, abgesehen vom ersetzen einer nicht geschriebenen Methode um TDD an zu wenden, vieles bietet (vgl. Listing 21). Wäre dieses Defizit nicht, würde das Tool im Aspekt der Anwendbarkeit sehr weit oben stehen. Flexmock vereint die Mocks und Stubs in einer Klasse, wodurch die gesamte Funktionalität mit einem Objekt geboten wird. Die Schreib weise der Tests ist hierbei, wie bei mocktest, deskriptiv. Auch die benötigten Abhängigkeiten von flexmock sprechen für das Tool, da keine weiteren zu sich selbst benötigt werden. Als letztes Tool im Aspekt der Anwendbarkeit ist doublex zu betrachten. Das Tool bietet zwar einiges an Funktionalität um zu mocken oder zu stuben, jedoch ist die Anwendung dieser Features für TDD nicht nutzbar. Es werden vier verschiedene Interfaces geboten, deren Funktionalität sich überschneidet und die jeweils ein Objekt hervorbringen, das ein Duplikat vom Zielobjekt ist. Die daraus resultierenden Anwendungszwecke bieten wenig Nutzbarkeit für TDD. Hinzu kommt, dass doublex drei weitere Abhängigkeiten benötigt und anscheinend nicht mehr aktiv weiter entwickelt wird.
Demnach ist im Aspekt der Anwendbarkeit mocktest also stubble und flexmock vor zu ziehen. doublex fällt aufgrund dessen, dass es nicht für TDD geeignet ist und derzeit nicht aktiv weiter entwickelt wird, komplett raus. Aus diesem Grund wird dieses Tool im weiteren Verlauf des Vergleichs nicht mehr berücksichtigt.

Als nächstes ist die Effizienz der jeweiligen Tools zu betrachten. Dabei spielt vor allem der benötigte Aufwand das Tool ein zu setzen eine Rolle. Nebenbei wird dabei auch noch auf die benötigte Vorarbeit geachtet, die geleistet werden muss um das Tool an zu wenden. Sowohl bei mocktest als auch bei flexmock ist der zu betreibende Aufwand sehr gering. Beide Tools haben ein ähnliches Interface mit dem deskriptiv beschrieben wird, was ein Objekt zu machen hat. Der dabei entstehende Aufwand ist sehr gering, da sich der Code so schreibt, wie ein Entwickler sich die Funktionalität vorstellt. Beide Tools benötigen die gleiche Menge an Vorarbeit und sind demnach im Bezug auf die Effizienz mit der sie angewendet werden können gleich auf. Lediglich stubble ist in diesem Aspekt etwas schlechter als die beiden anderen Tools. So ist die mocking Funktionalität ausschließlich mit einem vorgefertigten Stub möglich, wodurch dieser erst verstanden werden muss, bevor er effektiv eingesetzt werden kann. Zusätzlich muss jedes zu stubende Objekt überschrieben werden, wodurch die benötigte Vorarbeit sehr hoch ausfällt.

Demnach sollte im Aspekt der effizienten Anwendung der mock-testing Tools stubble nicht verwendet werden. Es ist dem Entwickler überlassen ob er in diesem Aspekt das Interface von mocktest oder flexmock lieber hat.

Zuletzt ist die Komplexität der mock-testing Tools zu betrachten. Dabei wird zum einen auf die gestellten Funktionalitäten abseits der Basis Funktionalität geachtet und zum anderen auf die Lesbarkeit des entstehenden Codes des Tools. Auch hier sind mocktest und flexmock auf einem Level, was die zusätzliche Funktionalität betrifft. Jedoch hat mocktest etwas mehr zusätzliche Funktionalität gegenüber flexmock geboten (vgl. Kapitel 3.2.2.2 und 3.2.2.3). stubble bietet auch in diesem Aspekt weniger als die beiden Tools. Abseits der Basisfunktionalität wird von stubble nichts weiter geboten. Im Bezug der Lesbarkeit sieht es ähnlich aus. mocktest und flexmock befinden sich auf einem Level, während stubble durch viel Code an Lesbarkeit verliert. Im Fall mocktest gegen flexmock ist es wieder dem Anwender überlassen, welches der beiden Tools lesbarer ist als das andere.
Im Bezug auf Komplexität geben sich mocktest und flexmock fast nichts, jedoch hat mocktest durch die zusätzliche Funktionalität hier die Nase vorne. stubble bleibt auch hier auf dem letzten Platz.

Im gesamten Vergleich ist mocktest durch seine Features und sein Interface an erster Stelle, wenn es um den Einsatz von mock-testing Tools geht. Der zweite Platz ist schwer zu wählen, da flexmock mit seiner fehlenden Funktionalität nicht implementierte Methoden hinter stubble liegt, das jedoch auch nur bedingt diese Anforderung erfüllt. In allen anderen Aspekten ist flexmock stubble voraus. Aus diesem Grund erhält flexmock den zweiten Platz der mock-testing Tools und stubble den dritten. Auf dem vierten wäre doublex, jedoch ist dieses Tool für die Anwendung von TDD ungeeignet, weshalb es keine Platzierung erhält.

3.4 Kombinierung von Tools

Um eine verbesserte Funktionalität zu bekommen, ist es ratsam verschiedene Tools miteinander zu kombinieren. Die in diesem Kapitel aufgezeigten Möglichkeiten zur Kombinierung ergeben sich aus den analysierten Features der jeweiligen Tools. Dabei wird zuerst geprüft ob die beiden, am besten geeigneten Tools aus unit- und mock-testing Tools miteinander verwendet werden können, um so einen Mehrwert zu erhalten.
Bei den unit-testing Tools ist doctest das am wenigsten geeignete Tool für die alleinige Anwendung mit TDD. Dennoch bietet das Tool Features, die für die Anwendung von

TDD nützlich sind. So kann doctest als testende Dokumentation gesehen werden. Manche Funktionen oder Methoden sind in ihrem Funktionsumfang so gering, dass ein ganzer Test als zu viel erscheint. Bei diesen Funktionen oder Methoden kann beispielsweise ein doctest geschrieben werden, der diese kleine Funktionalität prüft. Des-weiteren kann doctest als ergänzendes Tool für die Dokumentation des Codes gesehen werden, denn die Dokumentation ist ein wichtiger Bestandteil eines Projekts (van Rossum, Warsaw und Coghlan, 2001). doctests können dafür sorgen, dass die Dokumentation besser verstanden wird und eindeutiger ist. Aus diesen Gründen kann doctest immer als ergänzendes Tool in jede Entwicklung aufgenommen werden.
Da das Tool hypothesis alleiniger Vertreter seiner Kategorie, der fuzz-testing Tools ist, ist es das empfohlene Tool, wenn fuzz-testing benötigt wird. Da hypothesis mit jedem anderen Tool verwendet werden kann, empfiehlt es sich, dieses so oft wie es möglich ist ein zu setzen um eine bessere Testabdeckung zu erreichen.
Demnach ist es ratsam sowohl doctest als auch hypothesis immer ein zu setzen wo es möglich ist.

Grundsätzlich lässt sich ein jedes unit-testing Tool mit jedem mock-testing Tool kombinieren. Der dadurch erhaltene Mehrwert ist meist beachtlich gegenüber der Verwendung der Tools alleine. Es ist jedem Entwickler überlassen, welches Tool er mit welchem kombiniert, dennoch bietet es sich an, pytest zusammen mit mocktest zu verwenden. Wenn dazu noch doctest und hypothesis benutzt wird, erhält der Entwickler ein Toolset, das ihm alles an Funktionalität bietet, was er benötigt. Ist es erwünscht unittest statt pytest zu verwenden, so kann dies auch getan werden. Jedoch zeigt die Analyse, dass der Einsatz von pytest dem Einsatz von unittest vor zu ziehen ist.

4 Diskussion

Um für die Anwendung von TDD die passenden Tools zu finden, wurden verschiedene Tools der Programmiersprache Python analysiert und auf ihre Tauglichkeit im Bezug auf TDD geprüft. Daraus entstand eine Empfehlung für vier Tools, die gemeinsam für TDD genutzt werden können, um so optimale Funktionalität zum Schreiben von Tests, sowie eine optimalen Testabdeckung zu garantieren. Die dabei erhaltenen Ergebnisse zeigen auf, dass ein Tool alleine zwar ausreichend für die Anwendung von TDD sein kann, jedoch ist die Verwendung von mehreren Tools zusammen für den Entwickler und für das Produkt besser.

Im Rahmen dieser Arbeit wurden nur jene Tools analysiert, deren letzte Aktivität nicht weiter als ein Jahr zurück liegt. Tools, die eine Erweiterung zu einem anderen Tool darstellen, wurden nicht behandelt, da dies den Rahmen der Arbeit gesprengt hätte. Aus diesem Grund kann es sein, dass ein beliebtes oder bekanntes Tool in dieser Arbeit nicht behandelt wurde. Das bedeutet nicht, dass dieses Tool zwangsläufig nicht geeignet ist für TDD.
Im Bezug auf die Forschungsfrage, welches aktuelle Python Test-Tool das Beste ist, lässt sich Folgendes sagen: Viele Tools sind in ihrem Funktionsumfang sehr ähnlich, allerdings stellten sich pytest bei den unit-testing-, mocktest bei den mock-testing- und hypothesis bei den fuzz-testing Tools als die beste Wahl heraus. Dennoch ergab sich, dass doctest immer als ergänzendes Tool verwendet werden kann, egal welche anderen Tools verwendet werden.
Diese Arbeit hat sich jedoch lediglich mit einer begrenzten Anzahl von Tools beschäftigt und hat diese anhand eines spezifischen Beispiels getestet. Jeder Entwickler oder Software-Architekt sollte sich darüber informieren, welches Tool für den geplanten Anwendungszweck sinnvoll ist, da manche Tools vielleicht eine bessere Wahl, je nach Art der Anwendung, darstellen.

5 Fazit

Durch das Testen der verschiedenen Tools der Sprache Python, hat sich diese Arbeit mit dem Vergleich, sowie der Wertung dieser im Bezug auf TDD auseinander gesetzt. Das Ziel dabei war es, ein oder mehrere Tool(s) zu finden, welche für den allgemeinen Einsatz von Python und TDD am besten geeignet sind. Dieses Ziel wurde mithilfe der gewählten Methoden erreicht und ergab, dass vier Tools gemeinsam eine sehr stabile Basis für die Anwendung von TDD bieten. Diese vier Tools sind: pytest, mocktest, hypothesis und doctest. Die dabei verwendete Testmethode ermöglicht zwar eine hohe Validität, jedoch wäre es möglich die Reliabilität noch weiter zu steigern, indem die Testfälle mehr Funktionen abdecken. Jedoch war die gewählte Methode für diese Arbeit die Richtige und führt zu verwendbaren Ergebnissen.
Bei der Analyse der verschiedenen Tools ergab sich, dass die Anwendung von TDD komplexer ist, als zunächst erwartet. Um die Testmethode für diese Arbeit zu erstellen, musste eine „Software“ programmiert werden, welche diverse Anforderungen erfüllen muss, da sie die Funktionen eines Tools spiegeln muss. Dieser erste Schritt gleicht dem ersten Schritt der Anwendung von TDD, da die Architektur der Software entworfen werden

musste. Dies erwies sich schwieriger als zunächst erwartet, da die Methodik möglichst komplex sein sollte.
Diese Arbeit hat mir im Bezug auf TDD, Testen und Python viel Neues bei gebracht. So würde ich zwar TDD nicht unbedingt einsetzen wollen, dies beruht aber nicht darauf, dass ich es für ineffektiv halte, sondern vielmehr aus dem Grund, dass es ein sehr hoher Aufwand ist, der sich erst sehr spät bezahlbar macht. Ich selbst würde einen Ansatz nutzen, bei dem Tests neben dem Code geschrieben werden. Dies erspart viel Zeit und würde zu einer ähnlich hohen, wenn auch nicht so perfekten Testabdeckung führen.
Für mich wäre TDD nach dieser Arbeit zwar eine Methode, die ich definitiv einmal in einem professionellen Umfeld anwenden würde, jedoch aber eben auch nur dort. Meiner Meinung nach ist die Planung bei TDD das Wichtigste und sorgt dafür, dass die Entwickler auch Lust und Spaß an der Arbeit haben.

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.

Beck, K. (2002). *Test Driven Development: By Example*. Addison-Wesley Professional.

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.

Fowler, M. (2007). Mocks Aren't Stubs. Online erhältlich unter https://martinfowler.com/articles/mocksArentStubs.html; abgerufen am 21. Mai 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.

Percival, H. (2014). *Test-Driven Development with Python: Obey the Testing Goat!* O'Reilly Media, Inc.

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.

Stack Exchange Inc. (2019). Developer Survey Results 2019. Online erhältlich unter https://insights.stackoverflow.com/survey/2019; abgerufen am 10. Juni 2019.

van Rossum, G., Warsaw, B. & Coghlan, N. (2001). PEP 8 – Style Guide for Python Code | Python.org. Website. Online erhältlich unter https://github.com/python/peps/blob/master/pep-0008.txt oder https://www.python.org/dev/peps/pep-0008/; abgerufen am 26. 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.

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.

$Unterschrift:$ $Ort, Datum:$

Listingverzeichnis

Abbildungsverzeichnis

Fußnotenverzeichnis

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. 21, 30, 31

bug Ein Bug(zu deutsch Käfer) ist ein Fehler in einem Programm oder in einer Software. 4, 16

Command-line interface Eine Benutzerschnitstelle für das Terminal. 51

commit Ein commit ist die Sammlung von änderungen an Dateien, welche mithilfe eines VCS verwaltet werden. 16

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. 17, 20

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. 17, 20, 21

docstring Ein Docstring ist ein Block von Text der sich zwischen jeweils drei Anführungsstrichen befindet. Mit diesem Text wird ein Objekt in Python dokumentiert. 14, 33

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. 9, 11–15, 17–19, 22

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. 8–10, 17, 32, 33, 39

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. Je

nach implementierung kann es auch sein, dass ein Mock den Rückgabewert einer Funktion/Methode verändert (Fowler, 2007). 3, 8–10, 12, 14–19, 21–26, 29, 33–39

Pythonista Jemand, der Python als seine favorisierte Sprache verwendet. 4

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. 12, 13, 15

stub 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 (Fowler, 2007). 9, 10, 12, 17–27, 29, 34, 36, 37

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. 51

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

Listingverzeichnis

```
"""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

```
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 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 3: unittest einfaches Beispiel

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

Listing 4: 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 5: unittest einfaches Beispiel: Output misslungen

```
1  import unittest
2  import unittest.mock as mock
3
4  from my_package.my_module import (
5      Item,
6      Session
7  )
8
9  class TestMyDatabase(unittest.TestCase):
10     def setUp(self):
11         self.session = Session()
12         self.item = Item(id_=1, name='name', storage_location=7, amount=3)
13
14     def test_creation(self):
15         self.session.add(self.item)
16         self.session.commit()
17
18         self.assertIs(len(session.new), 0)
19         self.assertEqual(session.query(Item).first(), self.item)
20
21     @mock.patch('my_package.my_module.do_something_which_does_not_exist')
22     def test_external_function(self, mock_do_something_which_does_not_exist):
23         mock_do_something_which_does_not_exist.return_value = 42
24
25
26         self.assertIs(self.item.do_something(), 42)
27         mock_do_something_which_does_not_exist.assert_called_with(self.item)
28
29     def tearDown(self):
30         slef.session.close()
31
32 if __name__ == '__main__':
33     unittest.main()
```

Listing 6: unittest my_module

```
1  import random
2
3  def return_3(s=None, i=None):
4      """
5      >>> return_3(s=True)
6      '3'
7
8      >>> return_3(i=True)
9      3
10
11     >>> return_3(i=True)
12     '4'
13     """
14     if s:
15         return '3'
16     elif i:
17         return 3
18
19
20 if __name__ == '__main__':
21     import doctest
22     doctest.testmod()
```

Listing 7: doctest unterscheidet Typen

```
1  **********************************************************************
2  File "example.py", line 14, in __main__.return_3
3  Failed example:
4      return_3(i=True)
5  Expected:
6      '4'
7  Got:
8      3
9  **********************************************************************
10 1 items had failures:
11    1 of   3 in __main__.return_3
12 ***Test Failed*** 1 failures.
```

Listing 8: doctest unterscheidet Typen: Output

```
1  Trying:
2      from advanced import *
3  Expecting nothing
4  ok
5  Trying:
6      my_pow(2, 2)
7  Expecting:
8      4
9  ok
```

Listing 9: doctest verbose Output

```
1  def my_pow(a, b):
2      """
3      >>> my_pow(2, 2)
4      4
5      >>> my_pow(4, 8)
6      65536
7      """
8      result = a
9      for _ in range(1, b):
10         result = result * a
11
12     return result
13
14
15 class Item(Base):
16     """
17     Da Doctest keine Fixtures unterstuetzt muss hier der setUp Code stehen:
18     >>> Base.metadata.create_all(engine)
19     >>> item = Item(id_=1, name='name', storage_location=1, amount=1)
20     >>> session = Session()
21     >>> session.add(item)
22     >>> session.commit()
23
24
25     Check ob das item commited wurde
26     >>> len(session.new)
27     0
28
29     Check ob das item in der Datenbank ist.
30     Das item wird mit seiner __repr__() Methode representiert, da die
31     Werte bekannt sind kann ueberprueft werden ob diese uebereinstimmen.
32     >>> session.query(Item).first()
33     item<1, name, 1, 1>
34
35     do_something existiert nicht, da mit Doctest kein Mock erstellt werden
36     kann wird auf die Exception ueberprueft.
37     >>> item.do_something()
38     Traceback (most recent call last):
39         ...
40     TypeError: 'NoneType' object is not callable
41
42     Hier wird der tearDown Code ausgefuehrt
43     >>> session.close()
44     """
45     __tablename__ = 'items'
46
47     id_ = Column('id', Integer, primary_key=True)
48     name = Column(String(64))
49     storage_location = Column(Integer)
50     amount = Column(Integer)
51
52     def do_something(self):
```

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

Listing 10: 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 = Session()
15 >>> session.add(item)
16 >>> session.commit()
17
18
19 Check ob das item commited wurde
20 >>> len(session.new)
21 0
22
23 Check ob das item in der Datenbank ist.
24 Das item wird mit seiner __repr__() Methode representiert, da die
25 Werte bekannt sind kann ueberprueft werden ob diese uebereinstimmen.
26 >>> session.query(Item).first()
27 item<1, name, 1, 1>
28
29 do_something existiert nicht, da mit Doctest kein Mock erstellt werden kann
30 wird auf die Exceptioni ueberprueft.
31 >>> item.do_something()
32 Traceback (most recent call last):
33     ...
34 TypeError: 'NoneType' object is not callable
35
36 Hier wird der tearDown Code ausgefuehrt
37 >>> session.close()
```

Listing 11: doctest my_module: Textdatei

```
# Aufruf mit py.test -q -s <dateiname>
import pytest

@pytest.fixture
def setup_teardown():
    print ("setup")
    yield 1
    print ("\nteardown")

def test_setup_teardown(setup_teardown):
    assert setup_teardown == 1
```

Listing 12: pytest setUp und tearDown

```
setup
.
teardown

1 passed in 0.00 seconds
```

Listing 13: pytest setUp und tearDown Output

```
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'


```

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

Listing 14: pytest my_module

```
1 ============================= test session starts ==============================
2 platform linux -- Python 3.7.3, pytest-4.4.0, py-1.8.0, pluggy-0.9.0
3 rootdir: /this/is/my/secret/path
4 collected 2 items
5
6 pytest_my_module.py ..                                                  [100%]
7
8 =========================== 2 passed in 0.11 seconds ===========================
```

Listing 15: pytest my_module: Output

```
1 atomicwrites==1.3.0
2 attrs==19.1.0
3 more-itertools==7.0.0
4 pluggy==0.9.0
5 py==1.8.0
6 pytest==4.4.0
7 six==1.12.0
```

Listing 16: pytest Abhängigkeiten

```
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 17: 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 18: 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 19: 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 20: flexmock my_mock_module

```
1  E.......
2  ======================================================================
3  ERROR: test_call_helper_help (__main__.TestFlexmock)
4  ----------------------------------------------------------------------
5  Traceback (most recent call last):
6    File "example.py", line 40, in test_call_helper_help
7      helper_mock.should_receive('help').replace_with(lambda: True)
8    File "flexmock_/venv/lib/python3.7/site-packages/flexmock.py", line 731,
9      in should_receive
10     _ensure_object_has_named_attribute(obj, name)
11   File "flexmock_/venv/lib/python3.7/site-packages/flexmock.py", line 1150,
12     in _ensure_object_has_named_attribute
```

```
13      raise FlexmockError(exc_msg)
14 flexmock.FlexmockError: <my_package.my_mock_module.Helper object at
15      0x7f0e0c18a6a0> does not have attribute help
16
17 ----------------------------------------------------------------------
18 Ran 7 tests in 0.002s
19
20 FAILED (errors=1)
```

Listing 21: flexmock exception

```
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 22: 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 23: hypothesis @given() Beispiel