Universität Hamburg

Proseminar

Softwareentwicklung in der Wissenschaft

Prüfer: Dr. Hermann-Josef Lenhart; Prof. Dr. Thomas Ludwig
Betreuer: Michael Kuhn

Proseminar Hausarbeit

Thema: Testen

Eingereicht von: Benedikt Blume
Matr.-Nr.: 7261180

Abgabedatum: 31.08.2020

Inhaltsverzeichnis

1. Motivation

Bevor in dieser Arbeit darauf eingegangen wird was Sofwaretests sind wird zunächst auf die Relevanz von Softwaretests eingegangen.
Ein Beispiel dafür, welche Auswirkungen eine noch fehlerbehaftete Software haben kann, ist die missglückte Testfahrt eines autonomen Autos. Im Mai 2018 im US-Bundesstaat Arizona kam es zu einem Todesfall durch ein autonomes Testauto vom Mobilitätsdienst „Uber“. Eine Fußgängerin, die ihr Fahrrad über die Straße schieben wollte, wurde von dem Auto erfasst und ist verstorben. Eine Frau, die im Testauto saß, sollte bei Fehleinschätzungen oder Fehlern im Programm eingreifen, jedoch war sie zu diesem Zeitpunkt abgelenkt. Nach Auswertung des Protokolls vom Fahrzeug wurden gravierende Programmierfehler entdeckt. Es kam bei dem Unfall zu einer Aneinanderkettung von Fehleinschätzungen des Programms. [1]
Dieses Beispiel macht deutlich welche Folgen eine nicht ausführlich genug getestete Software haben kann. In den folgenden Abschnitten wird detaillierter darauf eingegangen, wodurch solche Fehler vermieden werden können.

1.1 Was ist Testen?

Beim Testen in der Softwareentwicklung wird geprüft, ob eine Software Fehler aufweist. Seit über 50 Jahren wird versucht durch verschiedene Softwareentwicklungstechnologien fehlerfreie oder wenigstens fehlerarme Software zu entwickeln. Doch selbst diese Technologien haben nicht geholfen Software fehlerfrei zu gestalten. Ausgelieferte Software weisen immer noch ein bis drei Fehler pro 1000 Anweisungen auf. Fehlerquellen finden sich in neuen aber auch in weiterentwickelten Programmen. Die Software selbst wird von Menschen geschrieben und Menschen machen Fehler. Neue Software kann bis zu mehreren Millionen von Anweisungen besitzen. Wenn die ein bis drei Fehler pro 1000 Anweisungen auf 100 Millionen Anweisungen erweitert werden ergibt das 100.00-300.000 Fehler allein in einem großen Programm. Aus diesem Grund gibt es mittlerweile ganze Teams die sich nur auf das Testen konzentrieren.
Um Fehlern vorzubeugen gibt es verschiedene Arten von Programmierstilen. Einer davon ist der Defensive Programmierstil zu dem auch der Test-First-Ansatz gehört. Bei diesem Vorgehen wird zu jeder testbaren Klasse direkt eine Testklasse geschrieben. Die Testklassen enthalten die Tests als geschriebenen Quellcode, wie z.B. Methoden, die es

ermöglichen Fehler zu finden. Es gibt jedoch noch viele weitere Möglichkeiten Fehler zu minimieren. Im nächsten Abschnitt werden zwei verschiedene Testverfahren erläutert. [2]

1.2 Statische Software-Testverfahren

Die Statischen Software-Testverfahren sind Verfahren, bei dem die Software nicht ausgeführt wird. Die Tests laufen nicht zur Laufzeit ab. Bei diesem Verfahren prüfen Menschen oder andere Programme den Quellcode der Software nach Fehlern. Der Quellcode wird unter anderem auf ausreichende Dokumentation geprüft. Die Dokumentation ist in dem Quelltext enthalten und erklärt was eine bestimmte Klasse oder Methode macht. Sie beschreibt auch welche Parameter eingegeben werden sollen und welche Werte ausgegeben werden. Die Dokumentation dient dazu den Quelltext verständlicher zu gestalten. Falls zu einem späteren Zeitpunkt der Quelltext verändert oder erweitert werden soll, wird dies durch die Dokumentation vereinfacht. Zudem wird der Quellcode nach Code Duplikaten untersucht, die nicht benötigt werden. Auch werden die Funktionen geprüft, ob Sie mit den entsprechenden Spezifikationen umgesetzt worden sind. Diese Prüfung der Software kann von verschiedener Software ausgeführt werden z.B. von dem Clang Static Analyzer. Dieses Quellcode-Analysetool findet automatisch Fehler in der Software. Der Clang Static Analyzer ist speziell für Software gemacht die in den Programmiersprachen C- und C++ geschrieben wurde. Solche Analysetools gibt es für viele andere verbreitete Programmiersprachen. Diese Tools übernehmen die oben genannten Aufgaben. Sie ähneln den Compiler-Warnungen die nützlich sind, um Codierungsfehler zu finden, gehen jedoch noch einen Schritt weiter. In den letzten Jahrzehnten haben sich diese statischen Analysen weiterentwickelt, sodass sie tiefe Fehler in der Software finden, indem sie die Semantik vom Code analysieren. Dieses Tool ist kostenlos, erweiterbar und von hoher Implementierungsqualität. [3] [9] [10]

1.3 Dynamische Software-Testverfahren

Die Dynamischen Software-Testverfahren sind im Gegensatz zu den Statischen Software-Testverfahren Tests, die zur Laufzeit durchgeführt werden. Sie eignen sich nur für lauffähige Software. Sie untersuchen die Software nach Fehlern, die in Abhängigkeit von dynamischen Laufzeitparametern auftreten. Diese Tests werden stichprobenartig

durchgeführt, da aufgrund der Verkettung eine 100% Abdeckung von Tests sehr zeitaufwändig ist. Bei den Tests werden die Ist-Ergebnisse mit den Soll-Ergebnissen verglichen und überprüft, ob diese übereinstimmen. Falls die Ergebnisse nicht übereinstimmen liegt ein Fehler vor, der behoben werden muss. Solche Tests werden hauptsächlich von den Programmierern geschrieben, können aber auch automatisch generiert werden, darauf wird in dieser Arbeit aber nicht weiter eingegangen. Um zu kontrollieren wie viel von der Software geprüft worden ist, können bestimmte Tools eingesetzt werden, die automatisch den Quellcode durchgehen und die Testabdeckung von Anweisungen, Zweigen und Bedingungen messen. Die Tools nennen sich Test-Metriken z.B. Code Coverage. Die Tools verdeutlichen mit bestimmten Kennzahlen, wie den Prozentsatz der bereits getesteten Testfälle und Anforderungen aber auch die Anzahl gefundener Fehler. Die Metriken können jedoch nicht absolut behandelt werden, sondern müssen im Zusammenhang gesehen und verglichen werden. Nach dem Durchlauf dieser Test-Metriken können Stellen im Quellcode, die noch nicht getestet werden nachträglich getestet werden. [4][8]

1.4 Warum testen?

Wie in Kapitel 1 der vorliegenden Arbeit wurde bereits ein Beispiel genannt, welches gezeigt hat, warum Softwaretest notwendig sind. Es gibt jedoch noch weitere Aspekte, weswegen getestet sollte, die im folgenden Abschnitt erläutert werden.

Ein weiteres Beispiel aus der Realität zeigt die Kostspieligkeit von Softwarefehlern. Durch fehlerhafte Software in der Produktion von Automobil- und Flugzeug-Branche in den USA sind allein für das Jahr 2000, ein Schaden von 1,8 Milliarden US-Dollar entstanden. Diese Summe entspricht ca. 16% des Softwareumsatzes. Die Hochrechnung der einzelnen Branchen auf die gesamte amerikanische Volkswirtschaft ergibt die Zahl von 59,5 Milliarden US-Dollar pro Jahr als Folge fehlerhafter Software. Die rechtzeitige Beseitigung dieser Fehler würde 70% der Fehlerbehebungskosten und das Schadenspotential um 50% verringern. Dazu passt eine Faustformel bei der Beseitigung von Softwarefehlern „Je später Fehler entdeckt werden, desto aufwändiger ist ihre Behebung“. [6] Aufgrund dieser Faustformel ist es wichtig Fehler so früh wie möglich zu beseitigen. Die Software soll funktionieren und gravierende Fehler sollten vermieden werden. Zudem wird mittlerweile sehr viel Code erweitert oder kopiert, was zu Folge hat, dass Folgefehler weitergetragen werden. Durch das Testen kann auch die Qualität der

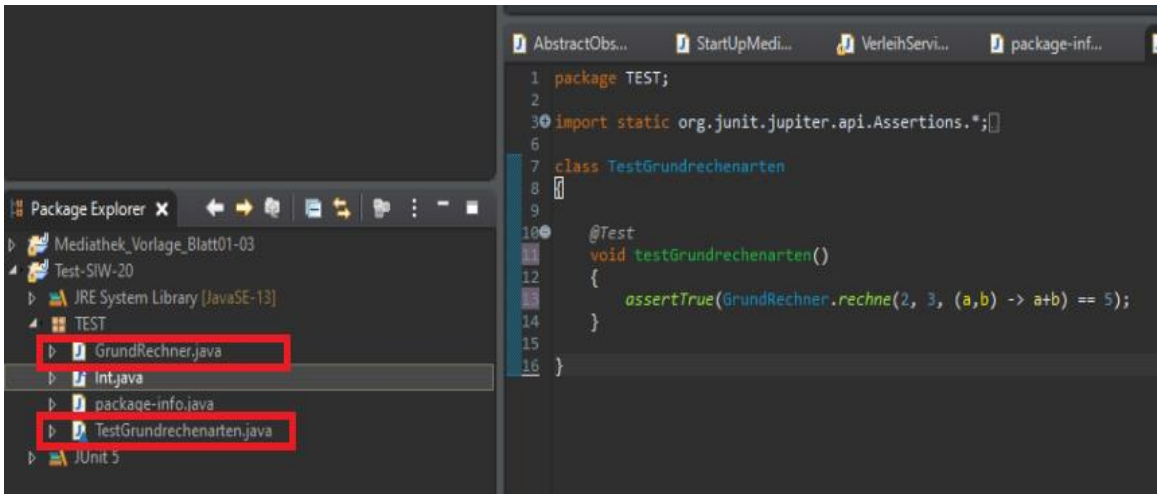
Software bestimmt werden, was dazu führt, dass der Kunde mehr Vertrauen in das Programm hat. Wenn eine Software genügend getestet wurde welche z.B. durch Test-Metriken verdeutlicht werden hat ein Kunde größeres Vertrauen in die Software und wird diese eher benutzen. [5][7]
Im Anschluss wird erklärt was ein Test-Framework ist und wie dieses funktioniert.

2. Test-Framework (JUnit)

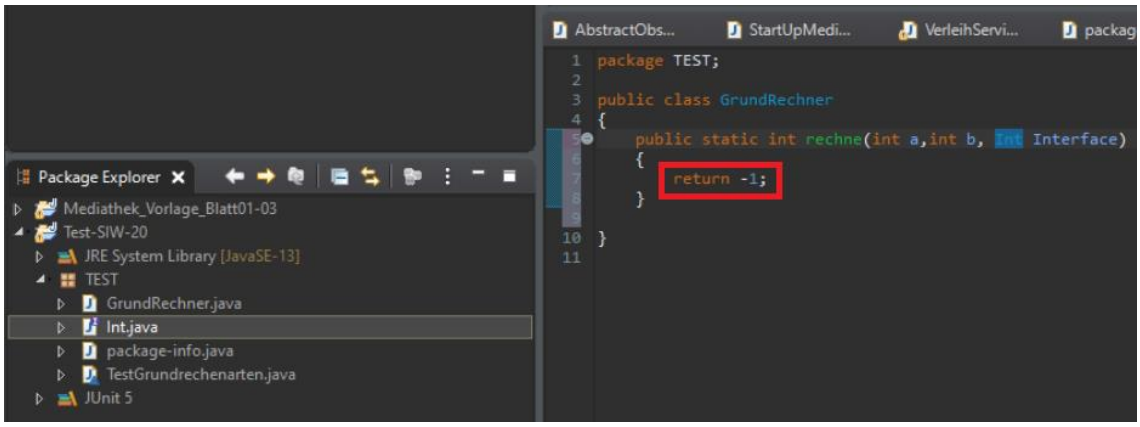
Um das Schreiben von Tests in der Software zu vereinfachen können Test-Frameworks genutzt werden. Diese dienen als Programmiergerüst. Sie selbst sind noch kein fertiges Programm, sondern dienen als Hilfe. Sie geben unter anderem die Anwendungsarchitektur vor. Solche Test-Frameworks gibt es für alle bekannten Programmiersprachen. Ein mögliches Testframwork ist JUnit, das speziell für Software entwickelt wurde, das in der Programmiersprache Java geschrieben wurde. Meistens besitzen diese Frameworks ein „Unit“ im Namen wie auch bei JUnit. Die aktuelle Version ist derzeit JUnit 5, die immer noch weiterentwickelt wird. Unit steht dabei für das Testen von Klassen oder Methoden. Für die Tests werden in JUnit eigene Klassen geschrieben, in denen die Tests implementiert werden. Dies dient der Übersichtlichkeit. Es sollte für jede testbare Klasse eine eigene Testklasse geschrieben werden. In JUnit können bei dem Test nur zwei Ergebnisse ausgegeben werden. Entweder der Test gelingt oder er misslingt. Misslingt ist unterteilt in Failure oder Error. Wenn der Test gelingt führt die jeweilige Klasse oder Methode das zu erwartende Ergebnis aus, falls der Test misslingt muss geprüft werden welcher Fehler vorliegt. Dabei wird zuerst geprüft, ob ein Failure oder Error vorliegt. Bei einem Failure wurde der Test richtig ausgeführt jedoch ist das Ergebnis nicht das erwartete, sodass der getestete Code umgeschrieben werden muss. Bei einem Error hingegen handelt es sich um einen unerwarteten Fehler. Es kann sich um einen Fehler in dem Test selbst handeln, der dann umgeschrieben werden muss. Dies kann aber auch bedeuten, dass ein Fehler entdeckt wurde, für den noch keinen Test geschrieben wurde. Der Test müsste dann neu geschrieben werden. [11][12]
In dem letzten Abschnitt der vorliegenden Arbeit wird erklärt, wie ein Test geschrieben wird.

3. Wie wird ein Test geschrieben?

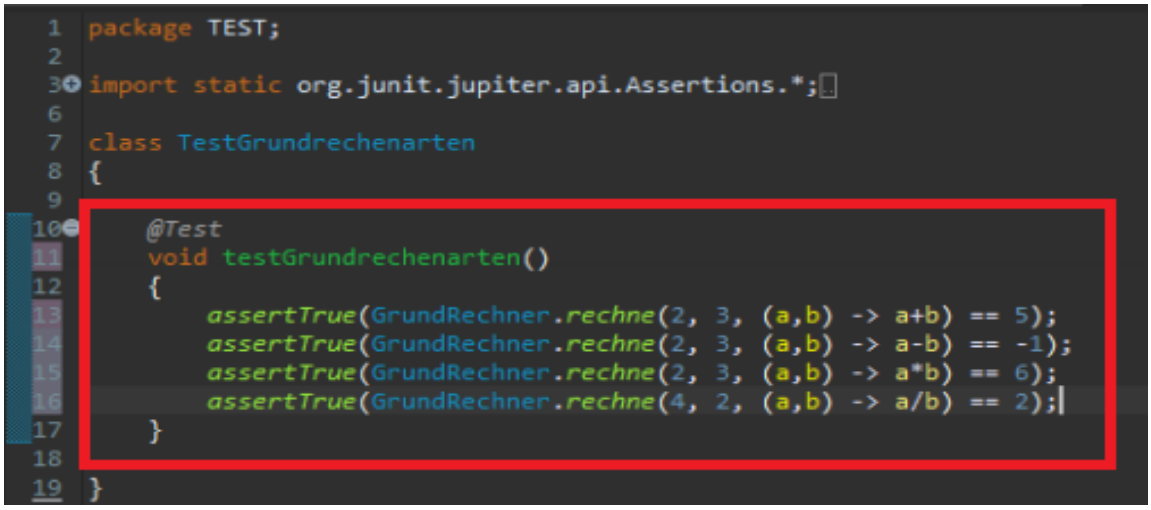
In dem folgenden Abschnitt wird erklärt wie ein Dynamischer Software-Test, in der Programmiersprache Java, mit Hilfe von JUnit in Eclipse geschrieben wird.
Zuerst muss ein neues Projekt in Eclipse angelegt werden. In dem Projekt kann ein Package mit dem gewünschten Namen erstellt werden. Danach werden die Klassen und Methoden geschrieben. Erklärt wird das Vorgehen nach dem Test-First Ansatz, sodass direkt zu der jeweiligen Klasse auch die passende Testklasse geschrieben wird. Es werden in diesem Beispiel die Grundrechenarten getestet, daher muss eine neue Klasse mit dem Namen „GrundRechner“ und der dazugehörigen Testklasse „TestGrundrechenarten“ geschrieben werden. Letzteres ist im folgenden Screenshot rot umkreist zu erkennen.



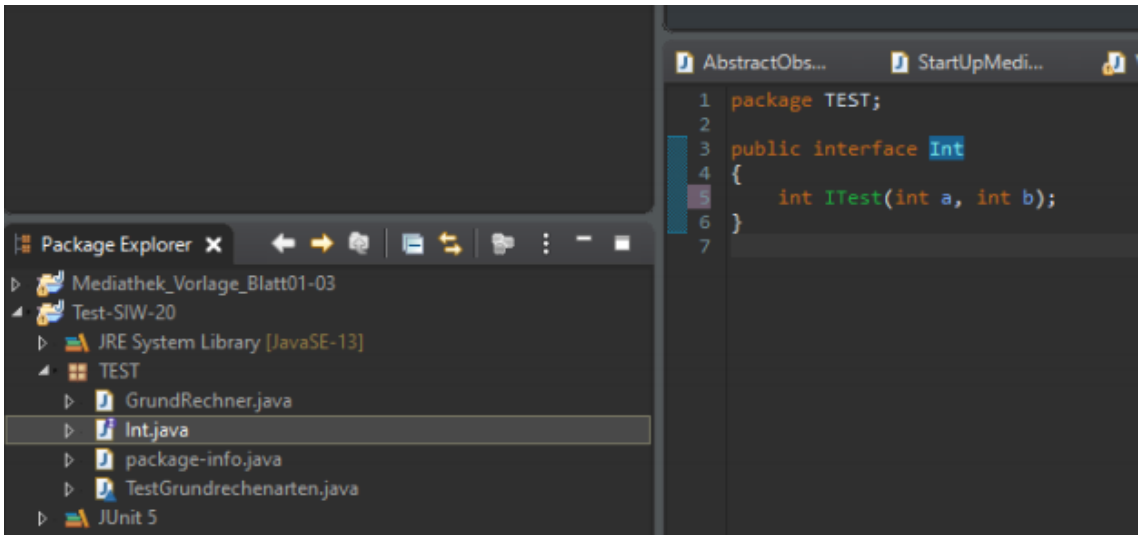
Anschließend werden für die beiden Klassen die jeweiligen Methoden geschrieben. In „GrundRechner““ wird die Methode „public static int rechne(int a, int b, Int Interface)“ geschrieben. Als Rückgabewert sollte zuerst die -1 eingetragen werden. Wie im folgenden Screenshot rot markiert zu erkennen ist.



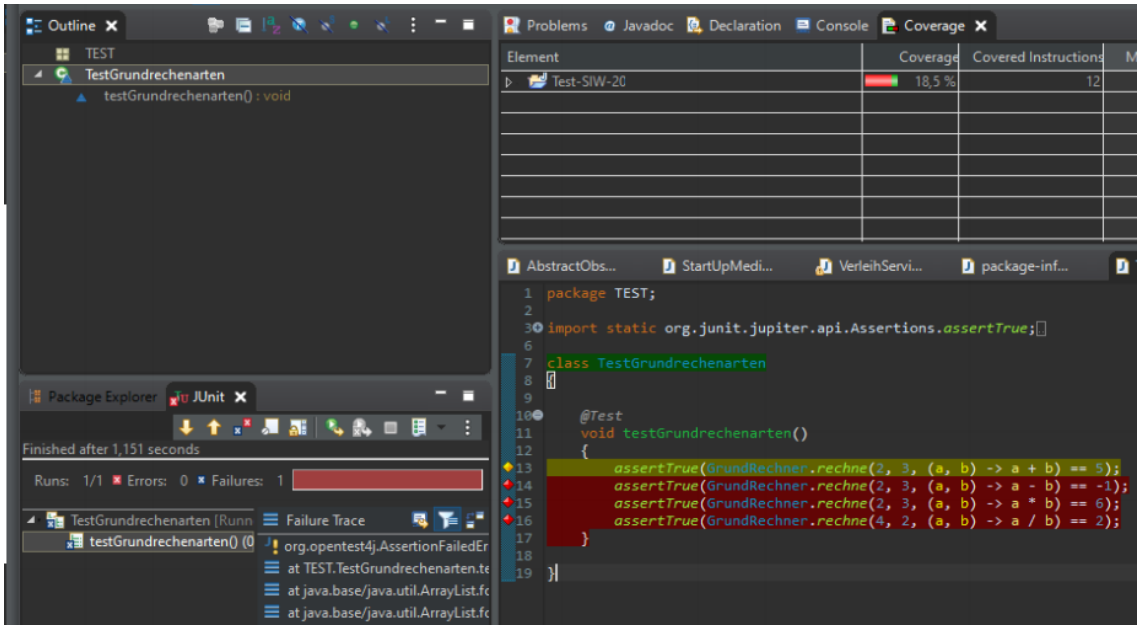
In der Testklasse „TestGrundrechenarten“ wird dem Programm durch „@Test“ vor der Methode signalisiert, dass es sich um einen Test handelt. Der Test wird durch das Test-Framework JUnit welches in Java schon importiert ist erkannt. Im folgenden Screenshot ist der geschriebene Test der Testklasse rot markiert.



In dem Test werden die verschiedenen Grundrechenarten plus, minus, mal und geteilt getestet. Durch die Anweisung „assertTrue“ wird das Ergebnis auf seinen Wahrheitswert geprüft. Der Wahrheitswert ist in diesem Fall das richtige Ergebnis nach der Berechnung. Es kann jedoch auch auf einen Vergleich getestet werden, indem die Anweisung „assertEquals“ verwendet wird. Zur Hilfe für die Rechnung wird in diesem Fall ein Interface erstellt, um das Ergebnis als Lambda Ausdruck zurück zu geben. Dieses Interface fügt zwei variable Zahlen zu einer zusammen. Im folgenden Screenshot ist die Interface Klasse zu erkennen.

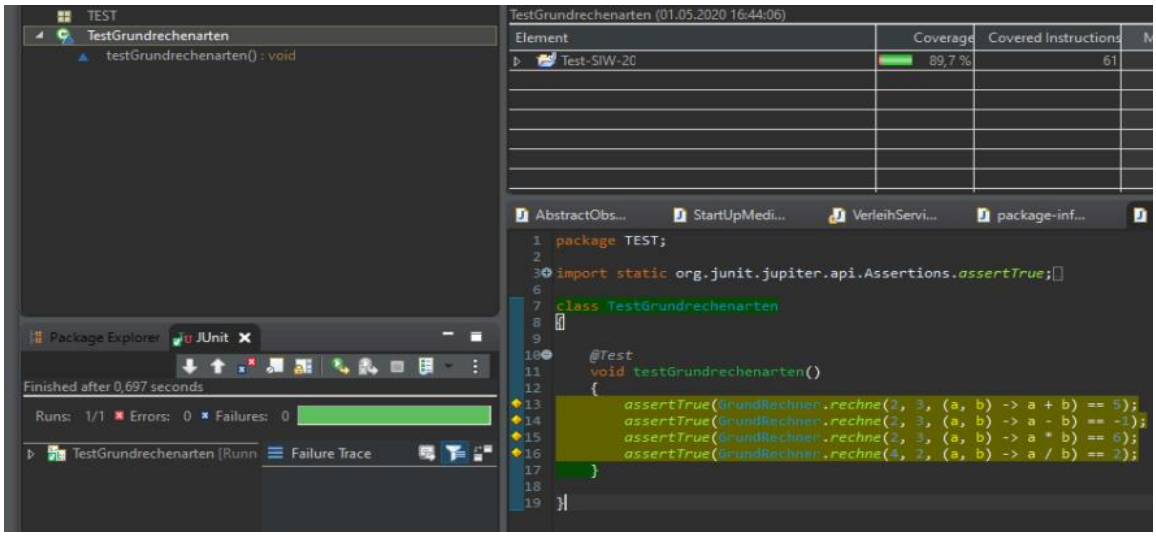


Nachdem alle benötigten Klassen und Methoden geschrieben wurden, wird der Test über Rechtsklick auf die Klasse „TestGrundrechenarten“ und der anschließenden Auswahl von „Coverage As“ und „1JUnit Test“ ausgeführt. In diesem Fall wird das Testergebnis wie folgt aussehen:



Das Programm hat den Test ausgeführt und hat einen Failure entdeckt, welcher durch den roten linken Balken angezeigt wird. Die rechte farbige Untermalung des Quellcodes zeigt an, wie oft der Test an der jeweiligen Stelle ausgeführt wurde. Der entdeckte Fehler muss behoben werden, indem der return Wert aus der „GrundRechner“ Klasse von -1 auf „Interface.ITest(a, b)“ geändert wird. Somit kann die Klasse auf das Interface zugreifen

und kann den richtigen Wert nach der Berechnung zurückgeben, sodass der Test gelingt. Im folgenden Screenshot ist der zuvor rote linke Balken grün. Das bedeutet es werden keine Error oder Failures mehr angezeigt. Damit ist der Test abgeschlossen und es könnte mit dem nächsten Test begonnen werden.



4. Fazit

Zusammenfassend kann gesagt werden, dass die Relevanz Software zu testen darin besteht Fehler zu finden und zu vermeiden, da diese gravierende Auswirkung auf z.B. hohe Kosten für Unternehmen bis hin zur Gefährdung von Menschenleben haben. In der vorliegenden Arbeit wurden daher Statische und Dynamische Software-Testverfahren betrachtet, die für Findung und Vermeidung von Fehlern in Software eigesetzt werden. Zudem wurden Test-Frameworks erläutert die als Programmiergerüst für Softwaretest dienen. In diesem Zusammenhang wurde genauer auf das Test-Framework JUnit eingegangen. Abschließend wurde dargestellt, wie ein Softwaretest in der Programmiersprache Java mit Hilfe von JUnit geschrieben wird und wie ein Softwarefehler behoben werden kann.

Literaturverzeichnis

[1] https://www.welt.de/wirtschaft/article203121966/Autonomes-Fahren-0-2-Sekunden- vor- dem-toedlichen-Aufprall-reagierte-das-Uber-Auto.html

[2] Der Systemtest: anforderungsbasiertes Testen von Software-Systemen / Harry M. Sneed; Manfred Baumgartner; Richard Seidl. 2011 Seite 1f

[3] https://de.wikipedia.org/wiki/Statisches_Software-Testverfahren

[4] https://de.wikipedia.org/wiki/Dynamisches_Software-Testverfahren

[5] Der Systemtest: anforderungsbasiertes Testen von Software-Systemen / Harry M. Sneed; Manfred Baumgartner; Richard Seidl. 2011 Seite 7f

[6] Pol, Koomen, Spillner: Management und Optimierung des Testprozesses

[7] Andreas Spillner, Tilo Linz, Basiswissen Softwaretest, Dpunkt Verlag 2005

[8] https://www.qa-systems.com/tools/cantata/code-coverage/

[9] https://clang-analyzer.llvm.org/

[10] https://clang.llvm.org/

[11] https://de.wikipedia.org/wiki/Framework

[12] https://de.wikipedia.org/wiki/JUnit