Aktuelle Herausforderungen für Testing & Software Verification

Ausarbeitung, Grundseminar

Alexander Menk

Wintersemester 18/19
Prof. Dr. Kai von Luck & Prof. Dr. Jan Sudeikat
HAW Hamburg

abstract Der Anstieg an Komplexität moderner Software scheint sich nicht abzuschwächen und während schon um die Jahrtausendwende der Vorteil von Modultests gezeigt wurde; während automatische Testmethoden, statische Analysewerkzeuge und Model Checking geschrieben, verbreitet und in der Wissenschaft betrieben wurden, scheint es so, dass in vielen Firmen, wie Open-Source Gruppen immer noch nur manuell getestet wird. Doch durch den bleibenden Anstieg an kritischen Anwendungen für Software, entsteht gleichzeitig ein größerer Bedarf an Softwarequalität denn je. Diese Ausarbeitung soll dazu dienen den Forschungsbedarf zu identifizieren.

Inhaltsverzeichnis

1 Einführung

Die Verifikation von *Software* scheint ein, sowohl im Studium, als auch in der Arbeitswelt vernachlässigtes Thema. Fehler und Probleme von traditionellen Programmen sind über die Jahre nicht weniger zum Problem geworden. Dabei sollte die, immer noch steigende Verwendung in kritische Anwendungen im Alltag (Automobilindustrie [3] und damit auch im öffentlichen Verkehr; Energieversorgung, Flugsicherung, ...) mit unterschiedlichen Technologien und Entwicklungsformen (Internet of Things (IOT) [36], Microservice-Architekturen, Cloud) dagegen eine gegenteilige Entwicklung zur Folge haben. In einer Welt, in der die Komplexität moderner *Software* zusehends durch mannigfaltige Abstraktionsschichten, Bibliotheken, Betriebssystemen aus Millionen von Zeilen von Code verdeckt wird [15], sollte die Verifikation von *Software* über 'das eine' Primärziel, der Funktionalität, hinaus, eine wichtige Rolle spielen. Gegenteiliges scheint jedoch der Fall.

Dabei besteht der Fokus jedoch nicht nur auf dem Projektmanagement, den Kostenabrechnungen und den technischen Möglichkeiten. Die Entwicklerkultur selbst, bestehend aus den unterschiedlichsten Gruppen und nicht selten ohne direkten Hintergrund in der Informatik, scheinen dazu beizutragen, dass es immer noch einen Mangel an Verifikationsmethoden innerhalb der Software-Entwicklung gibt.

Die offene Natur dieser Softwarelandschaft, die zum Teil rein auf Online-Plattformen lebt (Github[1], Gitlab[2] und weitere), trägt, zusammen mit dieser modernen Softwarekultur, die eben jene offenen Projekte für die eigene Software nutzt, dazu bei, die Situation noch weiter zu eskalieren.

1.1 Motivation

In nur wenigen Projekten innerhalb des Studiums, sind Verifikationsmethoden Teil der Anforderung gewesen und wenn doch, dann auf sehr grundsätzlichem Niveau. Das ist nicht überraschend, für den Entwickler scheint der interessante und wichtige Teil der Softwareentwicklung in anderen Teilgebieten zu finden.

Wo selbst die Wissenschaften mit Trends zu tun haben, findet in der Industrie ein durchgehender Wechsel von Technologien statt [29, 30]. Die Natur der Verifikation von *Software*, als Teil eines großen Ganzen (von der Formalisierung von Abläufen, bis hin zur Bereitstellung von Werkzeugen) sorgt dafür, dass erst eine Technologie entwickelt und auf längere Zeit verwendet werden muss, bis hinreichende Verifikationswerkzeuge [18] entstehen können. Dies im Zusammenhang mit dem *Image-Problem* und der gleichzeitigen Wichtigkeit der Verifikation von Software, erzeugt einen hohen Forschungsbedarf.

[1] www.github.com
[2] www.gitlab.com

Von den Entwicklern, den Menschen, über genutzte Programm-Bibliotheken bis hin zur Natur des entstehenden Produktes, gibt es also viele Ansätze und potentielle Probleme.

1.2 Gründe für Forschungsbedarf

Neuere *Software*-Architekturen gehen immer stärker dazu über, einzelne Computer durch Virtualisierung und andere Abstraktionsschichten von der Software zu lösen. Microservice-Architekturen, sowie weitere verteilte Anwendungen, die nicht an physische Systeme gebunden sind [22,26], können in die Cloud verschoben und einfacher ausgeliefert werden. Damit steigt die Komplexität, und damit auch die Schwierigkeit, ein *gutes* Software-Produkt liefern zu können, jedoch ein weiteres Mal [15]. Weitere Bestrebungen, eine Auslieferung von Software zu vereinfachen, die Infrastruktur mitliefern zu können und den Prozess zu automatisieren („Infrastructure As Code“, „Continouous Integration“), führen zu einem immer weiter automatisierten Prozess, bei dem ein Software-Release nur noch eine Bestätigung eines Entwicklers und nicht zwingend eine überwachte Abfolge ist. Gleichzeitig steigt die Gefahr durch Softwarefehler, die zu Unannehmlichkeiten in der Anwendungssoftware, aber auch bis hin zu großen Problemen in potentiell lebenswichtigen Situationen führen können [9] [19].

2 Grundlagen der Softwareverifikation

Die Verifikation von *Software* kann auf verschiedene Weisen mit verschiedenen Zielen durchgeführt werden. So kann eine Verifikation durch „Tests“, als auch durch statische Methoden aufgrund von theoretischen Überlegungen und Gesetzen stattfinden. Ein Teil dieser Ausarbeitung beschäftigt sich mit der Rolle des Testings innerhalb der Softwareverifikation und damit um die Frage, wie die Richtigkeit echter Programme besser nachgewiesen werden kann.

2.1 Softwarequalität

Um das Verifizieren von *Software* definieren zu können, bedarf es eine Definition für die Ziele der Verifikation. Und damit eine Definition für die Softwarequalität. Eine Formalisierung dieser Definition stellt der ISO/IEC-2500 (und zuvor ISO/IEC-9126) [32], jedoch bestehen viele weitere Definitionen [14, 33].

Um den Begriff einzuleiten soll eine *Software* für das Online-Banking in diesem Zusammenhang als Beispiel dienen. Der Begriff *Software* bezeichnet hier die Gesamtheit aller, zu einem Programm/einer Bibliothek zugehörigen Aspekte (Ausführbare Dateien, Daten, Sourcecode, Dokumentation [14]). Zunächst muss eine *Software* ein Problem lösen. Eine Banking-*Software* muss gewährleisten,

dass kein Geld verschwindet oder neu gewonnen wird. Diese und weitere Eigenschaften (dies lässt sich noch auf die Vollständigkeit, Korrektheit erweitern) werden innerhalb der Softwarequalität meist unter Funktionalität zusammengefasst und sind nur abstrakt zu definieren, da die meisten Aspekte der Funktionalität stark im Zusammanhang mit der Domain (also dem Einsatzgebiet, dem Kontext der *Software*) gebracht werden muss. Die Benutzung von Floating-Arithmetik wäre in einer Bank-Simulation für Kinder kein Problem, aber ein sehr großer Fehler in einer richtigen Banking Applikation.

Neben der offensichtlichen Anforderung einer *Software* innerhalb der Domain wahre (oder zumindest hinreichende) Aussagen zu liefern, besteht die Softwarequalität aus einer Reihe weiterer Kriterien [33].

Benutzbarkeit Einfachheit der Eingaben (Umgang mit UI Elementen, Einfachheit der API), Lernaufwand, Attraktivität. Die Web-App des Online-Banking Services sollte das Banking vereinfachen und auch Menschen ohne Erfahrung mit Computer-Oberflächen keine Probleme bereiten. Für das Ansehen der Bank ist ein angenehmes UI-Design zu bevorzugen.

Wartbarkeit Sourcecodequalität, Stabilität nach Veränderungen, Testbarkeit. Eine Banking-App muss nicht zwingend erweiterbar sein. Aufgrund der langlebigen Natur einer solchen Anwendung, muss die Software an neue Umstände anpassbar sein (bei Umsetzung einer neuen Währung). Je nach Auslegung kann Erweiterbarkeit hier also sehr sinnvoll sein.

Effizienz Speicherbedarf, Geschwindigkeit in Anbetracht der Nutzung. Fällt somit auch in Benutzbarkeit. Die Banking-App läuft auf unterschiedlichen Heimcomputern, aber die serverseitige Anwendung muss mit vielen Zugriffen umgehen können. Wartezeiten sind zu vermeiden.

Zuverlässigkeit Umgang mit Fehleingaben vom Nutzer. Umgang mit *Faults* und *Failures* (siehe Abschnitt 2.3), Wiederherstellung von Daten. Datenverlust wäre eines der schlimmsten Probleme für eine Banking-App.

Sicherheit Sicherheit vor Datenklau und Identitätsraub und anderen nichterwarteten Veränderungen durch Dritte. kann auch als Funktionalität gesehen [33] werden. Für eine Banking-App wird dies der Bereich mit der größten Anforderung sein.

Softwarequalität unterliegt noch vielen weiteren Einschränkungen und wird maßgeblich durch die Erstellung des ursprünglichen Angebots (Vertrags) beeinflusst und kann, vor allem im Hinblick auf das Einsatzgebiet, auch im Hinblick auf (indirekten) Spezifikationen schwanken [14, p.19]. In Kombination mit der Entwicklungsgeschichte der Applikation und den Tests, entsteht ein komplexes Gefüge, wohin genaue Aussagen über Problemlösungen nicht immer eindeutig möglich sind, selbst wenn auf dem Papier eindeutige Fehler auszumachen sind (siehe Abschnitt 3.1). Je nach Perspektive ist die Anforderungsanalyse der *Softwareverifikation* unterlegen oder eine Voraussetzung.

2.2 Testing in der Softwareverifikation

Das Verifizieren von *Software* basiert auf mehreren Aspekten.

- Das gezielte Durchführen des Endprodukts (dem „Testen“), bei dem die *Software* ausgeführt wird. In diesem Zusammenhang fallen auch die Bewertung von sekundären Qualitätsmerkmalen (der Softwarequalität) über das primäre Ziel der *Software* hinaus (siehe Abschnitt 2.1). Bestimmte Qualitätsmerkmale, wie Benutzbarkeit können nur hier bestimmt werden.
- Betrachtung des Sourcecodes (mit Hilfe von *Software*) im Hinblick auf Fehlern und Qualitätsmerkmalen (Wartbarkeit, Sicherheit, Speicherverbrauch) [31, S.10-11]. Oft wird dies ebenfalls zu Softwarequalität gezählt [31].
- Formale Verifikation der *Software* über Models. Simulationen können auf Eigenschaften der *Software* hinweisen (Petrinetze [22], Computation Tree Logic* [7]). Formale Methoden können Eigenschaften von Modulen beweisen.
- Theoretisches Wissen über die Verifikation von *Software*. Ein großer Aspekt stellt hier die gesammelte Erfahrung im Schreiben und Testen von *Software* dar. Aber auch das komprimierte Wissen aus Büchern und anderen Medien. Andrienko et al. beschreiben dies im Blick auf Visual Analytics [5].
- Softwarepakete, die verwendet werden, wie
  - Unit-Testing Frameworks ({j, scheme}-unit [1] [38], clojure.test [28])
  - Statische Analyse Werkzeuge (sonarlint [23])

2.3 Fehler in Software

Fehler können in unterschiedliche Kategorien eingeteilt werden [31, p.4-3] [14, p.4]: So genannte *Faults* oder *Defects*, also (im ersten Moment ungesehene) Fehler im System, sowie *Failures*, welche das Fehlverhalten der *Software* (für den Benutzer bemerkbar) darstellen. Es ist davon auszugehen, dass es weitaus mehr *Faults* im System gibt. Ein großer Nachteil des Testens ist, dass bei weitem nicht alle *Faults* im System gefunden werden können und das Ziel sich darauf beschränkt möglichst viele *Failures* zu finden.

2.4 Statische und formale Verifikation

Um bestimmte Fehler im System ausschließen zu können, werden neben dem Testing auch unterschiedliche formale Methoden benutzt. Model-Checking, sowie statische Analysewerkzeuge helfen dabei bestimmte Aspekte der Softwarequalität zu gewährleisten oder Hinweise zu liefern. In diesem Zusammenhang können die Werkzeuge auch auf *Faults* hinweisen (siehe Abschnitt 2.3). Besonders statische Werkzeuge werden zur Unterstützung der Entwicklung eingesetzt[3]. Beyer et al. zeigten 2017, dass Model Checking dem Testing (basierend auf automatischer Testerzeugung) in bestimmten Bereichen zur Findung von *Faults* überlegen ist [8] und beides zum finden von Fehlern genutzt werden sollte.

[3] Siehe Werkzeuge, wie Eraser [25], oder Sonar Lint [23]

2.5 Testing

Die einleitende Satz aus dem so genannten SWEBOK der *IEEE Computer Society* fasst den Kern des Testens von *Software* zusammen: „Software testing consists of the dynamic verification that a program provides expected behaviors on a finite set of test cases, suitably selected from the usually infinite execution domain.“ (*Guide to the Software Engineering Body of Knowledge* [31, p.4-1]) Testing befasst sich also mit dem Durchführen von ausgewählten Programmabläufen des zu testenden Produkts, womit sich Testing auch von statischen und formalen Methoden abgrenzt[4]. Tests können sowohl manuell (durch einen Tester), als auch automatisch durch weiteren Sourcecode als Teil der Software, oder durch fremde Programme stattfinden. So gibt es eine Reihe von typischen Abläufen für Tester, die jedoch immer wieder auf neue Programmiersprachen, Umgebungen und Architekturen angepasst werden müssen [18]. Typisch sind Begriffe, wie das *Blackbox Testing*, welches sich vom *Whitebox Testing* darin unterscheidet, dass der zu testende Code unbekannt ist. Beispiele hierfür sind Methoden, wie das *Random-Testing* [11] oder das *Failure-Testing* [4]. Schon vor Jahrzehnten sind Techniken entwickelt worden, die automatische Tests erlauben [13].

Manuelle und automatische Tests Neben dem manuellen Testen, sind unterschiedliche Werkzeuge zur automatischen Testgenerierung [2, 11, 16, 27], sowie automatischen Testausführung [1, 11] veröffentlicht worden. Die Vorteile der Verwendung von automatischen Modul- Integrations- und Systemtests wurden vielfach gezeigt [20]. Meist werden praktisch jedoch sämtliche automatische Testformen unter dem Titel *Unit-Testing* zusammengefasst (siehe Abschnitt 3). Von den bis her genannten Verifikationsmethoden scheint das automatische Testing neben dem manuellen Testen noch die Verbreiteste zu sein.

Ebenen des Testings Testing wird allgemein über mehrere Ebenen durchgeführt [31, p.1-4].

Modul & Modulgruppen Die bekannteste Form von Modultests sind Komponententests, meistens *Unit-tests* genannt (oft sind Modul- und Komponententests als Synonym zu verstehen), jedoch können einzelne Module auch durch andere Methoden (manuell) getestet werden. Auf dieser Ebene finden Tests an kleinstmöglichen Komponenten statt (Klassen, Dateien, Module). Es kann ebenfalls Sinn ergeben mehrere solcher Module auf einmal in Gruppen zu testen.

Integration Sind Module einzeln verifiziert, muss die Interaktion von mehreren Modulen überprüft werden. Schnittstellen, aber auch durch das Aufkommen von Seiteneffekten, sind typische Probleme für diese Kategorie von Tests.

[4] Die Grenzen sind fließend. Unit-Tests können auch zu Verifikation oder Debugging gezählt werden.

Wie in Abschnitt 3 beschrieben, sind die Übergänge zwischen Modul- und Integrationstests oft nicht ganz eindeutig. Durch die Hinzunahme von immer mehr Komponenten können Integrationstests von wenigen Modulen, über viele Hierarchien bis zum System sehr unterschiedlich (komplex) ausfallen.

System Tests an der *Software* als ganzes, werden als Systemtests beschrieben und stellen die komplexeste Form von manuellen Tests dar. Die Auswahl der zu testenden Elemente gestaltet sich als sehr schwierig, da die Möglichkeiten typischerweise unendlich sind (siehe Anfang dieses Abschnitts).

Ziele des Testings Als Teildisziplin der *Softwareverifikation* beinhaltet das Testing den Versuch hinreichende Nachweise für viele Aspekte der Softwarequalität zu finden. Einige Aspekte lassen sich nicht durch Beweise, statische oder formale Methoden beurteilen. Die Attraktivität einer Software definiert sich aus verschiedenen Aspekten, die auch mit dem „Gefühl“ der Nutzer zu tun haben. Statische Methoden können *Faults* verhindern, Model Checking kann den modellierten Aufbau beweisen - aber ein manueller Tests ist nötig, um Qualitätsmerkmale, wie die Attraktivität festzustellen.

Nichtsdestotrotz scheint es, dass in der Industrie weiterhin ausschließlich manuell getestet wird und die Vorzüge der Alternativen (Model Checker, statische Methoden) als zu teuer abgetan oder gleich ignoriert werden.

Ein weiteres Ziel (vor allem für automatische {UI, Unit}-Tests) ist ein verhindern von neu hinzugefügten *Failures* und *Faults* durch Veränderungen des bestehenden Sourcecodes [31, 4-6], so genannte „Regression“. Ein typisches Vorgehen ist hier bestehenden Sourcecode mit Modul-Tests zu versehen und erst dann umzuschreiben (ein *Refactoring*, in einer idealen Welt wäre dies das Vorgehen auch bei der initialen Erstellung der Software).

Typischwerweise wird aktiv versucht einen großen großen Teil des Sourcecodes mit automatischen Tests abzudecken (genannt *Testabdeckung* bzw. *Coverage*) zu erreichen.

Probleme des Testings Das Testen von *Software* beschäftigt sich so sehr mit dem Testenden, wie mit dem Getesteten (siehe auch Abschnitt 3.1). Das Testen von *Software* ist die einfachste, weit verbreiteste Form der Verifikation.

Softwarequalitäts-erhaltende Maßnahmen, wie Tests müssen im Vertrag integriert sein. Es bedarf außerdem gut ausgebildete Tester und Entwickler, um automatische Tests sinnvoll umsetzen zu können. Conzalez et al. zeigen, dass nur wenige Programmierer gute (und überhaupt) Tests schreiben [12].

Im ersten Moment können Unit-Tests die Entwicklungszeit stark verlängern. Dazu handelt es sich bei Modultests, ähnlich zu der Dokumentation, direkt um veraltete Aussagen (*Legacy Code*). Was als *Regressions-Test* ein Vorteil sein kann, muss gepflegt werden und kann damit auch den Entwicklungsprozess aufhalten.

Eines der offensichtlichen Probleme des Testens ist die Festlegung eines korrekten Ergebnisses. Zusammengefasst unter dem Titel *The Oracle Problem* [6]. Schon bei einfachen Modulen, bei denen auf Naturgesetze zurückgegriffen werden kann, können aufgrund von Rand- und Extremwerten (Vergleich: Umgang mit `Null`, `-1` und leeren Strukturen) Schwierigkeiten erzeugt werden und Quelle für viele Fehler sein. Die Ausnahmebehandlung und Integration mehrere Module können für viele Fehler verantwortlich sein.

Trotzdem besteht wohl ein starker Gegensatz zu dem aktuellen Wissensstand und dem in der in der Industrie im Gebrauch befindlichen Methoden. Schon 2003 haben unter anderem Olan et al. gezeigt, dass Modultests die Softwarentwicklung verbessern und auf Dauer Zeit sparen [20]. Erst schleppend scheint sich sowohl in der Open-Source Gemeinde [12, 35], sowie in der Industrie [3] diese Haltung zu verbreiten.

Nicht zuletzt können nicht alle *Faults* (oder sogar *Failures*) einer Software durch Tests gefunden werden, da die Möglichkeiten selbst bei sehr kleinen Programmen zu groß sind [31].

3 Softwareverifikation in Industrie und Open Source

Wie in Abschnitt 2.5 beschrieben sind die Vorteile des Testens in der Wissenschaft bekannt. Aber auch die Verifikation über Model Checking und automatische Testgeneration sind kürzlich gezeigt worden [8].

Gonzalez et al. untersuchten 82450 Open-Source Projekte und fanden, dass nur 17.2 % aller Projekte überhaupt Tests schrieben [12]. Von diesen 17 % nutzten nur 24 % überhaupt Testmuster um die Lesbarkeit von Tests zu verbessern. Zu diesen Mustern gehörten *Simple Test*, bei dem ein Test nur eine *Assertion* besitzt, so dass innerhalb eines Moduls alle *Assertions* unabhängig vom Ausgang getestet werden. *Implicit Teardown* welches festlegt, dass eine `teardown` Methode vorhanden ist, in der die erzeugte Testumgebung für einen möglichen nächsten Test abgerissen wird. Desweiteren sollen Nachrichten in *Assertions* enthalten sein, um das Verständnis über Tests zu verbessern. Und schließlich soll für jedes Modul eine entsprechende Testklasse existieren (*Test Class Per Class*). Dies vereinfacht die Testabdeckung und sorgt für mehr Bereitschaft Tests umzusetzen (diese also zu erweitern). Diesen Mustern wurden Qualitätsattribute zugewiesen (so verbessert das *Assertion Message* Muster die Lesbarkeit des Tests) und die Tests der Projekte unter diesen Umständen bewertet. Von diesen Mustern wurden nun teilweise sehr wenig umgesetzt (von 24 % benutzten nur 153 Projekte das *Implicit Teardown* Muster). Auffällig sei, dass ein Großteil der Umsetzung von Testmustern sich auf kleine Projekte verteilt, während die allgemeine Anzahl von Tests mit der Größe der Projekte steigt. Eine genauere Analyse einiger weniger Projekte, zeigte die schon in Abschnitt 2.5 angesprochene Beobachtung, dass nur wenige Programmierer überhaupt den Großteil der Tests schreiben.

Trautsch et al. zeigen anhand von 10 Python Projekten, dass es noch weitere Probleme mit der Definition und Nutzung von Modultests gibt [35]. Durch die Komplexität des Mockings, wird dieses gerne durch direkten Einsatz von eigenem Code ersetzt, weswegen es mehr Integrationstests als Modultests in den Projekten gibt.

Auf der anderen Seite sind die Entwickler der Industrien, in denen es auf eine hohe Softwarequalität ankommt, eher dazu geneigt Verifikationsmethoden zu verwenden. Altinger et al. [3] sammelten in Gesprächen mit entwickelnden Kollegen, dass die Verifikationsmethoden je nach Entwicklungsziel (formuliert wurden Forschung, Vorentwicklung, Serienfertigung) durchaus schwanken und das immerhin um die 30 % in allen Bereichen mit Modultests arbeiten.

Statische Analysewerkzeuge Altinger et al. zeigten, dass in der Automobilindustrie in 2014 nur in der Forschungsabteilung mit formalen Methoden gearbeitet wurde [3].

Die Vor- und Nachteile der Benutzung von statischen Analysewerkzeugen wurde ebenfalls betrachtet [10, 24]. Die Ergebnisse fallen jedoch nicht eindeutig für die Nutzung von statischen Werkzeugen aus, mehr noch, sie deuten auf verschiedene Probleme der Softwareverifikation allgemein an. Sadowski et al. schlagen vor, dass statische Analysewerkzeuge in den Hintergrund treten müssen, sie formulieren „Developer happiness is key“ ([24, p.65]). Konsens scheint, dass verschiedene Probleme, wie falsche Alarme oder auch zu viele Ausschläge bestehender *Faults*, die Arbeit des Programmierers stören. So kann es vorkommen, dass ein statisches Analysewerkzeug, welches das Problem lösen und die Softwarequalität merklich verbessern würde, verworfen wird. Diese Überlegung kann auch auf das Schreiben und Auswerten von automatisch generierten und ausgeführten Tests bezogen werden.

Eine vorgeschlagene Lösung scheint die Nutzung von Werkzeugen, wie Linter [17, 23, 24] zu sein, die lokale Teile der Software analysieren und auf Probleme bei der Entwicklung hinweisen, statt eine allgemeine Analyse vorzunehmen (Kombinationen dieser Werkzeuge sind möglich, siehe [23]). Jedoch können diese Werkzeuge nicht alle Probleme lösen [10] und weitere Eingriffe sind nötig. Diese und weitere Probleme haben jedoch nicht nur mit den technischen Systemen zu tun, alternative Möglichkeiten müssen gesucht werden.

3.1 Psychologische Überlegungen

Unabhängig von den gewählten Verifikationsmethoden und Komplikationen der Spezifizierung einer *Software*, lassen sich viele Probleme auch auf die Psychologie zurückführen. Verschiedene Versuche wurden unternommen, um die Psychologie des Programmierens zu verstehen. Sowohl im Bezug auf das Lösungsverhalten [21], als auch im Bezug auf das soziale Gefüge und Arbeitsverhalten von Programmierern [37].

Die Schaffung von *Rockstars* und der Rolle der *Seniors* einer Gruppe von Programmierern, hat eine eigene Dynamik erschaffen. Schon 1971 wurde deswegen das *Egoless Programming* mit seinen zehn Commandments [37] geschaffen, um diesen Problemen entgegenzutreten. Teil dieser Commandments sind die Einsicht, dass Fehler entstehen, dass es keinen 'besten' und keinen 'schlechtesten' Programmierer gäbe, und das keiner sich für den eigenen fehlerhaften Code angegriffen fühlen dürfe. Gleichzeitig, müsse man allerdings auch mit Respekt an den Sourcecode eines anderen annähern. Dennoch scheint der soziale Hintergrund einer Gruppe Programmierer maßgeblich das Ergebnis zu beeinflussen.

Insbesondere beim Testing sind weitere Betrachtungen nötig, die mit der Entwicklungskultur zu tun haben können. Ein großes Problem scheint der Programmierer selbst zu sein. Gelöste Probleme werden als uninteressant empfunden. Nur wenige Testkonzepte konnten große Aufmerksamkeit auf sich ziehen. Ein Beispiel ist eine Testimplementation der Firma Netflix[5] für die Schlagworte, wie „Chaos-Engineering“ und „Chaos Monkey“, sowie ein neueres reißerisches Konzept nötig waren, um sich interessant zu gestalten.

Ein weiteres Problem stellt den reinen Zusammenhang dar, dass ein Test für den Programmierer ein Problem nicht verbessert oder gar löst. So müssen Programmierer scheinbar durch einen Arbeitgeber (oder Vertrag) zur Qualitätssicherung (und damit zum Erstellen von Tests) gezwungen werden. In vielen Open-Source Projekten waren wenig Programmierer für die meisten Tests verantwortlich [12]. Sie entschieden auch über die Qualität dieser Tests. Es ist also (weiterhin) so, dass Programmierer ein spezielles Interesse in *Softwareverifikation* entwickeln müssen, um diese auch umzusetzen. Selbst die weite Verbreitung von Begriffen, wie „Test first“ werden nicht von allen Programmierern ernst genommen oder gar von Firmen aufgrund des Zeit-Kosten-Denkens unterdrückt. Neben der Bildung der Programmierer muss also auch ein weiteres Umdenken an anderer Stelle stattfinden.

Fraglich ist auch, wieso Programmierer bekannte Probleme, die durch gefunden statische Analysewerkzeuge, aufgrund von einigen wenigen Fehlschlägen direkt komplett ignorieren können, obwohl diese offensichtliche Fehler zeigen.

Genauere Recherche und tiefgehendere Untersuchungen in diesem Bereich scheinen noch nicht vorhanden, wären aber nötig.

3.2 Forschungsbedarf

Studien zu Testing und Verification in der Praxis Neue Wege müssen gesucht werden Probleme der aktuellen Umsetzung von Softwarverifikationsmethoden umzusetzen. Fabian Trautsch fasst über das Schreiben von Modultests zusammen: „Furthermore, it revealed that developers intend to write unit tests, but they fail to do so.“ (Fabian Trautsch [34]). Neben dem offensichtlichen Bedarf an weiteren Werkzeugen zur Lösung von nebenläufigen Programmen und

[5] https://github.com/netflix/chaosmonkey, Aufgerufen 27.2.2019

anderen Problemen, besteht viel mehr Bedarf an Bildung über die bereits in der Wissenschaft anerkannten Methoden.

Es ist außerdem nicht sicher, ob und wie viel weiter die Industrie in diesem Bereich ist. Zum Teil wird dies durch die Auskunft der Automobilindustrie (als Qualitäts-kritischer Bereich) bestätigt [3], andererseits wird hier keine Aussage über die Qualität dieser Tests ausgesagt. Ob Tests (und andere Model Checking/Analysen) auf Basis aktuellem wissenschaftlichen Wissen durchgeführt werden, bleibt damit fraglich und muss näher untersucht werden.

Außerdem müssen Vergleiche zu anderen Entwicklungsbereichen folgen. Erst wenn eine Basis für eine nutzbare Form der Testumsetzung geschaffen ist, lohnt es sich wirklich weitere Werkzeuge, mit dem Ziel der allgemeinen Verbesserung der Softwarequalität, zu entwickeln.

Es ist außerdem fraglich, wie schnell die Situation sich verschieben wird. Wird die aktuelle Generation später in *Senior* Position die jetzt gelehrten Praktiken, wie *Test first*, umsetzen oder bedarf es noch eine weitere Generation an Programmierern?

Visual Analytics Methoden zur Analyse von Software Eine weitere Vorstellung, wäre die Softwareanalyse viel enger als formalisiertes Vorgehen umzusetzen. Eine Softwareanalyse steht damit über Tests und anderen Vorgehen, ähnlich, wie Andrienko et al. es im Bezug zu Visual Analytics vorschlagen [5] Einige Analysewerkzeuge, wie sonarlint [23] liefern bereits Visualisierungen, jedoch bestehen noch keine Studien über die möglichen Vorteile über die Vorteile eines solchen Vorgehens.

4 Ausblick

Diese Ausarbeitung soll zeigen, dass es nicht nur weitere Verifikationswerkzeuge, sondern inhaltliche Ideen und Forschungsansätze bedarf. Ein Werkzeug, das nicht verwendet wird hat keinen Wert. Fast jede *Software* kann jedoch durch ausgebildete Tester an Qualität gewinnen, wenn nur 16 % aller Projekte überhaupt mit automatischen Tests bestückt sind.

Die Anforderungen an Software sind so unterschiedlich, wie ihre Einsatzgebiete. Aber besonders wenn Open-Source Bibliotheken und Werkzeuge in kritischen Anwendungen zum Einsatz kommen [3], muss die Funktion dieser Werkzeuge ebenfalls gewährleistet werden. Diesen Umstand konnten Gonzalez et al. jedoch nicht nachweisen [12]. Gleichzeitig zeigen verschiedene Studien, dass eine allgemeine Bereitschaft zu Testing und anderen Verifikationsmethoden nicht vorliegt. Die wissenschaftlichen Grundlagen, die die Funktion und den Wert der Werkzeuge aufgezeigen, sind weitesgehend vorhanden. Psychologische und ausbildungstechnische Überlegungen müssen folgen, um diese für Entwickler anwendbar zu gestalten, mit dem Ziel die Qualität von Software allgemein zu verbessern.

Literatur

1. Junit 5. https://junit.org/junit5/, aufgerufen 24.2.19, 17:18
2. Randoop, automatic unit test generation for java. https://randoop.github.io/randoop/, aufgerufen 28.10.18 um 16:04
3. Altinger, H., Wotawa, F., Schurius, M.: Testing methods used in the automotive industry: Results from a survey. In: Proceedings of the 2014 Workshop on Joining AcadeMiA and Industry Contributions to Test Automation and Model-Based Testing. pp. 1–6. JAMAICA 2014, ACM, New York, NY, USA (2014). https://doi.org/10.1145/2631890.2631891, http://doi.acm.org/10.1145/2631890.2631891
4. Alvaro, P., Andrus, K., Sanden, C., Rosenthal, C., Basiri, A., Hochstein, L.: Automating failure testing research at internet scale. In: Proceedings of the Seventh ACM Symposium on Cloud Computing. pp. 17–28. SoCC '16, ACM, New York, NY, USA (2016). https://doi.org/10.1145/2987550.2987555, http://doi.acm.org/10.1145/2987550.2987555
5. Andrienko, N., Lammarsch, T., Andrienko, G., Fuchs, G., Keim, D., Miksch, S., Rind, A.: Viewing visual analytics as model building. Computer Graphics Forum **37**(6), 275–299 (2018). https://doi.org/10.1111/cgf.13324, https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.13324
6. Barr, E.T., Harman, M., McMinn, P., Shahbaz, M., Yoo, S.: The oracle problem in software testing: A survey. IEEE Transactions on Software Engineering **41**(5), 507–525 (May 2015). https://doi.org/10.1109/TSE.2014.2372785
7. Berard, B., Bidoit, M., Finkel, A., Laroussinie, F., Petit, A., Petrucci, L., Schnoebelen, P.: Systems and Software Verification: Model-Checking Techniques and Tools. Springer Publishing Company, Incorporated, 1st edn. (2010)
8. Beyer, D., Lemberger, T.: Software verification: Testing vs. model checking. In: Strichman, O., Tzoref-Brill, R. (eds.) Hardware and Software: Verification and Testing. pp. 99–114. Springer International Publishing, Cham (2017)
9. Bilton, N.: Nest thermostat glitch leaves users in the cold, https://www.nytimes.com/2016/01/14/fashion/nest-thermostat-glitch-battery-dies-software-freeze.html, aufgerufen 21.2.19, 17:20
10. Calcagno, C., Distefano, D., Dubreil, J., Gabi, D., Hooimeijer, P., Luca, M., O'Hearn, P., Papakonstantinou, I., Purbrick, J., Rodriguez, D.: Moving fast with software verification. In: Havelund, K., Holzmann, G., Joshi, R. (eds.) NASA Formal Methods. pp. 3–11. Springer International Publishing, Cham (2015)
11. Claessen, K., Hughes, J.: Quickcheck: A lightweight tool for random testing of haskell programs. SIGPLAN Not. **35**(9), 268–279 (Sep 2000). https://doi.org/10.1145/357766.351266, http://doi.acm.org/10.1145/357766.351266
12. Gonzalez, D., Santos, J.C.S., Popovich, A., Mirakhorli, M., Nagappan, M.: A large-scale study on the usage of testing patterns that address maintainability attributes: Patterns for ease of modification, diagnoses, and comprehension. In: Proceedings of the 14th International Conference on Mining Software Repositories. pp. 391–401. MSR '17, IEEE Press, Piscataway, NJ, USA (2017). https://doi.org/10.1109/MSR.2017.8, https://doi.org/10.1109/MSR.2017.8
13. Kotik, G., Markosian, L.: Automating software analysis and testing using a program transformation system. SIGSOFT Softw. Eng. Notes **14**(8), 75–84 (Nov 1989). https://doi.org/10.1145/75309.75318, http://doi.acm.org/10.1145/75309.75318

14. Laporte, C., April, A.: Software Quality Assurance (12 2017). https://doi.org/10.1002/9781119312451
15. Lawson, H.B.: The march into the black hole of complexity. Commun. ACM **61**(5), 43–45 (Apr 2018). https://doi.org/10.1145/3201606, http://doi.acm.org/10.1145/3201606
16. Lin, M., Hou, X., Liu, R., Ge, L.: Enhancing constraint based test generation by local search. In: Proceedings of the 6th International Conference on Software and Computer Applications. pp. 154–158. ICSCA '17, ACM, New York, NY, USA (2017). https://doi.org/10.1145/3056662.3056672, http://doi.acm.org/10.1145/3056662.3056672
17. Loskutov, A.: Findbugs. https://github.com/findbugsproject/findbugs, aufgerufen 20.2.2019
18. Márki, A., Lindström, B.: Mutation tools for java. In: Proceedings of the Symposium on Applied Computing. pp. 1364–1415. SAC '17, ACM, New York, NY, USA (2017). https://doi.org/10.1145/3019612.3019825, http://doi.acm.org/10.1145/3019612.3019825
19. Mathews, L.: Hackers use ddos attack to cut heat to apartments. https://www.forbes.com/sites/leemathews/2016/11/07/ddos-attack-leaves-finnish-apartments-without-heat/#68bf84ec1a09, aufgerufen 21.2.19, 17:20
20. Olan, M.: Unit testing: Test early, test often. J. Comput. Sci. Coll. **19**(2), 319–328 (Dec 2003), http://dl.acm.org/citation.cfm?id=948785.948830
21. Peitek, N., Siegmund, J., Apel, S., Kästner, C., Parnin, C., Bethmann, A., Leich, T., Saake, G., Brechmann, A.: A look into programmers' heads. IEEE Transactions on Software Engineering pp. 1–1 (2018). https://doi.org/10.1109/TSE.2018.2863303
22. Ramaswamy, S.R., Neelakantan, R.S.: Software design and testing using petri nets : A case study using a distributed simulation software system * (2002)
23. S.A, S.: sonarlint: Fix issues before they exist. https://www.sonarlint.org, aufgerufen 24.2.19, 17:23
24. Sadowski, C., Aftandilian, E., Eagle, A., Miller-Cushon, L., Jaspan, C.: Lessons from building static analysis tools at google. Commun. ACM **61**(4), 58–66 (Mar 2018). https://doi.org/10.1145/3188720, http://doi.acm.org/10.1145/3188720
25. Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., Anderson, T.: Eraser: A dynamic data race detector for multithreaded programs. ACM Trans. Comput. Syst. **15**(4), 391–411 (Nov 1997). https://doi.org/10.1145/265924.265927, http://doi.acm.org/10.1145/265924.265927
26. Savchenko, D., Ashikhmin, N., Radchenko, G.: Testing-as-a-service approach for cloud applications. In: Proceedings of the 9th International Conference on Utility and Cloud Computing. pp. 428–429. UCC '16, ACM, New York, NY, USA (2016). https://doi.org/10.1145/2996890.3007890, http://doi.acm.org/10.1145/2996890.3007890
27. Shahbazi, A., Miller, J.: Black-box string test case generation through a multi-objective optimization. IEEE Transactions on Software Engineering **42**(4), 361–378 (April 2016). https://doi.org/10.1109/TSE.2015.2487958
28. Sierra, S.: Api for clojure.test. http://clojure.github.io/clojure/clojure.test-api.html, aufgerufen 24.2.19, 17:20
29. Society, I.C.: Ieee computer society predicts the future of tech: Top 10 technology trends for 2019. https://www.computer.org/web/pressroom/ieee-cs-top-technology-trends-2019, aufgerufen 26.2.19, 16:23

30. Society, I.C.: Top 10 technology trends for 2018: Ieee computer society predicts the future of tech. https://www.computer.org/web/pressroom/top-technology-trends-2018, aufgerufen 26.2.19, 16:36
31. Society, I.C., Bourque, P., Fairley, R.E.: Guide to the Software Engineering Body of Knowledge (SWEBOK(R)): Version 3.0. IEEE Computer Society Press, Los Alamitos, CA, USA, 3rd edn. (2014)
32. Software, I.J.S.., systems engineering: Iso/iec 25000:2014. https://www.iso.org/standard/64764.html, aufgerufen 26.2.19, 16:56
33. Spinellis, D.: Code quality: The open source perspective (effective software development series) (02 2019)
34. Trautsch, F.: Reflecting the adoption of software testing research in open-source projects. In: 2017 IEEE International Conference on Software Testing, Verification and Validation (ICST) (March 2017). https://doi.org/10.1109/ICST.2017.77
35. Trautsch, F., Grabowski, J.: Are there any unit tests? an empirical study on unit testing in open source python projects. In: 2017 IEEE International Conference on Software Testing, Verification and Validation (ICST). pp. 207–218 (March 2017). https://doi.org/10.1109/ICST.2017.26
36. Voas, J., Kuhn, R., Laplante, P.: Testing iot systems. In: 2018 IEEE Symposium on Service-Oriented System Engineering (SOSE). pp. 48–52 (March 2018). https://doi.org/10.1109/SOSE.2018.00015
37. Weinberg, G.M.: The Psychology of Computer Programming. John Wiley & Sons, Inc., New York, NY, USA (1985)
38. Welsh, N.: Schemeunit: Unit testing for scheme. http://download.plt-scheme.org/doc/html/schemeunit/index.html, aufgerufen 24.2.19, 17:17