Bachelorarbeit

Eine empirische Studie über die Korrelation zwischen Sicherheitsschwachstellen und Qualitätseigenschaften von Software-Designs

Brigitte Wiebe

12. Dezember 2017

Gutachter: Prof. Dr. Jan Jürjens
Sven Peldszus

Prof. Dr. Jan Jürjens
Institut für Softwaretechnik
Institut für Informatik
Universtität Koblenz
Universitätsstraße 1
56070 Koblenz
https://rgse.uni-koblenz.de

Brigitte Wiebe
bpede@uni-koblenz.de
Matrikelnummer: 213200151
Studiengang: Bachelor Informatik
Prüfungsordnung: PO2012

Bachelorarbeit
Thema: Eine empirische Studie über die Korrelation zwischen Sicherheitsschwachstellen und Qualitätseigenschaften von Software-Designs

Eingereicht: 12. Dezember 2017

Betreuer: Sven Peldszus und Daniel Strüber

Prof. Dr. Jan Jürjens
Institut für Softwaretechnik
Institut für Informatik
Universtität Koblenz
Universitätsstraße 1
56070 Koblenz

Ehrenwörtliche Erklärung

Hiermit bestätige ich, dass die vorliegende Arbeit von mir selbständig verfasst wurde und ich keine anderen als die angegebenen Hilfsmittel – insbesondere keine im Quellenverzeichnis nicht benannten Internet-Quellen – benutzt habe und die Arbeit von mir vorher nicht in einem anderen Prüfungsverfahren eingereicht wurde.

Koblenz, den 12. Dezember 2017

Brigitte Wiebe

Abstrakt

Da Software heute nahezu alle Bereiche des Alltags durchdringt, ist Sicherheit von Softwaresystemen ein wichtigeres Anliegen als je zuvor. Die Sicherheit eines Softwaresystems zu bewerten stellt in der Praxis jedoch eine Herausforderung dar, da nicht viele Metriken existieren, die Sicherheitseigenschaften von Sourcecode auf Zahlenwerte abbilden. Eine geläufige Annahme ist, dass das Auftreten von Sicherheitslücken in Korrelation mit der Qualität von Software-Designs steht, allerdings fehlt es momentan an klaren Belegen für diese Annahme. Der Nachweis einer vorhandenen Korrelation kann dazu beitragen die Messung von Programmsicherheit zu optimieren, da man dann gezielt Qualitätsmetriken zum Messen von Sicherheit einsetzen könnte. Zu diesem Zweck wurden in dieser Arbeit für 50 Android-Apps aus dem Open-Source-Bereich drei Sicherheits- und sieben Qualitätsmetriken, sowie Korrelationen zwischen diesen Metriken berechnet. Bei den Qualitätsmetriken handelt es sich um einfache Code-Metriken bis hin zu High-Level-Metriken, wie objektorientierte Antipatterns, die zusammen ein umfassendes Bild der Qualität vermitteln. Um die Sicherheit darzustellen wurden zwei Sichtbarkeitsmetriken zusammen mit einer Metrik, die die minimale Rechteanforderung mobiler Applikationen berechnet, ausgewählt. Es wurde festgestellt, dass auf Basis der betrachteten Evaluationsprojekte deutliche Korrelationen zwischen den meisten Qualitätsmetriken liegen. Zu den Sicherheitsmetriken wurden dahingegen keine signifikanten Korrelationen gefunden. Es werden diese Korrelationen und deren Ursachen diskutiert und auf dieser Basis Empfehlungen formuliert.

Since software influences nearly every aspect of everyday life, the security of software systems is more important than ever before. The evaluation of the security of a software system still poses a significant challenge in practice, mostly due to the lack of metrics, which can map the security properties of source code onto numeric values. It is a common assumption, that the occurrence of security vulnerabilities and the quality of the software design stand in direct correlation, but there is currently no clear evidence to support this. A proof of an existing correlation could help to optimize the measurements of program security, making it possible to apply quality measurements to evaluate it. For this purpose, this work evaluates fifty open-source android applications, using three security and seven quality metrics. It also considers the correlations between the metrics. The quality metrics range from simple code metrics to high-level metrics such as object-oriented anti-patterns, which together provide a comprehensive picture of the quality. Two visibility metrics, along with a metric that computes the minimal permission request for mobile applications, were

selected to illustrate the security. Using the evaluation projects, it was found that there is a clear correlation between most quality metrics. By contrast, no significant correlations were found using the security metrics. This work discusses the correlations and their causes as well as further recommendations based on the findings.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

Die Vision eines fehlerfreien Softwaresystems ist reizvoll, in der Praxis jedoch bei weitem unerreicht. Im Entwicklungs- und Lebenszyklus eines Projektes ist es nahezu vorhersehbar, dass Sicherheitslücken und Bugs auftreten werden und anschließend gefixt werden müssen, damit die Software den Qualitätsansprüchen der Nutzer gerecht bleibt. Dies hat zwei problematische Auswirkungen: Zum einen gilt, dass je später im Entwicklungsstadium Fehler entdeckt werden, desto teurer wird die Überarbeitung des Codes. Zum anderen werden die meisten Schwachstellen erst nach Fertigstellung und Veröffentlichung des Systems bekannt, wobei sie oft erst durch einen Angriff des Systems offengelegt werden, d.h. viel zu spät.

Demnach ist es nicht verwunderlich, dass es bereits viele Ansätze gibt, Software dahingehend zu analysieren, wie gut verschiedene Anforderungen an das System umgesetzt sind. Diese Anforderungen können sich zum Beispiel auf Software-Designs oder Sicherheitsaspekte beziehen. Gutes Software-Design, das sich an der Realisierung von Designpatterns ausrichtet, ist ausschlaggebend für die Wartbarkeit und Erweiterung, bzw. Überarbeitung der bestehenden Funktionalität eines bestehenden Systems, da ein initial schlechtes Design die damit verbundenen Aufgaben maßgeblich erschwert und dadurch die Fehleranfälligkeit steigt [Par94]. Es ist nachvollziehbar, dass es in einem solchen System schwerer wird Sicherheitskonzepte erfolgreich einzubauen, was das Risiko für Sicherheitslücken erhöht.

Um solche Einschätzungen zu ermöglichen hat sich der Gebrauch von Metriken in der Softwaretechnik etabliert. Eine Metrik ist die Abbildung einer speziellen Eigenschaft auf einen numerischen Messwert [LM10]. Durch diese Methode werden verschiedene Systeme gegeneinander vergleichbar. Anhand festgelegter Schwellwerte lässt sich einschätzen, ob geforderte Eigenschaften ausreichend zutreffen. Die meisten Arbeiten in diesem Kontext konzentrieren sich entweder auf die Bewertung der Qualität von Sourcecode oder auf die Bewertung von verschiedenen Sicherheitskriterien. Obwohl angenommen wird, dass die Qualität des Software-Designs ein Maß für die Sicherheit der Software ist, wurde dieser Zusammenhang noch nicht zufriedenstellend anhand realer Systeme nachgewiesen [LT06].

Ausgangsbasis der zu erstellenden Ausarbeitung bilden deshalb verschiedene Qualitäts- und Sicherheitsmetriken, deren mögliche Korrelation ich an diversen Open-Source-Projekten analysieren werde.

1.1 Motivation

Die Qualität des Software-Designs und die Softwaresicherheit sind zwei Aspekte, die bei der Entwicklung von Softwareprojekten oft getrennt voneinander betrachtet werden. Es existiert jedoch die Annahme, dass eine Korrelation zwischen Design-Qualität des Sourcecodes und Sicherheit besteht. Bisher wurde diese Vermutung in aktueller Softwaretechnik-Forschung mehrfach aufgestellt [LT06, AFC10, CCZ08], aber meiner Recherche nach noch nicht an einer großen Menge von Projekten getestet. Eine solche Analyse kann entweder eine Basis für weitere Analysen liefern oder zur Bestätigung der Korrelation führen. Letzteres würde einen großen Mehrwert zum Entwicklungszyklus von Softwareprojekten beisteuern, da Entwickler dann die Möglichkeit haben ihre Produkte anhand solcher Methoden sicherheitstechnisch zu untermauern. Außerdem können sie teure Überarbeitungskosten sparen, indem sie von Anfang an in ein qualitatives Design investieren [HSD+04]. In dieser Arbeit werden die folgenden drei Forschungsfragen betrachtet, um auf die erläuterte bestehende Lücke im wissenschaftlichen Kontext der Evaluation von Softwareprojekten einzugehen:

- RQ1: Existiert eine innere Konsistenz zwischen einzelnen Qualitätseigenschaften innerhalb von Softwarecode?
- RQ2: Existiert eine Korrelation zwischen diversen Qualitäts- und Sicherheitseigenschaften einer einzelnen Version eines Softwareprojektes?
- RQ3: Existiert eine Korrelation zwischen diversen Qualitäts- und Sicherheitseigenschaften über mehrere Versionen hinweg?

Der Inhalt dieser Arbeit wird sich auf die Beantwortung dieser Fragen fokussieren. Ein notwendiger Schritt zur Untersuchung der RQs besteht im Festlegen geeigneter Evaluationsprojekte, die als Testbasis dienen.

In der heutigen Gesellschaft nimmt die Nutzung des Smartphones stetig zu, obwohl grade sicherheitsrelevante Aspekte wie beispielsweise Datenschutz oft sehr kritisch gesehen werden. Die Anzahl der Downloads mobiler Apps stieg in den letzten Jahren signifikant an. 2016 waren es laut des US-Marktforschungsunternehmens AppAnnie 149,3 Billionen Downloads weltweit [Ann17]. Für das Jahr 2017 rechnet AppAnnie mit einer Steigung auf 197 Billionen und 2021 bereits auf 352,9 Billionen Downloads. Smartphones und die damit einhergehenden Mobilen Anwendungen sind aus dem heutigen Alltag nicht mehr wegzudenken und besonders häufig in den Medien vertreten, wenn es um das Thema mangelnde Sicherheit geht. Heise Security veröffentlichte einen Artikel über eine Sicherheitslücke des Android-Browsers, der bis einschließlich Version 4.3 anfällig für Cross-Site-Scripting sei [Eik15]. Dadurch können bösartige Webseiten beliebige andere Seiten fernsteuern und Zugriff auf persönliche Daten, wie z.B. Emails erlangen, sofern der Nutzer zu dem Zeitpunkt eingeloggt ist. Im August 2017 entfernte Google laut ComputerWeekly um die 300 Apps aus dem Playstore,

nachdem Sicherheitsforscher feststellten, dass die Apps Androidgeräte angriffen, um Denial-of-Services-Attacken durchführen zu können [Ash17]. Diese permanente Medienpräsenz mangelnder Sicherheit von Smartphones kann damit zusammenhängen, dass die mobile Internetnutzung in den letzten Jahren allgemein massiv angestiegen ist. Das unabhängige Web-Analyse-Unternehmen StatCounter veröffentlichte 2016 eine Pressemitteilung, aus der hervorgeht, dass die mobile Internetnutzung im Oktober 2016 zum ersten Mal den Desktop in der Internetnutzung übertraf, wie es in Abbildung 1.1 zu sehen ist.

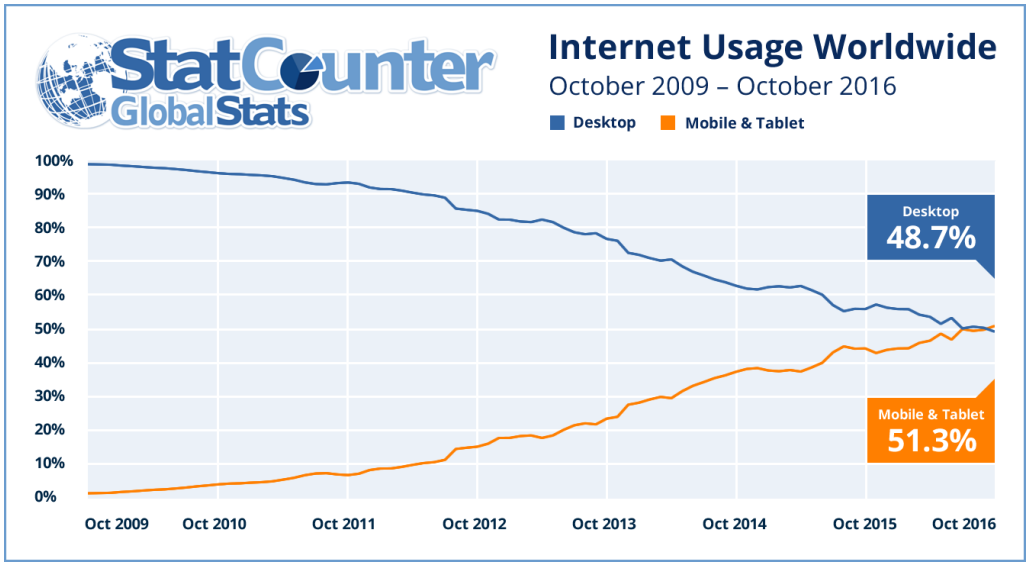


Abbildung 1.1: Mobile Internetnutzung übertrifft Desktop zum ersten Mal weltweit [mob16]

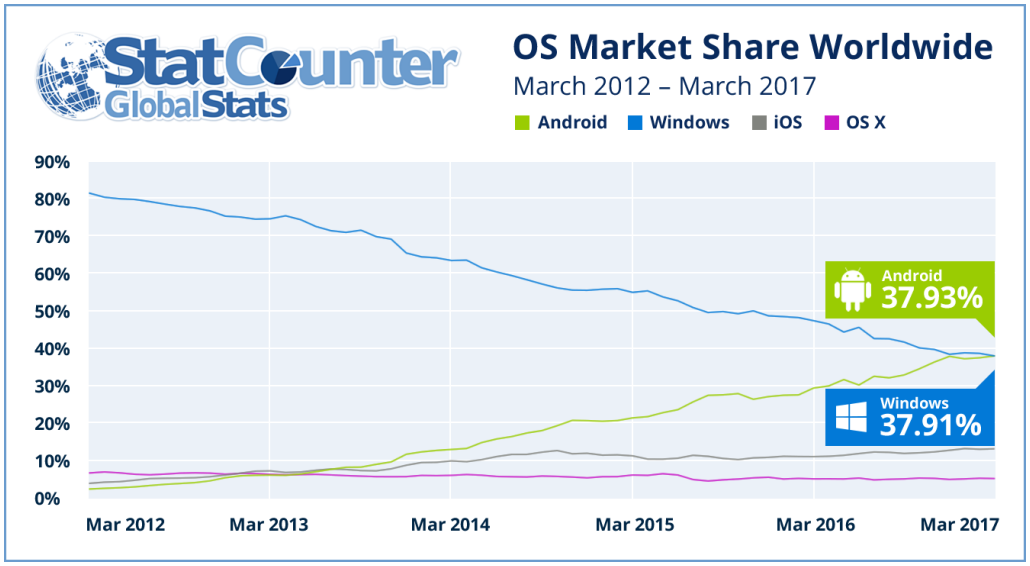


Abbildung 1.2: Android übertrifft Windows zum ersten Mal bezüglich Internetnutzung [and17]

In Abbildung 1.2 ist eine weitere Statistik zu sehen, die das selbe Unternehmen ein Jahr später veröffentlichte. Android überragt hier im März 2017 zum ersten Mal Windows als beliebtestes Betriebssystem bezüglich der Internetnutzung. Diese aktuellen Entwicklungen, in denen sich das Betriebssystem Android und speziell die mobile Internetnutzung so rasant ausbreiten, favorisieren die Möglichkeit Android-Apps als Evaluationsprojekte in dieser Arbeit zu nutzen.

1.2 Zielsetzung

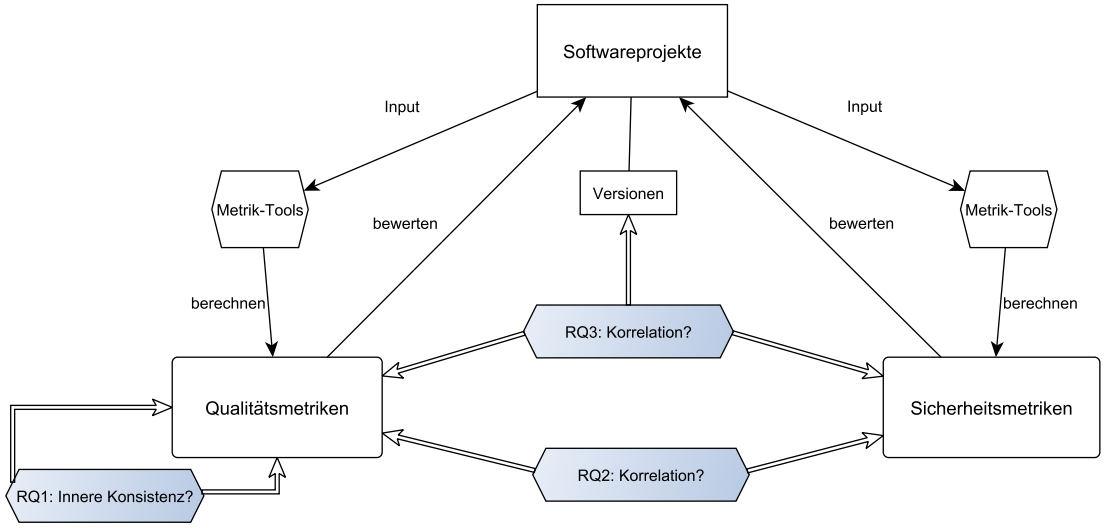


Abbildung 1.3: Die Konzeptgrafik für die Bachelorthesis

Im vorliegenden Abschnitt werden die zugrundeliegenden Ziele dieser Bachelorarbeit erläutert. Anhand der Abbildung 1.3 wird das grobe Konzept zur Untersuchung der Forschungsfragen aus Abschnitt 1.1 vorgestellt. Dazu werden die Forschungsfragen zunächst genauer erläutert:

- *RQ1*: Existiert eine innere Konsistenz zwischen einzelnen Qualitätseigenschaften innerhalb von Softwarecode?
  Diese Frage zielt auf eine vermutete innere Konsistenz zwischen verschiedenen *Qualitätsmetriken* im Sourcecode ab. Es wird spezifisch untersucht, ob die Verschlechterung einer *Metrik* dazu führt, dass sich auch die anderen in ihrem Wert negativ entwickeln. Mittels dieses Vorgehens kann evaluiert werden, ob signifikante Abhängigkeiten zwischen unterschiedlichen Qualitätsmetriken existieren.

- *RQ2*: Existiert eine Korrelation zwischen diversen Qualitäts- und Sicherheitseigenschaften einer einzelnen Version eines Softwareprojektes?
  Hiermit wird der vermutete Zusammenhang zwischen Code-Qualität und sicherheitsspezifischen Schwachstellen im Code untersucht. Es soll ermittelt werden, ob ein hoher Qualitätsstandard das Risiko für Sicherheitslücken reduziert oder ob Qualität keine Auswirkung auf die Sicherheit hat.

- *RQ3*: Existiert eine Korrelation zwischen diversen Qualitäts- und Sicherheitseigenschaften über mehrere Versionen hinweg?
  Mithife dieser Frage wird der gleiche Zusammenhang wie in *RQ2* näher betrachtet, allerdings dieses Mal in Anbetracht verschiedener *Versionen* des gleichen *Softwareprojektes*. D.h. hier wird der Faktor Zeit mitberücksichtigt. Es wird untersucht, ob eine möglicherweise auftretende Korrelation in mehreren *Versionen* zu beobachten ist.

Um auf die Forschungsfragen eingehen zu können, müssen zuerst *Metriken* ausgewählt werden, anhand derer eine Bewertung von Sourcecode bezüglich Qualitäts- und Sicherheitseigenschaften möglich ist. Für die *Metriken* gelten folgende Auswahlkriterien:

- Die einzelnen *Qualitätsmetriken* sollen unterschiedliche Qualitätseigenschaften bewerten, damit die Abhängigkeiten aus *RQ1* untersucht werden können.
- Die *Qualitätsmetriken* sollen ein möglichst weites Spektrum abdecken, d.h. von Code-Level-Metriken bis hin zu High-Level-Metriken reichen.
- Die *Sicherheitsmetriken* dürfen nicht auf *Qualitätsmetriken* basieren, da in dem Fall die Korrelation mathematisch bedingt ist.
- Es sollen möglichst anerkannte *Metriken* sein, die in verschiedenen Publikationen auftreten, um den Wert der Arbeit zu erhöhen und für andere Studien brauchbar zu machen.
- Es muss bereits existierende *Metrik-Tools* geben, die die *Metriken* berechnen, da die Metrik-Tool-Entwicklung nicht Teil meiner Arbeit ist, sondern die empirische Beantwortung der Forschungsfragen.

Anschließend müssen *Metrik-Tools* gefunden werden, mit deren Hilfe die ausgewählten *Metriken* berechnet werden können. Bei der Auswahl der *Metrik-Tools* müssen folgende Kriterien beachtet werden:

- freie Verfügbarkeit
- gute Dokumentation der Tools bezüglich ihrer Nutzung
- möglichst über die Konsole ansteuerbar oder stellen eine Java API zur Verfügung, damit sie aus dem geplanten Programm heraus abrufbar sind

Desweiteren müssen *Softwareprojekte* ausgewählt werden, auf die die *Metriken* anwendbar sind. Es können nur brauchbare Ergebnisse erwartet werden, wenn Sicherheit in den jeweiligen *Projekten* eine Rolle spielt. Vielversprechend ist eine Beschränkung auf mobile Applikationen, da in diesem Kontext bereits die Kriterien erfüllende *Sicherheitsmetriken* existieren, die beispielsweise überprivilegierte Rechteanforderungen oder sicherheitsbezogene Konfigurationseinstellungen untersuchen [FCH+11, EOM09]. Diese Eigenschaften können in Bezug zu verschiedenen Qualitätsanforderungen gestellt werden. Für *RQ3* werden außerdem verschiedene *Versionen* der einzelnen *Softwareprojekte* benötigt, da untersucht werden soll, ob sich die

vermutete Korrelation zwischen Code-Qualität und Sicherheitslücken im Laufe der verschiedenen *Versionen* bestätigt. Das bedeutet, dass die ausgewählten *Softwareprojekte* über genügend *Versionen* verfügen sollen, um dies möglich zu machen. Für die Auswahl der *Softwareprojekte* bietet es sich an die Plattform Github zu nutzen, da hier Millionen von Open-Source-Projekten gehostet werden. Zusätzlich realisiert Github eine ausgeprägte Versionsverwaltung, die besonders für die Diskussion von *RQ3* nützlich ist. Demnach gelten für die *Softwareprojekte* folgende Auswahlkriterien:

- Open-Source-Projekte der Plattform Github
- Verfügung über mehrere Versionen
- sinnvolle Anwendungsmöglichkeit aller ausgewählten Metriken
- evtl. Beschränkung auf mobile Applikationen

Für die Anwendung der *Metrik-Tools* auf die ausgewählten *Softwareprojekte* soll ein Programm implementiert werden, welches folgende Funktionalitäten erfüllt:

- automatisiertes Auschecken von *Softwareprojekten* aus Github anhand der jeweiligen Git-URLs
- automatisiertes Ausführen der ausgewählten *Metrik-Tools* auf die *Softwareprojekte*
- Ausgabe einer Statistik, die die Werte aller *Metriken* für alle *Softwareprojekte* auflistet

Im Anschluss daran soll anhand der berechneten *Metriken* auf die genannten Forschungsfragen eingegangen werden. Zu diesem Zweck muss die Programmausgabe mithilfe geeigneter Methoden ausgewertet werden, um auf die Forschungsfragen eingehen zu können. Zur Auswertung werden statistische Methoden genutzt, um die Metrikwerte der Stichproben zu vergleichen. Bei Korrelationsbetrachtungen werden häufig der Korrelationskoeffizient nach Pearson oder der nach Spearman berechnet. Der Korrelationskoeffizient nach Spearman ist ein nichtparametrisches Verfahren und unterliegt somit weniger Grundvoraussetzungen bezüglich der Werteverteilung der Stichprobe. [SW08]. Deshalb könnte er sich im Laufe der Arbeit als geeigneter herausstellen, als der Korrelationskoeffizient nach Pearson, der wiederum, im Fall erfüllter Voraussetzungen, eine etwas stärkere Aussagekraft bietet.

Mithilfe einer anschließenden Diskussion der Ergebnisse werde ich die Fragen beantworten oder gegebenenfalls erläutern, wieso eine Beantwortung zu diesem Zeitpunkt nicht möglich ist.

1.3 Aufbau der Arbeit

In Kapitel 2 wird zunächst ein Überblick über verwandte Arbeiten gegeben. Danach werden in Kapitel 3 essentielle Grundlagen erläutert, die zum Verständnis der Arbeit benötigt werden. Anschließend wird in Kapitel 4 die Methodik beschrieben, wozu die Auswahl der Evaluationsprojekte, Metriken und Tools gehört. In Kapitel 5 wird auf den praktischen Teil der Arbeit eingegangen und Anforderungen sowie Umsetzung der Implementation erläutert. Dann folgen die Ergebnisse der Studie in Kapitel 6 und die Diskussion derselbigen in Kapitel 7. Abschließend wird in Kapitel 8 eine kurze Zusammenfassung der gesamten Bachelorarbeit gebildet und ein kurzer Ausblick für zukünftige Arbeiten gegeben.

2 Verwandte Arbeiten

Es existieren viele Arbeiten, die allgemein die Code-Qualität von Software anhand von Metriken untersuchen. So bieten beispielsweise Kessentini et al. eine Methode zur automatischen Erkennung von Antipattern an [KVS10]. Dabei orientieren sie sich an der Vorgehensweise des biologischen Immunsystems, welches nicht direkt nach bekannten Viren und Bakterien sucht, sondern Abweichungen von einem als normal angenommenen Zustand erkennt und als mögliche Gefährdung einstuft. Die gleiche Herangehensweise nutzen sie für die Erkennung von Design-Defekten, indem sie analysieren, inwiefern die Implementation von erprobten und sich als gut bewährten Programmiermustern abweicht. Verschiedene weitere Strategien wurden entwickelt, um schlechtes Design zu erkennen und damit die Code-Qualität zu bewerten [Mar04, KVGS09, MGDT10]. In diesen Publikationen wird jedoch nicht der Bezug zu Sicherheitsaspekten gesucht.

Im Bereich der Sicherheitsanalyse sind ebenfalls einige Studien zu finden. Fahl et al. entwickelten das Tool *MalloDroid*, das mobile Anwendungen zum einen auf den Einsatz valider SSL-Zertifikate überprüft, zum anderen untersucht, ob der Gebrauch von SSL angemessen realisiert wurde [FHM+12]. Das Resultat ergab unter anderem, dass acht Prozent der 13.500 untersuchten Applikationen potentiell durch *Men-in-the-middle*-Angriffe verwundbar seien. Dies sei hauptsächlich verschuldet durch den falschen Gebrauch von SSL. Mithilfe des Tools *Stowaway* wurde eine große Anzahl von mobilen Applikationen auf ihre Rechteanforderungen hin untersucht [FCH+11]. Das Tool wurde entwickelt, um herauszufinden, ob Entwickler routinemäßig mehr Privilegien anfordern, als sie benötigen und dadurch mögliche Sicherheitslücken in Kauf nehmen.

Arbeiten, die Qualitätseigenschaften in Bezug zu Sicherheitslücken setzen, sind selten. Die am nächsten verwandte Arbeit, die mir bekannt ist, wurde von Shin und Williams durchgeführt. Sie analysieren den Zusammenhang zwischen Codekomplexität und auftretenden Sicherheitsschwachstellen an einem einzigen Softwareprojekt [SW08]. Sie führten dazu eine Studie mit neun Komplexitätsmetriken durch, die in hypothetischem Zusammenhang mit Sicherheitsproblemen stehen, konnten allerdings nur eine schwache Korrelation zeigen. Ihrer Empfehlung, weitere Studien in dem Bereich durchzuführen, folgen sie drei Jahre später und berechnen weitere Komplexitätsmetriken an zwei Softwareprojekten [SW11]. Allerdings fokussieren sie sich dieses Mal auf Metriken, die zur Laufzeit berechnet werden, während ich der ursprünglichen Empfehlung entsprechend eine statische Analyse durchführe.

Nach Liu und Traore wird die Angreifbarkeit eines Systems als externes Qualitätsattribut gesehen [LT06]. Um die generelle Qualität von Software zu steigern, muss ihnen zufolge eine Beziehung zu internen Qualitätsattributen gefunden werden. Sie versuchen, die Beziehung von Kopplung als interner Eigenschaft zu Angreifbarkeit von außen zu identifizieren, indem sie eine Studie durchführen, die auf *Denial-of-Service*-Angriffen basiert. Abschließend haben sie festgestellt, dass eine starke Korrelation existiert.

Verschiedene Metriken, die auf der Erkennung von Antipatterns basieren, werden in [TKZ+13] vorgeschlagen. Hier wird die Entwicklung von Antipatterns in Anbetracht verschiedener Versionen von Eclipse und ArgoUML untersucht. Die Autoren konnten dabei beobachten, dass die Fehlerdichte in antipatternsdurchsetzten Dateien höher sei als in Dateien, die guten Designmustern folgten. Ihr Hauptanliegen ist, mit dieser Studie ein Fehler-Vorhersage-Modell zu entwickeln. Das Thema Sicherheit wird dabei nicht von ihnen betrachtet.

In [CCZ08] werden Metriken entwickelt, die die Eigenschaft von Code-Qualität untersuchen und möglicherweise die Programmsicherheit erhöhen könnten. Es soll herausgefunden werden, ob ein Angreifer aufgrund von Unvollkommenheit der Codestruktur dem System Schaden zufügen könnte. Allerdings wird der Zusammenhang von Sicherheit zur Codequalität hier nicht mit konkreten Sicherheitsmetriken bewiesen, sondern nur eine Beziehung vermutet.

3 Grundlagen

In diesem Kapitel werden die Grundlagen erläutert, die für das Verständnis der Arbeit benötigt werden. Dafür werden in Abschnitt 3.1 relevante Begriffe eingeführt. In Abschnitt 3.2 folgt eine Erklärung dazu, was statische Programmanalyse ist und wie sie sich von der dynamischen Analyse unterscheidet. Abschließend werden in Abschnitt 3.3 statistische Methodiken erläutert, die in dieser Arbeit verwendet werden.

3.1 Begriffsbestimmungen

Zentrale Begriffe der Ausarbeitung sind Sicherheit, Codequalität, Software-Designs und weitere. In der Literatur werden diese häufig unterschiedlich verwendet und interpretiert. Um ein einheitliches Verständnis zu schaffen, werden die Begriffe hier kurz erklärt und eingegrenzt.

3.1.1 Sicherheit

Den allgemeinen Zustand von *Sicherheit* in einem Softwaresystem zu bewerten ist aufwendig und herausfordernd, wenn nicht sogar unmöglich. In der Praxis hat es sich deshalb als sinnvoll erwiesen, abhängig von der Art der zu untersuchenden Software, sicherheitsrelevante Eigenschaften auszuwählen, an denen man das Sicherheitsniveau messen kann [IF11]. Im Kontext dieser Ausarbeitung wird das Sicherheitsniveau eines Softwaresystems anhand ausgewählter Metriken bewertet. Welche Eigenschaften damit konkret abgedeckt werden, wird in Kapitel 4 erläutert. Der Begriff Sicherheit bezeichnet im Allgemeinen einen von Risiken und Gefahren freien Zustand. Diese Sicherheit kann bedroht werden durch kriminelle Angriffe, organisatorische Mängel oder auch durch technische Unfälle. Während diese komplexe Thematik im Deutschen im Begriff Sicherheit vereint ist, findet im englischen Sprachgebrauch durch die Bezeichnungen *security* und *safety* eine klare Differenzierung statt [MHTK15]. Unter *safety* versteht man den Schutz gegen zufällige oder unvorhersehbare Einflüsse. Das Hauptziel ist, die Umgebung vor einem Fehlverhalten des Systems zu schützen. *Security* hingegen meint den Schutz gegen vorsätzliche Angriffe, sodass das System nicht durch die Umgebung geschädigt werden kann.

Im Kontext dieser Ausarbeitung soll es um Sicherheit im Sinne von *security* gehen. Sicherheitsrelevante Softwareprojekte sollen auf ein potentielles Risiko für Sicherheitsschwachstellen untersucht werden, die durch kriminelle Angriffe ausgenutzt

werden können. Beispielsweise kann durch das Untersuchen mobiler Anwendungen auf einen validen Gebrauch von SSL das Risiko für potentielle *Men-in-the-middle*-Angriffe ermittelt werden [FHM+12].

3.1.2 Codequalität und Software-Design

Die *Codequalität* von Programmen ist ein sehr weitreichender Begriff und es existieren verschiedene Definitionen, die unterschiedlich viele Kriterien miteinbeziehen. Im Kontext dieser Ausarbeitung soll die Qualität von internen Objektstrukturen bewertet werden. Das beinhaltet wechselseitige Beziehungen zwischen Klassen, Komponenten und Methoden. Die zugrundeliegende Annahme ist, dass solche Messungen als objektives Maß genutzt werden können, um externe Qualitäts-Attribute zu beurteilen [BW02]. Als externe Qualitätsattribute werden nach ISO 9126 Funktionalität, Zuverlässigkeit, Benutzbarkeit, Effizienz, Wartbarkeit und Portabilität bezeichnet.

Wenn von *Software-Designs* die Rede ist, ist somit das Code-Design des fertig entwickelten Systems gemeint [Ree92]. In dieser Ausarbeitung werden keine Entwurfsmodelle und Spezifikationen verwendet, sondern ausschließlich der Sourcecode betrachtet und anhand dessen Qualitätseigenschaften des Designs bewertet. Dazu zählen sowohl strukturelle Designentscheidungen, wie beispielsweise die Modularität und Komplexität, als auch eine Bewertung hinsichtlich des (Nicht-)Vorhandenseins von objektorientierten strukturellen Antipatterns. Antipatterns werden in Abschnitt 3.1.4 beschrieben.

Unter Modularität versteht man im Allgemeinen die Zusammensetzung eines Ganzen aus verschiedenen Modulen. In der Softwaretechnik ist dies ein bewährtes Designprinzip, da man einzelne Module überarbeiten und ersetzen kann, ohne dadurch das gesamte System überarbeiten zu müssen [Par72]. Durch einfache Strukturrichtlinien innerhalb der Modularität lassen sich viele Fehlerquellen vermeiden, indem die Abhängigkeiten zwischen einzelnen Modulen überschaubar gehalten werden. Durch das Prinzip der Modularität lassen sich demnach komplexe Systeme einfacher handhaben und eine erhöhte Qualität sicherstellen.

3.1.3 Metrik

Für die Messung von Sicherheits- und Qualitätseigenschaften werden häufig Metriken eingesetzt. Eine *Metrik* ist die Abbildung einer speziellen Eigenschaft eines Softwaresystems auf einen konkreten Messwert. Durch diese Methodik können Systeme bezüglich verschiedener Kriterien bewertet werden und lassen sich mit anderen Systemen vergleichen [LM10]. Die Berechnung der Metrik liefert eine Zahl, die weiterer Eingrenzung und Interpretation durch den Anwender bedarf. Es existieren bereits zahlreiche Metriken, die interne strukturelle Eigenschaften messen (LOC, LCOM, WMC, DIT, u.a.). Die Beziehung zu externen Qualitätsattributen wurde für viele dieser Metriken ausreichend validiert, sodass dieser Zusammenhang für die vorlie-

gende Ausarbeitung vorausgesetzt werden kann. Wenn also im Folgenden die Rede von Qualitätsmetriken ist, sind damit Metriken gemeint, die oben beschriebene Qualitätseigenschaften bewerten.

McCabe's Zyklomatische Komplexität ist beispielsweise eine Qualitätsmetrik, die auf Methoden angewendet wird [LRH]. Sie berechnet die Anzahl der linear unabhängigen Pfade, die der Kontrollfluss einer Methode durchlaufen kann. Um die Metrikwerte interpretieren zu können, empfiehlt es sich, Randwerte zu definieren. *McCabe's Zyklomatische Komplexität* sollte üblicherweise unter dem Wert 10 bleiben, da man eine möglichst geringe Komplexität innerhalb einer Methode haben möchte. Eine weitere Qualitätsmetrik ist *Lack of Cohesion in Methods* [LRH]. Sie bewertet den Zusammenhalt von Methoden innerhalb einer Klasse. Ist der Zusammenhang zu gering, empfiehlt es sich, die Klasse in mehrere Subklassen mit jeweils höherem inneren Zusammenhang zu unterteilen.

Sicherheitsmetriken zielen in diesem Kontext auf die Bewertung eines speziellen Risikos ab. Oft sind es konkrete Angriffsmöglichkeiten, auf die ein Softwaresystem hin untersucht wird, sowie *Man-in-the-middle-* oder *Buffer-Overflow*-Angriffe, aber auch Konfigurationseinstellungen bzgl. Sicherheit oder Rechteanforderungen können ein potentielles Sicherheitsrisiko bergen und als Grundlage für die Definition einer Sicherheitsmetrik dienen.

3.1.4 Antipattern

Ein weiteres Kriterium für qualitatives Design ist das Implementieren von *Designpatterns*. Unter einem Designpattern versteht man ein Entwurfsmuster für einen bewährten Lösungsweg für ähnliche Probleme. Dadurch lassen sich Bausteine wiederverwenden und weisen eine optimale Änderungsfreundlichkeit auf [GHJV94]. *Antipatterns* beschreiben im Gegensatz dazu eine allgemein verbreitete Problemlösung, die entschieden negative Konsequenzen nach sich zieht [BMMM98]. Sie entstehen, weil vielen Entwicklern Wissen und Erfahrung fehlen und dadurch gute Designpatterns entweder unbekannt sind oder in dafür nicht geeigneten Kontexten verwendet werden. Die Verwendung von Antipatterns verhindert zwar keine funktional einwandfreie Implementierung, aber Qualitätsattribute wie Wartbarkeit und Erweiterung werden dadurch beeinträchtigt. Es existieren bereits viele Tools, die spezielle Erkennungsalgorithmen implementieren und Programme nach Antipatterns durchsuchen.

The Blob ist beispielsweise ein Antipattern, dass in Software-Designs auftritt, wo eine Klasse den gesamten Prozess monopolisiert und andere Klassen größtenteils nur Daten einkapseln [BMMM98]. Dieses Prinzip realisiert nicht das Ziel der Objektorientierung Prozesse und Datenmodelle in Objekten miteinander zu verschmelzen, sondern trennt sie voneinander. Gewöhnlich spricht man von *the Blob* wenn eine Klasse mehr als 60 Attribute und Operationen beinhaltet. Aus dieser Menge an

Funktionalitäten innerhalb einer Klasse kann man schlussfolgernd, dass der Zusammenhalt der Attribute und Operationen sich eher gering hält. Weitere Konsequenzen sind erhöhter Schwierigkeitsgrad beim Testen einer so umfangreichen Klasse und steigende Ineffizienz bei Wiederverwendung von Teilen ihrer Funktionalität. Änderungen an anderen Objekten des Systems haben meistens auch Auswirkungen auf *the Blob*, sodass eine Überarbeitung der Software teuer wird.

3.1.5 Android-Apps

Im Internet stehen massiv *Android-Apps* in Form von Sourcecode oder APK-Dateien kostenlos zum Download zur Verfügung. APK ist ein ausführbares Programmformat, das zur Verbreitung und Installation von Anwendungssoftware im Android Betriebssystem genutzt wird [COO13]. Es ist eine Archivdatei, in welcher verschiedene Dateien, wie Sourcecode, Bilder, Audio-, Videodateien etc. komprimiert werden. Vergleichen lässt sich dieses Format mit einem Java Archive (JAR-Datei), in dem kompilierte Java Klassen archiviert werden. Ein signifikanter Unterschied ist, dass in einer APK-Datei genau eine Binärdatei im DEX-Format vorliegt, die alle Sourcecode Klassen in kompilierter Form enthält, während in einer JAR-Datei alle Klassen einzeln in kompilierter Form im CLASS-Format vorliegen.
Android-Apps werden in Java programmiert. Um eine APK-Datei zu erhalten, wird zuerst der Java-Sourcecode in Java-Bytecode kompiliert und anschließend vom Dex-Compiler in Dalvik-Bytecode umgewandelt [Gui12]. Es existieren verschiedene Tools, wie beispielsweise Gradle, die diesen Kompilierungsprozess automatisiert durchführen.

3.2 Statische Programmanalyse

Ein Artefakt, das alle Softwareprojekte gemeinsam haben, ist der Sourcecode. Bei der Entwicklung von Sourcecode schleichen sich immer wieder Fehler und Schwachstellen ein. Auf viele Fehler weist bereits der Compiler hin, aber es gibt immer auch welche die unentdeckt bleiben. Zu Schwachstellen gehören sowohl qualitative Mängel, die nicht bewährten Standards entsprechen oder die Performance beeinträchtigen, als auch Sicherheitslücken, die unter Umständen zu Angriffen auf das System führen können.

Um solche Mängel aufzudecken und das System weitgehend zu optimieren, hat sich die *Programmanalyse* etabliert. Das allgemeine Ziel von Programmanalyse ist es, den Sourcecode und das Verhalten eines Softwareprojektes zu untersuchen und dabei Schwachstellen und Fehler zu lokalisieren. Es existieren verschiedene Arten der Programmanalyse, wobei die manuelle Analyse sehr zeitaufwendig ist. In dieser Arbeit konzentriere ich mich ausschließlich auf die automatische Programmanalyse, die in zwei Hauptkategorien, statische und dynamische Analyse, unterteilt werden kann.

Statische Programmanalyse kennzeichnet sich hauptsächlich durch das Nichtausführen des zu analysierenden Codes. Bereits zur Kompilierzeit werden semantische

Informationen aus dem Sourcecode extrahiert, wodurch die Analyse schon in früher Entwicklungszeit durchführbar ist [DuP13]. Fehler im Code können dem Entwickler zu einem Zeitpunkt bewusst werden, wo die Lösung des Problems noch relativ kostengünstig ist. Im Gegensatz dazu benötigt die dynamische Analyse ein lauffähiges Programm, das zur Laufzeit untersucht wird, weshalb diese Art der Analyse gewöhnlich erst zu einem späteren Zeitpunkt im Entwicklungsprozess anwendbar ist. Ein weiterer Vorteil statischer Analyse gegenüber dynamischer Analyse ist, dass alle möglichen Ausführungspfade und Variablenwerte untersucht werden und nicht nur diejenigen, die beim Testen zur Ausführung kommen [Int12]. Dieser Aspekt ist besonders wertvoll, wenn es um die Untersuchung von Sicherheit geht, da Sicherheitslücken und -angriffe häufig auf unvorhergesehenen und nicht getesteten Szenarien beruhen. Dadurch kann es zu einer hohen Rate an *False Positives* kommen. *False Positives* nennt man Schwachstellen oder Fehler, die angezeigt werden, aber in Wirklichkeit keine sind oder nicht im Code enthalten sind [CM04]. Das hat zur Konsequenz, dass der Analyst entscheiden muss, wie er mit dem Ergebnis eines Analysetools umgeht und welche Hinweise davon relevant sind. Diese Entscheidung ist schwierig zu automatisieren, da sie meistens vom Kontext des untersuchten Programmes abhängt. Es gibt beispielsweise Schwachstellen, deren Risiko bei einem Programm hingenommen werden kann, während die gleiche Schwachstelle in einem anderen Kontext schwerwiegende Konsequenzen nach sich ziehen kann, sodass sie dringend entfernt werden muss.

In dieser Bachelorarbeit wende ich statische Codeanalyse an, um Ergebnisse für die Untersuchung des Zusammenhanges zwischen Codequalität und Sicherheitsschwachstellen zu erhalten. Es existieren bereits viele Opensource-Tools, die anhand von statischer Analyse qualitative Mängel in Sourcecode untersuchen. Einige dieser Tools, wie beispielsweise SourceMeter [sou15] oder Eclipse Metrics Plugin [ecl04], führen eine solche Analyse in Form einer Metrikberechnung durch. Die Grenzen akzeptierter Metrikwerte müssen meistens vom Analysten selbst festlegt werden, um Aussagen über die Qualität des untersuchten Sourcecodes zu treffen. Die Grenzwerte werden normalerweise nicht vom Tool mitgeliefert, da sie abhängig vom Kontext des zu analysierenden Sourcecodes sind.

Das Erkennen von Antipattern kann ebenfalls anhand statischer Analyse erfolgen, wobei Tools in diesem Bereich auf Graphentheorien zurückgreifen. Beispielsweise ist Hulk ein Antipatterndetektions Tool, welches aus dem Abstrakten Syntax Baum (AST) eines Programmes eine Repräsentation des Programmmodells abstrahiert, welche für die Spezifikation von Antipattern relevante Details enthält [PKLS16]. Meyer entwickelte eine theoretische Methode zur Antipattern-Erkennung basierend auf der Untersuchung des Abstrakten Syntax Graphen (ASG), wobei die Antipatterns als ASG-Regeln definiert werden und mithilfe von Matching-Algorithmen entdeckt werden können [Mey06].

Auch in der Bewertung von Sicherheit ist statische Analyse weit verbreitet. An der North Carolina State Universität wurde eine Studie zur Effektivität von statischer

Analyse durchgeführt [ZWN+06]. Die Ergebnisse der statischen Analyse korrelierten dabei zu 83% mit der Anzahl der Fehlermeldungen aus dem PreRelease-Test, womit sie sich als sehr effektiv herausstellen lies.

Ein konkretes Beispiel für ein statisches Analysetool ist Splint, welches potentielle Schwachstellen in Programmen aufgrund von Inkonsistenzen zwischen Sourcecode und geforderten Eigenschaften verwendeter Annotationen, findet[EL02]. Ein weiteres Beispiel ist das Tool MalloDroid [FHM+12]. Hier wird statische Analyse dazu genutzt, mobile Anwendungen auf den Gebrauch einer validen SSL-Realisierung zu untersuchen, wodurch das System angreifbar wäre für Men-in-the-middle-Angriffe. Trotz der weiten Verbreitung des Ansatzes der statischen Analyse existieren in dem Bereich der Sicherheit noch nicht so viele Opensource-Tools, wie beispielsweise in der Qualitätsbewertung.

3.3 Statistik

Um statistische Aussagen treffen zu können, werden in der Regel Hypothesentests durchgeführt. Anhand der vorgegebenen Fragestellung wird dabei eine H_0-Nullhypothese und eine H_1-Alternativhypothese formuliert, entsprechend der geforderten Form des ausgewählten Tests. H_0 und H_1 werden so formuliert, dass ein Nichteintreffen von H_0 zwangsläufig zum Eintreffen von H_1 führt. Dadurch wird das Ziel des Tests, die Nullhypothese zu widerlegen und damit die Alternativhypothese nachzuweisen, möglich.

Da eine Hypothese nur mit einer gewissen Wahrscheinlichkeit widerlegt oder bestätigt werden kann und nie mit absoluter Sicherheit, wird ein Signifikanzniveau α festgelegt [HEK05]. α ist die Wahrscheinlichkeit mit der hier ein Fehler passieren darf. D.h. man lehnt die Nullhypothese ab, obwohl sie der Realität entspricht. Ein solcher Fehler kann eintreten, wenn die Stichprobe, auf deren Daten der Test basiert, einen stark abweichenden Mittelwert zur Grundgesamtheit aufweist. Je sicherer man mit der Entscheidung sein will, desto niedriger muss das Signifikanzniveau α gewählt werden. Ein üblicher Wert für α ist 5%. In besonders kritischen Anwendungsgebieten, wie der Medizin werden niedrigere Werte wie z.B. 1% gewählt. Um zu entscheiden, wie signifikant die Ergebnisse sind, wird außerdem ein P-Wert berechnet. Der P-Wert gibt die Wahrscheinlichkeit des Stichprobenergebnisses bei Gültigkeit der Nullhypothese an [SH09]. Das bedeutet, je kleiner der P-Wert ist, desto mehr spricht das Ergebnis gegen die Nullhypothese. Ist der berechnete P-Wert kleiner als das festgelegte Signifikanzniveau α wird die Nullhypothese verworfen und die Alternativhypothese angenommen.

Als nächster Schritt müssen Daten erhoben werden, die als Testbasis dienen. Abhängig vom Kontext der Studie muss hier entschieden werden, wie groß die Stichprobenmenge sein soll. Das hängt auch davon ab, welcher Grad der Übertragbarkeit der Ergebnisse durch die Studie erhofft wird. Nach der Datenerhebung werden die Daten unter der Annahme, dass die Nullhypothese gilt, ausgewertet. Das exakte Vorgehen

richtet sich dabei nach dem ausgewählten Hypothesentest. Besonders häufig wird der sogenannte *t-Test* durchgeführt. Dieser Hypothesentest testet, ob sich die Mittelwerte zweier unabhängiger Stichproben voneinander unterscheiden [AIC14].

Für die Analyse der Korrelation zwischen Qualität und Sicherheit, die den Kern dieser Ausarbeitung ausmacht, muss ein Korrelationskoeffizient berechnet werden, der die Stärke des Zusammenhangs angibt. Ein Korrelationskoeffizient beschreibt den linearen Zusammenhang zwischen zwei quantitativen Messgrößen und wird durch einen Dezimalwert zwischen -1 und 1 dargestellt [HEK05]. Es gibt verschiedene Möglichkeiten den Korrelationskoeffizienten zu berechnen. Eine Option bildet der Korrelationskoeffizient nach *Pearson*. Um ihn anwenden zu dürfen, muss mindestens eine der zwei Messgrößen aus einer normalverteilten Grundgesamtheit stammen. Das nicht-parametrische Äquivalent zu *Pearsons* Korrelationskoeffizient ist *Spearmans* Rank-Korrelationskoeffizient. Hier muss die Grundgesamtheit nicht normalverteilt sein. Als einzige Voraussetzung gilt, dass die Werte ordinalskaliert sein müssen. Bei *Spearman* werden nicht die konkreten Werte für die Berechnung der Korrelation genutzt, sondern nur die Rangfolge der Werte. Es gibt Meinungen darüber, dass hier ein Informationsverlust stattfindet, aber in der Praxis hat sich gezeigt, dass kaum Unterschiede zwischen den berechneten Korrelationsstärken nach *Spearman* und *Pearson* bestehen. Ein Vorteil der *Spearman* Variante ist, dass Ausreißerwerte eingedämmt werden, da die Abstände zwischen den Werten keine Rolle mehr spielen, sondern nur ihre Reihenfolge.

Bei beiden Varianten gilt für den Korrelationskoeffizienten r: Ist $r = 0$ liegt keine Korrelation vor, d.h. die verglichenen Merkmale sind vollkommen unabhängig voneinander. Ist $r > 0$, korrelieren die zwei Messgrößen positiv miteinander. Das bedeutet, dass das Ansteigen eines Merkmals auch ein Ansteigen des anderen Merkmals verursacht. Bei $r < 0$ liegt ein negativer Zusammenhang vor. Hier sinkt der Wert des einen Merkmals, wenn der Wert des anderen Merkmals steigt. Außerdem gibt es unterschiedlich starke Zusammenhänge:

$$
\begin{aligned}
0,0 \leq |r| \leq 0,2 &\rightarrow \text{kein bis geringer Zusammenhang} \\
0,2 \leq |r| \leq 0,5 &\rightarrow \text{schwacher bis mäßiger Zusammenhang} \\
0,5 \leq |r| \leq 0,8 &\rightarrow \text{deutlicher Zusammenhang} \\
0,8 \leq |r| \leq 1,0 &\rightarrow \text{hoher bis perfekter Zusammenhang}
\end{aligned}
$$

Bei der Berechnung einer Korrelation wird neben dem Korrelationskoeffizienten auch der oben eingeführte P-Wert berechnet. In diesem Kontext bestimmt der P-Wert auf Basis der Stichprobengröße die statistische Signifikanz des Korrelationskoeffizienten. Dabei gilt: Ist der P-Wert $\leq \alpha$ gilt die berechnete Korrelation als statistisch signifikant.

Um zu entscheiden, welche Art von Korrelationskoeffizienten man verwendet, muss das Ergebnis der Stichprobe auf Normalverteilung getestet werden. Unter vielen Tests zur Normalverteilung hat sich der *Shapiro-Wilk-Test* als eine Art Standardtest

für kleinere Stichprobenumfänge etabliert [SS97]. Dabei wird in Form eines Hypothesentests die Teststatistik W, die aus den Ergebnissen der Stichprobe berechnet wird, mit einem kritischen W-Wert W_{crit}, der aus einer vordefinierten Tabelle stammt, verglichen. Auch hier wird wieder ein P-Wert berechnet. Ist $W \leq W_{\text{crit}}$ und gleichzeitig der P-Wert $\leq \alpha$ wird die Nullhypothese, dass die Stichprobe normalverteilt ist, verworfen. Andernfalls wird die Nullhypothese angenommen und die Stichprobe gilt als normalverteilt.

4 Methodik

Um die geplante Studie durchzuführen, werden verschiedene Artefakte benötigt. In diesem Kapitel wird die Beschaffung und resultierende Auswahl dieser Artefakte erläutert. In Kapitel 4.1 gehe ich kurz auf meine Entscheidung für Android-Apps als Evaluationsdaten ein. Auf dieser Entscheidung aufbauend wurden verschiedene Qualitäts- und Sicherheitsmetriken ausgewählt, welche in Kapitel 4.2 vorgestellt werden. In 4.3 werden anschließend externe Metrik-Tools vorgestellt, die sich für die Metrikberechnung eignen und zudem eine automatisierte Ansteuerung zulassen. Außerdem wurden für die Entwicklung noch einige andere Hilfs-Tools verwendet, die in Kapitel 4.4 kurz vorgestellt werden.

4.1 Android-Apps als Evaluationsdaten

In Kapitel 1.1 wurde bereits die Entwicklung des Betriebssystems Android und speziell die rasante Ausbreitung der mobilen Internetnutzung geschildert. Diese aktuellen Entwicklungen haben dazu beigetragen, dass ich mich in meiner Bachelorarbeit auf Android-Apps als Evaluationsprojekte festlege. Ich werde anhand einer angemessenen Anzahl von Applikationen den Zusammenhang zwischen Qualitäts- und Sicherheitseigenschaften im Sourcecode untersuchen. Ein signifikanter Vorteil dieser Auswahl ist es, dass ausreichend Android-Apps in Form von Open-Sourcecode kostenlos zum Download zur Verfügung stehen. Es stellt demnach kein Problem dar, an genügend Daten zu gelangen, um die geplante Studie durchzuführen. Für die Beschaffung der Apps nutze ich die Plattform Github. Github ist eine Softwareentwicklungsplattform, auf der viele Open-Source-Projekte gehostet werden. Da Github eine effiziente Versionsverwaltung implementiert, ist es möglich auch frühere Softwareversionen herunterzuladen. Dadurch lässt sich der Zusammenhang von Qualität und Sicherheit auch über mehrere Versionen hinweg analysieren.

4.2 Metrikauswahl

Um aussagekräftige Ergebnisse für die Auswertung der Studie zu erhalten, mussten bei der Auswahl der Metriken verschiedene Anforderungen beachtet werden. Die Qualitätsmetriken sollten beispielsweise unterschiedliche Qualitätsmerkmale objektorientierten Designs bewerten. Aus diesem Grund wurde neben einfachen Code-Level-Metriken wie LOC (*Lines of Code*) auch aufwendiger zu berechnende High-Level-Metriken verwendet. Ein Beispiel für eine High-Level-Metrik ist das *Zählen von Antipatterns*, die im Code auftreten.

Bei der Auswahl der Sicherheitsmetriken musste beachtet werden, dass diese nicht einer mathematisch bedingten Korrelation zu den Qualitätsmetriken unterliegen. Das bedeutet, die Berechnung der Sicherheitsmetrik durfte nicht durch den Einsatz von Qualitätsmetriken erfolgen, wie es beispielsweise in der Studie von Alshammari et al. der Fall ist [AFC10]. Für beide Arten von Metriken, qualitäts- und sicherheitsbezogen, sollte gelten, dass die Metriken möglichst anerkannt und verbreitet sind. Das bedeutet, dass deren Gebrauch in verschiedenen wissenschaftlichen Arbeiten vorkommt und ausreichend validiert wurde. Außerdem musste es Metrik-Tools geben, die die Berechnung der Metriken vornahmen. Wenn kein konkreter Metrikwert berechnet wurde, musste das Tool zumindest ausreichendes Datenmaterial über den Sourcecode liefern, damit die Berechnung einer Metrik mit vertretbarem Aufwand implementierbar war.

4.2.1 Qualitätsmetriken

Ich habe mich für insgesamt sieben Qualitätsmetriken entschieden. Beruhend auf objektorientierten Designprinzipien ließ sich anhand der folgenden Auswahl eine gute Bewertung der Qualität durchführen. Jede Metrik wird zunächst durch ihre englische Bezeichnung eingeführt. In Klammern dahinter befindet sich der Namenskürzel, dessen Verständnis im weiteren Verlauf dieser Arbeit vorausgesetzt wird. Daraufhin folgt eine knappe Definition der Metrik mit anschließender Erläuterung.

- **Lines of Code per Class (LOCpC)**
  Metrikdefinition: Durchschnittliche Anzahl der Codezeilen pro Klasse ohne Leerzeilen und Kommentarzeilen.
  Erläuterung: Die reine Anzahl der Codezeilen in einem Software-Projekt gilt nicht als Qualitätsmetrik, da sie in Bezug auf die Qualität eines Projektes nichts aussagt, sondern nur die Größe misst. In Bezug auf eine Klasse weist ein hoher LOCpC-Wert allerdings auf schlechte Qualität hin, da eine Klasse ab einer gewissen Größe an Leserlichkeit und Übersichtlichkeit verliert. Dadurch ist sie schwer zu erweitern und zu warten. Aus diesem Grund kann LOCpC als Qualitätsmetrik betrachtet werden.

- **Weighted Methods per Class (WMC)**
  Metrikdefinition: Die Summe aller Methodenkomplexitäten innerhalb einer Klasse [CK94].
  Erläuterung: Um die Methodenkomplexität zu berechnen, wird eine Komplexitätsfunktion benötigt. In dieser Arbeit dient dafür *McCabe's Cyclomatic Complexity Number*. Dies ist eine häufig gewählte Komplexitätsfunktion für die Berechnung von WMC, da sie traditionelle Komplexitätsvorstellungen mit objektorientierten Klassenstrukturen kombiniert [Sch12]. So führen sowohl viele kleine Methoden, als auch wenige komplexe Methoden in einer Klasse zu einem ungünstigen WMC-Wert. Klassen mit einem hohen WMC-Wert weisen darauf hin, sehr applikationsspezifisch zu sein und haben dadurch tendenziell ein niedriges Wiederverwendungspotential. Je größer der WMC-Wert einer Klasse ist,

desto höher ist die Abhängigkeit ihrer Unterklassen von ihr. Bei Vererbung führen diese Abhängigkeiten zu hohen Kopplungen, was dem Designprinzip der Objektorientierung widerspricht und den Wartungsaufwand erhöht. Ein hoher WMC-Wert steht demnach für erschwerte Wiederverwendungsmöglichkeit und hohen Wartungsaufwand und spricht für mangelhafte Qualität.

- **Coupling between Objects (CBO)**
  Metrikdefinition: Anzahl der Klassen, von denen die betreffende Klasse direkt abhängt [CK94].
  Erläuterung: Abhängigkeiten zwischen Klassen entstehen beispielsweise durch Vererbung oder die Nutzung von anderen Klassenattributen und -methoden. Wenn Klassen zum selben Modul gehören, werden die Abhängigkeiten nicht zum CBO-Wert dazu gerechnet, da innerhalb eines Moduls das Prinzip der maximalen Kohäsion gilt und hohe Abhängigkeiten in diesem Kontext für Qualität sprechen. Zwischen verschiedenen Modulen sollen die einzelnen Klassen jedoch so unabhängig wie möglich sein. Je höher der CBO-Wert, desto wahrscheinlicher wird das Prinzip der schwachen Kopplung zwischen Modulen verletzt. Mit steigender Abhängigkeit sinkt die Wiederverwendbarkeit der Klasse und die Modularität des Codes. Abhängigkeiten deuten auf einen hohen Änderungsaufwand hin, da Änderungen an einer Stelle auch andere Codestellen beeinflussen, wobei sich häufig Fehler einschleichen. Ein hoher CBO-Wert indiziert somit komplexen Code, der schwierig zu warten und zu testen ist und ist ein Merkmal für minderwertige Codequalität.

- **Lack of Cohesion in Methods (LCOM)**
  Metrikdefinition: Anzahl der Methodenpaare in einer Klasse, die keine gemeinsamen Attribute haben, abzüglich der Anzahl von Methodenpaaren, die gemeinsame Attribute nutzen [CK94].
  Erläuterung: Ein hoher LCOM-Wert steht für einen geringen Zusammenhalt der Methoden innerhalb einer Klasse. Wenn eine Klasse Funktionen beinhaltet, die völlig unabhängig sind und nichts miteinander zu tun haben, steigert dies die Komplexität des Codes, da für den Entwickler oder den Wartungsbeauftragten kein Zusammenhang erkennbar ist. Ein Prinzip guten Codedesigns ist innerhalb einer Klasse eine starke Kohäsionskraft zu schaffen, um den Zusammenhang deutlich zu machen und den Code verständlicher zu gestalten. Wenn die Methoden voneinander unabhängig sind, deutet es darauf hin, dass dieses Prinzip verletzt wurde. Ein hoher LCOM-Wert steht für mangelnde Kohäsion innerhalb einer Klasse und reduziert dadurch die Codequalität.

- **Depth of Inheritence Tree (DIT)**
  Metrikdefinition: Maximale Länge des Vererbungspfades einer Klasse [CK94].
  Erläuterung: In der Objektorientierung gilt das Prinzip „Komposition vor Vererbung “, was im Fall eines hohen DIT-Werts schwer eingehalten werden kann. Mit der Anzahl geerbter Eigenschaften steigt die Komplexität einer Klasse. Außerdem können Klassen mit einem hohen DIT-Wert nicht isoliert betrachtet werden, was den Wartungsaufwand erheblich erschwert. Auch hier handelt es sich um ein Anzeichen mangelhaften Codedesigns.

- **Lines of Duplicate Code (LDC)**
  Metrikdefinition: Anzahl der Codezeilen, die Bestandteil von Codeduplikaten sind. [FB99]
  Erläuterung: Codeduplikate vergrößern unnötig den Code und erschweren Wartungsarbeiten, da jedes Duplikat einzeln überarbeitet werden muss. Beim Erstellen solcher Duplikate werden häufig Fehler mitkopiert, was den Weiterentwicklungsaufwand erschwert. Es gilt als wichtiger Aspekt von qualitativem Design, Codeduplikate zu eliminieren. Ein hoher LDC-Wert widerspricht diesem Aspekt und weist auf minderwertige Codequalität hin.

- **Occurences of the *Blob* Antipattern (BLOB)**
  Metrikdefinition: Das The-Blob-Antipattern tritt in einer Klasse auf, die einen Großteil des Prozesses monopolisiert und von Datenklassen umringt ist. Dieses Antipattern ist auch bekannt als Gottklasse [BMMM98].
  Erläuterung: Das The-Blob-Antipattern weist auf eine Trennung zwischen Daten und Prozess hin, was nicht dem Prinzip von Objektorientiertem Design entspricht. Klassen, die davon betroffen sind, enthalten häufig mehr als 60 Attribute und Methoden und sind sehr unübersichtlich. Die Existenz vom The-Blob-Antipattern steht für schlechte Wiederverwendbarkeit und erschwerte Wartbarkeit. Auch hierbei handelt es sich um qualitativ mangelhaften Code.

4.2.2 Sicherheitsmetriken

Es gestaltete sich als besonders herausfordernd, Sicherheitsmetriken auszuwählen, da die Auswahl stark davon abhing, ob passende Metrik-Tools vorhanden waren. Viele dieser Tools, die Sicherheit von Apps bewerten, sind veraltet, da sie zu Forschungszwecken entwickelt wurden und danach nicht weiter gepflegt wurden. Dadurch funktionieren sie nicht für neuere Android-Versionen. Weitere Hindernisse ergaben sich durch die Programmiersprache Java und das Anwendungsszenario Android-Apps. Manche Metrik-Tools sind ausschließlich für Softwareprojekte in anderen Programmiersprachen geeignet oder untersuchen Sicherheit in anderen Anwendungsszenarien, wie beispielsweise Webapplikationen. Durch die genannten Herausforderungen verkleinerte sich die Anzahl der zur Verfügung stehenden Sicherheitsmetriken. Nachfolgend wird die Auswahl erläutert, die abschließend daraus getroffen wurde.

- **Minimal Access Modifier**
  Metrikdefinition: Der minimale Zugriffsmodifikator eines Java-Typen ist derjenige, der die aktuell benötigte Nutzung des speziellen Java-Typen in einer gegebenen Codebasis zulässt, aber Nutzungen die darüber hinausgehen nicht zulässt [acc12].
  Erläuterung: Wenn beispielsweise eine Klassenvariable ausschließlich innerhalb der eigenen Klasse genutzt wird, ist der minimale Zugriffsmodifikator *private*. Wird diese Variable hingegen mit *protected* oder *public* deklariert, wird der Metrik-Wert erhöht, da dies nicht der minimale Zugriffsmodifikator ist. Die anzunehmenden Werte liegen im Bereich von 0,00 - 1,00. Zugriffsmodifikatoren regeln in Java die Sichtbarkeit von Attributen, Methoden und Klassen

und realisieren damit ein Level der Zugangskontrolle. Es existieren Ansichten darüber, dass Zugangskontrolle das Zentrum der Computersicherheit ist, denn hier wird kontrolliert, welche Personen, Prozesse, Funktionen etc. Zugriff auf welche Ressourcen im System haben [And08]. Je offener ein System ist, desto eher wird unbefugter Zugriff möglich. Aus diesem Grund habe ich mich dazu entschieden, *Minimal Access Modifier* als Sicherheitsmetrik aufzunehmen. Diese Metrik ist in zwei einzelne Metriken unterteilt:

  - **Inappropriate Generosity with Accessibility of Types (IGAT)**
    Erläuterung: IGAT gibt das Verhältnis an, in dem der Anteil der **Typen**, für die der tatsächliche Zugriffsmodifikator größer ist als der minimale, zur Gesamtanzahl der Typen in dem Sourcecode einer App steht.
  - **Inappropriate Generosity with Accessibility of Method (IGAM)**
    Erläuterung: IGAM gibt das Verhältnis an, in dem der Anteil der **Methoden**, für die der tatsächliche Zugriffsmodifikator größer ist als der minimale, zur Gesamtanzahl der Methoden in dem Sourcecode einer App steht.

- **Minimal Permission Request (MPR)**
  Metrikdefinition: Die Relation zwischen allen angeforderten Rechten einer App und den Rechten, die tatsächlich benötigt werden. [FCH+11]
  Erläuterung: Viele Entwickler fordern routinemäßig oder aufgrund von schlecht dokumentierten APIs mehr Rechte an, als ihre Software überhaupt benötigt. Eine sogenannte überprivilegierte Rechteanforderung kann zu ernsten Sicherheitsrisiken führen. Denn je offener ein System ist, desto mehr Raum bietet es für Sicherheitslücken, die ausgenutzt werden können. Die Überprüfung der minimalen Rechteanforderung bietet eine einfache Möglichkeit zu analysieren, wie sorgfältig Entwickler bei der Programmierung auf Sicherheit in diesem Bereich achten. Aus diesem Grund wurde diese Metrik ebenfalls als Sicherheitsmetrik ausgewählt.

4.3 Metrik-Tool-Auswahl

SourceMeter
SourceMeter ist ein kommerzielles Tool, das den Code von Java-, C/C++-, C#-, Python- und RPG-Projekten statisch analysiert [sou15]. Es wurde entwickelt, um Schwachpunkte im Code aufzudecken, damit Entwickler auf der Basis von Analysewerten die Qualität ihres Sourcecodes verbessern können. Eine kommerzielle Version wurde kostenlos bereitgestellt, aber die zusätzlichen Features konnten im Rahmen dieser Arbeit nicht genutzt werden. Deshalb wurde die kostenlose Version mit limitierter Funktionalität genutzt, die jedoch für das Ausmaß dieser Arbeit vollkommen ausreichte. Qualitätsmetriken, die mithilfe von SourceMeter berechnet wurden, sind:

- Lines of Code per Class

- Weighted Methods per Class
- Coupling between Objects
- Lack of Cohesion in Methods
- Depth of Inheritence Tree
- Lines of Duplicate Code

Hulk
Hulk ist ein Tool, das zu Forschungszwecken entwickelt wurde [PKLS16]. Es beinhaltet eine automatisierte Detektion objektorientierter *Code Smells* und *Antipatterns* in Java-Programmen. Außerdem implementiert es Metriken, die die Verwendung von Zugriffsmodifikatoren messen. Mithilfe von Hulk wurden folgende Metriken berechnet:

- Occurences of the Blob Antipattern
- Inappropriate Generosity with Accessibility of Types
- Inappropriate Generosity with Accessibility of Method

AndroLyze
AndroLyze ist ein statisches Analyse-Tool für Android-Apps [BGSF15]. Basierend auf Skripten ist es in der Lage Sicherheitschecks für große Anzahlen von Apps in effizienter Weise durchzuführen. Unter anderem beinhaltet es ein Skript zur Analyse der Rechteanforderung, welches ich für die Berechnung folgender Sicherheitsmetrik verwendete:

- Minimal Permission Request

4.4 Weitere Tools

Git - Ein Versionverwaltungs-Tool
Git ist ein frei verfügbares System zur Verwaltung von Versionen in der Softwareentwicklung [git05]. Mithilfe eines Git-Clients können verschiedene Entwicklungsschritte auf Servern gespeichert und alte Versionen wieder herunter geladen werden. Außerdem können unterschiedliche Versionen miteinander verschmolzen werden, wodurch ermöglicht wird, dass mehrere Entwickler gleichzeitig am selben Projekt arbeiten. Alle Softwareprojekte, die auf der Plattform Github gehostet werden, haben eine sogenannte Git-URL. Mit dem Tool Git können Projekte anhand dieser URL auf den eigenen Rechner geklont werden. Ich habe Git in dieser Arbeit genutzt, um Android-Apps automatisiert von der Plattform Github herunterzuladen.

Gradle - Ein Build-Management-Tool
Gradle ist ein Tool, das den Build-Prozess von Softwareprojekten steuert [gra07]. Es beinhaltet unter anderem Tasks zum Kompilieren und Testen einer Anwendung

und für die Definition von Abhängigkeiten. Besonders umfangreiche Projekte, die aus mehreren einzelnen Projekten bestehen, sind mithilfe von Gradle einfach zu bauen. Unter anderem hat sich Gradle für den Bau von Android-Systemen etabliert. In dieser Arbeit nutzte ich dieses Tool, um aus den Android-App-Projekten APK-Dateien zu bauen.

Android SDK Tools
Die Android SDK Tools werden von Gradle zum Bauen von apk-Dateien benötigt. Hier drin enthalten ist das Tool *sdkmanager*, womit verschiedene Android Plattformen und Build-Tools installiert werden können, die abhängig von der jeweiligen Android-Version der App sind [and09]. In dieser Arbeit wurden die Android SDK Tools von Gradle zum Bauen der Android-App-Projekte benötigt.

MongoDB - Eine Datenbank
MongoDB ist eine NoSQL-Allzweck-Datenbank mit offenem Quellcode [mon09]. Sie wurde für Skalierbarkeit, Leistung und hohe Verfügbarkeit entwickelt. Die MongoDB-Syntax liegt im Datenformat JSON vor. MongoDB ist gut dokumentiert und damit relativ simpel einzurichten. Ich habe MongoDB in dieser Arbeit auf indirekte Art und Weise verwendet, da es im Hintergrund laufen musste, damit das Tool AndroLyze Ergebnisse im Dateisystem speichern konnte.

5 Werkzeugunterstützung

Um Datenmaterial für die Durchführung der Studie zu einer möglichen Korrelation zwischen Qualitäts- und Sicherheitseigenschaften objektorientierten Codedesigns zu erhalten, habe ich ein *Metric-Correlation-Analysis*-Tool entwickelt. Der Code und die entsprechende Installationsanweisung sind auf meiner Github Seite verfügbar[1]. Für die Implementierung habe ich mich für die integrierte Entwicklungsumgebung Eclipse [Ecl16] entschieden, die durch das von mir entwickelte Plugin erweitert wurde. Eclipse gehört zu den am weitesten verbreiteten Programmierumgebungen. Es ist gut dokumentiert, intuitiv bedienbar und verfügt über eine weite Bandbreite an Open-Source-Plugins, die die Programmentwicklung erleichtern. Als Programmiersprache kam Java zum Einsatz.

5.1 Anforderungen

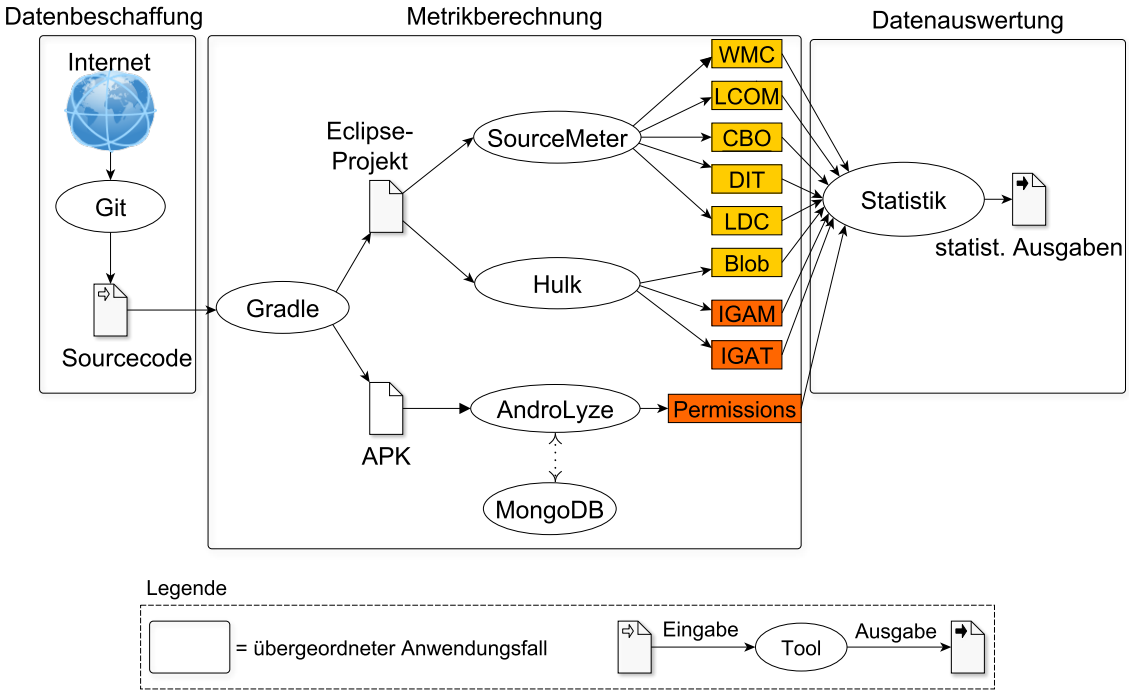


Abbildung 5.1: Konzeptgrafik der Implementierung

[1] https://github.com/biggiwiebe/metric-correlation-analysis

Abb. 5.1 veranschaulicht das grobe Konzept der Implementierung. Die genutzen Tools, die bereits in Kapitel 4 erwähnt wurden, sind durch Ellipsen dargestellt und Daten durch Rechtecke. Qualitätsmetriken sind gelb gekennzeichnet und Sicherheitsmetriken durch eine orange Farbmarkierung erkennbar. Anhand der Pfeilrichtung ist der Ablauf des Programmes zu erkennen, der in drei Hauptanwendungsfälle gegliedert ist. In den folgenden Abschnitten werden die funktionalen (FA) und die nicht funktionalen Anforderungen (NFA) an das Programm aufgelistet.

5.1.1 Funktionale Anforderungen (FA)

1. **Datenbeschaffung**
   (a) Ansteuerung der Versionverwaltunssoftware Git
   (b) Herunterladen eines Evaluationsprojektes anhand einer Git-URL
   (c) Herunterladen einer Menge von Evaluationsprojekten anhand einer Liste von Git-URLs
   (d) Herunterladen einer konkreten Version eines Evaluationsprojektes anhand der jeweiligen Commit-ID
2. **Datenaufbereitung**
   (a) Ansteuerung der Datenbank MongoDB
   (b) Ansteurung des Build-Tools Gradle
   (c) Duchführung eines Gradle-Imports für den Erhalt eines Eclipse-Projektes
   (d) Durchführung eines Gradle-Builds für den Erhalt einer APK-Datei
3. **Metrikberechnung**
   (a) Ansteuerung des Metrik-Tools Hulk
   (b) Ansteuerung des Metrik-Tools SourceMeter
   (c) Ansteuerung des Metrik-Tools AndroLyze
4. **Datenverwertung**
   (a) Herausfiltern aller nicht relevanten Sourcecode-Dateien aus einem Evaluationsprojekt
   (b) Extrahieren ausgewählter Metrikwerte aus einer CSV-Datei und Bilden eines Gesamtwertes aus allen Klassenwerten eines Projektes zu jeder Metrik
   (c) Berechnung eines Metrikwerts auf Basis einer JSON-Datei
5. **Datenauswertung**
   (a) Durchführung eines Tests auf Normalverteilung

   (b) Berechnung einer Korrelationsmatrix nach Pearson und nach Spearman

   (c) Darstellung der Metrikverteilung aller Klassen einer App als Boxplot-Diagramm

6. **Datenspeicherung**

   (a) Speicherung der numerischen Ergebnisse im Dateisystem in Form von CSV-Dateien

   (b) Speicherung der grafischen Ergebnisse im Dateisystem in Form von Bild-Dateien

5.1.2 Nicht funktionale Anforderungen (NFA)

1. **Systemunabhängigkeit**

   (a) Lauffähigkeit auf den Betriebssystemen Windows und Linux

2. **Erweiterbarkeit**

   (a) Einfache Integration weiterer Metriken und Tools

   (b) Einfaches Hinzufügen weiterer statistischer Funktionalitäten

5.2 Konzeption und Umsetzung

Die Umsetzung der erhobenen Anforderungen erfolgte durch die Implementation des Tools *Metric-Correlation-Analysis*. Das Tool besteht aus zwei voneinander unabhängigen Teilen, die in zwei Java-Pakete unterteilt sind. Im ersten Paket *metricTool* geht es um die Berechnung der Metrikwerte. Im zweiten Paket *statistic* wird die Ausgabe des ersten Programmteils weiterverarbeitet, indem verschiedene statistische Berechnungen auf den Metrikwerten ausgeführt werden. Bei der Beschreibung der verschiedenen Funktionalitäten wird in Klammern auf die jeweilige Anforderung verwiesen.

5.2.1 Berechnung der Metriken

Anhand des Klassendiagrammes in Abb. 5.2 lässt sich die Struktur des Paketes *metricTool*, welches für die Metrikberechnung zuständig ist, beschreiben. Die Hauptklasse des Paketes ist *Executer*. Hier befindet sich die Methode *mainProcess()* mit dem Parameter *src_location*, der den Ordner mit allen Evaluationsprojekten enthält und dem Parameter *result_file*, der die Datei angibt, wo die Ergebnisse gespeichert werden sollen. Die Methode *mainProcess()* steuert den Ablauf des Programmes, wie es in Abb. 5.1 dargestellt ist. Zu Beginn des Programmablaufs wird die Methode *startDatabase()* aufgerufen, die die MongoDB startet, die im Hintergrund für das Tool AndroLyze laufen muss (FA2a). Anschließend wird für jedes Projekt, dass sich in dem Parameter *src_location* befindet, die Methode *calculateMetrics()* aufgerufen.

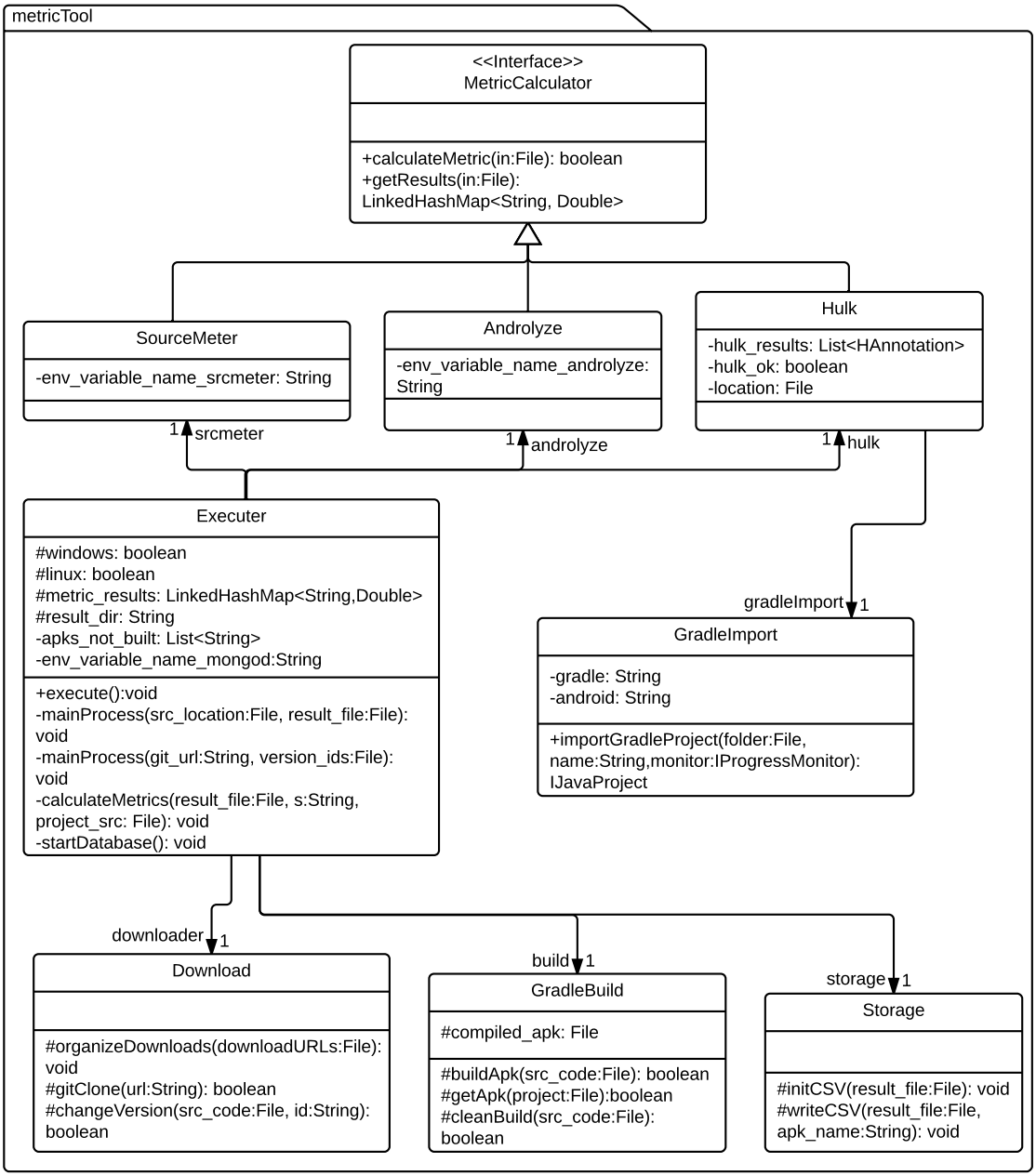


Abbildung 5.2: Klassendiagramm package metricTool

Diese Methode beinhaltet die Aufrufe der Methoden zur Berechnung der Metriken und die Speicherung der jeweiligen Rückgabewerte in *result_file*.

Datenbeschaffung
Der Erhalt der Android-Applikationen, die in dieser Studie als Evaluationsprojekte dienten, wird durch die Klasse *Download* implementiert. In der Methode *gitClone()* ist die Ansteuerung des Build-Tools Git realisiert (FA1a). Für den Download einer einzelnen App wird direkt *gitClone()* aufgerufen (FA1b). Will man eine ganze Liste von Android-Apps herunterladen, muss *organizeDownloads()* aufgerufen werden (FA1c). Beide Methoden bekommen die Download-URLs der jeweiligen Github-Projekte als Parameter übergeben. Diese Datenbeschaffung muss geschehen, bevor die Methode *mainProcess()* in der Hauptklasse aufgerufen wird.

Datenaufbereitung
Die Klasse *GradleBuild* steuert das Build-Tool Gradle an (FA2b). In der Methode *GradleBuild.buildApk()* wird die APK-Datei gebaut, die als Eingabeparameter für das Tool AndroLyze benötigt wird (FA2d). In *GradleBuild.getApk()* wird die generierte APK-Datei im jeweiligen Projektordner gesucht und in der Klassenvariablen *GradleBuild.compiled_apk* gespeichert. Die Klasse *GradleImport* nutze ich, um den Gradle-Import einer App in ein Eclipse-Workspace zu realisieren (FA2c). Dieser Vorgang wird benötigt, da das Tool Hulk als Eclipse-Plugin vorliegt und nur auf Eclipse-Projekten ausgeführt werden kann. Außerdem werden bei dem Import alle Klassen herausgefiltert, die Sourcecode von eingebundenen Bibliotheken verwenden, damit die Metrikberechnungen auf dem Sourcecode nicht verfälscht werden (FA4a).

Metrikberechnung und Datenverwertung
Für die externen Tools, die die Metrikberechnungen durchführen, werden jeweils Klassen erstellt, die das Interface *MetricCalculator* implementieren. Dabei beinhaltet die Methode *calculateMetric()* die Ansteuerung des jeweiligen Tools (FA3a/b/c), wobei die Ergebnisse entweder im lokalen Dateisystem oder in Datenobjekten gespeichert werden. Die Methode *getResults()* greift anschließend auf die Ergebnisse zu und führt entsprechende Berechnungen durch, damit konkrete Metrikwerte erhalten werden.

Hulk.calculateMetric() bekommt als Eingabeparameter den von Git heruntergeladenen Projektordner. Hier wird *GradleImport.importGradleProject()* aufgerufen, wodurch ein IJavaProject erstellt wird. Mit diesem Projekt wird das Hulk-Plugin angesprochen und zusätzlich spezifiziert, welche Metriken Hulk berechnen soll. In diesem Fall sollen *Blob*, *IGAM* und *IGAT* berechnet werden. Die Ergebnisse werden in einer Liste zurückgegeben, die in der Klassenvariable *Hulk.hulk_results* gespeichert wird. *Hulk.getResults()* greift auf *Hulk.hulk_results* zu und zählt zum einen die Anzahl der *Blobs* in der Liste und extrahiert zum anderen den *IGAM*- und den *IGAT*-Wert. Die drei Metrikwerte werden mit dem zugehörigen Metriknamen als Schlüssel in einer Hashmap gespeichert. Läuft bei der Metrikberechnung etwas schief, wird in die Hashmap als Metrikwert -1 eingetragen.

SourceMeter.calculateMetric() bekommt als Eingabeparameter den Sourcecode, der im Eclipse-Workspace liegt und speichert die Ergebnisse in einem spezifizierten Ordner im Dateisystem. *SourceMeter.getResults()* bekommt als Input den Ordner mit den Ergebnissen und extrahiert zwei Dateien namens SrcMeter-Class.csv und SrcMeter-Enum.csv (FA4b). In diesen Dateien befinden sich alle Metrikwerte für jede einzelne Klasse des Evaluationprojektes. Die Methode *getResults()* iteriert über alle Metriken in den Dateien und speichert für die in der Methode festgelegten Metriken (vgl. 4 Metrikauswahl) den Durchschnittswert aller Klassen in einer Hashmap. Auch hier wird -1 in die Hashmap eingetragen, wenn bei der Metrikberechnung etwas schief gelaufen ist.

Androlyze.calculateMetric() bekommt als Eingabeparameter den Ordner, in dem die Ergebnisdatei gespeichert werden soll. Die Methode *calculateMetric()* greift auf die Klassenvariable *GradleBuild.compiled_apk* zu, in welcher die APK-Datei gespeichert ist und steuert mit der APK das Tool AndroLyze an. Sobald AndroLyze die Berechnungen abgeschlossen und die Ergebnisse in Form einer JSON-Datei von der Datenbank in einen Default-Ordner im Dateisystem gespeichert hat, wird die JSON-Datei in den spezifizierten Ordner kopiert und im Default-Ordner gelöscht. *Androlyze.getResults()* bekommt als Input die JSON-Datei (FA4c). In der Datei sind alle Rechteanforderungen der App aufgelistet und auch in welchen Klassen diese Rechte benötigt werden. Wird ein Recht angefordert, aber in keiner Klasse verwendet, handelt es sich um überprivilegierte Rechteanforderung. Die Methode *getResults()* berechnet das Verhältnis zwischen der Gesamtanzahl von Rechteanforderungen und den tatsächlich benötigten Rechten und liefert den ermittelten Wert als Metrikwert in einer Hashmap zurück. Konnte aus irgendeinem Grund kein Metrikwert berechnet werden, wird in die Hashmap -1 eingetragen.

Datenspeicherung

Die berechneten Metrikwerte werden in den jeweiligen Hashmaps an *Executer.mainProcess()* zurückgeliefert und zu einer einzigen Hashmap *Executer.metric_results* zusammengefügt. *Executer.metric_results* wird anschließend an die Klasse *Storage* für die Speicherung in eine CSV-Datei weitergeleitet (FA6a).

Metrikberechnung für verschiedene Versionen

Ein zusätzliches Feature ist die Berechnung der Metriken für verschiedene Versionen einer App. Dafür wird die Methode *Executer.mainProcess()* mit folgenden Eingabeparametern ausgeführt: ein String mit der Git-URL des zu untersuchenden Projektes und eine CSV-Datei *version_ids*, in welcher die verschiedenen Commit-IDs der jeweiligen Versionen gespeichert sind. Innerhalb von *mainProcess()* wird *Download.gitClone()* aufgerufen. Die Methode lädt mit der Git-URL das Projekt von Github herunter. Anschließend wird mit *Download.changeVersion()* die Version geändert, woraufhin alle Metriken mit dem Aufruf von *calculateMetrics()* berechnet werden (FA1d). Dieser Vorgang wird für jede Commit-ID wiederholt, die in *version_ids* gespeichert ist.

5.2.2 Berechnung der Statistik

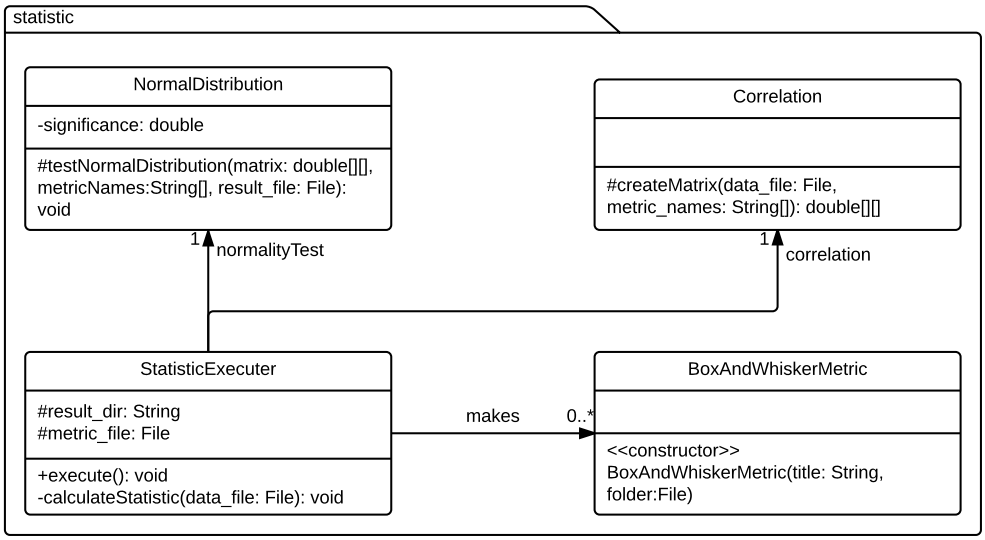


Abbildung 5.3: Klassendiagramm package statistic

Das Klassendiagramm in Abbildung 5.3 veranschaulicht die Struktur des *statistic* packages. Die Hauptklasse des Java-Paketes ist *StatisticExecuter*. Von der Methode *calculateStatistic()* aus werden alle Statistik-Tasks gesteuert. Die Klassenvariable *metric_file* stellt die CSV-Datei mit allen Metrikwerten dar, die von *metricTool.Executer* erstellt wird.

Datenauswertung und -speicherung
In den Klassen *NormalDistribution*, *Correlation* und *BoxAndWhiskerMetric* sind drei statistische Funktionen implementiert. Der Aufruf von *NormalDistribution.testNormalDistribution()* testet jede einzelne Metrik darauf, ob eine Normalverteilung vorliegt (FA5a). Dafür wird der Shapiro-Wilk-Test (vgl. Kapitel 3) durchgeführt. Der Shapiro-Wilk-Test ist ein statistischer Signifikanztest, der überprüft, ob die zugrundeliegende Grundgesamtheit einer Stichprobe normalverteilt ist. In dieser Klasse wird die Bibliothek *flanagan.analysis.Normality* genutzt [Fla15]. Das Signifikanzniveau *significance* ist standardmäßig auf 0,05 gesetzt und kann bei Bedarf angepasst werden. Das Ergebnis wird in Form einer CSV-Datei im Dateisystem gespeichert (NFA4a).

Durch den Aufruf von *Correlation.createMatrix()* werden zwei Korrelationsmatrizen erstellt (FA5b). Aus den Werten der Datei, die in *StatisticExecuter.metric_file* gespeichert ist, wird ein zweidimensionales Array gebildet, womit zum einen Pearsons Korrelation und zum anderen Spearmans Korrelation berechnet wird. Für die Berechnung wird die *apache.commons.math* Bibliothek verwendet. Als Ausgabe werden zwei CSV-Dateien mit den jeweiligen Korrelationsmatrizen im Dateisystem gespeichert (FA6a).

Eine dritte statistische Funktion stellt die Klasse *BoxAndWhiskerMetric* dar. Mithilfe von JFreeChart wird ein Diagramm erstellt, in welchem für jedes Evaluationsprojekt alle verwendeten Metriken von SourceMeter als Boxplots dargestellt werden (FA5c). Das Diagramm wird durch den Aufruf des Konstruktors erstellt. Als Parameter wird ein Dateiordner, in dem ausgewählte Ergebnisse von SourceMeter liegen, mitgegeben. Das berechnete Diagramm wird im JPEG-Format im Dateisystem gespeichert (FA6b).

5.2.3 Systemunabhängigkeit

Das Metric-Correlation-Analysis-Tool kann auf der Basis von Windows- und Linux-Betriebssystemen genutzt werden (NFA1a). In der *Executer*-Klasse wurden zu diesem Zweck zwei Boolean-Werte *windows* und *linux* als Klassenvariablen deklariert, die automatisiert das zugrundeliegende Betriebssystem abfragen und abhängig davon auf true oder false gesetzt werden. Bei allen systemspezifischen Aufrufen kann das Betriebssystem damit abgerufen werden und das Programm wechselt in die dafür vorgesehene Verzweigung. Das Programm wurde sowohl auf einem Windows- als auch auf einem Linuxsystem installiert, getestet und die Funktionalität sichergestellt.

5.2.4 Erweiterbarkeit

Das Interface *MetricCalculator* bietet eine einfache Möglichkeit, das *Metric-Correlation-Analysis*-Tool um weitere Metriken zu erweitern (NFA2a). Für jedes externe Tool, das zusätzlich eingebunden werden soll, kann eine neue Klasse erstellt werden, die *MetricCalculator* implementiert. Dabei muss der Aufruf der Methoden *calculateMetric()* und *getResults()* der neuen Klasse in *Executer.calculateMetrics()* ergänzt werden. Die genaue Stelle, wo die Aufrufe ergänzt werden müssen, ist mit einem Kommentar im Code gekennzeichnet. Die verarbeiteten Metrikwerte, die *getResults()* liefert, werden der Klassenvariablen *Executer.metric_results* zugefügt, damit sie mit den anderen Metrikwerten zusammen an *Storage* zur Speicherung übergeben werden können.

Der Statistik-Teil des Programmes kann ebenfalls leicht erweitert werden, indem man neue Klassen einbindet, die neue statistische Funktionen implementieren (NFA2b). Die neuen Funktionen müssen durch die Methode *calculateStatistic()* in der Klasse *StatisticExecuter* angesteuert werden. An welcher Stelle dies genau geschehen muss ist nicht relevant, da die bisher implementierten statistischen Methoden unabhängig voneinander sind.

5.3 Benutzung

Um mit dem Programm *Metric-Correlation-Analysis* zu arbeiten, müssen die externen Tools installiert und verschiedene Attributwerte gesetzt werden. Für die Installation existieren Installationsanweisungen auf den spezifischen Webseiten. Für die in Abbildung 5.4 aufgelisteten Tools müssen anschließend Umgebungsvariablen

gesetzt werden, damit sie vom *Metric-Correlation-Analysis*-Tool angesteuert werden können. Außerdem muss in *metricTool.Executer* die Klassenvariable *result_dir* gesetzt werden, die den Speicherort aller Ordner und Dateien festlegt, die während des Programmverlaufs generiert werden. In *statistic.StatisticExecuter* muss der gleiche Speicherort in der Klassenvariablen *metric_file* festgelegt werden.

Tool	Variablenname	Variablenwert
Gradle	GRADLE_HOME	gradle Datei im bin-Ordner im Gradle Verzeichnis
Android SDK Tools	ANDROID_HOME	Root-Ordner im Android-SDK-Tools Verzeichnis
MongoDB	MONGOD	mongod Datei im bin-Ordner im mongo-DB Verzeichnis
SourceMeter	SOURCE_METER_JAVA	SourceMeterJava Datei im SourceMeter/ Java Verzeichnis
AndroLyze	ANDOLYZE	Root-Ordner im AndroLyze Verzeichnis

Abbildung 5.4: Zu setzende Umgebungsvariablen

6 Ergebnisse

6.1 Stichprobenbeschreibung

Um das Metric-Correlation-Analysis Tool zu testen und Datenmaterial für die Auswertung zu erhalten wurden insgesamt 90 Android-Apps aus dem Open-Source Bereich ausgewählt. Die Auswahl richtete sich nach dem Resultat der Suchanfrage „Android Applikationen Java“ auf der Plattform Github. Es wurden die ersten 90 Apps ausgewählt, die über eine *gradlew*-Datei verfügten, die für den Bau der APK-Datei benötigt wird. Die Auswertung der Ergebnisse des Tools beruhte auf einer Stichprobe von 50 Android-Apps, die fehlerfrei durch das Programm durchgelaufen sind. Der Grund für die hohe Verlustrate an Evaluationsprojekten liegt vermutlich darin begründet, dass die Apps aus dem Open-Source Bereich stammen und ihre Funktionalität nicht immer vollständig getestet wurde. Der Hauptanteil der Projekte, die nicht durchliefen, scheiterte am Gradle-Build-Prozess, der für die Metrik-Tools Hulk und AndroLyze benötigt wurde. Wenn für eine App nicht alle Metriken berechnet werden konnten, wurde sie komplett aus der Bewertung herausgenommen.

6.2 Diskussion des Repräsentationswertes der Qualitätsmetriken

In Abbildung 6.1 handelt sich um eine Auswahl der Evaluationsprojekte, deren SourceMeter-Metrikwerte als Boxplot-Diagramm dargestellt sind. Als Auswahlkriterium galt, dass die Apps verschieden groß sind und aus unterschiedlichen Bereichen stammen. Beispielsweise ist *Notepad* eine Texteditor-App, *Prey* eine Sicherheits-App und *Silence* eine Messenger-App. Das Diagramm soll einen Eindruck über die Streuung der Metrikwerte in den einzelnen Klassen verschaffen. Die Metrik LOCpC wurde hier aus Gründen der Übersichtlichkeit nicht mit dargestellt. SourceMeter berechnet die verschiedenen Qualitätsmetriken für jede Klasse einer gegebenen Sourcecode-Menge. Man erhält dadurch für jede Metrik die Anzahl von Werten, die der Anzahl der Klassen des Projektes entsprechen. Um die Ergebnisse auszuwerten wurde jedoch nur ein einziger Metrikwert benötigt, der das gesamte Projekt repräsentiert. Eine naheliegende Option war, den Durchschnitt über alle Klassenwerte zu bilden und den resultierenden Wert als repräsentativen Metrikwert für die gesamt App zu wählen. Abbildung 6.1 soll dazu beitragen, diese Entscheidung zu begründen.

Die Klassenmetrikwerte der einzelnen Apps streuen sehr stark und bei den meisten Boxplots liegen der Median und der Durchschnittswert verhältnismäßig weit

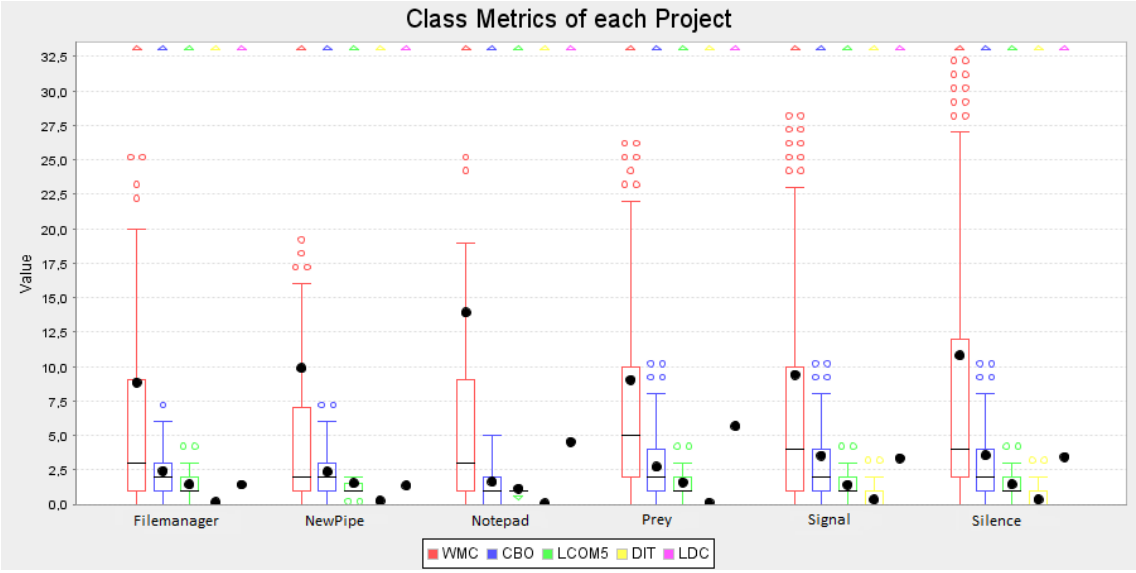
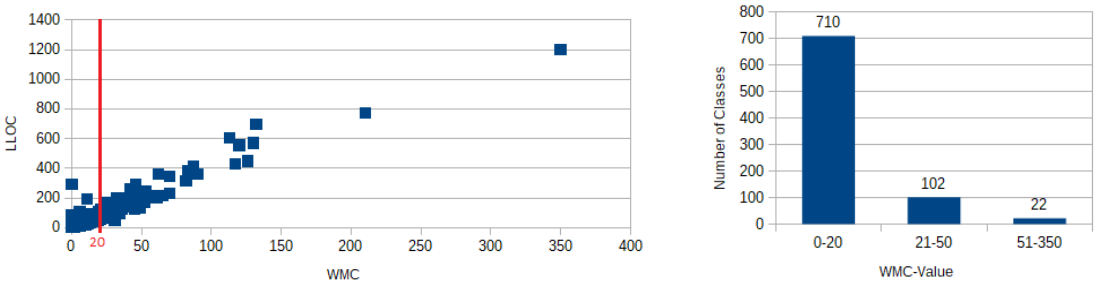


Abbildung 6.1: SourceMeter - Streuung der Klassenmetriken

auseinander. Der Median stellt den mittleren Wert aller Metrikwerte dar und berücksichtigt nicht die Größe der anderen Werte. Es wird lediglich ersichtlich, dass sich die gleiche Anzahl an Klassenwerten sowohl unterhalb als auch oberhalb des Medians befindet. Beim Durchschnitt hingegen ist die Größe jeder einzelnen Klassenmetrik im Durchschnittswert enthalten. Wenn der Median und der Durchschnitt sehr weit auseinander liegen, ist dies ein Zeichen dafür, dass es extrem hohe Ausreißer-Werte geben muss, die den Durchschnittswert nach oben ziehen.

Man kann gut erkennen, dass die Werte in Abbildung 6.1 einen weiten Zahlenbereich von 0 bis 32,5, zuzüglich der Ausreißerwerte abdecken. Dabei gilt für qualitativen Sourcecode, dass sich die Metrikwerte innerhalb bestimmter Grenzen aufhalten sollten. Es existieren Studien, die belegen, dass die Fehleranfälligkeit von Sourcecode steigt, wenn eine gewisse Grenze überschritten wird. Beispielsweise ist 20 eine validierte Metrikgrenze für WMC [Sha06]. An den rot umrandeten Boxplots erkennt man, dass das Maximum der WMC-Metrik für die meisten Apps sehr weit nach oben gezogen ist. Das obere Quartil geht dahingegen maximal bis zu einem Wert von 11. Das lässt ebenfalls darauf schließen, dass es sehr hohe Klassenwerte geben muss, die die Maximum-Grenze nach oben ziehen. Bei allen Boxplots sieht man außerdem am oberen Rand des Diagrammes ein kleines Dreieck. Das bedeutet, dass es noch weitere Ausreißer gibt, die aufgrund der Skalierung nicht mehr darstellbar sind.

Die Grafiken in Abbildung 6.2 beschreiben die Streuung der WMC-Werte für die App Silence etwas detaillierter. Bei dieser App ist das Phänomen der hohen Ausreißerwerte besonders gut sichtbar. Man sieht auf den ersten Blick, dass 710 von insgesamt 834 Klassen einen WMC-Wert innerhalb der Grenze 20 haben. Jedoch befinden sich viele Klassen weit über der maximalen Metrikgrenze, die für die Ausreißerwerte in Abbildung 6.1 verantwortlich sind. Standardmäßig sind dies auch die Klassen mit



(a) Korrelation zwischen WMC und LLOC in Klassen (b) Klassenwerte von WMC

Abbildung 6.2: Streuung der Metrik WMC am Beispiel der App Silence

den höchsten LLOC-Angaben, was an der starken Korrelation in Abbildung 6.2(a) abgelesen werden kann. Die App Silence hat einen WMC-Klassendurchschnittswert von 10,81. Da der Wert weit unter der Grenze 20 liegt, würde man vermuten, dass dies für gute Qualität spricht. Anhand des Korrelationsdiagrammes sieht man jedoch, dass dieser Wert durch viele kleine Klassen mit niedrigen WMC-Werten und einige sehr große Klassen mit hohen WMC-Werten zustande gekommen ist. Beispielsweise haben die Klassen, deren WMC-Werte unter der Grenze 20 liegen, außer zwei Ausnahmen, einen LOCpC-Wert unter 200. Dahingegen sind Klassen, deren WMC-Werte im dreistelligen Bereich liegen, doppelt so groß mit LOCpC-Werten von 400 - 1200. Wenn man den Durchschnitt über die Klassen als repräsentativen Metrikwert für die gesamte App nimmt, wird die Größe der Klasse nicht berücksichtigt. Da große Klassen aber einen Großteil des Sourcecodes ausmachen, enthalten sie gewöhnlich auch einen Großteil der Funktionalität einer App und sind für die Bewertung der Qualität besonders bedeutend. Berücksichtigt man also nicht die Größe der qualitativ mangelhaften Klassen, besteht die Gefahr, die Qualitätsbewertung zu verfälschen. Hätten wir beispielsweise eine App mit ebenfalls 843 Klassen, bei welcher jede Klasse einen ungefähr konstanten WMC-Wert von ca. 10 hat, würde diese App als qualitativ identisch mit Silence eingeschätzt werden, was ein gravierendes Missverständnis wäre.

Für die Entscheidung, welcher Wert als repräsentativer Metrikwert für die gesamte App gewählt wurde, war dieser Sachverhalt sehr entscheidend. Im Kontext dieser Ausarbeitung sollten die SourceMeter-Metriken möglichst genau die Qualität des Sourcecodes bewerten. Gerade die Klassen mit extrem hohen Metrikwerten, sind hierbei die relevanten Werte, da dies die Klassen sind, die die Qualität einer App negativ beeinflussen. Aus diesem Grund sollten die Metrikwerte großer Klassen stärker im Repräsentationswert gewichtet werden, als die Metrikwerte kleiner Klassen. Ich habe mich deshalb dazu entschieden, den Durchschnittswert nicht in Bezug auf die Anzahl der Klassen einer App, sondern auf die Anzahl der Codezeilen zu berechnen. So erhalten Klassen mit einem hohen LLOC-Wert eine stärkere Gewichtung.

Nach dieser Berechnung ergibt sich für die App Silence ein WMC-Wert von 45,69, der die Qualität dieser App präziser abbildet, als ein Wert von 10,81.

6.3 Normalverteilung

In Kapitel 3 wurde bereits erläutert, dass für die Durchführung einiger statistischer Tests eine Normalverteilung der Grundgesamtheit, aus der die Stichprobe stammt, vorausgesetzt wird. In diesem Abschnitt wird deshalb erläutert, ob für die einzelnen Metrikverteilungen eine Normalverteilung vorliegt. Das Ergebnis des Tests soll als Grundlage für die folgenden Abschnitte dienen, in denen mögliche Korrelationen untersucht werden. Zu diesem Zweck wurde der Shapiro-Wilk Test genutzt. In Abbildung 6.3 ist die statistische Ausgabe des Metric-Correlation-Analysis Tools visualisiert, die den Shapiro-Wilk Test auf die Metrikergebnisse anwendete.

Metric Name	W-Value	P-Value	Normal Distribution
LOCpC	0.88	0.00	No
WMC	0.78	0.00	No
CBO	0.84	0.00	No
LCOM	0.97	0.19	Yes
DIT	0.93	0.00	No
LDC	0.62	0.00	No
BLOB	0.78	0.00	No
IGAM	0.96	0.07	Yes
IGAT	0.92	0.00	No
MPR	0.47	0.00	No

Abbildung 6.3: Shapiro Wilk Test mit W-crit=0,96, α=0,05 und n=50

Es wurde die Nullhypothese getestet, ob eine Stichprobe n aus einer normalverteilten Grundgesamtheit stammt. Vorab mussten das Signifikanzniveau α und der kritische W-Wert festgelegt werden. Die Stichprobe n besteht aus den Ergebniswerten einer einzelnen Metrik. α wurde dabei auf 0,05 gesetzt, was ein typischer Wert hierfür ist. W-crit = 0,96 wurde abhängig vom Signifikanzniveau und der Stichprobengröße aus einer dem Test zugehörigen Tabelle entnommen [SSBW65]. Anschließend wurde abhängig von der jeweiligen Stichprobe die Teststatistik W (W-Value) und der P-Wert (P-Value) berechnet. Ist W-Value $\geq$ W-crit und P-Value $\geq \alpha$ wird die Nullhypothese angenommen. Andernfalls wird sie abgelehnt und angenommen, dass die Stichprobe n nicht aus einer normalverteilten Grundgesamtheit stammt. In den durchgeführten Tests wurde die Nullhypothese in den meisten Fällen abgelehnt. Wie in Abbildung 6.3 zu sehen ist, stammen nur die Metriken LCOM und IGAM aus normalverteilten Grundgesamtheiten. Für die restlichen Metriken darf nicht von einer Normalverteilung ausgegangen werden.

6.4 Korrelation

Im vorangegangenen Abschnitt wurde geprüft, ob eine Normalverteilung der Grundgesamtheit für die hier betrachteten Metriken vorliegt. Da diese zwingende Voraus-

setzung für den Korrelationskoeffizienten nach Pearson nicht bei allen Metriken erfüllt ist, wird in den folgenden Abschnitten die Korrelationsmatrix nach Spearman betrachtet, um mögliche Korrelationen zu untersuchen. Die Berechnung beider Korrelationsmatrizen wurde vom *Metric-Correlation-Analysis Tool* durchgeführt. Als Ausgabe wurden sowohl die Korrelationskoeffizienten nach Pearson, als auch die Korrelationskoeffizienten nach Spearman in zwei separierten Matrizen gespeichert. Die genauen Metrikwerte, auf denen die Korrelationsberechnungen beruhen, sind im Anhang in Abbildung A.1 zu finden. Eine Visualisierung der Matrix nach Spearman ist in Abbildung 6.4 zu sehen.

	LOCpC	WMC	CBO	LCOM	DIT	LDC	BLOB
LOCpC	X						
WMC	0.86	X					
CBO	0.48	0.67	X				
LCOM	0.41	0.64	0.37	X			
DIT	0.23	0.26	0.64	0.01	X		
LDC	0.62	0.66	0.59	0.42	0.25	X	
BLOB	0.29	0.47	0.73	0.34	0.42	0.51	X
IGAM	-0.01	-0.05	-0.31	0.06	-0.11	0.01	-0.27
IGAT	0.04	0.03	0.00	0.12	0.15	0.13	0.14
MPR	-0.02	-0.18	-0.39	0.07	-0.27	-0.20	-0.26

Abbildung 6.4: Matrix mit den Korrelationskoeffizienten nach Spearman

Die Felder in der Korrelationsmatrix sind entsprechend der jeweiligen Korrelationsstärke markiert. Keine bis geringe Korrelationen von 0,0 bis 0,2 sind in schwachem gelb markiert. Schwache bis mäßige Korrelationen von 0,2 bis 0,5 sind in kräftigem gelb markiert, deutliche Korrelationen von 0,5 bis 0,8 sind in orange markiert und rot steht für eine hohe bis perfekte Korrelation von 0,8 bis 1,0.

In Abhängigkeit der Stichprobengröße n wurde der P-Wert für die Korrelationskoeffizienten berechnet. Der P-Wert muss kleiner als ein festgelegtes Signifikanzniveau sein, damit die Korrelation als statistisch signifikant gilt. Für die abgebildete Matrix gilt: Bei einem Signifikanzniveau von $\alpha = 0,05$ und $n = 50$ muss der Betrag des Korrelationskoeffizienten $\geq 0,28$ sein, damit die Korrelation statistische Signifikanz hat. Bei einem Korrelationskoeffizienten von 0,28 liegt der P-Wert bei 0,049 und ist damit $<$ 0,05. Damit ist die Voraussetzung für statistische Signifikanz erfüllt. Anhand Abbildung 6.4 sieht man, dass dies bei der Korrelationsbetrachtung zwischen einzelnen Qualitätsmetriken für 17 von 21 Korrelationskoeffizienten zutrifft. Bei der Korrelationsbetrachtung zwischen Qualitäts- und Sicherheitsmetriken kann nur für 2 von 21 Korrelationskoeffizienten statistische Signifikanz angenommen werden.

Die Zahlenwerte in den ersten sieben Zeilen der Matrix beschreiben Korrelationsstärken zwischen verschiedenen Qualitätsmetriken. Auf diesen Teil wird in Zusammenhang mit der ersten Forschungsfrage in Abschnitt 6.4.1 näher eingegangen. In Abschnitt 6.4.2 werden die unteren drei Zeilen der Matrix untersucht, die von Kor-

relationen zwischen Qualitäts- und Sicherheitsmetriken handeln und zur Untersuchung der zweiten Forschungsfrage dienen. In Abschnitt 6.4.3 wird auf die dritte Forschungsfrage eingegangen, welche von Zusammenhängen zwischen Qualitäts- und Sicherheitsmetriken innerhalb verschiedener Versionen eines Softwareprojektes handelt.

6.4.1 RQ1: Korrelation der Qualitätsmetriken

In diesem Abschnitt geht es um die Analyse der ersten Forschungsfrage dieser Arbeit.

→ RQ1: Existiert eine innere Konsistenz zwischen einzelnen Qualitätseigenschaften innerhalb von Softwarecode?

Hinter dieser Forschungsfrage stehen verschiedene Annahmen. Beispielsweise ist eine Erwartung, dass Entwickler, die ein Designprinzip der Programmierung nicht einhalten, auch andere Designprinzipien nicht genau nehmen könnten. Es kann auch sein, dass ein schlecht erfülltes Designprinzip es dem Entwickler schwerer macht, andere Prinzipien einzuhalten, wodurch Korrelationen auftreten könnten. Eine positive Beantwortung der Forschungsfrage kann nur erfolgen, wenn die Mehrheit der Metriken zumindest mäßige Korrelationen aufweist. Dabei gilt, je höher die Werte der Korrelationskoeffizienten sind, desto signifikanter ist das Ergebnis. Für auftretende Korrelationen sollen logische Erklärungen gefunden werden, um Scheinkorrelationen weitestgehend auszuschließen.

Im weiteren Textverlauf werden zunächst die Korrelationen zwischen den Qualitätsmetriken in der Korrelationsmatrix nach Spearman betrachtet. Dieser Teil ist in Abbildung 6.5 erneut dargestellt. Ich gehe auf die Metriken in der Reihenfolge ein, wie sie in der Matrix aufgelistet sind und werde auftretende Korrelationen untersuchen.

	LOCpC	WMC	CBO	LCOM	DIT	LDC
LOCpC	X					
WMC	0.86	X				
CBO	0.48	0.67	X			
LCOM	0.41	0.64	0.37	X		
DIT	0.23	0.26	0.64	0.01	X	
LDC	0.62	0.66	0.59	0.42	0.25	X
BLOB	0.29	0.47	0.73	0.34	0.42	0.51

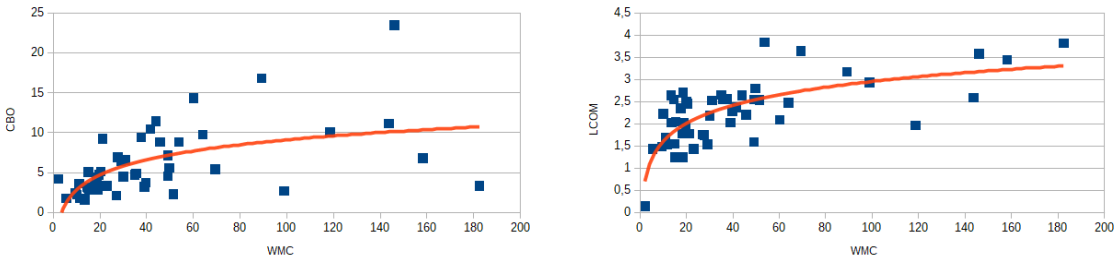
Abbildung 6.5: Ausschnitt aus Abb. 6.4 mit den Qualitäts-Korrelationskoeffizienten nach Spearman

Man kann klar erkennen, dass zwischen allen Metriken bis auf LCOM und DIT Korrelationen existieren. Zwischen LOCpC und WMC wird eine hohe Korrelation erwartet, da sich in der Berechnung von WMC die Anzahl der Methoden als Faktor befindet. Je mehr Codezeilen in einer Klasse vorhanden sind, desto wahrscheinlicher

ist auch eine höhere Anzahl der Methoden. In Abbildung 6.5 wird diese Erwartung durch einen sehr hohen Korrelationskoeffizienten von 0,86 bestätigt.

Es scheint ebenfalls ein deutlicher Zusammenhang zwischen LOCpC und LDC mit einem Korrelationskoeffizienten von 0,62 vorhanden zu sein. Auch hier ist es eine zu erwartende Entwicklung, dass mit mehr Codezeilen pro Klasse auch die Anzahl duplizierter Codezeilen steigt. Duplizierter Code ist vermutlich in jedem Softwareprojekt zu finden: Je größer das Projekt ist, desto eher ergibt sich die Möglichkeit, eine bereits implementierte Lösung noch einmal verwenden zu können und gegebenenfalls leicht anzupassen. Dies könnte eine Erklärung für den berechneten Zusammenhang liefern.

Die Metrik WMC weist außer zu LOCpC auch zu anderen Metriken deutliche Zusammenhänge auf. In den Abbildungen 6.6(a) und 6.6(b) sind die Korrelationen zwischen WMC und CBO mit einer Stärke von 0,67 und zwischen WMC und LCOM mit einer Stärke von 0,64 veranschaulicht. In beiden Grafiken kann entsprechend der logarithmischen Trendlinie ein deutlicher Zusammenhang beobachtet werden. Je höher der WMC-Wert ist, desto höher sind meistens auch der CBO- und der LCOM-Wert, was sich mit den Korrelationsberechnungen in Abbildung 6.5 deckt. Eine Klasse mit hohem WMC-Wert besteht aus vielen Methoden mit jeweils hohen Komplexitäten und ist oft unübersichtlich. Nach Abbildung 6.6(a) zu urteilen, liegt in den meisten Evaluationsprojekten ein Zusammenhang vor zwischen dem Auftreten solcher Klassen und dem Kopplungsgrad. Dieses Phänomen liegt vermutlich darin begründet, dass Klassen mit vielen komplexen Methoden sehr unstrukturiert wirken und es schwierig wird, Kopplungen zu vermeiden. Allerdings sind auch einige Punkte mit hohen WMC-Werten und sehr niedrigen CBO-Werten zu sehen. Hier kann es sein, dass Entwickler besonders viele Methoden in eine Klasse schreiben, um keine Aufrufe in andere Klassen zu haben. Dadurch wird die Kopplung schwächer, aber der WMC-Wert steigt. In Abbildung 6.6(b) ist der Korrelationszusammenhang dagegen noch klarer zu sehen. Evaluationsprojekte mit hohen WMC-Werten haben immer auch hohe LCOM-Werte.

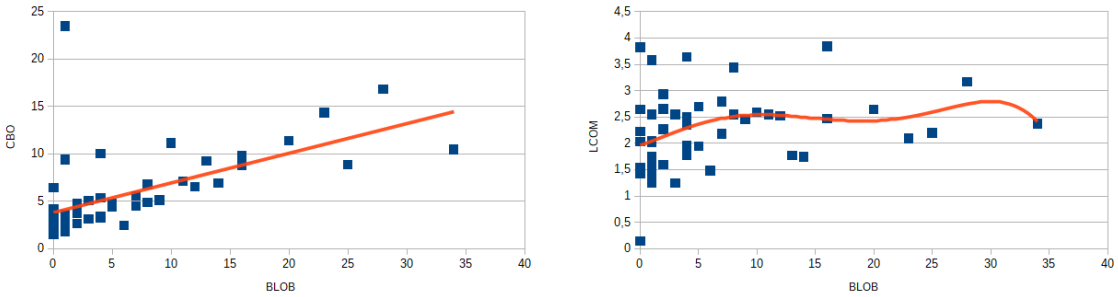


(a) Korrelation zwischen WMC und CBO (b) Korrelation zwischen WMC und LCOM

Abbildung 6.6: Deutliche Korrelationen der Metrik WMC

Der Grund hierfür ist, dass es aufgrund der Unübersichtlichkeit in Klassen mit hohem WMC-Wert nahezu unmöglich wird einen Zusammenhang zwischen der großen Menge an Methoden zu schaffen. Dadurch kommt eine deutliche Korrelation zustande. Andersherum haben Klassen mit hohen LCOM-Werten aber nicht zwangsläufig auch hohe WMC-Werte, denn auch Klassen mit wenigen Methoden können einen geringen Methodenzusammenhalt aufweisen. Dieser Zusammenhang wird durch die Punkte mit hohen LCOM-Werten und niedrigen WMC-Werten bestätigt. Zwischen WMC und LDC befindet sich mit 0,66 ebenfalls eine deutliche Korrelation. Demzufolge zeigt sich der WMC-Wert deutlich korrelativ zu anderen Qualitätsmetriken.

CBO weist ebenfalls sehr prägnante Korrelationen zu anderen Metriken auf. Die Korrelationsstärke zu DIT liegt bei 0,64 und lässt sich dadurch erklären, dass Vererbung ebenfalls eine Kopplungsart ist und in die Berechnung von CBO mit einfließt. Das heißt diese Korrelation ist zum Teil mathematisch bedingt. Aber auch zu LDC und BLOB zeigt CBO deutliche Zusammenhänge. In Abbildung 6.7 sind die Korrelationen zwischen BLOB und CBO und zwischen BLOB und LCOM zu sehen.



(a) Korrelation zwischen BLOB und CBO (b) Korrelation zwischen BLOB und LCOM

Abbildung 6.7: Zusammenhänge der Metrik BLOB

Die Trendlinie visualisiert den Zusammenhang zwischen den beiden Metriken. Mit Ausnahme von einem Punkt mit extrem hohem CBO-Wert, aber sehr geringem BLOB-Wert, entwickeln sich die anderen Punkte recht nah an der linearen Trendlinie. In Abbildung 6.7(b) hingegen kann kein klarer Zusammenhang erkannt werden, da es sehr viele Evaluationsprojekte mit hohem LCOM-Wert, aber geringem BLOB-Wert gibt. Es liegt eine starke Streuung vor und auch mit einer polinomiellen Trendlinie weichen die Punkte sehr stark von der Linie ab. Besonders bei den Evaluationsprojekten mit niedrigen BLOB-Werten sind sehr verschiedene Höhen von LCOM-Werten zu finden, was den Zusammenhang abschwächt. Diese dargestellten Zusammenhänge waren nicht zu erwarten, denn das Metrik-Tool Hulk, welches für die Berechnung von BLOB zuständig ist, verwendet zur Erkennung des *The Blob*-Antipatterns die Metrik LCOM. Das heißt, der Zusammenhang zwischen LCOM und BLOB müsste erwartungsgemäß sehr deutlich sein. Hier ist er allerdings mit einem Wert von 0,34 nur mäßig. Dass der Zusammenhang zu LCOM so gering ausfällt könnte zum Teil daran liegen, dass Hulk eine andere Berechnungsart von LCOM durchführt, als SourceMeter. Andere von Hulk berechnete Faktoren könnten

außerdem maßgeblicher zur Erkennung des *The Blob*-Antipatterns führen, als die Metrik LCOM, was die verhältnismäßig schwache Korrelation erklären würde. Dahingegen indiziert der Korrelationskoeffizient von BLOB und CBO mit einem Wert von 0,73 eine starke Korrelation, obwohl CBO nicht in Hulk genutzt wurde, um *The Blob* ausfindig zu machen. Eine Klasse, die von dem *The Blob*-Antipattern betroffen ist, beinhaltet einen Großteil des Prozesses der App und ist häufig umringt von Datenklassen. Es liegt nahe, dass der starke Zusammenhang zu CBO durch die massive Nutzung der Daten aus anderen Klassen zustande kommt. Zudem werden viele primitiv implementierte Methoden mit wenig Funktionalität aus anderen Klassen aufgerufen, wodurch sich hohe Kopplungen ergeben.

Die schwache Korrelation zwischen BLOB und LOCpC scheint auf den ersten Blick mit einem Wert von 0,29 zu überraschen. Die Vermutung, dass Klassen mit einer großen Anzahl von Codezeilen eher anfällig für ein Auftreten von *The Blob* sind liegt nahe und ist auch begründet. Von *The Blob* betroffene Klassen haben tatsächlich verhältnismäßig viele Codezeilen. Allerdings wird der LOCpC-Wert einer App als Durchschnitt über alle Klassen gebildet. Wenn eine App einige große Klassen mit *The Blob*-Antipatterns hat, aber auch sehr viele kleine Klassen besitzt, wird der LOCpC-Wert heruntergezogen. Die BLOB-Metrik wird dahingegen nicht als Durchschnitt über alle Klassen berechnet, sondern als reine Aufzählung der Vorkommnisse in der gesamten App. Das heißt, wenn der Großteil der Klassen nicht vom Antipattern betroffen ist, wirkt sich das nicht dekrementell auf den Metrik-Wert aus. Dadurch entwickelt sich der LOCpC-Wert weitgehend unabhängig zum BLOB-Wert.

Weiterhin ist es bemerkenswert, dass die schwächsten Korrelationskoeffizienten im Zusammenhang mit DIT auftreten, wie beispielsweise LCOM und DIT mit einem extrem niedrigen Wert von 0,01, DIT und LDC mit 0,25 und DIT und WMC mit 0,26. Abgesehen von CBO scheint die Länge des Vererbungspfades relativ unabhängig zu anderen Qualitätsmetriken zu sein. Da die Korrelationskoeffizienten kleiner als der Grenzwert 0,28 sind, kann man hier auch nicht von statistischer Signifikanz ausgehen.

Wie in diesem Abschnitt erläutert wurde, existieren tatsächlich deutliche Korrelationen zwischen einzelnen Qualitätsmetriken. Allerdings gibt es auch einige Zusammenhänge, die eher gering ausfallen. Da RQ1 aber von inneren Konsistenzen einzelner Qualitätsmetriken handelt, kann diese Forschungsfrage bejaht werden. Wie in Abbildung 6.5 gut zu sehen ist, existieren hier mit Ausnahme von LCOM und DIT überall mäßige oder sogar bedeutende Korrelationen.

6.4.2 RQ2: Korrelation zwischen Qualität und Sicherheit

Die zweite Forschungsfrage dieser Arbeit lautet:

→ RQ2: Existiert eine Korrelation zwischen diversen Qualitäts- und Sicherheitseigenschaften innerhalb von Softwarecode?

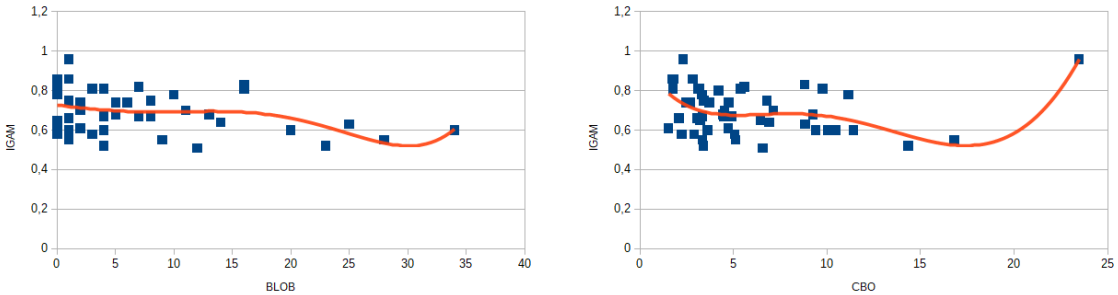
Die zugrundeliegende Erwartungshaltung ist, dass Entwickler, die nicht viel Wert auf die Qualität ihres Sourcecodes legen, auch mit der Umsetzung von Sicherheitseigenschaften nachlässig umgehen. Um auf diese Frage einzugehen, wird in Abbildung 6.8 ein Ausschnitt der Korrelationsmatrix nach Spearman gezeigt.

	LOCpC	WMC	CBO	LCOM	DIT	LDC	BLOB
IGAM	-0.01	-0.05	-0.31	0.06	-0.11	0.01	-0.27
IGAT	0.04	0.03	0.00	0.12	0.15	0.13	0.14
MPR	-0.02	-0.18	-0.39	0.07	-0.27	-0.20	-0.26

Abbildung 6.8: Ausschnitt aus Abbildung 6.4 mit Korrelationskoeffizienten von Qualitäts- und Sicherheitsmetriken

Die Diskussion der Korrelationskoeffizienten zwischen einzelnen Qualitäts- und Sicherheitsmetriken erfolgt in der Reihenfolge der Sicherheitsmetriken, wie sie in Abbildung 6.8 zu sehen ist. Anhand der farblichen Markierung kann man auf den ersten Blick feststellen, dass keine deutlichen Korrelationen zwischen den Qualitäts- und Sicherheitsmetriken existieren. Die meisten Metriken weisen keine bis eine sehr geringe Korrelation auf. Einige wenige Werte weisen auch auf einen schwachen bis mäßigen Zusammenhang hin.

Die Metrik IGAM zeigt beispielsweise eine mäßige negative Korrelation zu CBO und BLOB auf. Mit einer Stärke von -0,27 wird die Korrelation zwischen BLOB und IGAM in Abbildung 6.9(a) dargestellt. Die IGAM-Werte verlaufen ungefähr auf der gleichen Höhe zwischen 0,5 und 1, während BLOB sich auf einen Zahlenbereich von 0 bis 35 erstreckt. Es sind einige Punkte mit sehr geringem BLOB-Wert zu sehen, wo der IGAM-Wert extrem hoch ist. Dagegen haben die Punkte mit höherem BLOB-Wert etwas niedrigere IGAM-Werte. Dadurch kommt die negative Korrelation zustande. In Abbildung 6.9(b) kann ein ähnliches Verhalten beobachtet werden.



(a) Korrelation zwischen BLOB und IGAM (b) Korrelation zwischen CBO und IGAM

Abbildung 6.9: Negative Korrelationen zwischen der Sichtbarkeit und CBO und BLOB

Hier ist die Korrelation zwischen CBO und IGAM mit einer Stärke von -0,31 zu sehen. Es kann eine leichte Abwärtsentwicklung beobachtet werden, mit Ausnahme eines herausragenden Punktes, wo sowohl der IGAM- als auch der CBO-Wert

extrem hoch sind. Da es sich bei diesem Punkt um eine absolute Ausnahme handelt, ist die Aussagekraft begrenzt. Bei den anderen Punkten lässt sich die negative Korrelation recht gut festmachen. Es gibt viele Punkte nach dem Schema: Je höher der CBO-Wert, desto geringer der IGAM-Wert. Bei anderen Punkten kann jedoch keine Veränderung beim IGAM-Wert festgestellt werden, während der CBO-Wert immer größer wird. Durch diesen Sachverhalt wird die negative Korrelation etwas abgeschwächt, sodass es nur zu einer mäßigen Stärke von -0,31 kommt. Die Erklärung für beide negativen Korrelationen in Abbildung 6.9 liegt darin, dass sowohl in Klassen, wo das Antipattern *The Blob* auftritt, als auch in Klassen mit hohem CBO-Wert viele Aufrufe in andere Klassen getätigt werden. Der starke Zusammenhang zwischen BLOB und CBO wurde bereits im vorherigen Abschnitt diskutiert. Um diese Aufrufe in andere Klassen machen zu können, muss die Sichtbarkeit der Methoden, die aufgerufen werden, erhöht werden. Da diese hohe Sichtbarkeit durch den Aufruf auch tatsächlich benötigt wird, kommt es zu einem guten IGAM-Wert, denn dieser steigt nur an, wenn eine Methode sichtbarer ist, als sie unbedingt sein muss. Dadurch entsteht der Zusammenhang: Je höher der CBO- und der BLOB-Wert sind, desto niedriger ist der IGAM-Wert.

Zu anderen Qualitätsmetriken weist IGAM eine so geringe Korrelationsstärke auf, dass die Werte keine statistische Signifikanz haben. Der Grund für die geringen Korrelationskoeffizienten ist vermutlich, dass die IGAM-Werte bei allen Apps größer als 0,5 sind und damit alle für schlecht realisierte Sichtbarkeit sprechen. So ist es generell schwierig einen Zusammenhang zu den Qualitätsmetriken zu finden, da bei den Qualitätsmetriken sehr gute und auch sehr schlechte Metrik-Werte auftreten, sodass hier die Entwicklung innerhalb der Metrik viel größer ist als bei IGAM.

Als nächstes werden die Korrelationen der Metrik IGAT betrachtet. IGAT ist die einzige Sicherheitsmetrik, die keine negativen Korrelationen aufweist. Zwischen IGAT und CBO konnte absolut keine Korrelation berechnet werden, wie in Abbildung 6.8 zu sehen ist. Dieses Resultat war nicht vorherzusehen, da man wie schon zwischen CBO und IGAM eine negative Korrelation erwarten würde. Allerdings existiert hier kein Zusammenhang. Die einzigen nennenswerten, aber dennoch sehr geringen Korrelationen treten zwischen IGAT und LCOM mit 0,12, IGAT und DIT mit 0,15, IGAT und LDC mit 0,13 und IGAT und BLOB mit 0,14 auf. Da alle Koeffizienten aber unter dem Grenzwert von 0,28 liegen, liegt für diese Ergebnisse keine statistische Signifikanz vor.

Zwischen der Sicherheitsmetrik MPR und den jeweiligen Qualitätsmetriken können wieder etwas höhere Korrelationen als bei IGAT beobachtet werden. Der höchste Korrelationskoeffizient bezieht sich auf MPR und CBO, deren Zusammenhang in Abbildung 6.10 dargestellt ist. Auf den ersten Blick sind sechs Punkte zu sehen, die einen MPR-Wert von 0 haben und etwas abseits der anderen Punkte liegen. Die zugehörigen CBO-Werte reichen von 3 bis 17, was dafür spricht, dass es hier keinen Zusammenhang gibt. Bei den anderen Punkten kann aber mithilfe der polynomiellen Trendlinie ein leicht abfallendes Hauptballungsgebiet beobachtet werden, wodurch

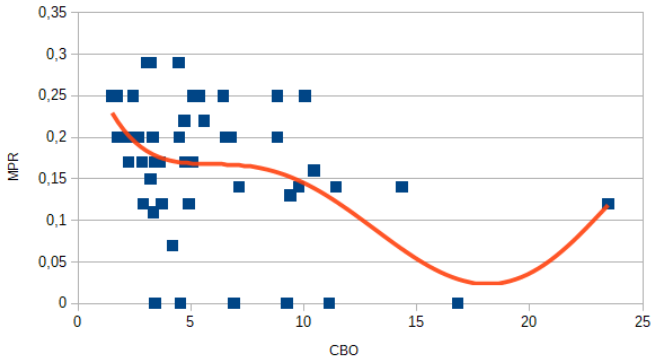
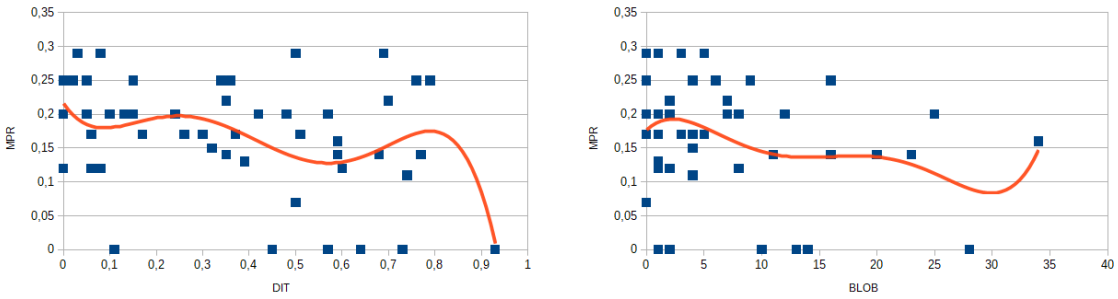


Abbildung 6.10: Zusammenhang zwischen dem Kopplungsgrad und der minimalen Rechteanforderung

der negative Korrelationskoeffizient zustande kommt. Übertragen auf die Bedeutung der hier betrachteten Metriken bedeutet dieser Zusammenhang: Je weniger das Prinzip der schwachen Kopplung beachtet wird, desto mehr ist das Prinzip der minimalen Rechteanforderung realisiert. Für diese Korrelation kann ich keine logische Erklärung finden, weshalb die Möglichkeit in Betracht gezogen werden muss, dass es sich bei der Werteverteilung um ein zufälliges Ergebnis handelt. Weitere mäßige Korrelationen existieren zu DIT und BLOB. Die Verteilung von DIT und MPR ist in Abbildung 6.11(a) zu sehen. Auch hier indizieren die Punkte mit MPR-Werten von 0 die Abwesenheit eines Zusammenhangs, denn die zugehörigen DIT-Werte reichen von sehr niedrig bis sehr hoch. Die anderen Punkte weisen auf eine mäßige negative Korrelation hin, die entlang der polynomiellen Trendlinie erkennbar ist.



(a) Korrelation zwischen DIT und MPR

(b) Korrelation zwischen BLOB und MPR

Abbildung 6.11: Negative Korrelationen der minimalen Rechtanforderung

In Abbildung 6.11(b) ist der Zusammenhang von MPR und BLOB zu sehen. Hier beträgt die Korrelationsstärke -0,26 und zeugt damit ebenfalls von einer schwachen negativen Korrelation. Die Mehrheit der Punkte hat niedrige BLOB-Werte, aber keine Tendenz was die Höhe des MPR-Wertes angeht. Dadurch ist die Korrelation in der Grafik nur schwer zu erkennen. Man kann wieder beobachten, dass die Punkte mit einem MPR-Wert von 0 sowohl niedrige, als auch hohe BLOB-Werte haben. Insgesamt wird für beide Korrelationen in Abbildung 6.11 keine statistische Signifikanz

erreicht, weshalb man davon ausgehen muss, dass zwischen der minimalen Rechteanforderung und der Länge des Vererbungspfades kein Zusammenhang besteht. Das gleiche gilt für den Zusammenhang zum Auftreten von *The Blob* Antipatterns. Der einzige positive Korrelationskoeffizient von MPR bezieht sich auf LCOM, ist aber mit einem Wert von 0,07 so gering, dass man nicht von einer Korrelation ausgehen kann. Auf Basis der vorhandenen Evaluationsprojekte existieren innerhalb der betrachteten Qualitäts- und Sicherheitsmetriken die einzigen mäßigen Korrelationen zwischen IGAM und CBO und zwischen IGAM und BLOB.

Es bietet sich noch eine weitere Betrachtungsmöglichkeit an. Anstatt die Zusammenhänge einzelner Metriken zu untersuchen, könnte der Gesamtzusammenhang aller Sicherheits- und Qualitätsmetriken untersucht werden. Dafür muss eine sinnvolle Berechnungsmethodik gewählt werden, die alle Qualitätsmetrikwerte einer App auf einen Gesamtqualitätswert projiziert. Das gleiche muss für die Sicherheitsmetriken geschehen. So kann der Zusammenhang dieser zwei Wertemengen, in denen alle einzelnen Qualitäts- und Sicherheitswerte enthalten sind, untersucht werden. Eine einfache Strategie ist, den Durchschnitt über alle Qualitäts- und über alle Sicherheitsmetriken zu bilden. Allerdings sind die Wertebereiche der Qualitätsmetriken sehr unterschiedlich und bei der normalen Durchschnittsbildung würden die Metriken mit höherem Wertebereich stärkere Auswirkungen auf den Gesamtwert haben, als die Metriken mit niedrigen Wertebereichen. Deshalb muss eine Normalisierung durchgeführt werden. Es gibt die Möglichkeit Ränge für alle Metriken zu bilden. So erhält man für alle Metriken den gleichen Wertebereich von 1 bis zur Stichprobengröße n. Für die Berechnung des Sicherheitsgesamtwertes müssen nicht unbedingt die Ränge jeder Metrik gebildet werden, da alle Sicherheitsmetriken denselben Wertebereich von 0 bis 1 haben. Hier könnte es durch eine Rangbildung eher zu einem Informationsverlust kommen, da die konkreten Abstände der einzelnen Werte durch die Rangbildung verloren gehen.

In Abbildung 6.12 wurden deshalb für den Durchschnitt der Qualitätsmetriken Ränge gebildet und für den Durchschnitt der Sicherheitsmetriken die absoluten Werte genommen. Man kann gut sehen, dass mit Ausnahme einiger weniger Ausreißer, die

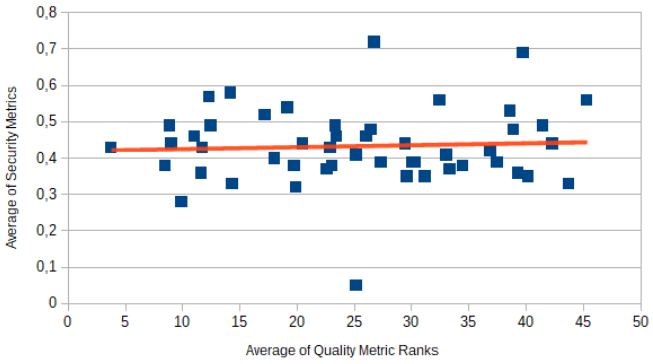


Abbildung 6.12: Korrelationsbetrachtung zwischen relativen Werten der Qualität und absoluten Werten der Sicherheit

Sicherheitswerte immer im gleichen Bereich von 0,3 bis 0,6 liegen. Bei den Qualitätswerten sind dagegen weite Streuungen von 3 bis 45 zu sehen. Wenn im Sicherheitswert keine klaren Differenzen zwischen den einzelnen Evaluationsprojekten auftreten und bei den Qualitätswerten gegensätzlich dazu deutliche Unterschiede erscheinen, ist es die einzige zulässige Schlussfolgerung, dass auf Basis der Stichprobe keine Korrelation vorhanden sein kann. Aus diesem Grund muss die Forschungsfrage RQ2 verneint werden. Anhand der betrachteten Metriken existieren nur zwei nennenswerte Korrelationen, wodurch keinesfalls auf eine allgemeine Korrelation zwischen Qualität und Sicherheit geschlossen werden kann.

6.4.3 RQ3: Korrelation zwischen Qualität und Sicherheit in verschiedenen Versionen

In diesem Abschnitt soll die letzte der drei Forschungsfragen dieser Arbeit betrachtet werden.

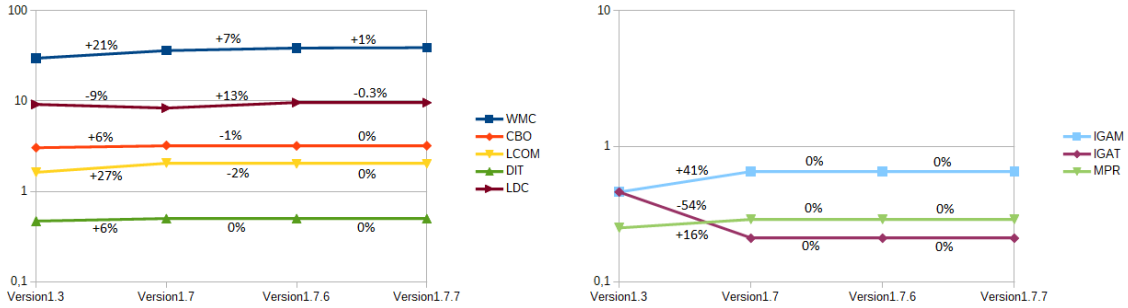
→ RQ3: Existiert eine Korrelation zwischen diversen Qualitäts- und Sicherheitseigenschaften über mehrere Versionen hinweg?

Es existiert die Annahme, dass sich ein möglicher Zusammenhang zwischen Qualität und Sicherheit zu Beginn der Entwicklung eines Softwareprojektes auf den weiteren Verlauf des Projektes auswirkt. Im Kontext dieser Arbeit bedeutet das, wenn sich von einer Version zur nächsten die Qualitätsmetriken einer App verschlechtern, dann verschlechtern sich auch die Sicherheitsmetriken. Wenn sich andersherum die Qualität positiv entwickelt, beispielsweise aufgrund von Refactoring-Maßnahmen, dann verbessert sich auch die Sicherheit. Diese Erwartungshaltung führte zur Entwicklung von RQ3, welche mithilfe von vier verschiedenen Versionen einer Android-App untersucht werden soll.

Zu diesem Zweck habe ich die App *Tinfoil-Facebook*[1] ausgewählt. Die App bietet eine Alternative zur Facebook-App, die durch die Realisierung eines Sandbox-Systems die Privatsphäre des Nutzers schützt. Durch die vorhandene Sicherheitsrelevanz bietet diese App eine gute Möglichkeit den Zusammenhang zwischen Qualität und Sicherheit zu untersuchen. Zur Auswahl dieser App führte außerdem die Tatsache, dass mehrere Versionen sowohl als Sourcecode, als auch als APK-Datei auf Github verfügbar sind. Wie bereits in Kapitel 6.1 erwähnt wurde, stellte der Gradle-Build-Prozess für viele Apps ein Problem dar. Für keine dieser Apps konnten mehrere Versionen gebaut werden, weshalb eine App ausgewählt werden musste, für die die APK-Dateien bereits für mehrere Versionen verfügbar sind. Aus diesem Grund konnten die Daten zur Beantwortung von RQ3 nicht vollständig von dem *Metric-Correlation-Analysis*-Tool berechnet werden und mussten teilweise manuell ergänzt werden. Die konkreten Metrikwerte sind im Anhang in Abbildung A.2 zu finden.

[1] https://github.com/velazcod/Tinfoil-Facebook

Abbildung 6.13 visualisiert die Metrikverläufe der App *Tinfoil-Facebook* über vier Versionen hinweg.



(a) Entwicklung der Qualitätsmetrikwerte (b) Entwicklung der Sicherheitsmetrikwerte

Abbildung 6.13: Metrikverläufe über verschiedene Versionen der App *Tinfoil-Facebook*

In Abbildung 6.13(a) ist der Verlauf der Qualitätsmetriken dargestellt. Anhand der relativen Änderungsangaben zwischen den Versionen sind teilweise starke Änderungen von 21 und 27% zu sehen. Bei der Metrik WMC kann eine durchgehende Steigung des Metrikwertes mit steigender Versionsnummer beobachtet werden. Bei allen anderen Metriken kann keine klare Tendenz in eine bestimmte Richtung festgestellt werden. LDC sinkt zuerst um 9%, steigt dann aber wieder um 13% an. CBO und LCOM verschlechtern sich von Version 1.3 auf Version 1.7, verbessern sich dann aber wieder zur Version 1.7.6 und bleiben schließlich unverändert. DIT steigt beim ersten Versionswechsel um 6% und bleibt dann gleich. Die Metrik BLOB bleibt konstant auf 0 und wird deshalb nicht mit dargestellt.

In Abbildung 6.13(b) sind die Werte der Sicherheitsmetriken visualisiert. Indem der Metrikwert von IGAM von Version 1.3 zu Version 1.7 eine Steigung um 41% hat, zeigt sich eine deutliche Verschlechterung. Auch bei MPR ist bei dem ersten Versionswechsel eine Steigung um 16% zu beobachten. IGAT zeigt im Gegensatz dazu eine deutliche Verringerung des Wertes um 54%. Von Version 1.7 bis Version 1.7.7 lassen sich bei den Sicherheitsmetriken keine Änderungen mehr beobachten.

Im vorherigen Abschnitt wurden einige mäßige Korrelationen zwischen Qualitäts- und Sicherheitsmetriken berechnet. Beispielsweise liegen zwischen der Qualitätsmetrik CBO und den Sicherheitsmetriken IGAM und MPR jeweils negative Korrelationen vor. Mit diesem Hintergrund erwartet man, dass in Abbildung 6.13 ebenfalls gegensätzliche Entwicklungen vorliegen. Das bedeutet, wenn CBO ansteigt, müssten IGAM und MPR abfallen. Bei allen drei Metrikverläufen ist jedoch eine gleiche Änderungstendenz zu beobachten, was den Erwartungen widerspricht.

Die Metrik DIT weist laut der Korrelationsbetrachtung in Abschnitt 6.4.2 ebenfalls eine negative Korrelation zu MPR auf, weshalb bei den verschiedenen Versionen ein antiproportionales Verhalten erwartet wird. Trotz der negativen Korrelation ist in Abbildung 6.13 aber ein sehr ähnlicher Werteverlauf über alle vier Versionen hin-

weg zu sehen. Beim ersten Versionswechsel steigen sowohl DIT, als auch MPR an und bleiben bei allen weiteren Versionswechseln unverändert. Dadurch ergibt sich wiederum eine unerwartete Entwicklung. Bis auf geringe Abweichungen von +-1-2% weisen die Metrikverläufe von CBO, LCOM, DIT, IGAM und MPR eine ähnliche Tendenz auf. Auch die Metrik WMC passt gut in dieses Bild rein, denn zu Beginn steigt sie sehr stark an, im weiteren Versionsverlauf aber nur noch gering. Hierbei muss allerdings beachtet werden, dass aufgrund der kleineren Versionssprünge auch die Änderungen im Sourcecode geringer werden, weshalb bei späteren Versionswechseln bei keiner Metrik mehr auffällige Änderungen auftreten.

Zwischen IGAT und LDC wurde im vorherigen Abschnitt eine Korrelationsstärke von 0,13 berechnet. Dabei handelt es sich um eine sehr geringe positive Korrelation, allerdings handelt es sich hierbei um einen der höchsten positiven Korrelationskoeffizienten, der zwischen den Qualitäts- und Sicherheitsmetriken berechnet wurde. Mit diesem Hintergrund bildet sich die Erwartungshaltung auch bei dem Versionsverlauf einer App eine ähnliche Entwicklung beider Metriken vorzufinden. Wenn man sich die Abbildungen in 6.13 anschaut, kann zwischen den Metrikverläufen von LDC und IGAT tatsächlich eine ähnliche Änderungstendenz beobachtet werden. Von Version 1.3 zu Version 1.7 verbessern sich beide Metrikwerte. Während LDC sich aber zu Version 1.7.6 hin verschlechtert und dann wieder minimal verbessert, bleibt IGAT unverändert. Allerdings sind dies die einzigen zwei Metriken, die sich beide im ersten Versionswechsel verbessern. Aufgrund dessen stimmen diese Ergebnisse teilweise mit den Erwartungen überein.

Insgesamt kann bei den Qualitätsmetriken zwischen der ersten und der letzten betrachteten Version immer eine Verschlechterung der Metrikwerte festgestellt werden. Diese Entwicklung stimmt mit den Sicherheitsmetriken bei zwei von drei Versionen überein. Dadurch lässt sich ein Zusammenhang zwischen Qualität und Sicherheit vermuten. Allerdings könnte es sein, dass dieser Zusammenhang mehr durch die Größe der Version zustande kommt, als durch das vorhandene Qualitäts- und Sicherheitsniveau. Bei den betrachteten Versionen vergrößert sich die Codebasis von 1247 auf 1465 Codezeilen. Es ist möglich, dass die Metrikentwicklung damit zusammenhängt. Anhand der erläuterten Ergebnisse kann auf Basis der untersuchten App eine ähnliche Tendenz zwischen Qualität und Sicherheit über den Verlauf mehrerer Versionen hinweg festgestellt werden. Dieser Zusammenhang kann allerdings nur für das betrachtete Evaluationsprojekt angenommen werden, da für die Verallgemeinerung auf weitere Projekte eine größere Stichprobe untersucht werden müsste. Im Kontext dieser Arbeit war dies nicht umsetzbar, da die dafür vorgesehene Automatisierung des *Metric-Correlation-Analysis* Tools aufgrund des häufig fehlgeschlagenen Gradle-Build-Prozesses nicht genutzt werden konnte.

7 Diskussion

In diesem Kapitel wird diskutiert, wie sich die Resultate aus Kapitel 6 erklären und in den Gesamtzusammenhang einordnen lassen. Dabei wird einzeln auf die Forschungsfragen eingegangen, wobei jeweils kurz die Ergebnisse zusammengefasst und die diesbezüglichen Erwartungen diskutiert werden. Anschließend wird die zugrundeliegende Methodik hinterfragt und aufgezeigt, welche Schlüsse sich daraus ziehen lassen.

7.1 Diskussion der Ergebnisse von RQ1

Mit der ersten Forschungsfrage wurde eine Korrelationsbetrachtung innerhalb der Qualitätsmetriken durchgeführt. Die Erwartung, dass die Qualitätsmetriken in einer gewissen Abhängigkeit zueinander stehen bzw. miteinander korrelieren, wurde durch das Resultat dieser Studie erfüllt. Bei insgesamt 21 Metrikpaaren konnten elf mäßige, acht deutliche und eine hohe Korrelation berechnet werden. Mit einer Stichprobe von 50 Apps konnte für 17 von insgesamt 20 aufgetretenen Korrelationen statistische Signifikanz ermittelt werden. Mit diesem Resultat wurden die Erwartungen in einem hohen Maß erfüllt und es ergibt sich die Möglichkeit, Schlüsse aus dem Ergebnis auf andere Softwareprojekte zu übertragen. Es wurde herausgefunden, dass es teilweise schwierig wird, gewisse Designprinzipien der Objektorientierung zu erfüllen, wenn ein anderes Prinzip bereits verletzt wurde, da hier viele Abhängigkeiten existieren. Dadurch ergibt sich der Rückschluss für Softwareentwickler, dass es wichtig ist, alle bewährten Designprinzipien der Objektorientierung zu beachten, um qualitativen Code zu entwickeln.

Bei den betrachteten Qualitätsmetriken handelt es sich um typische Metriken der Objektorientierung. WMC, CBO, LCOM und DIT gehören zum sogenannten CK-Metrik-Set und gelten als vielseitig eingesetzte und validierte Metriken im Kontext der Objektorientierung [CK94]. LDC und BLOB ergänzen das Metrik-Set um weitere wertvolle Qualitätseigenschaften. Mit dieser Auswahl an Metriken lässt sich die Qualität eines Softwareprojektes gut abbilden und bewerten. Man kann demnach davon ausgehen, dass die aufgetretenen Korrelationen bei den betrachteten Android Apps auf eine größere Grundgesamtheit übertragbar sind. Da die Metriken bereits in verschiedenen anderen Arten objektorientierter Softwareprojekte getestet wurden, sind die resultierenden Schlüsse nicht nur auf die Grundgesamtheit aller in Java entwickelten Android Apps übertragbar, sondern auch generell bei objektorientiertem Softwarecode. Man kann demnach davon ausgehen, dass eine innere Konsistenz zwischen den meisten Qualitätsmetriken existiert.

7.2 Diskussion der Ergebnisse von RQ2

Auf der zweiten Forschungsfrage lag der Hauptfokus dieser Arbeit. Es sollte der Zusammenhang zwischen Qualität und Sicherheit untersucht werden. Die Erwartung hinter dieser Forschungsfrage war, dass ein Zusammenhangsnachweis das Messen von Sicherheit vereinfachen könnte. Auf Basis der vorhandenen Stichprobe von 50 Apps konnten allerdings keine bedeutenden Korrelationen berechnet werden. Auch aus den mäßigen Korrelationen, für die statistische Signifikanz berechnet wurde, können keine Schlüsse gezogen werden, die diese Erwartung erfüllen würden.

Der Grund für dieses unerwartete Ergebnis liegt vermutlich in der Auswahl der Sicherheitsmetriken. In Kapitel 4 wurden bei der Aufzählung der Sicherheitsmetriken bereits die damit verbundenen Herausforderungen erläutert. Es stellte sich als besonders schwierig heraus, Metriken zu finden, die die Sicherheit von Softwarecode statisch berechnen. Bei der letztendlich resultierenden Auswahl handeln zwei der drei Sicherheitsmetriken von der Sichtbarkeit der Methoden und Typen im Sourcecode. Die Sichtbarkeit von Java-Typen auf einem minimalen Niveau zu halten, trägt eindeutig zur Sicherheit des Sourcecodes bei. Je offener ein System ist, desto mehr Angriffsfläche bietet es für unbefugten Zugriff. Wenn Entwickler bereits intern wenig Wert auf minimale Sichtbarkeit legen, liegt die Vermutung nahe, dass sie auch nach außen hin mehr Offenheit zeigen, als mit der Sicherheit des Systems zu vereinbaren ist. Allerdings ist die minimale Sichtbarkeit auch ein Anspruch von qualitativem Code, weshalb beide Metriken auch als Qualitätsmetriken interpretiert werden könnten. Ein weiterer Schwachpunkt der Metrikauswahl liegt darin, dass durch keine der verwendeten Sicherheitsmetriken konkrete Schwachstellen gefunden werden. Bei allen drei Metriken handelt es sich um die Abbildung einiger Sicherheitseigenschaften des Sourcecodes, die dazu führen könnten, dass Schwachstellen auftreten, aber es wird nicht konkret nach welchen gesucht. Dieser Umstand könnte dazu beigetragen haben, dass keine bedeutenden Korrelationen zwischen den Qualitäts- und Sicherheitsmetriken berechnet wurden. Die Anzahl der verwendeten Metriken ist außerdem zu gering, um ein ausreichendes Bild des Sicherheitsniveaus einer App zu erhalten. Deshalb war es schwierig, den allgemeinen Zusammenhang von Qualität und Sicherheit zu untersuchen und man musste mehr auf die Zusammenhänge einzelner Metriken zueinander eingehen.

Die Auswahl der Metriken hat außerdem von vornherein das mögliche Ausmaß der Übertragung der Ergebnisse verringert. Eine der drei Sicherheitsmetriken untersucht die Rechteanforderung von Apps, wodurch es nicht möglich ist, die Ergebnisse auf andere objektorientierte Anwendungsszenarien zu übertragen. Hinzu kommt, dass nur für zwei der insgesamt 21 Metrikpaare ein statistisch signifikanter Korrelationskoeffizient berechnet wurde. Das führt dazu, dass das Resultat der zweiten Forschungsfrage auch nicht auf die Grundgesamtheit von Android Apps übertragen werden kann.

Aufgrund der nicht vorhandenen Korrelationen kann man allerdings davon ausgehen, dass bei den Sicherheitsmetriken IGAM und IGAT Aspekte gemessen werden, die in den Qualitätsmetriken nicht enthalten sind. Deswegen wird empfohlen, diese beiden Metriken bei einer Qualitätskontrolle immer mitzuberechnen, um ein größeres Gesamtbild der Qualität zu erhalten.

7.3 Diskussion der Ergebnisse von RQ3

Anhand der dritten Forschungsfrage sollte der Zusammenhang zwischen Qualität und Sicherheit innerhalb verschiedener Versionen der gleichen App betrachtet werden. Die Erwartung hinter dieser Frage war, dass sich ein möglicher Zusammenhang zu Beginn der Softwareentwicklung auf den weiteren Entwicklungsverlauf auswirkt. Das würde heißen, dass ein einmal aufgetretener Zusammenhang im späteren Versionsverlauf verstärkt wird oder zumindest bestehen bleibt. Dadurch kommt es für Entwickler zu erhöhten Herausforderungen, die Qualität bzw. die Sicherheit ihres Projektes zu späteren Zeitpunkten (wieder) herzustellen.

Auf Basis der betrachteten App konnte ein tendenzieller Zusammenhang zwischen Qualität und Sicherheit festgestellt werden. Bei allen Qualitätsmetriken und bei zwei Sicherheitsmetriken konnte eine Verschlechterung der Metrikwerte zwischen der ersten und der letzten betrachteten Version beobachtet werden. In den mittleren Versionen waren teilweise unterschiedliche Entwicklungen zu sehen, aber zwischen einzelnen Metrikpaaren konnten ähnliche Tendenzen identifiziert werden. Dieses Resultat stimmt mit den oben genannten Erwartungen überein, allerdings ist die Stichprobe zu gering um Rückschlüsse auf weitere Softwareprojekte zuzulassen.

Die größte Herausforderung bei dieser Forschungsfrage war es, Datenmaterial zu erhalten. Die Idee war, dafür die Versionsverwaltung auf Github zu nutzen. Da die Evaluationsdaten der ersten beiden Forschungsfragen auf Apps der Plattform Github basieren, sollten anhand ausgewählter Commit-IDs verschiedene Versionen der gleichen Projekte heruntergeladen werden. Für die einzelnen Versionen sollten anschließend ebenfalls alle Metrikwerte berechnet werden. Zu diesem Zweck wurde eine zusätzliche Funktionalität im *Metric-Correlation-Analysis*-Tool implementiert, die diesen Prozess automatisiert steuert. Das unerwartete Problem an dieser Strategie war, dass der Gradle-Build-Prozess, der für den Erhalt der APK-Datei durchgeführt werden musste, sehr oft fehlschlug. Dadurch wurde bereits die Stichprobe der ersten beiden Forschungsfragen maßgeblich verkleinert. Für den Datenerhalt der dritten Forschungsfrage war dies noch gravierender. Es konnten keine Commit-IDs gefunden werden, bei denen der Gradle-Build-Prozess erfolgreich durchgeführt werden konnte.

Aus diesem Grund musste die Strategie verworfen werden und als Notfallplan auf eine App zurückgegriffen werden, bei welcher sowohl Sourcecode als auch APK-Dateien in verschiedenen Versionen verfügbar sind. Aus Zeitmangel und fehlender Verfügbarkeit weiterer Apps, die diese Voraussetzung erfüllen, konnten keine weite-

ren Evaluationsprojekte im Zusammenhang mit der dritten Forschungsfrage untersucht werden. Das führt dazu, dass sich anhand der betrachteten App nur Tendenzen erahnen, aber keine klaren Aussagen bezüglich des Zusammenhangs von Qualität und Sicherheit innerhalb verschiedener Versionen treffen lassen.

8 Fazit

In Kapitel 8.1 werden die grundlegenden Aspekte der Arbeit zusammengefasst. Damit soll ein Überblick über die praktische Errungenschaft der Bachelorarbeit und den Beitrag der Studie zur Wissenschaft verschafft werden. Ein Ausblick über zukünftige Erweiterungen, die im Bereich dieser Arbeit möglich sind, wird in Kapitel 8.2 gegeben.

8.1 Zusammenfassung

In dieser Bachelorarbeit wurde eine empirische Studie über die mögliche Korrelation zwischen Qualitäts- und Sicherheitseigenschaften von Objektorientierten Softwaredesigns durchgeführt. Um automatisiert Datenmaterial für die Auswertung zu erhalten, wurde ein Metric-Correlation-Analysis-Tool entwickelt. Das Tool ist für die Analyse von Android-Apps ausgelegt und integriert zum jetzigen Standpunkt drei externe Tools, die insgesamt sieben Qualitätsmetriken und drei Sicherheitsmetriken berechnen. Des Weiteren ist in dem Tool eine statistische Auswertung implementiert. Neben der Berechnung von Korrelationsmatrizen nach Spearman und Pearson, wird der Shapiro-Wilk-Test automatisiert zum Testen einer Wertemenge auf Normalverteilung durchgeführt. Außerdem ist auch eine Visualisierung der Metrikwerte als Boxplotdiagramm realisiert.

Die Vermutung, dass die Qualität von Software-Designs ein Maß für die Sicherheit der Software ist, wurde bereits häufig in der Forschung geäußert und bildete gewissermaßen die Grundlage dieser empirischen Studie [LT06, AFC10, CCZ08]. Das übergeordnete Ziel war es, diesen Zusammenhang zwischen Qualität und Sicherheit zu untersuchen und an einer Anzahl realer Softwareprojekte herauszufinden, ob er tatsächlich auftritt. Ausgehend von drei Forschungsfragen wurden deshalb die innere Konsistenz einzelner Qualitätsmetriken und die Korrelation zwischen Qualitäts- und Sicherheitsmetriken auf Basis einer Stichprobe von 50 Apps analysiert. Außerdem wurde der Zusammenhang zwischen Qualitäts- und Sicherheitsmetriken über verschiedene Versionen einer einzigen App hinweg untersucht. Dabei wurde festgestellt, dass zwischen den meisten Qualitätsmetriken signifikante Korrelationen existieren. Durch dieses Resultat lässt sich belegen, wie wichtig es für Softwareentwickler ist, sich an bewährten Designprinzipien zu orientieren und diese in ihrer Gesamtheit umzusetzen. Denn das Missachten eines Designprinzips kann durch auftretende Abhängigkeiten zu anderen Designprinzipien gravierende Auswirkungen auf die Gesamtqualität des Sourcecodes haben.

Bei der Korrelationsbetrachtung zwischen Qualität und Sicherheit konnten keine signifikanten Ergebnisse ermittelt werden. Dadurch lässt sich die Vermutung, dass die Sicherheit eines Programmes gewissermaßen mit der Programmqualität gemessen werden kann, nicht bestätigen. Allerdings wurde diese Vermutung auch nicht widerlegt, weshalb der Anreiz bestehen bleibt, weitere Studien in diesem Bereich durchzuführen. Außerdem wird empfohlen, die Sicherheitsmetriken IGAM und IGAT immer zusätzlich zur Messung der Qualität von Sourcecode einzusetzen. Man kann davon ausgehen, dass hier nochmal neue Aspekte der Qualität gemessen werden, die in den anderen Qualitätsmetriken nicht abgedeckt sind, da hier keine Korrelationen auftraten. So wird ein detaillierteres Gesamtbild der Qualität erhalten und gleichzeitig mehr Sicherheit umgesetzt.

8.2 Ausblick

Es existieren verschiedene Möglichkeiten, das in dieser Arbeit implementierte *Metric-Correlation-Analysis*-Tool, sowie die Korrelationsstudie zu erweitern.

Integration weiterer Metriken
Wie bereits in Kapitel 5 erläutert, existieren einfache Möglichkeiten, das Metric-Correlation-Analysis-Tool um weitere Metrik-Tools zu erweitern. Besonders im Bereich der Sicherheit wäre es vorteilhaft weitere Metriken einzubinden, um eine höhere Aussagekraft der Studie zu erreichen.

Integration einer Datenbank
Für eine effiziente Datenspeicherung ist es sinnvoll alle Ergebnisse in einer Datenbank zu speichern. MongoDB ist bereits integriert, wird bisher aber nur für AndroLyze im Hintergrund genutzt. Während des Programmdurchlaufs entsteht durch die heruntergeladenen Evaluationsprojekte, sowie die Zwischenergebnisse der einzelnen Tools viel Datenmaterial, welches bisher im lokalen Dateisystem gespeichert wird. Für die aktuelle Anzahl von 50 Android-Apps reicht dies aus. Will man die Anzahl aber um ein Vielfaches steigern, ist eine Umstellung auf Datenbankspeicherung sehr empfehlenswert.

Erweiterung der Beschaffung von Evaluationsprojekten
Bisher ist das Metric-Correlation-Analysis-Tool darauf ausgelegt Github-Projekte anhand einer gegebenen Git-URL herunterzuladen. Eine Erweiterungsmöglichkeit wäre die Funktionalität zu implementieren, dass das Programm auch mit einer Liste von verschiedenen URLs arbeiten und selbst entscheiden kann, welches Tool es zum Herunterladen einer URL ansteuern muss. So könnten beispielsweise SVN-Projekte als zusätzliche Evaluationsprojekte mit eingebunden werden. Eine weitere Möglichkeit, die Beschaffung von Evaluationsprojekten auszuweiten, ist die Erweiterung um eine Funktion, die automatisiert passende Projekte auf angegebenen Plattformen sucht, ohne die Eingabe konkreter URLs zu benötigen.

A Weitere Informationen

	Quality								Security		
Application Name	LLOC	LOCpC	WMC	CBO	LCOM	DIT	LDC	BLOB	IGAM	IGAT	MPR
android-bankdroid	14708	61,03	29	6,43	1,54	0,79	13,06	0	0,65	0,21	0,25
Android-BluetoothSPP	1202	31,63	18,34	3,4	1,24	0,11	14,15	1	0,75	0,24	0
android-bootstrap	3409	31,56	17,51	3,2	2,35	0,32	3,2	4	0,81	0,6	0,15
android-obd-reader	2014	57,54	34,99	4,71	2,65	0,35	7,71	2	0,61	0,35	0,22
Android-Password-Store	4143	34,82	39,72	3,7	2,27	0,06	20,2	2	0,74	0,61	0,12
Android-Spotify-MVP	1337	27,85	18,61	4,45	2,7	0,69	0	5	0,68	0,41	0,29
AndroidDrawing	885	35,4	13,66	1,75	2,03	0,15	8,06	0	0,81	0,27	0,2
AndroidSlidingUpPanel	1867	116,69	182,53	3,3	3,82	0,05	19,52	0	0,78	0,2	0,2
andstatus	38202	71,67	60,12	14,34	2,09	0,77	11,5	23	0,52	0,43	0,14
animate	1693	34,55	11,59	1,78	1,52	0,42	13,58	1	0,86	0,68	0,2
anyaudio-android-app	8395	35,57	31,06	6,55	2,52	0,1	23,37	12	0,51	0,35	0,2
AppOpsX	8482	35,64	20,39	5,11	2,45	0,15	11,09	9	0,55	0,44	0,25
CalculatorApp	1597	93,94	51,5	2,3	2,54	0	126,97	1	0,96	1	0,2
cgeo	53737	38,08	89,13	16,8	3,17	0,93	32,91	28	0,55	0,43	0
Conversations	38839	63,15	146,1	23,48	3,58	0,6	27,58	1	0,96	1	0,12
cv4j	8739	53,61	18,15	3,33	1,78	0,74	16,29	4	0,67	0,67	0,11
davdroid	157	15,7	2,17	4,19	0,14	0,5	0	0	0,8	0,59	0,07
document-viewer	39377	76,31	53,71	8,81	3,84	0,76	54,51	16	0,83	0,59	0,25
EhViewer	35971	68	63,96	9,77	2,47	0,68	41,46	16	0,81	0,52	0,14
Energize	1897	36,48	15,01	3,03	1,55	0,03	10,72	1	0,66	0,53	0,29
filemanager	5728	40,34	29,99	4,47	2,18	0,13	3,78	7	0,67	0,43	0,2
FlappyCow	2470	65	49,14	4,52	1,59	0,45	5,34	2	0,7	0,44	0
fly-gdx	11452	39,63	21,24	9,24	1,77	0,73	12,08	13	0,68	0,49	0
foregroundappchecker	293	22,54	10,08	2,23	2,22	0,17	0	0	0,58	0,38	0,17
froody-android	7404	39,81	49,11	7,12	2,55	0,35	29,75	11	0,7	0,3	0,14
gandalf	1210	31,03	11,09	3,61	1,68	0,26	0,9	1	0,6	0,31	0,17
ImmersiveDetailSample	803	40,15	27,01	2,08	1,75	0,24	15,77	1	0,66	0,33	0,2
InifiniteRecyclerView	295	26,82	15,46	2,82	2,04	0,37	0	1	0,86	0,69	0,17
iosched	25016	43,13	37,6	9,39	2,56	0,39	21,6	1	0,6	0,31	0,13
kcanotify	15716	122,78	158,25	6,77	3,44	0	142,91	8	0,75	0,48	0,2
MagicLight-Controller	712	32,36	19,01	2,88	2,01	0	0	1	0,58	0,15	0,12
MaterialAbout	2112	111,16	98,89	2,67	2,93	0	6,11	2	0,74	0,44	0,2
mdb-android-application	2411	42,3	19,65	4,73	1,94	0,3	7,72	5	0,74	0,39	0,17
MicroPinner	939	67,07	23,14	3,33	1,43	0,48	9,06	1	0,55	0,56	0,2
NewPipe	11285	39,74	49,65	5,58	2,79	0,7	5,11	7	0,82	0,65	0,22
Nimbus	6787	28,16	9,5	2,44	1,48	0,02	6,79	6	0,74	0,38	0,25
notepad	5319	77,09	118,61	10,05	1,96	0,34	14,15	4	0,6	0,25	0,25
offline-calendar	758	27,07	13,44	1,52	2,64	0	3,59	0	0,61	0,46	0,25
Piclice	634	19,21	5,52	1,74	1,43	0,05	1,48	0	0,86	0,19	0,25
prey	14913	43,1	35,58	4,9	2,55	0,08	19,98	8	0,67	0,54	0,12
Quicksend	1260	34,05	14,61	3,12	2,55	0,08	13,99	3	0,81	0,52	0,29
RedReader	27605	41,08	43,99	11,41	2,64	0,59	44,43	20	0,6	0,32	0,14
Rocket,Chat,Android	15629	40,81	27,64	6,91	1,74	0,64	38,46	14	0,64	0,6	0
Signal	57534	42,71	41,59	10,44	2,37	0,59	20,06	34	0,6	0,42	0,16
Silence	40040	47,61	45,69	8,83	2,2	0,57	24,1	25	0,63	0,42	0,2
smartnavi	5176	76,12	69,32	5,38	3,64	0,36	47,55	4	0,81	0,54	0,25
sopa	4209	31,65	15,03	5,06	1,24	0,51	11,94	3	0,58	0,4	0,17
Tinfoil-Facebook	1469	69,95	38,98	3,21	2,03	0,5	9,61	0	0,65	0,21	0,29
vanilla	18620	91,72	143,71	11,13	2,59	0,57	24,64	10	0,78	0,53	0
writeily-pro	2763	33,7	19,93	3,37	2,5	0,06	10,78	4	0,52	0,28	0,17

Abbildung A.1: Ergebnisse der Metrikberechnung für die gesamte Stichprobe

App-Version	WMC	CBO	LCOM	DIT	LDC	BLOB	IGAM	IGAT	MPR
Tinfoil-Facebook-1.3	29,81	3,05	1,63	0,47	9,19	0	0,46	0,46	0,25
Tinfoil-Facebook-1.7	36,21	3,24	2,07	0,5	8,38	0	0,65	0,21	0,29
Tinfoil-Facebook-1.7.6	38,65	3,2	2,03	0,5	9,64	0	0,65	0,21	0,29
Tinfoil-Facebook-1.7.7	38,98	3,21	2,03	0,5	9,61	0	0,65	0,21	0,29

Abbildung A.2: Ergebnisse der Metrikberechnung für verschiedene Versionen der App *Tinfoil-Facebook*

Literaturverzeichnis

[acc12] Sourceforge, accessanalysis, 2012.

[AFC10] Bandar Alshammari, Colin Fidge, and Diane Corney. Security metrics for object-oriented designs. In *Proceedings of the 2010 21st Australian Software Engineering Conference*, ASWEC '10, pages 55–64, Washington, DC, USA, 2010. IEEE Computer Society.

[AIC14] Mohammed Akeyede Imam, Usman and Moses Abanyam Chiawa. On consistency and limitation of paired t-test, sign and wilcoxon sign rank test. 2014.

[And08] Ross J. Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems*. Wiley Publishing, 2 edition, 2008.

[and09] Android sdk tools. `https://developer.android.com/studio/releases/sdk-tools.html`, 2009.

[and17] Android overtakes windows for first time. Technical report, StatCounter Global Stat, April 2017.

[Ann17] App Annie. App annie market forecast 2017. TechCrunch; App Annie, March 2017.

[Ash17] Warwick Ashford. Google removes play store apps used in wirex ddos botnet, August 2017.

[BGSF15] L. Baumgärtner, P. Graubner, N. Schmidt, and B. Freisleben. Andro lyze: A distributed framework for efficient android app analysis. In *2015 IEEE International Conference on Mobile Services*, pages 73–80, June 2015.

[BMMM98] William H. Brown, Raphael C. Malveau, Hays W. SSkip"McCormick, and Thomas J. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley & Sons, Inc., New York, NY, USA, 1st edition, 1998.

[BW02] Lionel C. Briand and Jürgen Wüst. Empirical studies of quality models in object-oriented systems. volume 56 of *Advances in Computers*, pages 97 – 166. Elsevier, 2002.

[CCZ08] Istehad Chowdhury, Brian Chan, and Mohammad Zulkernine. Security metrics for source code structures. In *Proceedings of the Fourth International Workshop on Software Engineering for Secure Systems*, SESS '08, pages 57–64, New York, NY, USA, 2008. ACM.

[CK94] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.*, 20(6):476–493, June 1994.

[CM04] B. Chess and G. McGraw. Static analysis for security. *IEEE Security Privacy*, 2(6):76–79, Nov 2004.

[COO13] V. Coskun, K. Ok, and B. Ozdenizci. *Professional NFC Application Development for Android*. Wrox programmer to programmer. Wiley, 2013.

[DuP13] Neil DuPaul. Static testing vs. dynamic testing. *Intro to AppSec, Veracode*, December 2013.

[ecl04] Eclipse metrics plugin. `https://sourceforge.net/projects/metrics/`, 2004.

[Ecl16] ECLIPSE FOUNDATION: Eclipse. https://www.eclipse.org/, June 2016.

[Eik15] Ronald Eikenberg. Android-exploit schleust beliebige apps ein, February 2015.

[EL02] David Evans and David Larochelle. Improving security using extensible lightweight static analysis. *IEEE Softw.*, 19(1):42–51, January 2002.

[EOM09] William Enck, Machigar Ongtang, and Patrick McDaniel. On lightweight mobile phone application certification. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, pages 235–245, New York, NY, USA, 2009. ACM.

[FB99] M. Fowler and K. Beck. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley object technology series. Addison-Wesley, 1999.

[FCH+11] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android permissions demystified. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 627–638, New York, NY, USA, 2011. ACM.

[FHM+12] Sascha Fahl, Marian Harbach, Thomas Muders, Lars Baumgärtner, Bernd Freisleben, and Matthew Smith. Why eve and mallory love android: An analysis of android ssl (in)security. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 50–61, New York, NY, USA, 2012. ACM.

[Fla15] Michael Thomas Flanagan. Normality - java library. `https://www.ee.ucl.ac.uk/~mflanaga`, 2015.

[GHJV94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software (Adobe Reader)*. Pearson Education, 1994.

[git05] Git. `https://git-scm.com`, 2005.

[gra07] Gradle. `https://gradle.org/`, 2007.

[Gui12] H. Guihot. *Pro Android Apps Performance Optimization*. Apressus Series. Apress, 2012.

[HEK05] Joachim Hartung, Bärbel Elpelt, and Karl-Heinz Klösener. *Statistik - Lehr- und Handbuch der angewandten Statistik ; mit zahlreichen, vollständig durchgerechneten Beispielen*. Oldenbourg, Deutschland, 2005.

[HSD+04] Bill Haskins, Jonette Stecklein, Brandon Dick, Gregory Moroney, Randy Lovell, and James Dabney. 8.4.2 error cost escalation through the project life cycle. *INCOSE International Symposium*, 14(1):1723–1737, 2004.

[IF11] S. Islam and P. Falcarin. Measuring security requirements for software security. In *2011 IEEE 10th International Conference on Cybernetic Intelligent Systems (CIS)*, pages 70–75, Sept 2011.

[Int12] Intel Corporation. *Improve Fortran Code Quality with Static Analysis*, 2012.

[KVGS09] F. Khomh, S. Vaucher, Y. G. Guéhéneuc, and H. Sahraoui. A bayesian approach for the detection of code and design smells. In *2009 Ninth International Conference on Quality Software*, pages 305–314, Aug 2009.

[KVS10] Marouane Kessentini, Stéphane Vaucher, and Houari Sahraoui. Deviance from perfection is a better criterion than closeness to evil when identifying risky code. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pages 113–122, New York, NY, USA, 2010. ACM.

[LM10] Michele Lanza and Radu Marinescu. *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*. Springer Publishing Company, Incorporated, 1st edition, 2010.

[LRH] Dr. Linda, H. Rosenberg, and Lawrence E. Hyatt. Software quality metrics for object oriented system environments, a report of satc's research on oo metrics.

[LT06] Michael Yanguo Liu and Issa Traore. Empirical relation between coupling and attackability in software systems:: A case study on dos. In *Proceedings of the 2006 Workshop on Programming Languages and Analysis for Security*, PLAS '06, pages 57–64, New York, NY, USA, 2006. ACM.

[Mar04] Radu Marinescu. Detection strategies: Metrics-based rules for detecting design flaws. In *Proceedings of the 20th IEEE International Conference on Software Maintenance*, ICSM '04, pages 350–359, Washington, DC, USA, 2004. IEEE Computer Society.

[Mey06] M. Meyer. Pattern-based reengineering of software systems. In *2006 13th Working Conference on Reverse Engineering*, pages 305–306, Oct 2006.

[MGDT10] Naouel Moha, Anne-Françoise Le Guéhéneuc, Yann-Gaël and Meur, Laurence Duchien, and Alban Tiberghien. From a domain analysis to the specification and detection of code and design smells. *Formal Aspects of Computing*, 22(3):345–361, 2010.

[MHTK15] Nadja Menz, Petra Hoepner, Jens Tiermann, and Frank Koußen. Safety und security aus dem blickwinkel der öffentlichen it. 2015.

[mob16] Mobile and tablet internet usage exceeds desktop for first time worldwide. Technical report, StatCounter Global Stats, November 2016.

[mon09] Mongodb. `https://www.mongodb.com/de`, 2009.

[Par72] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, December 1972.

[Par94] David Lorge Parnas. Software aging. In *Proceedings of the 16th International Conference on Software Engineering*, ICSE '94, pages 279–287, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[PKLS16] Sven Peldszus, Géza Kulcsár, Malte Lochau, and Sandro Schulze. Continuous detection of design flaws in evolving object-oriented programs using incremental multi-pattern matching. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, pages 578–589, New York, NY, USA, 2016. ACM.

[Ree92] Jack W. Reeves. What is software design? *C++ Journal*, 1992.

[Sch12] K. Schneider. *Abenteuer Softwarequalität: Grundlagen und Verfahren für Qualitätssicherung und Qualitätsmanagement*. dpunkt.verlag, 2012.

[SH09] L. Sachs and J. Hedderich. *Angewandte Statistik: Methodensammlung mit R*. Springer, 2009.

[Sha06] Raed Shatnawi. An investigation of ck metrics thresholds. 01 2006.

[sou15] Sourcemeter. `https://www.sourcemeter.com`, 2015.

[SS97] A. Sen and M. Srivastava. *Regression Analysis: Theory, Methods, and Applications*. Springer Texts in Statistics. Springer New York, 1997.

[SSBW65] Samuel S. Shapiro and Martin B. Wilk. An analysis of variance test for normality. 52:591–, 12 1965.

[SW08] Yonghee Shin and Laurie Williams. Is complexity really the enemy of software security? In *Proceedings of the 4th ACM Workshop on Quality of Protection*, QoP '08, pages 47–50, New York, NY, USA, 2008. ACM.

[SW11] Yonghee Shin and Laurie Williams. An initial study on the use of execution complexity metrics as indicators of software vulnerabilities. In *Proceedings of the 7th International Workshop on Software Engineering for Secure Systems*, SESS '11, pages 1–7, New York, NY, USA, 2011. ACM.

[TKZ+13] Seyyed Ehsan Salamati Taba, Foutse Khomh, Ying Zou, Ahmed E. Hassan, and Meiyappan Nagappan. Predicting bugs using antipatterns. In *Proceedings of the 2013 IEEE International Conference on Software Maintenance*, ICSM '13, pages 270–279, Washington, DC, USA, 2013. IEEE Computer Society.

[ZWN+06] Jiang Zheng, Laurie Williams, Nachiappan Nagappan, Will Snipes, John P. Hudepohl, and Mladen A. Vouk. On the value of static analysis for fault detection in software. *IEEE Trans. Softw. Eng.*, 32(4):240–253, April 2006.