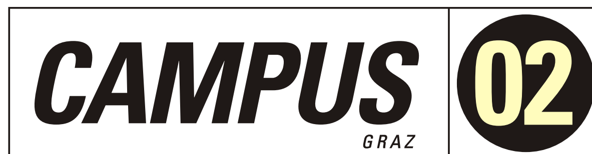
MASTERARBEIT

Erstellung und Anwendung von Metriken zur Gegenüberstellung von Softwaretestverfahren anhand ausgewählter Randbedingungen

ausgeführt am



Studiengang
Informationstechnologien und Wirtschaftsinformatik

Von: David Hitthaler
Pers. Kennz. 1810320016

Graz, am 5. Juli 2020

....
David Hitthaler

Ehrenwörtliche Erklärung

Ich erkläre ehrenwörtlich, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen nicht benützt und die verwendeten Quellen wörtlich zitiert sowie inhaltlich entnommene Stellen als solche kenntlich gemacht habe.

....
David Hitthaler

Danksagung

An dieser Stelle möchte ich mich bei all denjenigen bedanken, die mich während der Anfertigung dieser Masterarbeit unterstützt und motiviert haben.

Beginnen möchte ich mit Herrn DI(FH) Günther Zwetti, der meine Masterarbeit betreut hat. Für die hilfreichen Anregungen und die konstruktive Kritik von der Themeneingrenzung bis hin zur Fertigstellung dieser Arbeit möchte ich mich herzlich bedanken.

Ein herzlicher Dank gilt allen ExpertInnen, ohne die diese Arbeit nicht hätte entstehen können. Mein Dank gilt ihrer Informationsbereitschaft und ihren interessanten Beiträgen, Kommentaren und Antworten auf meine Fragen.

Ich danke meinen Vorgesetzten der Firma BearingPoint Technology GmbH für die Möglichkeit der flexiblen Zeitgestaltung und Urlaubseinteilung, um Studium und Arbeit in Einklang bringen zu können!

Abschließend möchte ich mich besonders bei meiner Frau, Mag.ª Theresa Hitthaler-Frank, für ihre besondere und immerwährende Unterstützung im Studium und für das Korrekturlesen der Arbeit und die daraus resultierenden Tipps und Hinweise herzlich bedanken.

Kurzfassung

Die Anforderungen an die moderne Softwareentwicklung sind hoch. Softwareunternehmungen stehen angesichts einer fast schon unübersichtlich großen Anzahl an Methoden und Entwicklungswerkzeugen vor der Herausforderung, die für ihre Anforderungen und Abläufe idealen Prozesse und Werkzeuge zu wählen. Diese Wahl ist essentiell, um die richtige Nische am Markt effizient und langfristig zu bedienen sowie auf Änderungen noch schneller reagieren zu können.

Die vorliegende Masterarbeit beschäftigt sich mit der Erstellung von Metriken, um Softwaretestverfahren miteinander vergleichbar zu machen. Ausgehend von der Literatur werden verschiedene Testverfahren beschrieben, analysiert und auf das Vorhandensein von Klassifizierungs- und Kategorisierungsmerkmalen für das Erstellen der Metriken überprüft. Durch die Kombination von Literaturrecherche und ExpertInneninterviews werden diese Ergebnisse Schritt für Schritt verfeinert. Im Anschluss wird eine Bewertungsmatrix erstellt und mit konkreten Werten auf Basis des ExpertInnenwissens angereichert.

Diese Masterarbeit legt den Grundstein für weitere Untersuchungen in EntwicklerInnenteams, um dort das Delta zwischen dem Ist-Stand und dem Soll-Stand bei der Verwendung der aktuellen Softwaretests zu ermitteln. Die sich daraus ergebenden Optimierungspotenziale können untersucht und in den jeweiligen Teams umgesetzt werden. Das Feedback der Teams bei der Umsetzung der Optimierungen ist ein weiterer möglicher Untersuchungsgegenstand.

Somit ist klar ersichtlich, dass der Vergleich von Testverfahren einen akademischen, aber auch stark praxisorientierten Anwendungsfall darstellt.

Abstract

The demands on software development are high. In view of an almost confusingly large number of methods and development tools, software companies face the challenge of choosing the ideal processes and tools for their requirements and procedures. This choice is essential in order to serve the right niche in the market efficiently and in the long run to be able to react faster to the changes.

This master thesis deals with the creation of metrics to make software test procedures comparable. Starting from the academic literature, different test procedures are described, analysed and reviewed for the presence of classification and categorisation features for the creation of metrics. By combining literature research and expert interviews, these results are refined step by step. When all that is concluded, an evaluation matrix is created and enriched with concrete data based on the expert knowledge.

This master thesis lays the foundation for further investigations in developer teams to determine the delta between the actual state and the target state when using the current software tests. The resulting optimization potentials can be examined and implemented for the respective teams. The feedback of the teams during the implementation of the optimizations is another possible object of investigation.

Thus, it is clearly evident that the comparison of test procedures represents an academic, but also strongly practice-oriented implementation.

Inhaltsverzeichnis

1. Einleitung **1**
1.1. Softwaretests . . . 1
1.2. Ausgangssituation . . . 1
1.3. Zielsetzung und Forschungsfragen . . . 2
1.4. Aufbau der Arbeit und Methodik . . . 3

2. Grundlagen und Definitionen **4**
2.1. Definition des Begriffs Software und Bedeutung für die Gesellschaft . 4
2.2. Auswirkungen von Fehlern und Motivation, Software zu testen 5
2.3. Definition des Begriffs Fehler . . . 5
2.3.1. Weitere Begrifflichkeiten rund um den Terminus Fehler 6
2.3.2. Fehlerkategorien . . . 7
2.3.3. Arten von Softwarefehlern . . . 11
2.3.4. Eigenschaften von Softwarefehlern . . . 13
2.4. Qualität von Software . . . 14
2.4.1. Funktionalität . . . 15
2.4.2. Zuverlässigkeit . . . 16
2.4.3. Effizienz bzw. Laufzeit . . . 17
2.4.4. Benutzbarkeit . . . 18
2.4.5. Übertragbarkeit . . . 18
2.4.6. Änderbarkeit bzw. Wartbarkeit . . . 19
2.4.7. Testbarkeit . . . 19
2.4.8. Transparenz . . . 20
2.4.9. Fazit . . . 21
2.5. Definition des Begriffs Softwaretest . . . 23
2.6. Ziele und Nichtziele von Softwaretests . . . 24

3. Die Testklassifikationen **25**
3.1. Die vier Prüfebenen . . . 26
3.1.1. Der Modul-, Komponenten- bzw. Unittest . . . 26
3.1.2. Der Integrationstest . . . 27
3.1.3. Der Systemtest . . . 31
3.1.4. Der Abnahmetest . . . 33
3.2. Die Prüfkriterien . . . 34
3.2.1. Der funktionale Softwaretest . . . 35
3.2.2. Der operationale Softwaretest . . . 36
3.2.3. Der temporale Softwaretest . . . 37
3.2.4. Der strukturbezogene Test . . . 38
3.2.5. Die änderungsbezogenen Tests . . . 38

3.3. Die Prüftechniken bzw. -methoden 39
3.3.1. Black-Box Testverfahren 39
3.3.1.1. Der Äquivalenzklassentest 40
3.3.1.2. Die Grenzwertbetrachtung 42
3.3.1.3. Der zustandsbasierte Softwaretest 43
3.3.1.4. Der Use-Case Test 45
3.3.1.5. Der entscheidungstabellenbasierte Test 48
3.3.1.6. Das paarweise Testen 51
3.3.1.7. Diversifizierende Verfahren 52
3.3.2. White-Box Testverfahren 54
3.3.2.1. Der Anweisungs- oder C_0 - Test 54
3.3.2.2. Der Zweig- bzw. C_1 - Test 56
3.3.2.3. Der Pfad- bzw. C_2 - Test 58
3.3.2.4. Der Bedingungstest 59
3.3.2.5. Die McCabe Überdeckung bzw. the basic path test . . 60
3.3.2.6. Die Defs-Uses Überdeckung 61
3.3.3. Erfahrungsbasierte Testverfahren 63
3.3.3.1. Intuitive Testfallermittlung bzw. Error Guessing . . . 63
3.3.3.2. Exploratives Testen 64

4. Metriken für die Testverfahren **65**
4.1. Methodik zur Ermittlung der Metriken 65
4.2. Die ExpertInneninterviews 66
4.2.1. Definition des Zieles 66
4.2.2. Ableitung des Leitfadens 67
4.2.3. Suche nach ExpertInnen 67
4.2.4. Durchführung und Informationsmanagement 69
4.3. Aufstellung und Beschreibung der Metriken 70
4.3.1. Kategorie: Anwendbarkeit 70
4.3.2. Kategorie: Komplexität 70
4.3.3. Kategorie: Abhängigkeit 71
4.3.4. Kategorie: Automatisierbarkeit 72
4.3.5. Kategorie: Aufwand des Testings 73
4.3.6. Kategorie: Nutzen des Testings 75

5. Resultate und Empfehlungen **77**
5.1. Ausgangslage 77
5.2. Anwendungsszenarien 80
5.3. Conclusio und Ausblick 82

A. ExpertInneninterviews **84**

B. Fragebögen **91**

C. Bewertungsmatrix **94**

Abbildungsverzeichnis **97**

Tabellenverzeichnis **98**

Listings **100**

Literaturverzeichnis **101**

1. Einleitung

1.1. Softwaretests

Seit die Bedeutung von Tests in der Softwareentwicklung erkannt wurde, sind diese oftmals ein fixer Bestandteil von Entwicklungsprozessen. Durch agile Praktiken wird dieses Verhalten noch beschleunigt.

Dennoch besteht zwischen dem "Soll-Stand", oft beschrieben in Literatur und Forschung, sowie dem "Ist-Stand", meist gelebt in der Praxis, eine gewisse Diskrepanz. Dies liegt nicht zuletzt an den mannigfaltigen Möglichkeiten in der IT-Branche, mit Softwaretests unterschiedlich umzugehen. Es ist oft nicht möglich, diese Tests über unterschiedliche Anforderungen hinweg vergleichbar zu machen. Weiters erschwert die Masse an Programmiersprachen sowie Entwicklungsumgebungen dies weiter. Vielfach wird auch ein "Perfektionismus" des Testprozesses angestrebt, der in der Praxis aus unterschiedlichen Gründen nicht gelebt werden kann.

Weiters werden Softwaretests oft als Overhead oder Benefit und nicht als essentielle Aufgabe gesehen, da ein Verständnis der Softwareindustrie für die Problemstellung des Testings bzw. im Umkehrschluss ein Verständnis der Forschung für die Softwareindustrie fehlt.

1.2. Ausgangssituation

Die Forschung hat von Beginn an im Bereich des Testens von Software jede Menge an Publikationen geliefert. Diese Ergebnisse wurden direkt oder in weiterverarbeiteter, aggregierter Form in unzähligen Niederschriften rund um den Globus veröffentlicht und so der breiten Masse zugänglich gemacht.

Davon profitieren die unterschiedlichen Stakeholder in der Forschung aber auch in der Softwareindustrie. Vor allem die Industrie hat sich daraus viele Handlungsweisen und Konzepte abgeleitet, um damit den Alltag in der Softwareentwicklung zu erleichtern, zu verbessern, die Effizienz zu steigern, die Kosten zu senken usw.

Heutzutage ist es außerdem möglich, das Testen von Software - vor allem theoretisch, aber auch praktisch - als Ergebnis der zuvor genannten Entwicklung zu erlernen und dafür verschiedene Nachweise in Form von Zertifikaten zu erhalten.

Eine mögliche Lücke im Wissens- bzw. Erfahrungsaustausch zwischen Industrie und Forschung besteht darin, dass zwischen Theorie und Praxis oft ein Unterschied in der Wahrnehmung und Durchführung von Softwaretests besteht. Dies kann sich z.B. dadurch äußern, dass die Industrie erforschte Ergebnisse nicht oder nur mangelhaft umsetzt bzw. umsetzten kann und Erfahrungen aus der Industrie in der Forschung nicht ankommen bzw. nicht weiter beforscht werden.

Natürlich wird es auch durch die schiere Masse an verfügbaren Informationen immer schwieriger, entsprechendes Wissen zu extrahieren und dieses für den jeweiligen Zweck der Forschung bzw. der Softwareindustrie zu verwenden. Da die Informationen im Laufe der Zeit altern, sich verändern und aktualisieren, ist und bleibt die Informationsverwaltung auch in diesem Bereich eine ständige Herausforderung.

1.3. Zielsetzung und Forschungsfragen

Aufgrund der eben dargestellten Diskrepanz zwischen Forschung und Softwareindustrie ergeben sich viele Themen, die bearbeitet werden können. Im Rahmen dieser Arbeit wird das Ziel verfolgt, verschiedene Testverfahren zu beschreiben, zu analysieren und auf das Vorhandensein von Klassifizierungs- und Kategorisierungsmerkmalen für das Erstellen von Metriken zu überprüfen. Diese Metriken sollen anhand verschiedener Parameter die jeweiligen Ansätze des Softwaretestings für Stakeholder vergleichbar machen.

Die Forschungsfrage mit den entsprechenden Hypothesen lautet wie folgt:

Inwieweit ist es möglich, konkrete Softwaretests mithilfe von Metriken vergleichbar zu machen bzw. zu optimieren?

> H1: Softwaretests lassen sich durch eine Mindestanzahl von Metriken vergleichbar machen.
> H0: Softwaretests lassen sich nicht durch eine Mindestanzahl von Metriken vergleichbar machen.

1.4. Aufbau der Arbeit und Methodik

Aufgrund der Erläuterungen im Abschnitt 1.3 ergeben sich eindeutige Meilensteine, die zur Beantwortung der Forschungsfrage erarbeitet werden sollen. Dazu gehören unter anderem:

- Erstellung von Definitionen und Ausführungen zu den Grundbegriffen, die im weiteren Verlauf der Arbeit als Basis verwendet werden
- Überblick über die verschiedenen Testmethoden
- Erstellung von Metriken, um Testmethoden vergleichen zu können
- Beantwortung und Diskussion der Forschungsfrage
- Abschließendes Fazit mit Rückblick, Status Quo und Ausblick bezogen auf die Thematik

Zu Beginn der Arbeit wird die Methode der Literaturrecherche verwendet, um einen Überblick über die aktuelle Situation des Softwaretesting zu erlangen. Die Systematisierung und Klassifikation der entsprechenden Inhalte aus verschiedenen Quellen ist wichtig, um im nächsten Schritt mithilfe von ExpertInneninterviews, Abstraktion, Selektion sowie Systematisierung die Metriken zu identifizieren, die für die Vergleichbarkeit von Softwaretests relevant sind.

Im Anschluss wird eine Bewertungsmatrix erstellt und mit konkreten Werten auf Basis des ExpertInnenwissens angereichert. Abschließend werden die in Summe gewonnenen Resultate und Ergebnisse aggregiert, diskutiert und anhand verschiedener Szenarien vorgestellt.

2. Grundlagen und Definitionen

2.1. Definition des Begriffs Software und Bedeutung für die Gesellschaft

Der Begriff Software setzt sich aus den beiden Wörtern *soft* und *ware* zusammen, was so viel wie *weiche Ware* bedeutet. Lassmann, Schwarzer und Rogge (2006, S. 127) definieren Software als „Sammelbegriff für die Gesamtheit der Programme, die zugehörigen Daten und die notwendige Dokumentation, die es erlauben, mit Hilfe eines Computers Aufgaben zu erledigen."

Die Tätigkeiten, die Computer und die darauf laufenden Softwaresysteme erledigen, werden immer umfangreicher. Im Alltag kommen Systeme, die gänzlich ohne Software arbeiten und/oder mit keinem Softwaresystem direkt oder indirekt agieren, kaum noch vor. Auch die gerade stattfindende Entwicklung weg von der Client-Server Architektur hin zum Paradigma "Computing überall", auch Pervasive Computing genannt, kurbelt diesen Prozess weiter an. Software übernimmt mehr und mehr Funktionen, die es in manchen Ausprägungen so nicht gab, oder schafft neue Anwendungsfelder und liefert so eine Grundlage für technischen und sozialen Fortschritt (Hehl, 2008).

Dadurch ist Software wie z.B. auch Energie bzw. Energieversorgung ein Schlüsselfaktor unserer modernen Gesellschaft und wird Teil der kritischen Infrastruktur, die es entsprechend der jeweiligen Einsatzgebiete und -zwecke zu schützen gilt. Daraus folgt eine wachsende Verantwortung der SoftwareentwicklerInnen sowie der Gesellschaft dafür Sorge zu tragen, Software entsprechend dem jeweiligen Anwendungsszenario gezielt und methodisch zu untersuchen, das Risiko zu bewerten und wenn nötig entsprechende Maßnahmen zu treffen (Birkmann et al., 2010).

2.2. Auswirkungen von Fehlern und Motivation, Software zu testen

Wie bereits in Kapitel 2.1 erwähnt, gibt es kaum noch Abläufe oder Systeme, die ohne Software funktionieren. Daher ist es logisch, dass Software auch Fehler enthält. Ein Beispiel für einen Fehler mit fatalen Auswirkungen liefert das Unglück des Lufthansa Flugs 2904 mit der Landung in Warschau-Okecie. Aufgrund widriger Wetterbedingungen und einer falschen Spezifikation - in diesem Fall die falschen Parameter zur Berechnung der Freigabe für die Schubumkehr an den Piloten - konnte der Flieger nicht rechtzeitig auf der Landebahn anhalten und prallte gegen den künstlichen Wall am Ende der Piste. Zwei Menschen konnten nicht mehr gerettet werden. (Hoffmann, 2013)

Solche und ähnliche Ereignisse, siehe dazu Cleff (2010), zeigen deutlich, wie wichtig es ist, Software zu testen und entsprechende Qualitätskriterien festzulegen.

2.3. Definition des Begriffs Fehler

Die Auswirkungen des Fehlers im Abschnitt 2.2 sind klar erkennbar. Schon für Belli, Pfleger und Seifert (1984, S. 1) ist ein Programm (nur) dann korrekt, „wenn es seiner Spezifikation genügt". Ebert und Dumke (1996, S. 290) definieren Jahre später den Fehler als „in der Software manifestierter falscher Inhalt, der zu einem Fehlverhalten führen kann". Weiters unterscheiden sie klar zwischen der Ursache des Fehlers, dem Entwicklungsfehler und der Wirkung desselben, auch Fehlverhalten genannt. Jahre später formulieren Spillner und Linz (2012, S. 7) eine Situation nur dann als fehlerhaft, „wenn vorab festgelegt wurde, wie die vorausgesagte, korrekte, also nicht fehlerhafte Situation aussehen soll".

Im Laufe der Zeit hat sich somit die Definition eines Fehlers an sich kaum verändert. Dennoch ist klar ersichtlich, dass unterschiedliche AutorInnen verschiedene Perspektiven des Fehlerbegriffs beleuchten.

Nach den Erläuterungen und Ausführungen des aktuellsten Werks von Spillner und Linz (2012) ist die Diskrepanz zwischen Sollzustand und Istzustand als Fehler definiert. Diese Definition stützt sich darauf, dass die Zustände bzw. möglichen Fehlerfälle im Vorfeld definiert werden müssen. Eine Situation bzw. ein Zustand, der nicht im Voraus geklärt und/oder spezifiziert wurde, kann somit trotz unerwünschtem Resultat niemals ein Fehler sein.

2.3.1. Weitere Begrifflichkeiten rund um den Terminus Fehler

Der Begriff *Fehler* wird vom *Mangel* dadurch unterschieden, dass Zweiterer als nicht angemessene Erfüllung einer gestellten Anforderung oder berechtigter Erwartung definiert wird. Weitere Begriffe, um Fehler zu beschreiben und zu kategorisieren, sind laut Spillner und Linz (2012):

- Die Fehlerwirkung
- Der Fehlerzustand
- Die Ursachenkette für Fehler
- Die Fehlermaskierung
- Die Fehlhandlung

Die *Fehlerwirkung* ist die sichtbare Fehlwirkung einer Software. Dies kann bedeuten, dass statt einem Punkt ein Beistrich ausgegeben wird. Als *Fehlerzustand* wird der *innere Fehler* bezeichnet. Dieser ist immer vorhanden und kann z.B. als fehlerhaft programmierte Schleife enthalten sein. Wichtig ist hierbei, dass der Fehlerzustand nicht auftreten muss und zwischen seinem Auftreten, der Fehlerwirkung, und seinem Vorhandensein, dem Fehlerzustand, unterschieden werden muss. Der Fehlerzustand wird oft als *Bug*, *Defekt engl. defect* oder *fault* bezeichnet (Spillner & Linz, 2012; Spillner, Roßner, Winter & Linz, 2014).

Die *Ursachenkette* beschreibt, dass Softwarefehler nicht durch Alterung entstehen, sondern seit Anbeginn der Existenz der Software, also bei deren Entwicklung, bereits zu Stande kommen. Wie bereits erwähnt, muss der Fehlerzustand nicht als Fehlerwirkung zutage treten und somit klar erkennbar sein. Ist dies der Fall nennt man das *Fehlermaskierung*. Diese Fehlermaskierung kann durch andere Fehlerzustände passieren. Eine Fehlerwirkung tritt in diesem Fall erst nach der Korrektur der anderen Fehlerzustände auf. Somit ist es möglich, dass Korrekturen zu Seiteneffekten führen können. Konkrete Tests sollen Fehlermaskierungen vermeiden (Spillner & Linz, 2012; Roitzsch, 2005).

Der Grund für den Fehlzustand liegt in einer *Fehlhandlung*, auch *error* genannt. Diese beschreibt die falsche Handlung der Person zum Zeitpunkt des Entstehens bzw. der Änderung des entsprechenden Quellcodes, der für diesen Fehlerzustand verantwortlich ist. Die Fehlhandlungen entstehen durch weitere Faktoren wie Komplexität der Software, Änderung an derselbigen, Komplexität der Infrastruktur, nicht umgesetzte Anforderungen, eine Vielzahl an Systeminteraktionen, die es zu berücksichtigen gilt, und falsches oder unvollständiges Wissen über die eingesetzte Technologie (Spillner & Linz, 2012). „Fehlhandlungen sind durch konstruktive Qualitätssicherungsmaßnahmen vermeidbar“ (Spillner, Roßner et al., 2014, S. 327).

2.3.2. Fehlerkategorien

Die Fehler einer Software können in Kategorien eingeteilt werden. Jede Kategorie liefert dabei eine Entstehungsart des Fehlers auf abstraktem Niveau. Hoffmann (2013) gliedert diese in *lexikalische und syntaktische Fehlerquellen*, *semantische Fehlerquellen*, *Parallelität als Fehlerquelle*, *numerische Fehlerquellen*, *Portabilitätsfehler*, *Optimierungsfehler*, *tickende Zeitbomben* und *Spezifikationsfehler*.

- **Lexikalische und syntaktische Fehlerquellen**
  Ohne die Möglichkeit der grafischen Programmierung zu berücksichtigen, entsteht ein Programm typischerweise durch das Anordnen von lexikalisch definieren Symbolen, engl. tokens genannt, in einer genau spezifizierten Weise. Damit wird eine entsprechende Funktionalität abgebildet. Die Symbole umfassen dabei den gesamten *Sprachschatz* der jeweiligen Programmiersprache. Der Compiler überprüft die *Grammatik*, also die Regeln der Sprache, sowie ob die Symbole an sich plausibel sind und kann somit zu lexikalischen und syntaktischen Fehlern Rückmeldungen geben (Hoffmann, 2013).

  Die lexikalische Analyse dieses Zeichenstroms wird zumeist als Standardaufgabe des Compilers gesehen und von diesem ausgeführt. Durch Unverständnis bzw. Fehlverständnis der EntwicklerInnen in Bezug auf die Grammatik oder Konditionen, die der Compiler nicht berücksichtigt hat oder berücksichtigen kann, können trotzdem Fehler in den Programmcode einfließen. Dies passiert z.B. dann, wenn in einem Kontext beide Ausprägungen einer Anweisung möglich sind, jedoch nicht dasselbe bedeuten (Hoffmann, 2013). Laut Steyer (2004) beruhen viele syntaktische Fehler auf typografischen Fehlern, aber eben nicht alle.

- **Semantische Fehlerquellen**
  Im Gegensatz zu lexikalischen und syntaktischen Fehlerquellen, die sich auf die sprachlichen Muster an sich konzentrieren, sind semantische Fehlerquellen mit der Frage nach dem Sinn der Anweisungen beschäftigt. Die Semantik behandelt somit die *Bedeutung bzw. die Interpretation* der einzelnen Konstrukte und Sprachbausteine (Hoffmann, 2013).

  Daher sind semantische Fehler meist schwieriger zu finden und erweisen sich als deutlich tiefgründiger als lexikalische oder syntaktische Fehler. Durch ein gewisses Maß an Disziplin beim Entwickeln von Software lassen sich jedoch viele Fehler im Ansatz vermeiden (Hoffmann, 2013). Ein typisches Beispiel hierfür ist der lesende Zugriff auf Elemente, die noch nicht initialisiert sind (Belli et al., 1984).

- **Parallelität als Fehlerquelle**
  Das Phänomen der Prioritäteninversion muss zum Tragen kommen, damit Parallelität als Fehlerquelle auftreten kann. Dieses ist in der Literatur als solches

bekannt und wird häufig beschrieben. Es besagt, dass ein Job mit mittlerer Priorität einen Job mit hoher Priorität verzögert. Infolgedessen können Fehler bis hin zum Absturz des Systems vorkommen. Dies hängt unter anderem auch davon ab, wie viel Zeit die Tasks im Gegensatz zu den jeweils anderen benötigen, siehe Abbildung 2.1 (Hoffmann, 2013).

Der hochpriore Job kommt deshalb nicht zum Zug, da ein weiterer, niederpriorer Prozess eine Ressource hält oder sich in einem kritischen Abschnitt befindet, den auch der hochpriore benötigt. Der mittelpriore Job hingegen unterbricht den niederprioren und so entsteht die Prioritäteninversion. Zur Entschärfung der Problematik kann die Prioritätenvererbung eingesetzt werden. Dieser Sachverhalt ist grafisch in Abbildung 2.1 dargestellt (Quade & Kunst, 2016).

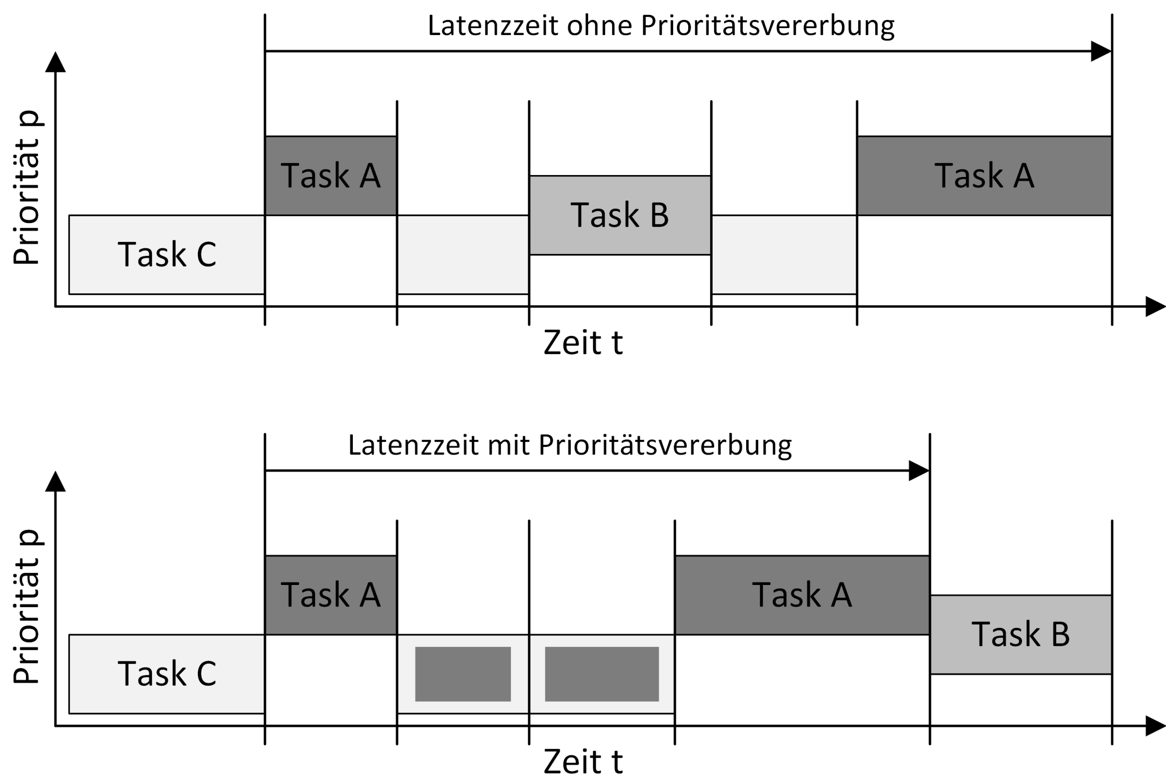


Abbildung 2.1.: Prioritäteninversion und mögliche Lösung durch Prioritätenvererbung. Dadurch, dass Task C die Priorität von Task A erbt, wird er vor Task B abgearbeitet und kann somit den kritischen Abschnitt schneller verlassen. Im Anschluss kann Task A, der diese kritische Ressource benötigt, abgearbeitet werden und wird nicht mehr von Task B ausgebremst (vgl. Quade und Kunst, 2016).

- **Numerische Fehlerquellen**
  Um numerische Fehlerquellen zu beschreiben bzw. exemplarisch gezielt erzeugen zu können, ist es nötig, die Verarbeitung von Daten in Computern nachzuvollziehen (Schwarz & Köckler, 2011).

Dabei löst die Disziplin der numerischen Mathematik genau diese Probleme, die zu numerischen Fehlern führen können. Sie beschriebt, wie Algorithmen Daten verarbeiten können, um daraus Ergebnisse zu bekommen, die in der Praxis verwendbar sind, obwohl sie Fehler enthalten. Die typischen Fehler in der Numerik sind *Datenfehler*, *Diskretisierungsfehler* und *Rundungsfehler*. Eine der Kernaufgaben der Numerik ist es nun, die Diskretisierungs- bzw. Rundungsfehler soweit in den Griff zu bekommen, damit die Ergebnisse verwertbar sind (Schwarz & Köckler, 2011).

Auch das Prinzip der Fehlerfortpflanzung muss berücksichtigt werden, damit sich Fehler, die an sich sehr gering sind, nicht aufschaukeln und im Endeffekt das System unbrauchbar machen (Hoffmann, 2013).

- **Portabilitätsfehler**
  Portabilität beschreibt auf Hardwareebene, wie einfach es ist, eine Software von einem Gerät, auf dem es aktuell eingesetzt wird, auf ein anderes zu übertragen und dort erfolgreich anzuwenden. Für die Softwareumgebungen, z.B. ein Betriebssystem, gilt dasselbe Prinzip. Portabilität bezieht sich somit auf die *Unabhängigkeit* von Hard- und Software (Schmitz, Bons & van Megen, 1983).

  Ein prominentes Beispiel für einen Portabilitätsfehler ist der Absturz der Ariane V am 4. Juni 1996. Aufgrund des sehr erfolgreichen Vorgängermodells Ariane IV wurden gewisse Softwareelemente einfach übernommen. Dazu gehörte auch der Programmcode zum Management der Horizontalgeschwindigkeit. Aufgrund der geänderten Parameter für die Ariane V, in diesem Fall unter anderem die Größe, das Gewicht sowie die Geschwindigkeit der Rakete an sich, wirkte sich der Typkonversionsfehler, der schon im Vorgängermodell vorhanden war, so stark aus, dass die Rakete nach 39 Sekunden aus Sicherheitsgründen zerstört werden musste (Hoffmann, 2013).

- **Optimierungsfehler**
  Die Optimierung von Programmelementen dient in den meisten Fällen zur *Verbesserung der Laufzeit und/oder des Speicherplatzbedarfs* kritischer Programmabschnitte. Eine weitere Optimierung ist es, durch handwerkliches Können den Programmcode im Sinne der Kompaktheit zu kürzen (Hoffmann, 2013).

  Dadurch, dass diese genannten Änderungen in Softwareprojekten oft sehr spät durchgeführt werden, muss hier mit besonderer Vorsicht agiert werden, um die ursprüngliche Intention nicht aus den Augen zu verlieren. Die dabei entstehenden Fehler, die ohne diese Änderungen nicht im Code enthalten wären, nennt man Optimierungsfehler (Hoffmann, 2013).

- **Bekannte Einschränkungen, die zu Fehlern ausarten können**
  Zu Beginn dieses Jahrtausends gab es grob gesagt zwei Ansichten: Entweder

man hielt die Umstellung von 1999 auf 2000 für ein echtes (technisches) Problem, oder man sah die Sache völlig gelassen (Fuchs & Hofkirchner, 2003).

Der technische Hintergrund zu dieser oftmals als "Glaubenssache" ausgetragenen Debatte war durchaus real. So wurden viele Computersysteme in den 1970er Jahren, als Hardware noch teuer war, mit zwei statt vier Stellen für die Datumsangabe versehen. Die bereits damals bekannte Schwachstelle, die ja erst in ca. einer Generation zum Tragen kommen sollte, wurde als Risiko bewusst akzeptiert, oder man war sich der langen Einsatzzeit mancher Computersysteme nicht bewusst. Zur Jahrtausendwende wurde dieser als Y2K - Bug in die Geschichte eingegangene Fehler von vielen Experten unterschiedlich bewertet (Fuchs & Hofkirchner, 2003).

Tatsächlich ging die Angst soweit, dass viele Flugzeuge die besagte Nacht nicht abhoben und zahlreiche Geldautomaten abgeschaltet wurden. Der große Knall bzw. große Schäden blieben jedoch aus. Auch hier sind sich Experten nicht einig, ob die Angst vor diesem Bug überschätzt wurde, oder ob die massiven Investitionen zur Erneuerung der alten Computersysteme größere Schäden verhindert haben (Hoffmann, 2013).

Auch heute noch gibt es diese Einschränkungen von Systemen, die zu Fehlern bzw. undefiniertem Verhalten führen können. 2038 wird sich für alle 32-bit Computersysteme, die das POSIX-Format verwenden, technisch dasselbe abspielen: Es kommt zu einem numerischen Überlauf der internen Darstellung des Datums. Semantisch bedeutet dies einen Sprung vom 19. Jänner 2038 zum 13. Dezember 1901, der innerhalb einer Sekunde stattfinden wird, siehe Listing 2.1. Dadurch, dass viele Systeme bereits jetzt auf 64-bit umgestellt werden, wird dieses Verhalten vermutlich weniger Systeme betreffen als im Jahr 2000 (Hoffmann, 2013).

Listing 2.1: POSIX konformes Verhalten aller 32-bit Computersysteme im Jahr 2038 mit numerischem Überlauf (vgl. Hoffmann, 2013).

```
1  Tue Jan 19 03:14:02 2038
2  Tue Jan 19 03:14:03 2038
3  Tue Jan 19 03:14:04 2038
4  Tue Jan 19 03:14:05 2038
5  Tue Jan 19 03:14:06 2038
6  Tue Jan 19 03:14:07 2038
7  Fri Dez 13 20:45:52 1901
8  Fri Dez 13 20:45:53 1901
9  Fri Dez 13 20:45:54 1901
10 Fri Dez 13 20:45:55 1901
```

- **Spezifikationsfehler**
  Alle bisherigen Fehler gehen auf Inkorrektheiten bei der Implementierung zurück. Daraus folgt, dass die Spezifikation korrekt, die Umsetzung in Summe

aber nicht richtig war. Der Spezifikationsfehler bildet hierbei eine Ausnahme. Hier ist die Definition, was umgesetzt werden soll, im Grunde genommen inkorrekt. Dies kann auf *fehlerhaften, vagen oder nicht vorhandenen Spezifikationen* beruhen (Hoffmann, 2013).

Da dieser Mangel der Software bereits anhaftet, ohne dass diese implementiert sein muss, gilt diese Fehlerkategorie als kritisch. Die Erkennung der fehlerhaften Spezifikation ist daher essentiell. Das Unglück des Lufthansa Flugs 2904 mit der Landung in Warschau-Okecie, erläutert im Kapitel 2.2, liegt einem solchen Spezifikationsfehler zugrunde (Hoffmann, 2013).

2.3.3. Arten von Softwarefehlern

Laut Trauboth (1996) können Softwarefehler nach verschiedenen Gesichtspunkten in Kategorien eingeordnet werden. Eine Gewichtung des Fehlers an sich kann anhand der Kategorisierung und der Sicht des Betrachters abgeleitet werden. Aus den verschiedenen Kategorien lassen sich Klassen oder Arten von Fehlern ableiten. Fehler lassen sich prinzipiell nach folgenden Kriterien kategorisieren:

- **Ort:** Wo ist der Fehler entstanden (Trauboth, 1996)?
  Den Systemkomponenten werden entsprechend dem Aufbau des Systems Fehlernamen zugeordnet. Der Aufbau dieser kann hierachisch gestaltet werden. In der Praxis wird dabei zwischen dem Entdeckungs- und dem Fehlerort nicht unterschieden. Einige Beispiele für mögliche Orte sind:

  - Systemfehler (Teilmenge des Softwarefehlers)
  - Betriebssystemfehler (Teilmenge des Systemfehlers)
  - Anwendungsfehler (Teilmenge des Softwarefehlers)
  - Ein-/Ausgabefehler (Teilmenge des Anwendungsfehlers)

- **Ursache:** Wo hat der Fehler seine Wurzeln (Trauboth, 1996)?
  Die Ursachen von Fehlern beziehen sich immer auf menschliche Tätigkeiten bzw. auf die Ergebnisse dieser Tätigkeiten. In Tabelle 2.1 sind einige Beispiele dargestellt.

  Im Vergleich dazu sehen Frühauf, Ludewig und Sandmayr (2007) die Fehlerentstehung und Fehlerentdeckung im Zusammenhang mit einem vereinfachten V-Modell. So ist jeder Fehler auf dem Niveau zu finden, auf dem er begangen wird. In Tabelle 2.2 ist dies dargestellt.

- **Entwicklungsphase:** In welchem Prozessschritt der Entwicklung trat der Fehler auf (Trauboth, 1996)?

Fehler in der Problemdefinition	Falsches Problem erfasst und/oder Vernachlässigung von essentiellen Parametern
Fehler in der Systemdefinition	Widersprüche und/oder Unvollständigkeiten treten auf
Fehler in der Umsetzung bzw. Programmierung	Verwendung falscher Syntax und/oder Logik
Fehler in der Interpretation	Ergebnisse werden falsch interpretiert

Tabelle 2.1.: Modell von Trauboth (1996): Nennung einiger Beispiele, wo Fehler ihre Wurzeln haben können.

Fehler in der Anforderung	Kann im Betrieb, bei der Abnahme oder beim Systemtest gefunden werden
Fehler im Entwurf	Kann bei Integrationstests gefunden werden
Fehler im Code	Kann bei Einzeltests gefunden werden

Tabelle 2.2.: Vergleichsmodell von Frühauf et al. (2007) in Bezug auf Fehler und deren Wurzeln.

Im Rahmen der Entwicklung von Software lassen sich die einzelnen Schritte in Phasen einteilen. In diesen einzelnen Prozessabschnitten können beispielsweise folgende Fehler passieren:

- Rechenfehler
- Logikfehler
- Konfigurationsfehler
- Bedienungsfehler

Voges (1989) gibt hier einen weiteren Aspekt zu bedenken. Durch das Testen, einen Prozessschritt in der Entwicklung, ist es möglich Fehler aufzudecken. Je nach Testverfahren können so gewisse Kategorien von Fehlern in der Regel aufgedeckt werden, andere wiederum sind bei selbigem Verfahren meist nicht auffindbar. Zum Beispiel sind beim Vergleichstest sehr wohl unterschiedliche Fehler sichtbar, identische Fehler maskieren sich jedoch gegenseitig.

- **Betriebsart:** In welchem Betriebsmodus trat der Fehler auf (Trauboth, 1996)? Nach der Fertigstellung durchläuft die Software unterschiedliche Betriebsarten. Ein möglicher Fehler hierbei ist der Übersetzungsfehler, der beim Übersetzten der Software entsteht.

- **Zeitverhalten:** Wie tritt der Fehler über einen gewissen Zeitraum hinweg auf (Steinhorst, 1999)?

Grundsätzlich haben Softwarefehler kein Zeitverhalten. Sie sind seit Beginn des Lebenszyklus' der Software vorhanden. Bemerkenswert dabei ist, dass sie über lange Zeit verborgen sein können und somit nicht fehlerwirksam sind.

Der triviale Fall besteht darin, dass der Fehler ununterbrochen und reproduzierbar auftritt. Dieser Fehler heißt *permanenter Fehler*. Im Gegensatz dazu tritt ein *transienter Fehler* nur kurzzeitig auf. Sollte ein Fehler immer wieder zu unregelmäßigen Zeitpunkten auftreten, nennt man dies einen *sporadischen Fehler* (Trauboth, 1996).

Auch für Gerlich und Gerlich (2005) gibt es den permanenten Fehler. Die anderen Fehlerkategorien werden bei ihnen jedoch etwas anders beschrieben. So ist der *sporadische Fehler* nicht oder nur schwer reproduzierbar und daher nicht oder nur schwer behebbar. Der *schlafende Fehler* wiederum ist noch gar nicht aufgetreten und ist daher noch nicht entdeckt. Der *verborgene Fehler* hat schließlich die Eigenschaft, dass er aufgetreten ist, jedoch nicht als Fehler erkannt wurde.

Es gibt somit unterschiedliche Betrachtungsweisen, was das Zeitverhalten von Fehlern betrifft. Es versteht sich von selbst, dass der permanente Fehler leichter zu erfassen ist als die jeweils anderen Kategorien.

- **Art der Fehlererkennung:** Auf welche Art und Weise wurde der Fehler erkannt (Trauboth, 1996) ?
  Der Fehler wird nach der Art der Erkennung bzw. Meldung benannt. Der *Parity Check* weist so z.B. darauf hin, dass Daten verfälscht wurden oder verloren gingen. Der *CRC Check*, der vor allem bei Übertragungen und der Speicherung von Daten zum Einsatz kommt, weist genauso auf diese Fehler hin. Diese Methode ist weitaus komplexer und erkennt mehr Fehler, als der einfache Parity Check. Über die Ursache des Fehlers machen weder der Parity Check noch der CRC Check eine Aussage (Trauboth, 1996; Müller, Käser, Gübeli & Klaus, 2005).

2.3.4. Eigenschaften von Softwarefehlern

Laut Trauboth (1996) gibt es eine weitere Möglichkeit Softwarefehler zu klassifizieren. Diese beruht auf der Aufschlüsselung der Fehler nach ihren Eigenschaften:

- **Gewicht:** Wie kritisch ist der Fehler?
  Untersucht wird, wie stark sich der Fehler im Eintrittsfall auswirkt. Dazu werden anhand von Risikoklassen die Fehler eingeteilt. Im Anschluss kann das Risiko aufgrund der folgenden Faktoren bewertet werden (Broy & Kuhrmann, 2013).

- *Eintrittswahrscheinlichkeit*: Beschreibt die Wahrscheinlichkeit, dass das Risiko eintritt.
- *Schadenshöhe*: Gibt den erwarteten Schaden an.
- *Risikokennzahl*: Beschreibt die Kritikalität des Fehlers, die sich aus der Eintrittswahrscheinlichkeit und der Schadenshöhe ergibt.

- **Häufigkeit:** Wie oft tritt der Fehler auf?
  Die Häufigkeit des Auftretens des Fehlers innerhalb eines definierten zeitlichen Rahmens ist von großer Bedeutung. Sie korreliert direkt mit der Verfügbarkeit des Systems für den konkreten Anwendungsfall (Trauboth, 1996).

  Die Anzahl der Fehler in einem Zeitraum, die Häufigkeit, hängt natürlich vom Aufwand ab, der vor der Freigabe der Software für das Vermeiden oder Beseitigen der Fehler investiert wurde (Riedermann, 1997).

Zu den genannten Punkten sind weitere Klassifizierungen denkbar. Einen Ansatz dazu liefert die Methode der *Standardisierung der Fehlerklassifizierung*. Anhand diese Methode ist es leichter möglich, die Fehler zu vergleichen. Dazu müssen drei Hauptattribute beachtet werden:

- *Quelle*: Beschreibt die Phase im Lebenszyklus der Software, in der der Fehler gemacht wurde.
- *Ursache*: Kausale Beschreibung für den Fehler. Auf symptomatische Beschreibungen wird hier bewusst verzichtet.
- *Schwere*: Die Wirkung auf die Aufgabe durch den Fehler.

Weitere Attribute wie z.B. der Zeitpunkt der Entdeckung des Fehlers sind möglich (Trauboth, 1996).

2.4. Qualität von Software

„Nahezu jeder Programmierer entwickelt im Laufe seiner Karriere ein intuitives Verständnis für den Begriff Software Qualität (Hoffmann, 2013, S. 6).“ Dieser Aussage wohnt eine Ahnung inne, dass der Begriff Softwarequalität nicht trivial zu verstehen bzw. zu definieren ist. Laut ISO-Norm 9126 (ISO/IEC 9126 2001, zitiert nach Hoffmann 2013, S. 6) wird der Begriff Softwarequalität wie folgt definiert:

> „Software-Qualität ist die Gesamtheit der Merkmale und Merkmalswerte eines Software-Produkts, die sich auf dessen Eignung beziehen, festgelegte Erfordernisse zu erfüllen.“

Die Definition des Begriffs Softwarequalität laut ISO-Norm 9126 beschreibt weiters, dass es sich bei selbigem um eine multikausale Größe handelt. Es gibt somit nicht das eine Kriterium, das Softwarequalität an sich ausreichend beschreibt (Hoffmann, 2013; Liggesmeyer, 2009). „Vielmehr verbergen sich hinter dem Begriff eine ganze Reihe vielschichtiger Kriterien, von denen sich einige zu allem Überdruss auch noch gegenseitig ausschließen.“ (Hoffmann, 2013, S. 6)

In Abbildung 2.2 ist dieser Sachverhalt dargestellt. Hier wird eine Gruppierung der Qualitätskriterien in Abhängigkeit zum Anwendungsgebiet vorgenommen. Im Folgenden sollen diese Kriterien aufgezählt und näher beschrieben werden.

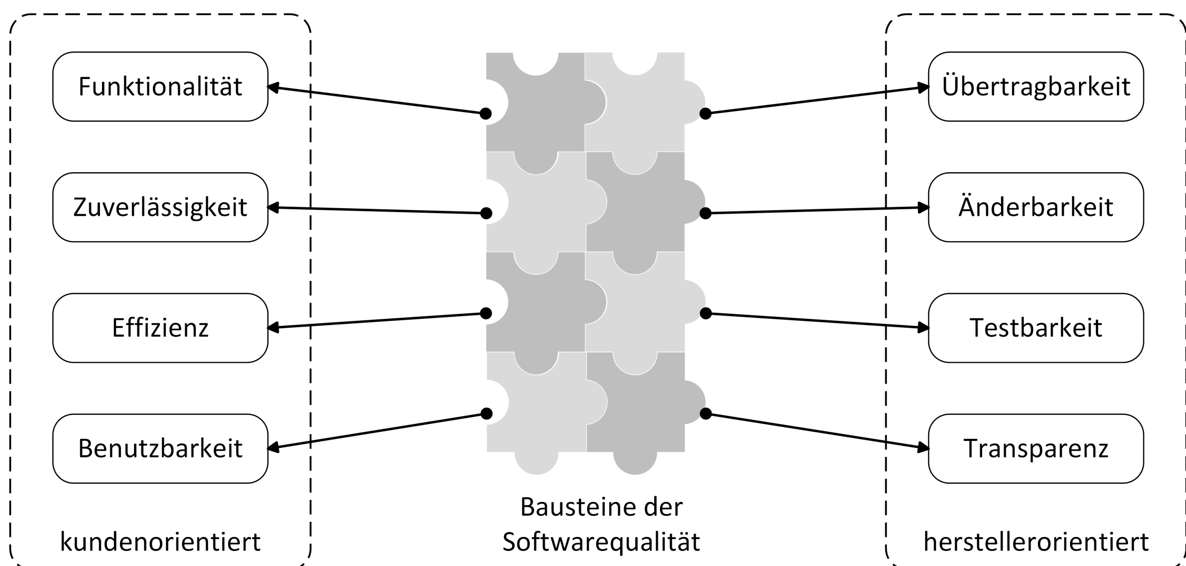


Abbildung 2.2.: Qualitätsmerkmale eines Software-Produkts (vgl. Hoffmann, 2013).

2.4.1. Funktionalität

Dieses Qualitätsmerkmal umfasst alle Charakteristiken, welche die geforderten Fähigkeiten des Systems beschreiben. Diese entsprechen definierten Reaktionen auf zuvor spezifizierte Ein-/Ausgabemuster, beschreiben aber auch Wirkungen des Systems auf entsprechende Eingaben. Laut ISO-Norm 9126 gliedert sich das Qualitätsmerkmal Funktionalität in fünf Teilmerkmale: *Angemessenheit*, *Richtigkeit*, *Interoperabilität*, *Ordnungsmäßigkeit* bzw. *Konformität* und *Sicherheit* (ISO/IEC 9126, 2001; Cleff, 2010; Spillner & Linz, 2012).

Die *Angemessenheit* definiert dabei, ob jede geforderte Fähigkeit vorhanden ist und entsprechend implementiert wurde. Infolgedessen übernimmt die *Richtigkeit* beim Testen die Aufgabe, die Wirkungen bzw. spezifizierten Reaktionen vom System auf

die Korrektheit zu prüfen. Der Begriff der *Interoperabilität* beschreibt, wie das System mit anderen Systemen, mit denen es zusammenarbeiten soll, reagiert. Eine möglichst reibungsfreie Zusammenarbeit der Systeme soll beim Testen aufgezeigt werden (Cleff, 2010; Spillner & Linz, 2012).

Die *Ordnungsmäßigkeit* bzw. *Konformität* deckt die Vorgaben von Gesetzen, Normen und anderen Vorschriften ab. Der Aspekt der *Sicherheit* soll gewährleisten, dass kein unberechtigter Zugriff auf die Daten sowie das Programm, sowohl absichtlich als auch unabsichtlich, stattfinden kann. Dies wird unter dem Begriff der Datensicherheit zusammengefasst (Cleff, 2010; Spillner & Linz, 2012).

Funktionale Fehler können in verschiedenen Varianten vorkommen und verletzen somit eines der eben genannten Kriterien. Oft fußen sie dabei auf unabsichtlich falsch verstandenen Anforderungen, fehlen gänzlich oder werden aus Zeitnot, z.B. kurz vor einem Release Termin, fehlerhaft umgesetzt (Hoffmann, 2013).

Den Großteil der Fehler beim Qualitätskriterium Funktionalität stellt jedoch der klassische Implementierungsfehler, auch Bug genannt, dar. Dieser tritt so häufig auf, dass der Begriff Softwarequalität meist damit in Verbindung gebracht wird und dabei die folgenden Qualitätskriterien meist vernachlässigt werden (Hoffmann, 2013).

2.4.2. Zuverlässigkeit

Solange die Software aufgrund der gegebenen, beschränkten Hardware relativ einfach und übersichtlich war, solange konnte die *Zuverlässigkeit* für Software außer Acht gelassen werden. Mit aktueller Hardware gilt dies nicht mehr. Die Komplexität von Software wird aktuell durch die Aufgabenstellung und die Fähigkeiten der EntwicklerInnen begrenzt (Becker, 1989).

Das Qualitätskriterium der Zuverlässigkeit beschreibt im Kern die Fähigkeit des Systems, die Leistungsfähigkeit unter definierten Bedingungen über einen festgelegten Zeitraum zu bewahren. Die entsprechenden Teilmerkmale wie *Reife*, *Fehlertoleranz*, *Wiederherstellbarkeit* und *Fehlertoleranz* beschreiben dies sehr gut (ISO/IEC 9126, 2001; Franz, 2007; Cleff, 2010).

Die Frage nach der Zuverlässigkeit einer Software ist verwandt mit der Frage nach den Mitteln bzw. vorhandenen Ressourcen, die für die Feststellung der Qualität der Software verwendet werden. Der Einsatz unterschiedlicher Mittel sowie der zur Verfügung stehenden begrenzten Ressourcen kann somit etwas über das Niveau der Zuverlässigkeit von Software aussagen (Becker, 1989).

Da der Ausfall einer Software, gleichbedeutend mit dem Zuverlässigkeitsverlust derselben, eine *Zufallsgröße* ist, sind auch die dazugehörigen Parameter, die messbar

sind, Zufallsgrößen. Das Maß für die Zuverlässigkeit einer Software ist somit ein *propabilistisches Maß* (Becker, 1989).

Das Kriterium der Zuverlässigkeit ist mit vielen anderen Kriterien stark gekoppelt. Daher kann eine hohe Systemzuverlässigkeit nicht singulär erreicht werden, sondern bedingt die Optimierung einer ganzen Reihe anderer Kriterien (Hoffmann, 2013).

2.4.3. Effizienz bzw. Laufzeit

Jedes Programm oder Teilprogramm unterliegt einer gewissen Anforderung an die *Effizienz* bzw. *Laufzeit*, auch *Performance* genannt. Diese kann explizit oder implizit gefordert werden. Ein eher schwammiges Beispiel aus der Praxis für eine implizite Anforderung kann lauten, dass ein Programm "immer sofort", was so viel wie "zeitnah" bedeutet, reagieren soll.

Das Qualitätskriterium Effizienz bzw. Laufzeit lässt sich in folgende Teilmerkmale gliedern: *Zeitverhalten*, *Verbrauchsverhalten* und *Konformität*. Ersteres beschreibt für die NutzerInnen sichtbare Eigenschaften wie Antwort- und Verarbeitungszeiten sowie den Durchsatz der Funktionen. Zweiteres beschreibt den Grad der Nutzung der vorhandenen Kapazitäten, was in Wechselwirkung mit der IT-Infrastruktur steht. Drittes beschreibt, inwieweit die Software Standards, Normen und Gesetze erfüllt (Franz, 2007; Cleff, 2010).

Da Computer meist sehr schnell rechnen, reicht es bei der Entwicklung häufig aus, Software systematisch zu entwickeln, um performant genug zu sein. Sind jedoch sehr rechenintensive Anwendungen gefragt, z.B. die Analyse oder Visualisierung großer Datenmengen, ist es wichtig, sich den Algorithmus für die auszuführende Tätigkeit genauer anzusehen und eventuell performantere Lösungen zu entwickeln (Kleuker, 2019).

Die Laufzeit wirkt vor allem bei *Echtzeitsystemen* als kritischer Faktor. Hier kann es durchaus sein, dass Kriterien erfüllt werden müssen, ansonsten gilt die Software als nicht geeignet für den Einsatz. In diesen Fällen sind die Kriterien Laufzeit und Funktionalität eng miteinander verwoben und von gleichrangiger Bedeutung (Hoffmann, 2013)

Ein sehr interessanter Aspekt ergibt sich, wenn die Performance mit der Skalierbarkeit verglichen wird. Der Begriff Performance beschreibt in diesem Lichte die Effizienz, z.B. einer Berechnung, während die Skalierbarkeit den Trend der Performance mit steigender Last beschreibt. Ein System kann demnach hochperformant aber dennoch nicht skalierbar sein (Liu, 2009).

2.4.4. Benutzbarkeit

Die *Benutzbarkeit* einer Software stellt die Interaktion der AnwenderInnen mit der Software in den Vordergrund. Diese sollen dabei möglichst einfach und effizient mit dem System arbeiten können. Ein wichtiger Punkt dabei ist, dass die AnwenderInnen das System möglichst leicht verstehen und verstehen lernen. Dies wiederum führt zu einer hohen *BenutzerInnenakzeptanz*, von der beide Seiten profitieren. Vor allem bei Systemen, die direkt für die EndkundInnen gedacht sind, ist dieser Faktor enorm wichtig (Brunst, Raszczyk & Schneider, 2004; Franz, 2007; Cleff, 2010).

Hoffmann (2013) fügt dieser Definition von Benutzbarkeit ein paar interessante Details hinzu. Früher waren BenutzerInnenschnittstellen eher spartanischer Natur. Anders formuliert bedeutet dies, dass Benutzbarkeit in Softwareprojekten eine geringe Rolle spielte. Heute sind diese Schnittstellen individuell auf die jeweilige Anwendung abgestimmt und haben dafür passende Bedienkonzepte in petto. Weiters besitzt die Benutzbarkeit von Softwareprodukten, die in Nischen eingesetzt werden, eine geringere Wichtigkeit als ihre Pendants außerhalb dieser Nischen.

2.4.5. Übertragbarkeit

Die *Übertragbarkeit* bzw. *Portierbarkeit* von Software beschreibt die bewusste Entscheidung, die Architektur der Software so zu designen, dass minimale oder kleine Anpassungen derselben für die Übertragung auf eine andere Plattform bzw. ein anderes Softwaresystem nötig sind. Ein Softwaresystem, das übertragbar bzw. portierbar ist, soll unter anderem folgende Elemente aufweisen:

- Die Verwendung von standardisierten Entwicklungssprachen ohne Nutzung von compilerspezifischen Erweiterungen.
- Logische Kapselung des Codes in einzelne Module als essentielles Merkmal. Alle Schnittstellenelemente einer Kategorie, z.B. alle benötigten Systemaufrufe, gehören so in ein eigenes Modul. Bei Bedarf kann das gesamte Modul zentral, einfach und schnell getauscht oder adaptiert werden. Das mühsame Suchen und Finden der verstreuten Aufrufe entfällt.

Unter Einhaltung dieser Kriterien ist es einfach oder einfacher möglich, Softwaresysteme auf anderen Hard- und Softwareplattformen in Betrieb zu nehmen (Goll, 2011).

Die Umstellung von 32-bit- auf 64-bit Anwendungen zeigt auf, wie manche Programme sehr einfach, da nach den Grundsätzen der Übertragbarkeit bzw. Portierbarkeit entwickelt, adaptiert werden können bzw. ohne Adaptierung sofort laufen, andere wiederum um Millionensummen angepasst werden müssen (Hoffmann, 2013).

2.4.6. Änderbarkeit bzw. Wartbarkeit

Die *Änderbarkeit* bzw. *Wartbarkeit* von Software beschreibt den dafür nötigen Aufwand sowie die resultierenden Auswirkungen auf die Software, wenn *Änderungen* aufgrund unterschiedlicher Faktoren wie z.B. Behehung von Fehlern, Hinzufügen von Features, Anpassung an eine neue Plattform usw. nach dem Fertigstellen des Softwareprojekts durchgeführt werden. Damit die Eigenschaft der Wartbarkeit auf ein Softwaresystem zutrifft, müssen gewisse Faktoren gegeben sein. Sind diese nur sporadisch oder gar nicht vorhanden, ist die Wartbarkeit der Software eingeschränkt oder gar nicht gegeben. Zu diesen Eigenschaften zählen unter anderem:

- Eine entsprechend ausführliche Dokumentation zu den Schnittstellen, egal ob von statischer oder dynamischer Natur.
- Eine gut erweiterbare, tragfähige und skalierfähige Architektur.
- Ein übersichtliches Design, welches sich bewährte Design Patters der Softwareentwicklung zu Nutze macht.
- Die Verwendung des DRY ("don't repeat yourself") Prinzips.
- Die Verwendung von Kommentaren auf lokaler Codeebene, um das Verständnis des Codes auf Zeilenebene zu garantieren.
- Ein möglichst großes Set an automatischen Tests.
- Eingebaute und/oder dokumentierte Hilfsmittel/Vorgehensweisen für die spätere Diagnose von Zuständen sowie Wartung der Software.

Diese Eigenschaften zeigen auf, dass der Wartungsgedanke von Software von Beginn des Lebenszyklus bis zur Fertigstellung verfolgt werden muss. Dieses Vorgehen erhöht die Wartbarkeit des Systems. Andernfalls ist diese Eigenschaft, die Wartbarkeit des Systems, dem Zufall überlassen (Bommer, Spindler & Barr, 2008; Cleff, 2010).

2.4.7. Testbarkeit

Der Begriff *Testbarkeit* beschreibt die Fähigkeit eines Softwaresystems, die Planung und Durchführung von Tests zu ermöglichen bzw. zu erleichtern. Dazu gehören unter anderem die Zurechenbarkeit eines Fehlers zur Software, die Verfügbarkeit des Systems zu Testzwecken und die Strukturiertheit der Software, Tests gezielt ansetzen zu können. Alle diese Faktoren ermöglichen bzw. erleichtern das Testen signifikant (Schneider, 2012).

Der Begriff Testbarkeit in Bezug auf Softwarequalität kann auf "Sichtbarkeit und Steuerung" heruntergebrochen werden. Diese Begriffe beschreiben, dass ein testbares System Einblicke in sein Inneres gewährt, sowie dort genügend Steuerungspunkte anbietet, um sein Verhalten testen zu können. Um diese beiden Konzepte zu verwirklichen, gibt es einige Möglichkeiten, die im Folgenden aufgelistet sind:

- Einsatz von systemeigenen oder systemfremden Tools wie z.B. Betriebssystemüberwachung durch den Task Manager (Sichtbarkeit).
- Eigene Skripte benutzen bzw. entwickeln und einsetzen, um z.B. Operationen des Dateisystems zu überwachen (Sichtbarkeit).
- Eben genannten Punkt um Aktionen erweitern, z.B. Starten einer Codesequenz nach 10.000 Operationen des Dateisystems (Sichtbarkeit und Steuerung).
- Testdaten in die Datenbank schreiben lassen. Bei ausgewählten Datensätzen soll es möglich sein, einen Trigger mit darauf folgenden Aktionen einzurichten (Steuerung).

Daraus folgt, dass das Aufsplitten der Testbarkeit in die beiden Elemente Sichtbarkeit und Steuerung wichtig für das tiefere Verständnis des Qualitätskriteriums ist (Hendrickson, 2014).

Zwei wesentliche Faktoren erschweren das Testen bzw. die Testbarkeit eines Softwaresystems: Die Komplexität und der Black-Box Charakter. Die Komplexität beschreibt dabei, dass es mit Ausnahme von sehr sehr kleinen Programmen nicht möglich ist, alle Verhaltensweisen eines Softwaresystems zu testen. Der Black-Box Charakter zeigt auf, dass viele Softwaresysteme erst dann testbar sind, wenn die internen Variablen und Datenstruktuen von außen sichtbar sind. Dies bedeutet, dass oft Software um Testcode ergänzt werden muss, um diesen Black-Box Charakter zu umgehen (design for test). Dieser Blickwinkel zur Testbarkeit von Hoffmann (2013) weist auf weitere Faktoren zum Qualitätskriterium Testbarkeit hin.

2.4.8. Transparenz

Transparenz in Softwareprojekten beschreibt grundsätzlich, ob die einzelnen Module und Bausteine sowie ihr Zusammenspiel auf Codeebene intuitiv verständlich sind. Anders formuliert bedeutet dies, dass EntwicklerInnen sich ohne Kenntnisse der konkreten Software, aber mit Kenntnissen der verwendeten Entwicklungssprache im Code zurechtfinden können. Ist dies der Fall, deutet dies auf die Transparenz der Software hin. Einige Bedingungen für transparente Software können folgendermaßen formuliert werden:

- Für das entsprechende Programm, Teilprogramm oder Codeschnipsel sollen die entsprechenden Zusammenhänge betreffend der dazugehörigen Aufgabe verstanden werden. Daraus folgend kann dieses Wissen abstrahiert, dargestellt und anschließend in Code umgesetzt werden.
- Die Zuverlässigkeit eines transparenten Codes ist gegeben. Fehler springen sofort ins Auge und werden nicht durch verschiedene Faktoren maskiert.
- Die Wartbarkeit des Codes ist gegeben. Durch die Transparenz können Fehler, Änderungen und neue Anforderungen abgebildet sowie die entsprechenden Folgen abgeschätzt werden.

Daraus folgt, dass Transparez kein Feature ist, auf das man verzichten kann bzw. sollte, sondern die Grundlage für andere Qualitätskriterien ist und daher immer im Auge behalten werden sollte (Schmidt, 1994).

Die Abnahme der Transparenz bzw. die Zunahme der Unordnung innerhalb eines langlebigen Softwaresystems scheint einer unsichtbaren Gesetzmäßigkeit zu folgen. Daraus folgt, dass Software altert. Dieser Umstand der Abnahme der Transparenz bzw. der Zunahme der Unordnung wird als Software-Entropie bezeichnet (Hoffmann, 2013).

2.4.9. Fazit

Die aufgezählten Qualitätskriterien werden in Abbildung 2.2 in zwei Gruppen eingeteilt. Die Kriterien *Funktionalität*, *Laufzeit*, *Zuverlässigkeit* und *Benutzbarkeit* sind nach außen hin sichtbar und bilden aus Sicht des BenutzerInnen die direkt bewertbare Qualität der Software ab. Daraus ergibt sich weiters, dass potenzielle KundInnen diese Paramater für ihre Kaufentscheidung berücksichtigen. Infolgedessen kann ein Softwareprodukt aufgrund dieser Sachlage einen kurzfristigen Erfolg am Markt haben (Hoffmann, 2013).

Die Parameter, die die KundInnen in den meisten Fällen nicht ohne Weiteres direkt einsehen können, sind die inneren Werte von Software. Diese Kriterien *Transparenz*, *Übertragbarkeit*, *Wartbarkeit* und *Testbarkeit* können die KundInnen unter anderem dann beurteilen, wenn sie entsprechendes Fachwissen haben, diese Kriterien offen dargelegt werden oder der Austausch zwischen den KundInnen und dem Entwicklungsteam vorhanden ist. Diese Kriterien sind vor allem für den langfristigen Erfolg einer Software wichtig und sind so ein strategisches Investment für die Zukunft (Hoffmann, 2013).

Wie bereits festgestellt, kann die Hälfte der Kriterien durch kurzfristige Planungen, Gewinnstreben oder Sonstiges unberücksichtigt bleiben. Weiters könnten so wichtige Investitionen für die Zukunft auf der Strecke bleiben. Durch Nutzung von weiteren Tools, Vorgehensweisen oder Entwicklungsmethoden wie z.B. Scrum, die die Förderung der inneren Werte von Software direkt oder indirekt unterstützen bzw. zwingend vorraussetzten, werden diese kurzfristig gedachten Ziele aufgeweicht oder neutralisiert und langfristige Investitionen in die Software möglich gemacht. Dies steigert insgesamt die Qualität des Produkts (Nyamsi, 2019).

Nach der Betrachtung der Kriterien stellt sich natürlich die Frage, ob eine Optimierung aller Kriterien auf ein Höchstmaß möglich ist und diese eventuell gleichbedeutend mit der höchst möglichen Qualität von Software ist. Nach eingehender Betrachtung aller Kriterien kann dies sofort ausgeschlossen werden. Dazu reicht es, ein Gegenbeispiel zu finden. Exemplarisch werden dazu die beiden Kriterien *Laufzeit* und

Übertragbarkeit herangezogen. Ein Programm, das einfach portierbar ist, wird in der Regel möglichst abstrakt bzw. allgemein gehalten. Ein hochperformantes Programm jedoch, das eine sehr gute Laufzeit aufweist, nutzt dazu die Entwicklungssprache sowie die Hardware möglichst gut aus. Somit können nicht beide Ziele gleichzeitig erreicht werden (Hoffmann, 2013).

Weitere Beeinflussungen sind gut in Abbildung 2.3 ersichtlich. So hat die Effizienz bzw. Laufzeit von Software auf alle anderen Qualitätskriterien mit Ausnahme der Benutzbarkeit einen negativen Einfluss. Dies lässt den Schluss zu, dass sehr (hoch)-effizienter Code wohlüberlegt eingesetzt werden sollte. Alle anderen Kriterien beeinflussen sich gegenseitig entweder neutral oder positiv. Vor allem die Funktionalität und die Transparenz wirken sich sehr positiv auf die anderen Kriterien aus, mit Ausnahme von Effizienz bzw. Laufzeit. Spannend ist weiters, dass die Benutzbarkeit sich stets neutral zu den anderen Kriterien verhält.

	Transparenz	Testbarkeit	Änderbarkeit	Übertragbarkeit	Benutzbarkeit	Effizienz	Zuverlässigkeit	Funktionalität
Funktionalität	+	+	+	+		-	+	
Zuverlässigkeit	+	+				-		
Effizienz	-	-	-	-				
Benutzbarkeit								
Übertragbarkeit	+							
Änderbarkeit	+							
Testbarkeit	+							
Transparenz								

Abbildung 2.3.: Korrelationsmatrix der verschiedenen Qualitätskriterien. Ein Plus- bzw. ein Minuszeichen drückt jeweils eine positive oder negative Korrelation zwischen den zwei betrachteten Qualitätskriterien aus (vgl. Hoffmann, 2013).

Erwähnenswert bei dieser Art von Gestaltung in Bezug auf die Qualitätskriterien von Software ist, dass sechs dieser Qualitätskriterien, nämlich Funktionalität, Zuverlässigkeit, Effizienz bzw. Laufzeit, Benutzbarkeit, Übertragbarkeit und Änderbarkeit bzw. Wartbarkeit, denen der ISO-Norm 9126 entsprechen. Die Eigenschaften des Qualitätskriteriums Testbarkeit sind als Teilmerkmal im Kriterium Änderbarkeit bzw. Wartbarkeit der ISO-Norm 9126 enthalten. Das Qualitätskriterium Transparenz ist in

der ISO-Norm 9126 nicht in Form eines Qualitätskriteriums oder Teilmerkmals enthalten.

Daraus ist ersichtlich, dass es je nach Betrachtungsweise unterschiedliche Möglichkeiten der Kategorisierung in Bezug auf die Qualitätskriterien von Software gibt. Diese sind einem ständig evolutionären Charakter ausgesetzt. Ein typisches Beispiel dafür ist die Weiterentwicklung der ISO-Norm 9126, die 2011 von der ISO-Norm 25010 abgelöst wurde bzw. in dieser aufgegangen ist. Wenngleich die Kategorisierungen ähnlich sind, bieten sie eine etwas andere Perspektive auf die Qualitätskriterien (ISO/IEC 9126, 2001; ISO/IEC 25010, 2011; Franz, 2007; Cleff, 2010; Spillner & Linz, 2012; Hoffmann, 2013).

Für folgende Ausführungen wird auf die Aufteilung nach (Hoffmann, 2013), wie in Abbildung 2.2 dargestellt, zurückgegriffen. Diese Kategorisierung hat den Vorteil, dass sich mit ihr im weiteren Verlauf der Arbeit zu den jeweils vier kundInnenenorientierten bzw. herstellerInnenorientierten Qualitätsmerkmalen Metriken ableiten lassen, siehe dazu Abschnitt 4, um die Testverfahren, siehe dazu Abschnitt 3.3, entsprechend bewerten zu können. Dies wäre mit anderen Aufteilungen nicht in dieser Klarheit möglich. Weiterführende Informationen sind an den entsprechenden Stellen zu finden.

2.5. Definition des Begriffs Softwaretest

Die Grundüberlegung des Softwaretests besteht darin, das zu untersuchende Programm mit Hilfe eines vordefinierten Satzes konkreter Eingabedaten zu bestücken und auszuführen. Dieses Ist-Ergebnis wird mit dem zuvor ermittelten Soll-Ergebnis verglichen (Hoffmann, 2013). Aufgabe des Tests ist es somit, „Fehlerwirkungen (die auf Defekte hinweisen) gezielt und systematisch aufzudecken“ (Spillner & Linz, 2012, S. 9).

Trotz dieser klaren und einfachen Darstellungen erweist sich der Begriff Softwaretest in der Praxis als zum Teil breit gefächert. So definiert zum Beispiel das IEEE (1990, zitiert nach Hoffmann 2013, S. 157) den Softwaretest folgendermaßen:

> „Test: (1) An activity in which a system or component is executed under specified conditions, the results are observed or recorded, and an evaluation is made of some aspect of the system or component.“

Diese Definition stellt klar die Aktion in den Vordergrund der Tätigkeiten und ist somit praxisorientiert. Im Gegensatz dazu sehen die Autoren Craig und Jaskiel (2002, zitiert nach Hoffmann 2013, S. 157) den Begriff des Softwaretests nicht sonderlich nahe an einer handwerklichen Tätigkeit, sondern klar auf Prozessebene angesiedelt:

> „Testing is a concurrent lifecycle process of engineering, using and maintaining testware in order to measure and improve the quality of the software being tested.“

Der hier verwendete Begriff *Testware* beschreibt dabei die gesamte Testumgebung und die entsprechenden Testfälle. Diese können später den KundInnen ausgeliefert werden und enthalten in vielen Projekten oft mehr Code, als die Produktkomponenten an sich (Hoffmann, 2013).

2.6. Ziele und Nichtziele von Softwaretests

Das Testen an sich verfolgt laut Spillner und Linz (2012) mehrere Ziele:

- Ausführung des Programms mit dem Ziel, Fehlerwirkung nachzuweisen.
- Ausführung des Programms mit dem Ziel, die Qualität zu bestimmen.
- Ausführung des Programms mit dem Ziel, Vertrauen in das Programm zu erhöhen.
- Analysieren des Programms oder der Dokumente, um Fehlerwirkungen vorzubeugen.

Cleff (2010) sieht in den Zielen von Softwaretests die Basis für die Planung und Durchführung der Softwaretests. Ein Testvorhaben wird aufgrund einer oder mehrerer Anforderungen der in Folge genannten Faktoren durchgeführt:

- Die Reduktion der Fehlerkosten wird angestrebt.
- Die Softwarequalität soll (anhand bestimmter Kriterien) bestimmt werden.
- Die Einhaltung von Standards und Normen soll sichergestellt werden.
- Die BenutzerInnen sollen durch Softwaretests Vertrauen zum Produkt aufbauen.

Daraus ist klar ersichtlich, dass Testen nicht nur dazu dient, Fehlerwirkungen nachzuweisen. Vor allem durch die Informationen, die sich aus den Tests in Bezug auf die oben genannten Faktoren ergeben, ist es möglich, weitreichende Entscheidungen wie z.B. die Freigabe zur Integration dieser getesteten Komponenten in weitere Systemteile zu erteilen oder abzulehnen (Spillner & Linz, 2012).

Zu den klassischen Nichtzielen des Softwaretests gehört die Feststellung von Fehlerfreitheit durch das Testen. Dies gilt für alle Systeme, mit der einzigen Einschränkung auf sehr sehr kleine Programme, die alle Zustände klar und definiert abbilden und somit zu 100% getestet werden können. Da diese jedoch die Ausnahme darstellen und in der Praxis nicht vorkommen, wird davon ausgegangen, dass beim Testen immer etwas übersehen wird. Ein mögliches Beispiel kann hier die Vernachlässigung der Auswirkungen von Schaltjahren auf ein System sein (Spillner & Linz, 2012).

3. Die Testklassifikationen

Die Softwareentwicklung ist ständig in Bewegung. Dies bezieht sich auf die Neu- bzw. Weiterentwicklung der sogenannten Programmiersprachen, der dahinterstehenden Konzepte, der Entwicklungsumgebungen sowie der Methoden, Software zu entwickeln. In den letzten Jahren hat vor allem eine Methode die Softwareentwicklung revolutioniert: Scrum (H. Wolf, van Solingen & Rustenburg, 2014; Sutherland, 2015). Diese Methode bricht mit den traditionellen Konzepten der Entwicklung. Dies wirkt sich auch auf das Testing aus (Sandhaus, Knott & Berg, 2014; Meyer, 2018; Lehmbach, 2012).

In der traditionellen Literatur des Testings findet sich häufig das sogenannte allgemeine V-Modell (Spillner & Linz, 2012; Spillner, Roßner et al., 2014; Cleff, 2010). Dieses steht exemplarisch für das Wasserfallmodell in der Softwareentwicklung, erweitert mit dem sogenannten Testing in allen Stufen. Dafür gibt es eigene Testphasen. Im Gegensatz dazu findet das Testing in Scrum kontinuierlich statt (Linz, 2014).

Laut Hoffmann (2013) lässt sich jeder Testfall in drei Klassen einteilen. Die erste Klasse, die *Prüfebene*, bescheibt dabei den Zeitpunkt der Entwicklungsphase, in der der Test durchgeführt wird. Die zweite Klasse, das *Prüfkriterium*, gibt an, welche inhaltlichen Aspekte beim Testen berücksichtigt werden. Die dritte Klasse gibt schlussendlich an, welche *Prüftechnik* bzw. *Prüfmethodik* auf den Testfall angewandt wird.

Der Vorschlag von Hoffmann (2013) in Bezug auf die Aufteilung der Prüftechniken bzw. Prüfmethodiken wird etwas angepasst. Statt der Grey-Box Methode findet das erfahrungsbasierte Testen Einzug in die Grafik. Dies wird in der Einleitung des Abschnitts 3.3 vertiefend behandelt. Diese drei angepassten Klassen sind in Abbildung 3.1 dargestellt.

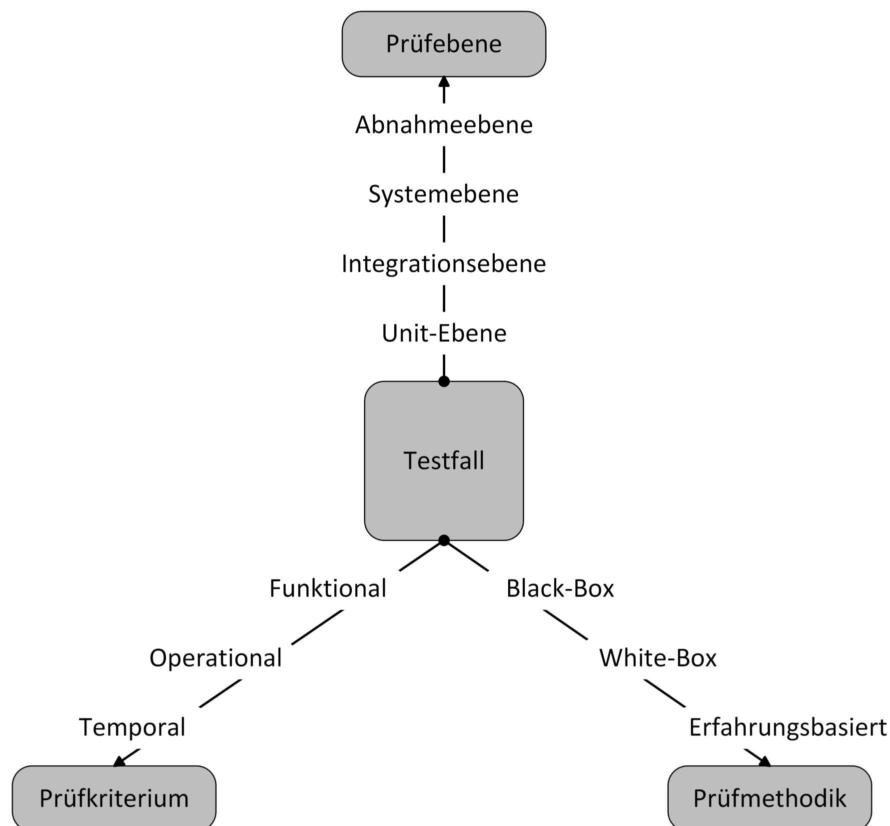


Abbildung 3.1.: Darstellung der Merkmalsräume der Testklassifikation (in Anlehnung an Hoffmann, 2013).

3.1. Die vier Prüfebenen

Wie in Abbildung 3.2 zu sehen, lassen sich die vier Prüfebenen anhand der Programmstruktur und der zeitlichen Enwicklungsphase kategorisieren. Weiters ist in der Grafik ersichtlich, dass der Unit- sowie Integrationstest programmorientiert ablaufen, der System- sowie Abnahmetest sich jedoch an der Funktionalität orientieren (Hoffmann, 2013).

3.1.1. Der Modul-, Komponenten- bzw. Unittest

Eine *Unit* beschreibt eine *atomare Programmeinheit*, die groß genug ist, um als solche eigenständig und systematisch getestet zu werden. In der Praxis stellt sich meist heraus, dass diese Formulierung recht vage ist und einiges an Spielraum zulässt. Dies kann sich dadurch äußern, dass sich Units von einzelnen Funktionen bzw. Klassen hin zu Klassen- bzw. Modulverbänden in Form von Paketen und/oder Bibliotheken erstrecken. Daher nennt man den Unittest oft auch *Modultest* bzw. *Komponententest* (Hoffmann, 2013; Spillner & Linz, 2012; Cleff, 2010).

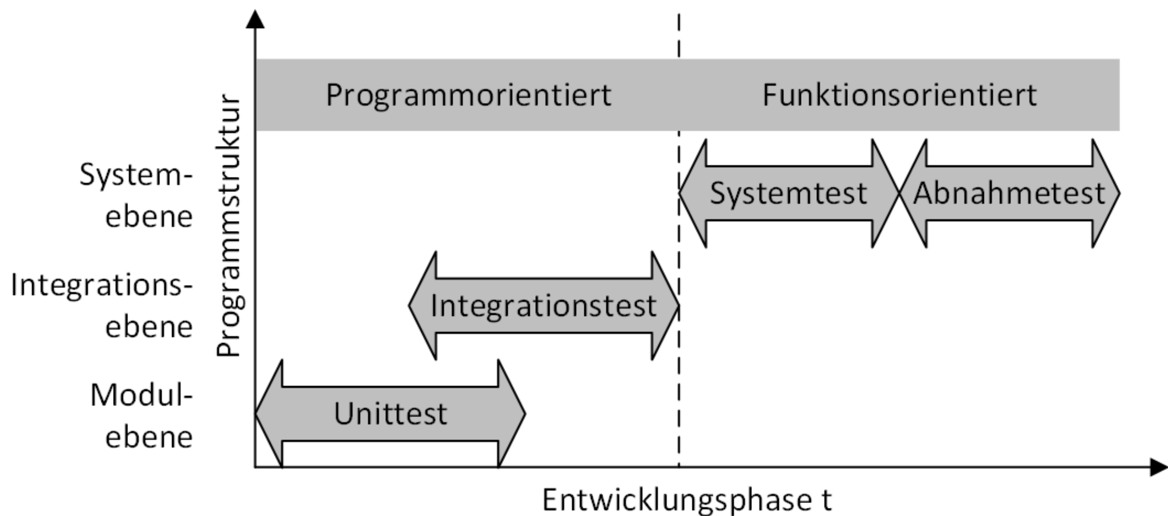


Abbildung 3.2.: Die vier Prüfebenen im Kontext der Entwicklungsphase sowie Programmstruktur (vgl. Hoffmann, 2013).

Typisch für den Unittest ist, dass dieser nahe an der Entwicklung durchgeführt wird. Die Unit wird dabei isoliert betrachtet getestet. Schnittstellen nach außen werden bei Bedarf emuliert. Eine weitere Möglichkeit ist, die Entwicklung nach dem Konzept *test driven developement* voranzutreiben. Dieses besagt, dass zuerst die Tests geschrieben werden. Anhand dieser wird der Code so lange verbessert, bis keine Fehler mehr auftreten (Grechenig, Bernhart, Breiteneder & Kappel, 2010; Cleff, 2010; Spillner & Linz, 2012).

Das Ziel des Unittests ist es, die Funktionalität, die die Komponente in der Theorie liefern soll, zu bestätigen und ein mögliches Fehlverhalten frühzeitig zu erkennen und bei Bedarf zu entfernen (Spillner & Linz, 2012).

3.1.2. Der Integrationstest

Der Integrationstest setzt voraus, dass der Unittest abgeschlossen ist und eventuelle Defekte nach Möglichkeit bereits korrigiert wurden. Der Integrationstest selbst hat das Ziel, das Zusammenspiel der Komponenten zu testen und zu kontrollieren, ob dieses entsprechend der definierten Spezifikation abläuft. Mögliche Ergebnisse des Tests sind die Aufdeckung von Schnittstellenfehlern sowie die Entdeckung von Fehlen beim Datenaustausch. Am Ende des Tests und der eventuell notwendigen Behebung der Defekte liegt ein System vor, das funktionsfähig ist (Leach, 2016; Puntambekar, 2009; Roitzsch, 2005; Spillner & Linz, 2012; Hoffmann, 2013).

Es gibt unterschiedliche Möglichkeiten, die Integrationsstrategien zu clustern. Eine mögliche Kategorisierung ist in Abbildung 3.3 dargestellt. Hier gibt es drei unter-

schiedliche Strategien mit ihren jeweiligen Ausprägungen, die im Folgenden genauer dargestellt werden (Hoffmann, 2013):

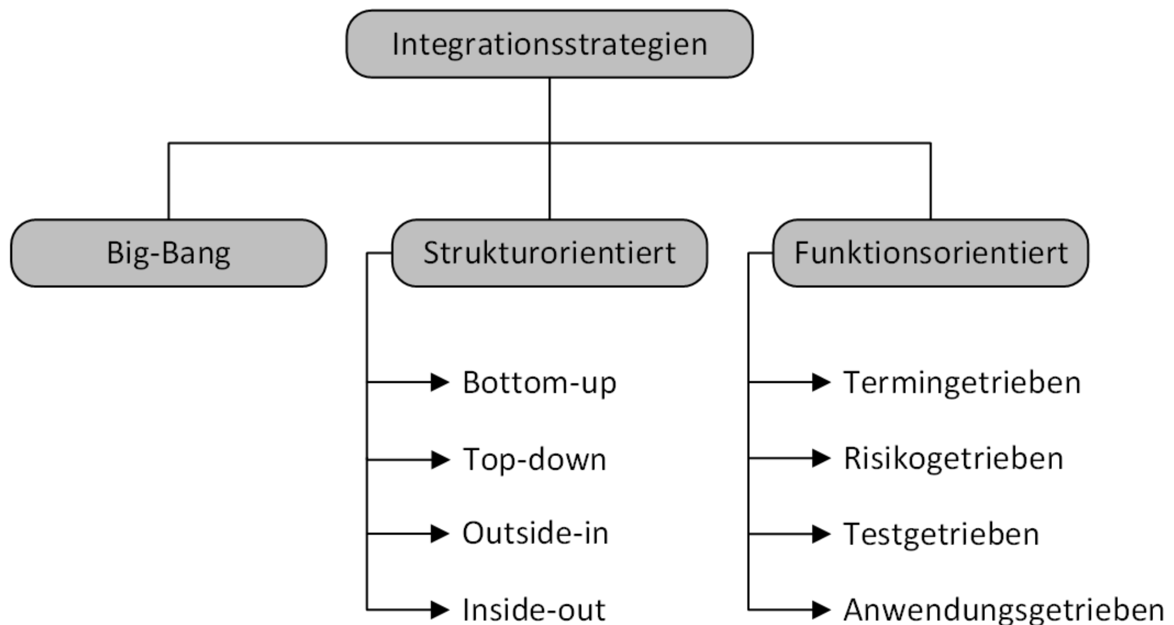


Abbildung 3.3.: Ein möglicher Überblick über die verschiedenen Integrationsstrategien.

- **Big-Bang-Integration**
  Hinter dem Begriff *Big-Bang-Integration* steht keine Integrationsstrategie im eigentlichen Sinne. Die Units werden zuerst vollständig entwickelt und im Anschluss auf einen Schlag integriert. Aufgrund dieser Tatsache ist es offensichtlich, dass dieses Vorgehen zwei wesentliche Nachteile beinhaltet:

  1. Zum einen kann mit der Integration erst begonnen werden, wenn die letzte Unit fertig entwickelt ist. Dies hat einen zeitlichen Einfluss auf das Projekt, da Fehler in der Integration so erst später entdeckt werden können.
  2. Zum anderen lassen sich Fehler bei der Big-Bang-Integration schwerer auf einzelne Komponenten zurückführen, als dies bei der inkrementellen Integration der Fall ist. Dadurch steigt der Aufwand für das Debugging massiv an.

  Insgesamt wirkt sich diese Vorgehensweise negativ auf die Reaktionszeiten sowie auf das Risiko des Projekts aus und soll vermieden werden (Leach, 2016; Puntambekar, 2009; Spillner & Linz, 2012; Hoffmann, 2013).

- **Strukturorientierte Integration**
  Die *Strukturorientierte Integration* ist im Gegensatz zur Big-Bang-Integration eine echte Integrationsstrategie. Die einzelnen Units werden inkrementell zu einem

Gesamtsystem zusammengefügt. Die Reihenfolge richtet sich dabei nach den strukturellen Abhängigkeiten, die zwischen den einzelnen Modulen auf Architekturebene bestehen (Roitzsch, 2005; Hoffmann, 2013).
Wie in Abbildung 3.4 dargestellt, sind folgende Integrationen möglich:

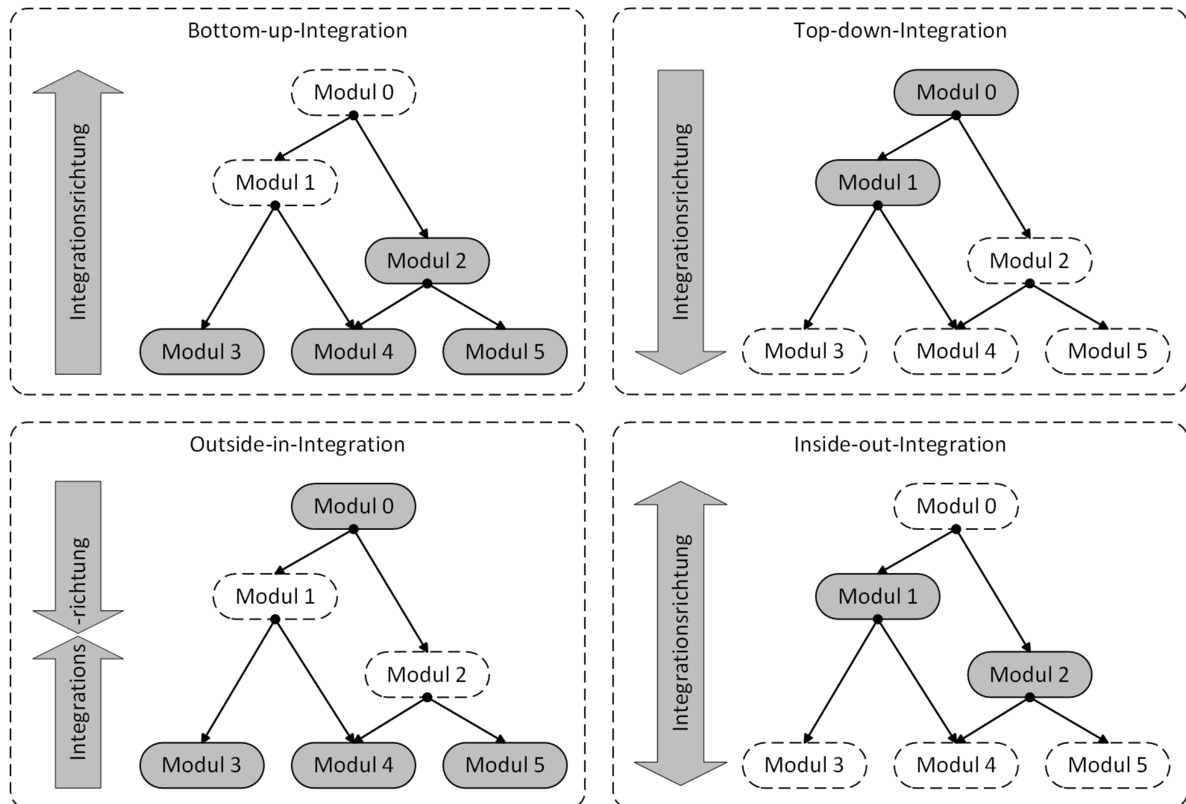


Abbildung 3.4.: Die verschiedenen strukturorientierten Integrationsstrategien im Vergleich (vgl. Hoffmann, 2013).

– **Bottom-up-Integration**
  Bei dieser Integrationsstrategie werden zuerst alle Units integriert, die keine Abhängigkeiten zu anderen Units besitzen. Diese nennt man Basiskomponenten. Im Anschluss folgen alle weiteren Elemente, die Abhängigkeiten zu anderen aufweisen. Die Reihenfolge ist hier so festgelegt, dass die Komponenten mit den wenigsten Abhängigkeiten zuerst integriert werden. Durch die Verwendung eines Schichtenmodells oder einer Baumbeschreibung der Programmstruktur, siehe dazu Abbildung 3.4, ist es möglich, zuerst alle Blätter des Baumes und dann sukzessive nacheinander alle Ebenen des Modells zu integrieren (Leach, 2016; Puntambekar, 2009; Spillner & Linz, 2012; Hoffmann, 2013).

  Solange das so konstruierte Softwaresystem nicht vollständig ist, werden die noch nicht integrierten Komponenten durch Testtreiber, sogenannte *driver*, ersetzt. Diese driver simulieren bestimmte Aspekte der späteren Im-

plementierung. Weiters sind die driver nicht vollständig und behandeln nur bestimmte, ausgewählte Testszenarien des später fertiggestellten Softwaresystems (Leach, 2016; Puntambekar, 2009; Spillner & Linz, 2012; Hoffmann, 2013).

  - **Top-down-Integration**
    Die *Top-down-Integration* weist im Vergleich zur Bottom-up-Integration die umgekehrte Integrationsrichtung auf, siehe dazu Abbildung 3.4. Die Basiskomponenten werden demnach zuletzt integriert. Noch nicht integrierte Elemente werden durch Platzhalter, sogenannte *stubs*, ersetzt. Diese bilden die Funktionalität der Komponente ab. Stubs sind normalerweise unvollständig und liefern nur für einige wenige Eingabeparameter die entsprechenden Ergebnisse (Leach, 2016; Puntambekar, 2009; Spillner & Linz, 2012; Hoffmann, 2013).

    Vergleicht man diese Integrationsstrategie mit der Bottom-up-Integration, erweist sich erstere in den meisten Fällen als schwieriger. Daher stößt sie bei den EntwicklerInnen öfter auf Ablehnung. Für KundInnen des Systems ist diese Strategie durchaus von Vorteil, da sich sehr früh ein Prototyp zeigen lässt, der sich äußerlich kaum vom Endprodukt unterscheidet (Leach, 2016; Puntambekar, 2009; Spillner & Linz, 2012; Hoffmann, 2013).

  - **Outside-in-Integration**
    Diese Vorgehensweise verbindet die Vorteile und vermeidet einige Nachteile der beiden eben genannten Integrationsstrategien. Es werden also die obersten und untersten Architekturebenen zuerst integriert. Im Anschluss folgen sukzessive die nächsten Schichten auf beiden Seiten, bis das System in der Mitte zusammenwächst. Durch diese Vorgehensweise ist es möglich, früh ein prototypisches Modell zu erhalten. Gleichzeitig entfällt die Entwicklung von Platzhaltern für die Komponenten der unteren Architekturebenen, was wiederum Zeit spart (Liggesmeyer, 2009; Hoffmann, 2013).

  - **Inside-out-Integration**
    Hierbei wird die Outside-in-Integration umgedreht. Mit der Integration der Lösung wird in der Mitte begonnen. Diese Strategie wird in der Praxis kaum verwendet, da sie die Nachteile der Bottom-up-Integration und Top-down-Integration kombiniert (Hoffmann, 2013).

- **Funktionsorientierte Integration**
  Im Gegensatz zur strukturorientierten Integration wird die Reihenfolge durch funktionale oder operative Kriterien bestimmt. Folgende Integrationen sind möglich (Hoffmann, 2013):

  - **Termingetriebene bzw. Ad-hoc Integration**
    Die verschiedenen Units werden aufgrund ihrer Verfügbarkeit integriert.

Dieses *first come first serve* Prinzip bestimmt die Integration (Hoffmann, 2013; Spillner & Linz, 2012).

– **Risikogetriebene Integration**
Dieser Ansatz verfolgt die Idee, die risikoreichsten Units, *hardest first*, zuerst zu integrieren. Dadurch können Risiken, z.B. Schnittstellen, entsprechend frühzeitig behandelt werden (Hoffmann, 2013).

– **Testgetriebene Integration**
Bei der *testgetriebenen Integration* werden anhand von Testcases die dafür unbedingt notwendigen Units integriert. Die restlichen Units werden mithilfe einer strukturorientierten Integration zusammengefügt (Hoffmann, 2013).

– **Anwendungsgetriebene Integration**
Diese Integrationsstrategie ist eng mit der testgetriebenen Integration verwandt. Anstatt eines einzelnen Testszenarios wird hier ein vollständiger Geschäftsfall bzw. ein konkretes Anwendungsszenario, auch *use case* genannt, verwendet. Die weitere Vorgehensweise ist identisch (Hoffmann, 2013).

3.1.3. Der Systemtest

Um einen Systemtest durchführen zu können, ist es notwendig, dass alle Units, die als Teil des zu testenden Systems bzw. Teilsystems gelten, den Integrationstest erfolgreich absolviert haben. Der Systemtest geht nun einen Schritt weiter und testet im Gegensatz zum Unittest bzw. Integrationstest nicht mehr den Code, sondern ausschließlich die funktionalen Kriterien, wie in Abbildung 3.2 dargestellt (Cleff, 2010; Spillner & Linz, 2012; Hoffmann, 2013).

Dies bedeutet, dass die produzierende Firma bzw. EntwicklerInnengemeinde aus Sicht der KundInnen und späteren AnwenderInnen die Software betrachtet und testet. Die AuftraggeberInnen und spätere AnwenderInnen müssen nicht dieselbe Personengruppe bzw. Organisation darstellen und können demnach unterschiedliche Interessen in Bezug auf das System repräsentieren. Für die Firma bzw. die EntwicklerInnengemeinde ist es die letzte Teststufe in ihrem Verantwortungsbereich, bevor die Software im nächsten Schritt von den AuftraggeberInnen getestet wird (Cleff, 2010; Spillner & Linz, 2012; Hoffmann, 2013).

Der Aufwand, den Systemtest durchzuführen, sollte nicht unterschätzt werden. Dies kann bedeuten, dass der Systemtest mehr Zeit in Anspruch nimmt wie der Unit- und Integrationstest zusammen. Folgende Faktoren können das Testen erheblich erschweren (Cleff, 2010; Spillner & Linz, 2012; Hoffmann, 2013):

- **Unvorhergesehene Fehler der Units als Systemverbund**
Erfahrungsgemäß treten viele Fehler erst im Zusammenspiel der einzelnen Units als Systemkomponente auf. Diese fallen in den Unit- bzw. Integrationstests oft nicht auf (Spillner & Linz, 2012; Hoffmann, 2013).

- **Unklare bzw. unscharfe Anforderungen**
Je nachdem wie scharf bzw. klar die Anforderungen spezifiziert sind, gestalten sich die entsprechenden Spielräume. Je größer diese sind, desto schwieriger ist es, das Systemverhalten exakt und zweifelsfrei festzustellen. Wichtig dabei ist es, die gefundenen Annahmen und Unterschiede zwischen den Anforderungen und dem tatsächlichen Verhalten des Systems lückenlos zu dokumentieren (Spillner & Linz, 2012; Hoffmann, 2013).

- **Der Einfluss der Testumgebung**
Der Systemtest soll nach Möglichkeit auf einer Plattform durchgeführt werden, die der des Kunden im möglichen Produktiveinsatz entspricht. Dennoch sollte es nicht ein gerade aktives Produktivsystem des Kunden sein. Das kann unter Umständen sehr aufwändig sein, zeigt aber mögliche Schwachstellen frühzeitig auf (Cleff, 2010; Spillner & Linz, 2012; Hoffmann, 2013).

- **Eingeschränkte Debugging Möglichkeiten**
Dadurch, dass die Tests sich auf die Funktionen des Systems bzw. Subsystems konzentrieren und den Code außen vor lassen, sind die Debugging Möglichkeiten (stark) eingeschränkt oder gar nicht mehr vorhanden (Hoffmann, 2013).

- **Vorhandene Testdaten**
Um das System testen zu können, müssen ausreichend Testdaten vorhanden sein. Diese müssen konsistent, aktuell und vollständig sein. Können keine Daten aus der Produktion verwendet werden, stellt die Anonymisierung oder Generierung von Pseudodaten eine Herausforderung dar (Cleff, 2010; Spillner & Linz, 2012).

- **Eingeschränkte Handlungsfähigkeit**
Da die Tests nun sehr weit forgeschritten sind, können weitreichende Entscheidungen im Sinne der Risikominimierung für das Projekt, wie z.B. Architekturentscheidungen, nicht mehr geändert werden. Eingriffe finden demnach auf mikrochirurgischer Ebene statt und zielen darauf ab, Fehlereinpflanzungen und deren Folgen möglichst klein zu halten. Bei großen Diskrepanzen zwischen dem Ist- und dem Sollzustand kann der Systemtest offiziell das Scheitern des Projekts attestieren (Spillner & Linz, 2012; Hoffmann, 2013).

3.1.4. Der Abnahmetest

Der Abnahmetest findet nach erfolgreichem Systemtest statt und wie dieser wird auch der Abnahmetest, wie in Abbildung 3.2 dargestellt, rein funktional durchgeführt. Im Gegensatz zu allen bisherigen Tests hat er das Ziel, dass die AuftraggeberInnen bzw. KundInnen das Produkt entsprechend den vertraglich festgelegten Kriterien abnimmt. Weiters sollen die AuftraggeberInnen bzw. KundInnen feststellen, dass das Ergebnis der Bemühungen auch juristisch als Erfolg gewertet werden kann (Cleff, 2010; Spillner & Linz, 2012; Hoffmann, 2013).

Ein weiterer Unterschied zum Systemtest ist, dass die AuftraggeberInnen bzw. KundInnen für die Durchführung dieses Tests verantwortlich sind. Die produzierende Firma bzw. EntwicklerInnengemeinde kann dabei, wenn von den AuftraggeberInnen bzw. KundInnen gewünscht oder dies vertraglich festgelegt ist, unterstützend mitwirken. Dies muss aber nicht der Fall sein. Am Ende entscheiden die AuftraggeberInnen bzw. KundInnen über den erfolgreichen Test. Folgende Vorgehensweisen sind für Abnahmetests üblich (Cleff, 2010; Spillner & Linz, 2012; Hoffmann, 2013):

- **Vertragliche Abnahme**
  Wie bereits erwähnt, ist es wichtig, dass die AuftraggeberInnen bzw. KundInnen anhand der vertraglich festgelegten Kriterien abklären, ob das gelieferte System dem Inhalt des Vertrags entspricht. Nach Möglichkeit haben die KundInnen dafür selbst die Testszenarien und Vorgehensweisen zu entwickeln, um diese dann am System durchführen zu können (Cleff, 2010; Spillner & Linz, 2012; Hoffmann, 2013).

- **BenutzerInnenakzeptanz**
  Sollte das entwickelte System nicht von den AuftraggeberInnen selbst, sondern von einer anderen AnwenderInnengruppe benutzt werden, kann es sinnvoll sein, das System mit diesen AnwenderInnen auf dessen Akzeptanz zu testen. Damit beim Test *BenutzerInnenakzeptanz* nicht unvorhergesehene Dinge auftreten, ist es wichtig, diese Gruppe beim Design des Systems bereits einzubinden (Spillner & Linz, 2012).

- **Akzeptanz der SystembetreiberInnen**
  Durch das Testen der Akzeptanz der SystembetreiberInnen bzw. AdministratorInnen wird der operative Betrieb des Systems in der jeweiligen IT-Landschaft festgelegt. Wichtig für diese AnwenderInnengruppe ist, dass diese genügend Informationen über das System haben, um ihre Aufgaben in Bezug auf das System, wie z.B. das Verwalten von UserInnen, das Erstellen und Einspielen von Backups usw., wahrnehmen zu können (Cleff, 2010; Spillner & Linz, 2012).

- **Feldtests in Form von Alpha- und Beta-Tests**
  Soll die Software auf sehr vielen Produktivsystemen eingesetzt werden, kann

meistens nicht jedes Verhalten zuvor definiert bzw. vorhergesehen werden. Dafür werden Alpha- und Beta-Tests eingesetzt.

Beim *Alpha-Test* wird ausgewählten BenutzerInnen das System zur Verwendung vorgelegt. Alle Rückmeldungen gehen direkt an die HerstellerInnen bzw. an die EntwicklerInnen. Während der Alpha-Test in der Anwendungsumgebung der HerstellerInnen stattfindet, wird der *Beta-Test* in der Umgebung der KundInnen durchgeführt.

Im Beta-Test wird das Produkt mit vorgegebenen Testszenarien bzw. durch den Einsatz des Systems im vorläufigen Produktionsbetrieb getestet. Die Fehlersuche erweist sich hier aufgrund folgender Faktoren als schwierig:

- Die Software befindet sich in einer authentischen KundInnenumgebung.
- Die AnwenderInnen sind oft auf sich alleine gestellt.
- Es fehlen die Debugging Möglichkeiten von Seiten der HerstellerInnen (Cleff, 2010; Spillner & Linz, 2012; Hoffmann, 2013).

3.2. Die Prüfkriterien

Die Prüfebenen, wie im Abschnitt 3.1 beschrieben, beschäftigen sich mit der zeitlichen Abfolge der Testfälle im Softwareentwicklungsprozess. Die Prüfkriterien, auch Testarten genannt, behandeln die inhaltlichen Aspekte der Testfälle. Dies ist ebenso in Abbildung 3.1 zu erkennen.

Die Gliederung der Prüfkriterien bzw. Testarten wird in der Literatur verschieden vorgenommen, siehe dazu Abbildung 3.5. Hoffmann (2013) gliedert die Tests in die Kategorien *funktional*, *operational* und *temporal*. Dabei beleuchtet er den Umfang der Software z.B. aus der Sicht der AuftraggeberInnen bzw. EntwicklerInnen, den Einsatz der Software z.B. aus der Sicht der BetreiberInnen und das Verhalten der Software in zeitlichen Ausnahmefällen z.B. aus der Sicht der BetreiberInnen, der AnwenderInnen und der EntwicklerInnen.

Spillner und Linz (2012) gliedern die Tests primär nach der ISO-Norm 9126. Demnach gibt es eine Einteilung der sechs Qualitätsmerkmale bezüglich der Tests in die funktionalen- (ein Merkmal) und die nicht funktionalen (fünf Merkmale) Softwaretests. Ergänzt wird diese Einteilung durch zwei weitere Kategorien von Tests, dem *strukturbezogenen Test* und dem *änderungsbezogenen Test*.

Die ISO-Norm 25010, die ein Jahr vor dem überarbeiteten Werk von Spillner und Linz (2012) erschienen ist, die ISO-Norm 9126 ersetzt und ergänzt, wird im Werk nicht verwendet. Die Unterschiede der Normen sind jedoch nicht gravierend (ISO/IEC

9126, 2001; ISO/IEC 25010, 2011; Wagner, 2013; Spillner, Roßner et al., 2014; Spillner, Linz & Schaefer, 2014).

Eine Ausarbeitung konkreter Tests führen Spillner und Linz (2012) nicht durch. Sie verweisen bei den funktionalen Tests mehr oder weniger auf die Normen. Hoffmann (2013) bietet hier einen konkreteren Weg an, um die Tests zu benennen und diese zu kategorisieren. Diese Einteilung wird in Folge für die Erläuterungen der einzelnen Prüfkriterien herangezogen und näher behandelt. Ergänzt wird diese durch die Ausführungen von Spillner und Linz (2012).

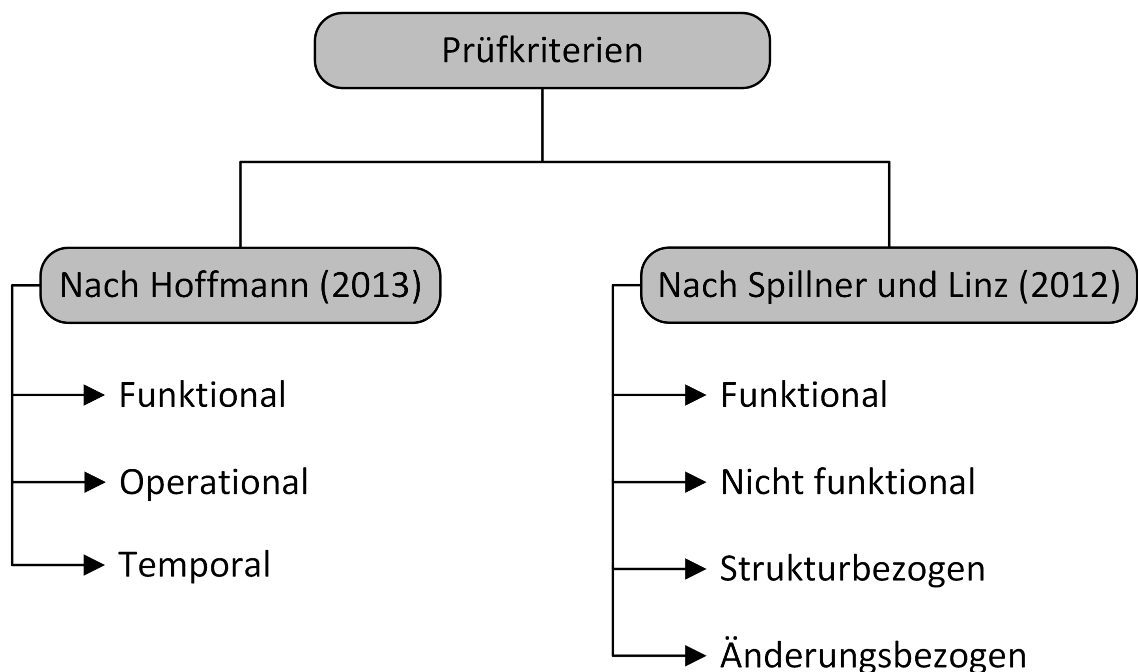


Abbildung 3.5.: Mögliche Gliederung der Prüfkriterien nach Hoffmann (2013) sowie Spillner und Linz (2012).

3.2.1. Der funktionale Softwaretest

- **Der Funktionstest**
  Dieser Test soll feststellen, ob das System bei entsprechenden Eingabewerten die erwarteten Ausgabewerte liefert. Diese Testkategorie kommt in der Praxis so häufig vor, dass sie oft generell mit dem Testen an sich gleichgesetzt wird (Spillner & Linz, 2012; Hoffmann, 2013).

- **Der Trivialtest**
  Der Trivialtest soll Grenzwerte abtesten. Dies kann z.B. die Sortierung einer leeren Liste oder die Streckung eines Vektors um den Faktor eins bedeuten. Diese Tests sind besonders einfach strukturiert (Hoffmann, 2013).

- **Der Crashtest**
  Tests, die darauf ausgelegt sind, das System zum Absturz zu bringen, indem sie Schwachstellen suchen und ausnutzen, nennt man Crashtests. Diese führen dazu, dass das System viel robuster gemacht werden kann. Besonders für sicherheitskritische Systeme sind diese Tests von großer Bedeutung (Hoffmann, 2013).

- **Der Kompabilitätstest**
  Der Test auf Kompabilität verfolgt das Ziel, die Eigenschaft der Portierbarkeit des Systems festzustellen. Die Einhaltung von Standards spielt hier eine große Rolle (Hoffmann, 2013).

- **Der Zufallstest**
  Alle bisher genannten Tests aus dieser Kategorie haben gemeinsam, dass die Eingabedaten anhand ausgewählter Parameter bestimmt werden. Der Zufallstest bricht mit dieser Vorgehensweise und verwendet als Eingabedaten zufällig generierte Werte. Dadurch wird das Testspektrum erweitert. Weiters ist es möglich, Fehler zu finden, die den anderen Tests verborgen geblieben wären (Hoffmann, 2013).

Die Definition des funktionalen Softwaretests nach Spillner und Linz (2012) führt alle Anforderungen auf die ISO-Norm 9126 zurück, die 2011 durch die ISO-Norm 25010 ersetzt und ergänzt wurde (ISO/IEC 9126, 2001; ISO/IEC 25010, 2011; Wagner, 2013; Spillner, Roßner et al., 2014; Spillner, Linz & Schaefer, 2014). Laut Spillner und Linz (2012) sind die Merkmale der Funktionalität, entsprechend der ISO-Norm 9126, Angemessenheit, Interoperabilibät, Ordnungsmäßigkeit, Richtigkeit und Sicherheit.

Der Funktions- und Trivialtest sind in diesen Merkmalen enthalten. Der Kompabilitäts- sowie Crashtest gehören bei Spillner und Linz (2012) in die Kategorie der nicht funktionalen Softwaretests, da sich der Kompabilitätstest laut ISO-Norm 9126 dem Qualitätsmerkmal Übertragbarkeit sowie der Crashtest dem Qualitätsmerkmal Zuverlässigkeit zuordnen lässt.

Aus der Sicht von Spillner und Linz (2012) gehört der Zufallstest als Erweiterung aller möglichen Tests je nach Anwendungsfall entweder zu den funktionalen oder zu den nicht funktionalen Softwaretests.

3.2.2. Der operationale Softwaretest

- **Der Installationstest**
  Dieser Test hat die Aufgabe festzustellen, wie die Inbetriebnahme des Systems

auf unterschiedlichen Hard- und Softwaresystemen abläuft. Auch die Aktualisierung bzw. das Upgrade auf eine neue Version der Software ist Teil dieses Tests (Hoffmann, 2013).

- **Der Ergonomietest**
  Der *Ergonomietest* zeigt auf, inwieweit ein Softwaresystem für die BenutzerInnen intuitiv verwendbar ist. Dazu gehören die grafischen Elemente, deren Anordnung und Kategorisierung, genauso wie die integrierte Hilfe oder die Installationsdokumentation (Spillner & Linz, 2012; Hoffmann, 2013).

- **Der Sicherheitstest**
  Um die Unbedenklichkeit, z.B. in Bezug auf Schwachstellen, Systemzugänge oder die vertrauliche Speicherung von Daten, zu testen, werden sogenannte Sicherheitstests durchgeführt. Bei Software, die in sicherheitskritischen Bereichen eingesetzt wird, muss nachweisbar sein, dass im Regelbetrieb keine Gefahr für Leib und Leben besteht (Spillner & Linz, 2012; Hoffmann, 2013).

Laut Spillner und Linz (2012) lassen sich der Installations- und der Ergonomietest in die Kategorie der nicht funktionalen Softwaretests eingliedern, da sich der Installationstest dem Qualitätsmerkmal Übertragbarkeit und der Ergonomietest dem Qualitätsmerkmal Benutzbarkeit zuordnen lässt. Der Sicherheitstest ist der Kategorie der funktionalen Softwaretests zuzuordnen, da dieser das Qualitätsmerkmal Funktionalität aufweist.

3.2.3. Der temporale Softwaretest

- **Der Komplexitätstest**
  Dieser Test stellt sicher, dass die implementierten Algorithmen in der zuvor spezifizierten Komplexitätsklasse liegen. Dadurch lassen sich Laufzeitprobleme im Vorfeld aufzeigen und bei frühzeitigem Start der Tests einschränken bzw. vermeiden (Hoffmann, 2013).

- **Der Laufzeittest**
  Vereinbarte Laufzeiten von Kernfunktionen wie z.B. von Geschäftsfällen oder der Ausführung bestimmter Funktionen lassen sich durch den *Laufzeittest* messen bzw. feststellen (Spillner & Linz, 2012; Hoffmann, 2013).

- **Der Lasttest**
  Der *Lasttest* untersucht das Verhalten des Softwaresystems im Grenzbereich der Spezifikation, also unter sehr hoher Last. Auch unter diesen Bedingungen soll das System die entsprechenden Prozesse richtig abarbeiten. Die Vorbereitung und Durchführung dieser Tests erfordert einiges an Aufwand, kann aber mitunter für den Geschäftsfall essentiell sein (Spillner & Linz, 2012; Hoffmann, 2013).

- **Der Stresstest**
  Der *Stresstest* ist eine Erweiterung des Lasttests. Hier wird bewusst die Grenze des Systems überschritten und beobachtet, wie das System damit umgeht. So ist es möglich, experimentell die realen Belastungsgrenzen des Systems zu ermitteln. Eine weitere wichtige zu klärende Frage ist, ob das System nach dem Wegfall der Überlast wieder in den Normalbetrieb zurückfindet. Dies nennt man auch *stress recovery* (Spillner & Linz, 2012; Hoffmann, 2013).

Wie bereits in den Abschnitten 3.2.1 und 3.2.2 erwähnt, führen Spillner und Linz (2012) alle Anforderungen auf die ISO-Norm 9126 zurück. Alle eben genannten Tests fallen daher laut Spillner und Linz (2012) in die Kategorie der nicht funktionalen Softwaretests und verteilen sich auf mehrere Qualitätsmerkmale.

3.2.4. Der strukturbezogene Test

Dieser Test dient dazu, die internen Strukturen und/oder Architekturen der Software zu untersuchen. Analysiert werden unter anderem Kontrollflüsse innerhalb der Units und die Aufrufhierarchien von Abläufen oder Menüstrukturen. Der Fokus dieses Tests liegt darauf, möglichst alle Elemente der zu untersuchenden Strukturen durch die Tests zu erreichen bzw. abdecken zu können. Dies erfordert eine entsprechende Planung bei der Gestaltung der Testszenarien. Diese Art von Test kommt vor allem bei Unit- und Integrationstests vor (Spillner & Linz, 2012).

3.2.5. Die änderungsbezogenen Tests

Durch Fehlerkorrekturen bzw. Wartungsarbeiten am Code werden verschiedene Elemente und Verhaltensweisen des Systems geändert. Hier muss sichergestellt werden, dass diese entsprechenden Tests nochmals durchgeführt werden und die korrekten Ergebnisse liefern. Diese Art von Test nennt man *Fehlernachtest*. Weiters muss festgestellt werden, dass die Änderungen keine unbeabsichtigten Seiteneffekte hervorrufen. Diese Art von Test nennt man *Regressionstest*.

Der Regressionstest ist somit kein weiterer inhaltlicher Test, sondern beschreibt eine Vorgehensweise des Testens nach Änderungen. Dies ähnelt dem Zufallstest aus dem Abschnitt 3.2.1. Wenn also Regressionstests stattfinden, können diese inhaltlich funktionaler, operationaler, temporaler oder struktureller Natur sein (Spillner & Linz, 2012).

3.3. Die Prüftechniken bzw. -methoden

Die *Prüftechniken* bzw. *-methoden* beschreiben konkrete Vorgehensweisen, mit deren Hilfe Softwaretests *konstruiert* bzw. *entworfen* werden können. Daher werden sie auch *Testfallentwurfsverfahren*, in dieser Arbeit größtenteils *Testverfahren*, genannt (Spillner & Linz, 2012; Hoffmann, 2013).

Diese lassen sich prinzipiell folgenden Kategorien zuordnen (Saleh, 2009; Franz, 2014; Liggesmeyer, 2009; Spillner & Linz, 2012; Hoffmann, 2013):

- **Black-Box Testverfahren:** Diese Testverfahren beruhen auf der Annahme, dass ein Einblick in den Code nicht möglich ist. Das Testobjekt wird als *schwarzer Kasten* bzw. *black-box* angesehen und auch in Bezug auf die Erstellung, Durchführung und Steuerung der Tests so behandelt. Weitere Aspekte dazu werden im Abschnitt 3.3.1 erörtert.
- **White-Box Testverfahren:** Diese Testverfahren beruhen auf der Annahme, dass ein Einblick in den Code möglich ist. Das Testobjekt wird als *weißer Kasten* bzw. *white-box* angesehen und auch in Bezug auf die Erstellung, Durchführung und Steuerung der Tests so behandelt.Weitere Aspekte werden im Abschnitt 3.3.2 erörtert.
- **Grey-Box Testverfahren:** Diese Testverfahren sind ein Sonderfall. Personen, die Black-Box Testverfahren entwerfen, jedoch Zugriff auf den Code haben, nutzen klarerweise dieses Wissen für den Entwurf des Testverfahrens. Dazu gehören typischerweise EntwicklerInnen. Dadurch ist das Vorgehen kein klassiches Black-Box Testfallentwurfsverfahren. Ein White-Box Testverfahren ist es aber auch nicht, da hierfür die systematische Nutzung der Codestruktur vonnöten ist.
- **Erfahrungsbasierte Testverfahren:** Bei diesen Testverfahren spielen die testenden Personen und nicht die Methodiken und Systematiken eine große Rolle. Zum einen ist es wichtig, dass diese Stakeholder einiges an Erfahrung mitbringen, zum anderen die Verantwortung für den Testvorgang in weiten Teilen übernehmen. Weitere Aspekte dazu werden im Abschnitt 3.3.3 erörtert.

Im Folgenden sollen die einzelnen Kategorien mit einer Auswahl der jeweiligen Testverfahren bzw. Testfallentwurfsverfahren dargelegt werden.

3.3.1. Black-Box Testverfahren

Die Black-Box Testverfahren beruhen darauf, dass der Programmcode bzw. die Programmstruktur nicht bekannt ist. Für die Erstellung der Testfälle wird daher die Spezifikation des zu testenden Systems verwendet. Dabei werden die Verhaltensweisen der Ein- und Ausgabewerte des Systems berücksichtigt. Deshalb wird empfohlen, die

Testerstellung sofort nach dem Design für die zu entstehende Software zu entwerfen (Wang & Tan, 2006; Agarwal, Tayal & Gupta, 2010; Metzner, 2020; Saleh, 2009; Spillner & Linz, 2012; Hoffmann, 2013).

Black-Box Tests werden oft mit funktionalem Testing gleichgesetzt. Durch die Vorlage der Spezifikation sind natürlich die Funktionalität und die Schnittstellen bekannt. Die Spezifikation enthält neben den funktionalen eventuell auch nicht funktionale Requirements. Dies kann mit Black-Box Testverfahren abgebildet werden (Wang & Tan, 2006; Agarwal et al., 2010; Saleh, 2009; Hoffmann, 2013).

Black-Box Testverfahren sollen grundsätzlich Folgendes sicherstellen (Agarwal et al., 2010):

- Die Funktionalität des Systems wird auf dessen Validität überprüft
- Die Eingabeparameter werden auf deren Gültigkeit getestet
- Die Eingabeparameter erzeugen das erwünschte und spezifizierte Verhalten am Ausgang des Systems
- Das System ist gültig und valide in Abhängigkeit zum Volumen der Operationen am System
- Seltenere Kombinationen werden beim Testing berücksichtigt

Die Testvollständigkeit wird durch die Abdeckung der Eingabewerte anhand der Spezifikation beurteilt. Sollte daher die Spezifikation Lücken aufweisen, kann sich dies auf den Black-Box Test auswirken (Metzner, 2020).

3.3.1.1. Der Äquivalenzklassentest

Beim Testen der Eingabeparameter eines Systems ist es meist nicht machbar, alle möglichen Werte zu berücksichtigen und die Ergebnisse mit den Sollergebnissen des Systems zu vergleichen. Die Methode der *Äquivalenzklassenbildung* stellt dafür eine Lösung bereit (Frühauf et al., 2007; Daigl & Glunz, 2016; Liggesmeyer, 2009; F. Wolf, 2018; Spillner & Linz, 2012; Hoffmann, 2013).

Die möglichen Eingabeparameter werden in Klassen *aufgeteilt bzw. partitioniert* und somit voneinander klar abgegrenzt, siehe dazu das Beispiel in Tabelle 3.1. Die Bestimmung der Äquivalenzklassen beruht meist auf Erfahrung und Intuition und kann nicht durch starre Regeln abgebildet werden (Frühauf et al., 2007; Daigl & Glunz, 2016; Liggesmeyer, 2009; F. Wolf, 2018; Spillner & Linz, 2012; Hoffmann, 2013).

Sollte genau ein Parameter aufgeteilt bzw. partitioniert werden, wird dieser Vorgang *eindimensionale Äquivalenzklassenbildung* genannt. Bei der *mehrdimensionalen Äquivalenzklassenbildung* werden Parameter den einzelnen Wertebereichen entspre-

Eingabebereich:	$1 \leq$ Wert ≤ 99
Gültige Äquivalenzklasse:	$1 \leq$ Wert ≤ 99
Ungültige Äquivalenzklasse:	Wert ≤ 1
Ungültige Äquivalenzklasse:	Wert ≥ 99

Tabelle 3.1.: Beispiel zur Aufteilung in entsprechende Äquivalenzklassen (vgl. Liggesmeyer, 2009).

chend partitioniert und die zusammengehörigen Äquivalenzklassen im Anschluss verschmolzen (Hoffmann, 2013).

Die Vorgehensweise bei der Partitionierung hängt vom zu erwartenden Verhalten des Systems ab. Eine *partielle Partitionierung*, also eine Äquivalenzklassenbildung ohne Berücksichtigung der ungültigen Eingabewerte, macht dann Sinn, wenn sichergestellt ist, dass diese nicht vorkommen können oder die Beschreibung des Systems, z.B. einer Funktion, keine Angaben zu dem zu erwarteten Funktionswert macht.
Die *vollständige Partitionierung* dagegen bindet alle Optionen des Eingabewertes ein. Vor allem bei sicherheitskritischen Systemen ist dieses Vorgehen durchaus sehr beliebt (Hoffmann, 2013).

Der Testvorgang an sich ist simpel: Eine Klasse von Parametern soll immer dasselbe Verhalten im System haben. Somit muss pro Äquivalenzklasse genau ein Parameter getestet werden. Welcher das ist, spielt dabei keine Rolle. Dies reduziert die notwendigen Testfälle auf ein Minimum, ohne dass Verhaltensweisen ungetestet bleiben müssen (Frühauf et al., 2007; Daigl & Glunz, 2016; F. Wolf, 2018; Liggesmeyer, 2009; Spillner & Linz, 2012; Hoffmann, 2013).

Die Anzahl der Testfälle ergibt sich dabei aus der Anzahl der Parameter sowie deren Eigenschaften. Die Testfälle haben bei vollständiger Partitionierung mindestens eine gültige und eine ungültige Äquivalenzklasse. Die Summe der nötigen Tests ergibt sich aus der Summe vom Produkt der Anzahl der gültigen Äquivalenzklassen pro Parameter sowie der Summe der Anahl der ungültigen Äquivalenzklassen pro Parameter. Diese Sachverhalte sind in den Gleichungen 3.1, 3.2 und 3.3 dargelegt (Spillner & Linz, 2012).

$$Anzahl\ der\ Testsfälle = Summe\ der\ gültigen + ungültigen\ Testfälle \tag{3.1}$$

$$Anzahl\ der\ gültigen\ Testsfälle = Produkt\ der\ gültigen\ Äquivalenzklassen\ pro\ Parameter \tag{3.2}$$

$$Anzahl\ der\ ungültigen\ Testsfälle = Summe\ der\ ungültigen\ Äquivalenzklassen\ pro\ Parameter \quad (3.3)$$

Bei drei Parametern mit jeweils zwei gültigen Äquivalenzklassen ergeben sich daraus gemäß Gleichung 3.2 acht Testfälle. Bei drei Parametern mit jeweils zwei ungültigen Äquivalenzklassen ergeben sich daraus gemäß Gleichung 3.3 sechs Testfälle. Die Anzahl der Testfälle, siehe dazu Gleichung 3.1, beträgt demnach 14.

3.3.1.2. Die Grenzwertbetrachtung

Die Grenzwertbetrachtung ist eine durchaus sinnvolle Erweiterung des Äquivalenzklassentests. Dadurch ergibt sich, dass die Äquivalenzklassen bereits vorhanden sein sollten oder für diesen Test neu erstellt werden müssen. Damit die Methode anwendbar ist, müssen die Werte von numerischer Natur sein bzw. einen Wertebereich darstellen, für den sich Grenzen identifizieren lassen (Grünfelder, 2017; Kribernegg, 2013; Spillner & Linz, 2012; Hoffmann, 2013).

Die Idee der Methode sowie der entscheidende Unterschied zum Äquivalenzklassentest besteht darin, Fehler, die vor allem an den Grenzen von Äquivalenzklassen auftreten, aufzudecken. Dies geschieht durch die gezielte Auswahl der Werte, siehe dazu Tabelle 3.2 (Grünfelder, 2017; Kribernegg, 2013; Spillner & Linz, 2012; Hoffmann, 2013).

Laut Hoffmann (2013) wird dies bei eindimensionalen Äquivalenzklassen angewendet, indem der innere und der äußere Wert des unteren und des oberen Randes der Klasse als Testfall verwendet wird. Bei mehrdimensionalen Äquivalenzklassen wird das Randwerte-Tupel sowie alle weiteren Tupel, für die ein einzelner Wert außerhalb der Äquivalenzklasse liegt, als Testfall verwendet.

Die Anzahl der Testfälle unter Betrachtung der Äquivalenzklasse bzw. der Dimension n ergibt sich wie folgt: Pro Grenzpunkt enstehen $(n + 1)$ Testfälle. Weiters errechnen sich 2^n Grenzpunkte. In Summe ergibt dies in Abhängigkeit der Dimension n $(n + 1)\ 2^n$ Testfälle. In der Praxis kommen meist weit weniger Testfälle zustande, da die Äquivalenzklassen direkt aneinandergrenzen und somit verschmolzen werden können. Weiters sind manche Äquivalenzklassen zu einer oder mehreren Seiten offen und haben deshalb eine reduzierte Anzahl an Grenzpunkten (Hoffmann, 2013).

Spillner und Linz (2012) sowie Grünfelder (2017), der sich auf Spillner und Linz (2012) beruft, nehmen im Gegensatz zu Hoffmann (2013) noch einen dritten Punkt in ihre Betrachtung auf: Den exakten Grenzwert (siehe dazu Tabelle 3.2).

Erste Grenzwertbetrachtung					
Bsp. Nr.	Implementierte Abfrage	0	1	2	Anmerkung
1.1	$1 \leq x$ (korrekte Abfrage)	false	true	true	Erwartetes Ergebnis
1.2	$1 < x$	true	false	true	Fehler aufdeckend
1.3	$0 \leq x$	true	true	true	Fehler aufdeckend

Zweite Grenzwertbetrachtung					
Bsp. Nr.	Implementierte Abfrage	98	99	100	Anmerkung
2.1	$99 \leq x$ (korrekte Abfrage)	false	true	true	Erwartetes Ergebnis
2.2	$99 < x$	false	false	true	Fehler aufdeckend
2.3	$98 \leq x$	true	true	true	Fehler aufdeckend

Tabelle 3.2.: Grenzwertbetrachtungen zur Äquivalenzklasse aus Tabelle 3.1

3.3.1.3. Der zustandsbasierte Softwaretest

Die bisherigen Methoden bzw. Tests haben eines gemeinsam: Sie berücksichtigen den jeweiligen Eingabewert, nicht aber den bisherigen Ablauf der Eingaben in das System. Dies ist bei den sogenannten *gedächtnislosen Systemen* völlig ausreichend, bei den *gedächtnisbehafteten Systemen* jedoch nicht. Der Ausgabewert dieser Systeme hängt neben dem Input wesentlich vom aktuellen Status des Systems ab. In diesen Fällen bietet die Methode des zustandsbasierten Softwaretests Abhilfe (Grünfelder, 2017; F. Wolf, 2018; Liggesmeyer, 2009; Spillner & Linz, 2012; Hoffmann, 2013).

Das zu beschreibende gedächtnisbehaftete System oder Testobjekt kann ausgehend von einem Startzustand mehrere andere Zustände annehmen. Diese Zustandsänderungen oder -übergänge können durch verschiedene Ereignisse, wie z.B. Funktionsaufrufe, ausgelöst werden. Dargestellt werden diese Zusammenhänge meist in einem Zustandsautomaten. Dieser vermag es, die Zustände, die jeweiligen Übergänge und begleitenden Aktionen in kompakter und übersichtlicher Form darzustellen (Grünfelder, 2017; F. Wolf, 2018; Liggesmeyer, 2009; Spillner & Linz, 2012; Hoffmann, 2013).

Um die Zusammenhänge zu verdeutlichen, ist es sinnvoll, ein Beispiel exemplarisch zu betrachten. In der Literatur werden dazu viele genannt. Diese reichen vom Ringpuffer über einen Stapel hin zu Teilnehmeranschlüssen sowie einer Hebebühne. Alle haben gemeinsam, dass sie mit der Beschreibung des Systems starten (F. Wolf, 2018; Grünfelder, 2017; Liggesmeyer, 2009; Spillner & Linz, 2012; Hoffmann, 2013).

Wie in Abbildung 3.6 dargestellt, hat der Ringpuffer drei mögliche Zustände: Leer, befüllt und voll. Im leeren Zustand sind keine Daten vorhanden und die beiden Zeiger stehen an derselben Position. Im befüllten Zustand sind Daten vorhanden und der Schreibzeiger ist dem Lesezeiger voraus. Im vollen Zustand ist der Puffer befüllt und der Schreibzeiger dem Lesezeiger um eine Runde minus ein Element voraus. Sollten weitere Elemente beschrieben werden, wird das Element an der Position des Lesezeigers vom Schreibzeiger überschrieben und der Lesezeiger um eines weitergesetzt (F. Wolf, 2018; Hoffmann, 2013).

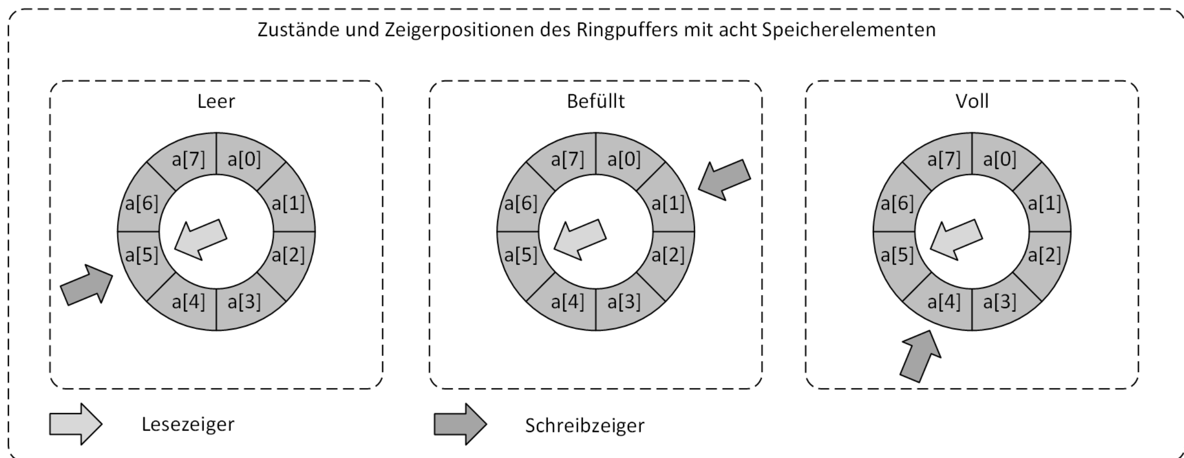


Abbildung 3.6.: Mögliche Zustände und Zeigerpositionen eines Ringpuffers unter der Annahme von acht Speicherelementen (vgl. Hoffmann, 2013).

Mit Hilfe von *Zustandsübergangsgraphen*, auch *endliche Automaten* genannt, lassen sich die entsprechenden Zustände als Kreise, der Startpunkt des Systems als Pfeil mit Start und die Zustandsübergänge als gerichtete Pfeile mit Beschriftung wie in Abbildung 3.7 darstellen (Grünfelder, 2017; F. Wolf, 2018; Liggesmeyer, 2009; Spillner & Linz, 2012; Hoffmann, 2013).

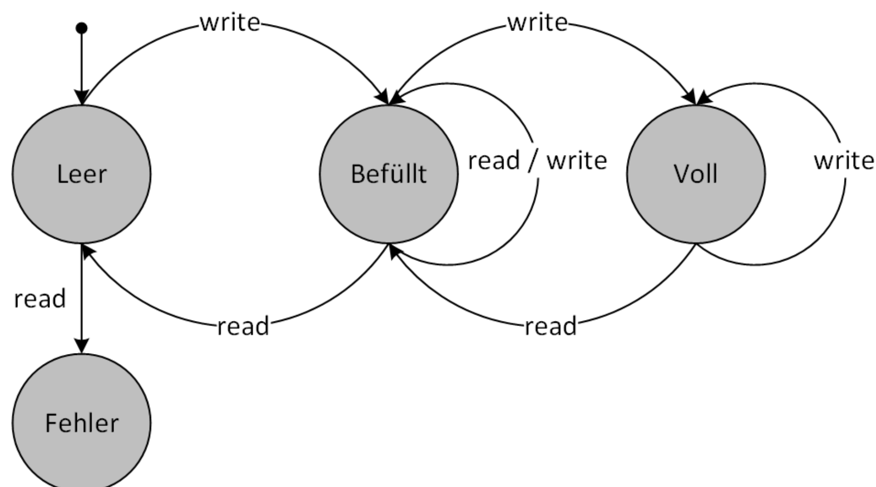


Abbildung 3.7.: Der Zustandsübergangsgraph des Rinpuffers (vgl. Hoffmann, 2013).

Mit Hilfe von *Zustandsübergangsgraphen*, auch *endliche Automaten* genannt, lassen sich die entsprechenden Zustände als Kreise darstellen. Die Beschriftung des Ausgangspunkts erfolgt als linearer Pfeil mit einem Punkt, die Zustandsübergänge werden mit gerichteten Pfeilen dargestellt (Grünfelder, 2017; F. Wolf, 2018; Liggesmeyer, 2009; Spillner & Linz, 2012; Hoffmann, 2013).

Aufbauend auf den Zustandsübergangsgraphen, wie in Abbildung 3.7 abgebildet, lässt sich durch das sogenannte *Ausrollen* des Zustandsübergangsgraphen der dazugehörige Zustandsbaum kreieren. Dieser ist in Abbildung 3.8 dargestellt (Grünfelder, 2017; F. Wolf, 2018; Spillner & Linz, 2012; Hoffmann, 2013).

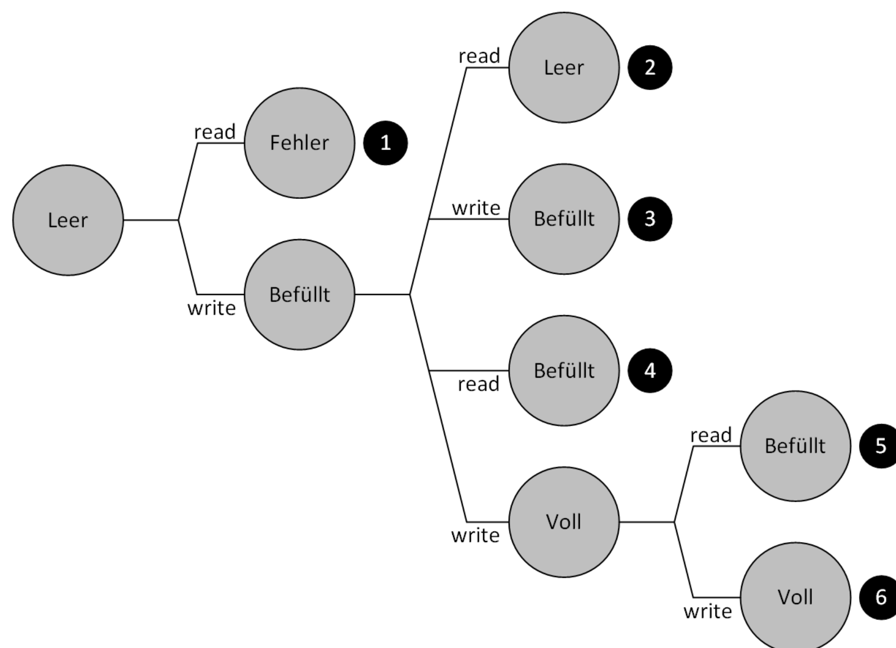


Abbildung 3.8.: Der Zustandsbaum des Rinpuffers (vgl. Hoffmann, 2013).

Für das Testen sind nun alle Fälle wichtig, die in diesem Baum ein Blatt darstellen. Diese Elemente werden mit einer schwarzen Nummer dargestellt und als Testfall angeführt. Die Testeingaben sowie die Soll-Ergebnisse, seien es die Puffer-Inhalte oder der Output des Systems, sind in Tabelle 3.3 aufgelistet. Insgesamt ergeben sich beim Ringpuffer sechs Testfälle (Grünfelder, 2017; F. Wolf, 2018; Liggesmeyer, 2009; Spillner & Linz, 2012; Hoffmann, 2013).

3.3.1.4. Der Use-Case Test

Die bis dato betrachteten Verfahren eignen sich sehr gut, um Tests für Klassen, Objekte und/oder Teilsysteme zu generieren. Daher werden diese oft für Unit- und/oder Integrationstests eingesetzt (Hoffmann, 2013).

Nr.	Testeingabe	Soll-Ergebnis (gelesen)	Soll-Ergebnis (Puffer-Inhalt)
1	read()	Fehler	{}
2	write(1), read()	{1}	{}
3	write(1), write(2)	{}	{1, 2}
4	write(1), write(2), read()	{1}	{2}
5	write(1), write(2), write(3), write(4), write(5), write(6), write(7), read()	{1}	{2, 3, 4, 5, 6, 7}
6	write(1), write(2), write(3), write(4), write(5), write(6), write(7), write(8)	{}	{2, 3, 4, 5, 6, 7, 8}

Tabelle 3.3.: Auflistung der notwendigen Tests aufgrund der Ergebnisse des ausgerollten Zustandsübergangsgraphen (vgl. Hoffmann, 2013).

Der *Use-Case Test*, auch *anwendungsfallbasierter Test* genannt, verlässt diese Einordnung und betrachtet das zu testende Softwarekonstrukt auf Systemebene. Dies hat zur Folge, dass die Implementationsdetails vernachlässigt werden und stattdessen die Arbeitsabläufe und vollständigen Anwendungsszenarien aus Kunden- oder Anwendersicht betrachtet werden (Franz, 2007; Kribernegg, 2013; Spillner & Linz, 2012; Hoffmann, 2013).

Für die Beschreibung bzw. Darstellung der Arbeitsabläufe und Anwendungsszenarien bietet sich zum einen die strukturierte, beschreibende Textform, z.B. die Use-Case Spezifikation oder das häufig verwendete *Use-Case Diagramm*, als übersichtliches, grafisches Tool an. Das Use-Case Diagramm wendet das *Unified Modeling Language* Konzept, auch *UML* genannt, an. Dieser 1997 eingeführte, 2005 erweiterte und sehr einfach gestaltete Standard liefert alle nötigen Elemente, die für die Darstellung nötig sind (Franz, 2007; Kribernegg, 2013; Desai & Srivastava, 2016; Spillner & Linz, 2012; Hoffmann, 2013).

Ein Use-Case Diagramm kann wie in Abbildung 3.9 dargestellt werden. Das System wird dabei durch ein Rechteck repräsentiert. Ein Use-Case wird stets von einem *Akteur*, dargestellt als *Person*, angestoßen. Die Beziehungen zwischen den Use Cases und den Akteuren werden durch simple Linien dargestellt. Die Zusammenhänge zwischen den Use Cases können als *include* oder *extend* Beziehungen skizziert werden. Während erstere immer zum Tragen kommen, trifft dies auf zweitere nicht zu. Wie zu sehen ist, hängt es von der sogenannten *Condition* ab, ob diese zum Einsatz kommt (Franz, 2007; Kribernegg, 2013; O'Regan, 2019; Spillner & Linz, 2012; Hoffmann, 2013).

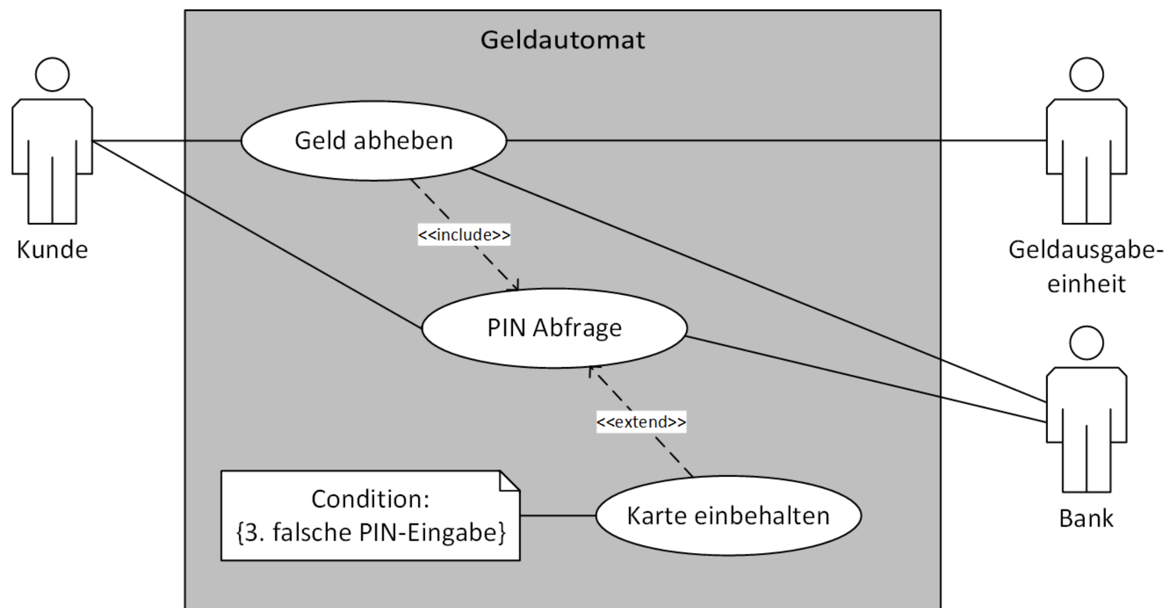


Abbildung 3.9.: Use-Case Diagramm (vgl. Spillner und Linz, 2012).

Für jedes Anwendungsszenario müssen je nach Rahmenbedingungen verschiedene Vor- bzw. Nachbedingungen eingehalten werden. Dies kann bedeuten, dass z.B. nach erfolgreicher PIN-Eingabe weitere Schritte wie die Verfügbarkeit der Geldmenge geprüft werden müssen (Kribernegg, 2013; O'Regan, 2019; Desai & Srivastava, 2016; Spillner & Linz, 2012).

Für die Konkretisierung der Tests ist es wichtig, folgende Informationen in Bezug auf den Use-Case bewusst einzuholen (Franz, 2007; O'Regan, 2019; Kribernegg, 2013; Desai & Srivastava, 2016; Spillner & Linz, 2012):

- Ausgangssituation und Vorbedingung
- Mögliche Rahmenbedingungen
- Vorausgesagte Ergebnisse und Resultate
- Nachbedingungen

Die Ermittlung der Eingabewerte, die eine bestimmte Ablaufsequenz innerhalb des Use-Cases auslösen, ist der nächste logische Schritt. Wichtig hierbei ist, dass alle möglichen Sequenzen betrachtet werden. Im Anschluss können diese Werte partitioniert werden. Dies ist z.B. mittels Äquivalenzklassen, siehe dazu Abschnitt 3.3.1.1, möglich. Im Anschluss an diese Tätigkeit werden konkrete Werte aus den Partitionierungen gewählt. Um auch die Grenzen der Parameter zu testen, empfiehlt es sich, die Grenzwertanalyse, siehe dazu Abschnitt 3.3.1.2, durchzuführen (Hoffmann, 2013).

Desai und Srivastava (2016) gehen hier einen etwas anderen Weg. Da sie kein Use-Case Diagramm verwenden, ist hier die Use-Case Spezifikation von zentraler Bedeutung. In dieser ist all das, was im Use-Case Diagramm abgebildet ist, textuell

enthalten. Zusätlich sind alle Informationen wie z.B. Ausgangssituation und Vorbedingungen notiert. Somit ist es einfach möglich, aus dieser Use-Case Spezifikation die Testfälle abzuleiten.

3.3.1.5. Der entscheidungstabellenbasierte Test

Der *entscheidungstabellenbasierte Test* kann wie der Use-Case Test auf Systemebene verwendet werden. Alle anderen Testebenen sind prinzipiell auch möglich. Die sogenannte *Entscheidungstabelle* mit den Testfällen lässt sich auf einfache Weise vom *Ursache-Wirkungs-Graphen* ableiten. Der Graph stellt die Zusammenhänge zwischen den *Ursachen*, auch *Bedingungen* genannt, und den *Wirkungen*, auch *Aktionen* genannt, plakativ grafisch dar. Im Gegensatz zu anderen Testverfahren betrachtet dieses die Abhängigkeiten der Eingabeparameter, konkret der Ursachen bzw. Bedingungen, zueinander (Kribernegg, 2013; Frühauf et al., 2007; Liggesmeyer, 2009; Spillner & Linz, 2012; Hoffmann, 2013).

In Abbildung 3.10 ist ein Ursache-Wirkungs-Graph dargestellt. Die Erstellung erfolgt, indem die Ursachen links und die Wirkungen rechts aufgelistet werden. Die Verbindungen zwischen ihnen erfolgen durch Pfeile, die eine Ursache mit einer oder mehreren Wirkungen verbinden *(Identität)* sowie mit den logischen Operatoren *und (Konjunktion)*, *oder (Disjunktion)* und *nicht (Negation)*, welche die Abhängigkeiten der Wirkungen von Ursachen weiter aufschlüsseln (Kribernegg, 2013; F. Wolf, 2018; Liggesmeyer, 2009; Spillner & Linz, 2012; Hoffmann, 2013).

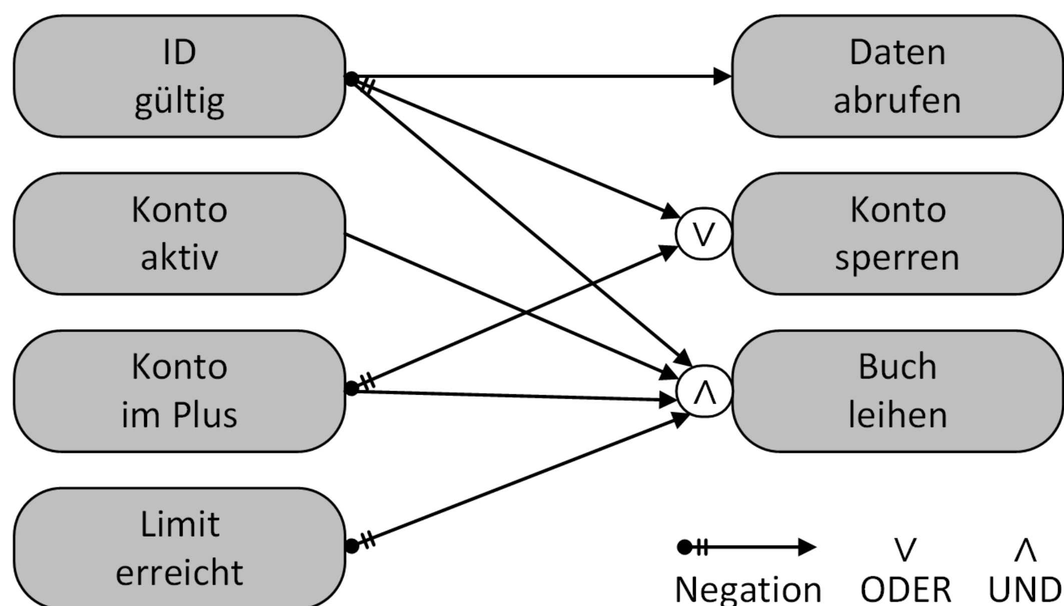


Abbildung 3.10.: Ursache-Wirkungs-Graph des Beispiels Buchausleihe (vgl. Hoffmann, 2013 bzw. F. Wolf, 2018).

Somit ergibt sich im Beispiel der Buchausleihe, wie in Abbildung 3.10 dargestellt, dass ein Buch nur dann ausgeliehen werden kann, wenn die ersten drei Ursachen positiv bestätigt werden und die letzte negiert wird. Weitere Wirkungen sowie Abhängigkeiten sind in der entsprechenden Abbildung dargestellt (F. Wolf, 2018; Hoffmann, 2013).

Im nächsten Schritt wird eine Entscheidungstabelle erstellt. Da die Ursachen n mal vorkommen und entweder mit dem Wert *ja* oder *nein* versehen sein können, ergeben sich 2^n mögliche Kombinationen. Diese werden durch die Anzahl der Spalten in der Tabelle abgebildet. Für das konkrete Beispiel ist dies in Tabelle 3.4 dargestellt. Jede Spalte stellt hierbei einen Testfall dar, wobei als Eingabeparameter des Testfalls die Werte aus den Bedingungen entnommen werden (Kribernegg, 2013; Hoffmann, 2013).

Bedingungen																
N	N	N	N	N	N	N	N	J	J	J	J	J	J	J	J	ID gültig
N	N	N	N	J	J	J	J	N	N	N	N	J	J	J	J	Konto aktiv
N	N	J	J	N	N	J	J	N	N	J	J	N	N	J	J	Konto im Plus
N	J	N	J	N	J	N	J	N	J	N	J	N	J	N	J	Limit erreicht

Aktionen																
N	N	N	N	N	N	N	N	J	J	J	J	J	J	J	J	Daten abrufen
J	J	J	J	J	J	J	J	J	J	N	N	J	J	N	N	Konto sperren
N	N	N	N	N	N	N	N	N	N	N	N	N	N	J	N	Buch leihen

Tabelle 3.4.: Auflistung der vollständigen Entscheidungstabelle zum Beispiel der Buchausleihe (vgl. Hoffmann, 2013).

In der Praxis wird auf eine vollständige Entscheidungstabelle meist verzichtet. Dies hat mannigfaltige Gründe. Auf der einen Seite ist es einfach nicht möglich, bei (sehr) vielen Ursachen eine vollständige Entscheidungstabelle zu generieren, da die Möglichkeiten exponentiell mit der Anzahl der Ursachen wachsen. Auf der anderen Seite beeinflussen sich Bedingungen gegenseitig und die sinnvolle Anzahl an Tests kann damit stark eingeschränkt werden, ohne die Testabdeckung merklich zu verschlechtern. Weiters sind in gewissen Szenarien die Werte der Bedingungen für die Aktionen ohne Bedeutung. Dies wird in der Entscheidungstabelle mit dem Symbol - dargestellt (Kribernegg, 2013; Frühauf et al., 2007; Spillner & Linz, 2012; Hoffmann, 2013).

Im Beispiel der Buchausleihe bedeutet dies, dass ohne einen gültigen Ausweis die Aktion des Ausleihens nicht möglich bzw. die Aktion des Konto Sperrens im-

mer möglich ist, egal welche Parameter die anderen Werte haben. Dies ist in Tabelle 3.5 dargestellt. Durch diese Aggregation werden einige Testfälle eingespart (F. Wolf, 2018; Hoffmann, 2013).

Bedingungen						
N	J	J	J	J	J	ID gültig
-	N	N	J	J	J	Konto aktiv
-	N	J	N	J	J	Konto im Plus
-	-	-	-	N	J	Limit erreicht

Aktionen						
N	J	J	J	J	J	Daten abrufen
J	J	N	J	N	N	Konto sperren
N	N	N	N	J	N	Buch leihen

Tabelle 3.5.: Auflistung der reduzierten Entscheidungstabelle, zum Beispiel der Buchausleihe (vgl. Hoffmann, 2013 bzw. F. Wolf, 2018).

Spillner und Linz (2012) gehen bei der Erstellung der reduzierten Entscheidungstabelle einen anderen Weg. Die vollständige Tabelle mit den 2^n Kombinationen entfällt komplett. Ausgehend vom Ursache-Wirkungs-Graphen und mit Hilfe des folgenden Ablaufs wird daher direkt die reduzierte bzw. optimierte Entscheidungstabelle erstellt:

1. Auswahl einer Wirkung.
2. Anhand dieser Wahl wird der Graphen nach Kombinationen von Ursachen bzw. Bedingungen durchsucht.
3. Dadurch erfolgt die Erstellung einer Spalte in der Entscheidungstabelle, in der alle Ursachen bzw. Bedingungen und die Kombinationen derer sowie die entsprechenden Aktionen eingetragen werden.
4. Sollten Einträge mehrfach auftreten, werden diese entfernt.

Nach Erstellung der reduzierten Entscheidungstabelle ergeben sich direkt die notwendigen Tests, ohne die Testabdeckung merklich zu verschlechtern. Die Szenarien werden am System durch mindestens einen konkreten Test abgedeckt. Die Bedingungen dafür werden eingestellt, entsprechende Werte konkret gesetzt sowie bei Bedarf durch weitere Vor-, Nach- bzw. Randbedingungen ergänzt. Anschließend findet durch den Vergleich von Soll- und Istzustand in Bezug auf die Aktionen die Überprüfung statt (Spillner & Linz, 2012; Hoffmann, 2013).

Durch das gezielte und systematische Vorgehen beim Erstellen des Ursache-Wirkungs-Graphen sowie der Entscheidungstabelle können Lücken aufgedeckt werden, die andere Tests so nicht finden können. Allerdings ist das Erstellen sowie das Reduzieren der Tabelle auf die notwendigen Testfälle fehleranfällig, sodass ohne ausreichende Toolunterstützung das gesamte Konstrukt schnell unübersichtlich wird und nur mehr schwer handhabbar ist (Spillner & Linz, 2012).

3.3.1.6. Das paarweise Testen

Das *paarweise Testen* greift im Gegesatz zu anderen Verfahren wie z.B. den entscheidungstabellenbasierten Test oder dem Use-Case Test nicht auf die semantische Analyse des Systems, sondern rein auf kombinatorische Möglichkeiten zurück. Das Ziel des Verfahrens ist es, die oft astronomischen Kombinationen von Testfällen auf ein Mindestmaß zu reduzieren (Kuhn, Kacker & Lei, 2016; Lee, 2004; Kribernegg, 2013; Hoffmann, 2013).

Damit paarweise getestet werden kann, sind zwei Parameter von großer Wichtigkeit: Zum einen die empirische Beobachtung, dass meist (ein bis) zwei Faktoren in Kombination zueinander für Fehler des Systems verantwortlich sind. Nur in wenigen Fällen sind es drei odere mehrere Faktoren in einer bestimmten Kombination, die nicht dem Ist-Verhalten des Systems entsprechen. Zum anderen das angewandte Pareto Prinzip. Dieses besagt, dass einige wenige Ursachen etwa 80% der Fehler erzeugen (Kribernegg, 2013; Hoffmann, 2013).

Zwei exemplarische Beispiele verdeutlichen den Bedarf, die Testfälle zu senken. Wenn ein Safe mit acht Einstellmöglichkeiten, die je vier Optionen anbieten, alle Testfälle abdecken soll, ergibt das entsprechend der Kombinatorik 4^8, also 65.536 Testfälle. Im nächsten Beispiel soll eine Website ausgeliefert und funktional korrekt dargestellt werden. Die möglichen Komponenten, die miteinander interagieren und getestet werden sollen, sind:

- 8 Browser: Verschiedene Browser und Versionen
- 3 Plug-in Möglichkeiten: 2 Plug-ins, einmal keines
- 6 Betriebssysteme: Verschiedene Betriebssysteme und Versionen
- 3 Programme: Anwendungen am Server, die die Website ausliefern
- 3 Server Betriebssysteme: Verschiedene Betriebssysteme und Versionen

Dadurch ergeben sich aus der Multiplikation der Möglichkeiten 1.296 mögliche Testfälle (Kuhn et al., 2016; Lee, 2004; Kribernegg, 2013; Hoffmann, 2013).

Wie erwähnt, verfolgt die Methode des paarweisen Testens das Ziel, die Anzahl der Kombinationen zu reduzieren. Die Idee dabei ist, dass jede paarweise Kombination, ein sogenanntes *Ausprägungspaar*, einmal aber nicht zwingend öfters vorkommen

muss. Die Testmenge, die dieses Kriterium erfüllt, nennt man *paarweise vollständig*. Wie in den Beispielen ersichtlich, ist es kaum möglich, die Erstellung händisch durchzuführen. Eine Möglichkeit der algorithmischen Erstellung von paarweise vollständigen Testsets basiert auf der Verwendung von *orthongonalen Feldern*, auch *orthogonal arrays* genannt (Lee, 2004; Hoffmann, 2013).

Das Grundprinzip des paarweisen Testens baut im Wesentlichen auf den Überlegungen des Mathematikers Leonhard Euler auf. Spezielle Eigenschaften der orthogonalen Felder ermöglichen es so, nach Generierung des paarweise vollständigen Sets, die Tests durchzuführen. Sollten Tests fehlschlagen, können die Ausprägungspaare verglichen und somit der Ursprung des Fehlers lokalisiert werden (Lee, 2004; Kribernegg, 2013; Hoffmann, 2013).

Durch die Vorbedingungen, die das Verfahren mit sich bringt, gibt es einige Einschränkungen. So ist es prinzipiell nicht möglich alle Fehler zu finden. Dies wird auch nicht als Ziel angeführt. Weiters sind die Paare ein Schlüssel im Design, können jedoch um weitere *n* Elemente erweitert werden, um eine höhere Abdeckung zu erreichen. Der Nutzen von n-fachen Kombinationen mit $n > 2$ ist wissenschaftlich nicht erwiesen. Das Ziel einer hohen Testabdeckung wird in der Praxis durch den Einsatz von Tools zur Generierung der paarweisen vollständigen Testmengen nach dem Pareto Prinzip durchaus erreicht (Kuhn et al., 2016; Lee, 2004; Kribernegg, 2013; Hoffmann, 2013).

3.3.1.7. Diversifizierende Verfahren

Die bisherigen erwähnten Black-Box Testverfahren vergleichen die Sollergebnisse der Anforderungen mit den Istergebnissen des Systems, indem sie das System mit entsprechendem Input befüttern und ausführen. Bei den diversifizierenden Verfahren geht es im Kern darum, verschiedene Programmversionen miteinander zu vergleichen. Ein Vorteil der diversifizierenden Verfahren ist, dass sie sich automatisieren lassen. Dies wird empfohlen, da sich das Testen ansonsten schnell als aufwändig und/oder unübersichtlich herausstellt (Liggesmeyer, 2009; Hoffmann, 2013).

Laut Hoffmann (2013) existieren folgende diversifizierende Verfahren:

- **Back-to-Back-Test**
  Das Ziel des *Back-to-Back-Tests* ist es, verschiedene Implementationen, die auf derselben Spezifikation aufbauen, zu testen, um anschließend die Ergebnisse der beiden Varianten miteinander zu vergleichen. Die wesentliche Voraussetzung dafür ist, dass die Teams unabhängig voneinander die Spezifikationen umsetzten. Die Maximierung der Heterogenität ist hierbei das Ziel, um gemeinsame Fehler zu vermeiden (Tomohiko, Takeshi & Zengo, 2013; Liggesmeyer, 2009; Hoffmann, 2013).

Aufgrund des hohen Aufwands werden diese Tests nur dann durchgeführt, wenn Systeme oder Teilsysteme extrem hohen Sicherheitsanforderungen unterliegen oder mit konventionellen Mitteln nur schwer testbar sind (Tomohiko et al., 2013; Liggesmeyer, 2009; Hoffmann, 2013).

- **Regressionstest**
  Der *Regressionstest* zielt darauf ab, nach Weiterentwicklungen oder Fehlerkorrekturen der Software diese Version mit der vorherigen zu vergleichen. Dazu werden Testfälle für die neue Funktionalität bzw. die Behebung der Fehler erstellt. Diese werden dann auf beide Varianten angewandt und sollen zeigen, dass Fehler, die sich im Rahmen der Weiterentwicklung einschleichen können, schnell erkannt werden und die behobenen Korrekturen tatsächlich greifen (Dustin, Rashka & Paul, 2001; Nörenberg, 2012; Hoffmann, 2013).

  Da für die Erstellung der Testszenarien sowie das Ausführen der Tests kaum Mehraufwand zu erwarten ist und damit die Softwarekorrektheit der aktuellen Version und der vorherigen gezeigt werden kann, erfreuen sich diese Tests einer gewissen Beliebtheit. Durch Testautomatisierung entsteht ein leistungsfähiges und kostenökonomisches Instrument zur Sicherstellung der Software-Integrität (Dustin et al., 2001; Nörenberg, 2012; Hoffmann, 2013).

- **Mutationstest**
  Der *Mutationstest* ist kein Testverfahren im eigentlichen Sinne. Vielmehr ist es eine Technik, die zur Bewertung von Testverfahren dient. Durch gezielten Einbau von Fehlern, z.B. *Konstantenfehlern*, *Variablenfehlern*, *Arithmetikfehlern* und *Logikfehlern*, wird eine sogenannte *Mutations-* bzw. *Fehlertransformation* durchgeführt (Basu, 2015; Ammann & Offutt, 2008; Hoffmann, 2013).

  Ein Anwendungsfall des Mutationstests ist es, die Leistungsfähigkeit einer Testsuite zu messen, auch *vergleichender Mutationstest* genannt. Dies geschieht, indem die Anzahl der identifizierten Mutationen durch die gesamte Anzahl der erzeugten Mutanten dividiert wird (Basu, 2015; Ammann & Offutt, 2008; Hoffmann, 2013).

  Ein weiterer Anwendungsfall ist die *prädizierende Variante*. Diese ermöglicht es, die Gesamtzahl der in einem Softwaresystem vorhandenen Fehler grob abzuschätzen. Dazu werden die gefundenen Fehler während eines Testlaufs mit der *Testfallmenge T*, die erzeugten und die identifizierten Mutanten als Eingabegrößen benötigt. Die Anzahl der Fehler errechnet sich wie folgt:

  $$Anzahl\ Fehler \approx Gefundene\ Fehler \cdot \frac{Erzeugte\ Mutanten}{Identifizierte\ Mutanten} \tag{3.4}$$

  So verlockend die Ergebnisse der Formel 3.4 sein mögen, so eingeschränkt müssen diese im jeweiligen Kontext interpretiert werden. Erst bei genügend großen

Test- und Mutationsmengen können statistisch relevante Informationen gewonnen werden (Hoffmann, 2013).

3.3.2. White-Box Testverfahren

White-Box Testverfahren, auch *Strukturtests*, *codebasierte Testverfahren* bzw. *glass-box Testverfahren* genannt, setzten im Gegensatz zu *Black-Box Testverfahren* ihren Fokus auf die Analyse der inneren Programmstrukturen des zu untersuchenden Softwaresystems. Die White-Box Testverfahren sollen grundsätzlich Folgendes sicherstellen:

- Alle Codefragmente sollen mindestens einmal zur Ausführung kommen.
- Alle logischen Entscheidungen sollen auf ihr Verhalten überprüft werden.
- Alle Schleifen und Grenzen sollen überprüft werden.
- Interne Datenstrukturen sollen auf deren Gültigkeit überprüft werden.

Dies setzt voraus, dass der entsprechende Programmcode offen, veränderbar und unter Umständen auch manipulierbar vorliegen muss (Agarwal & Tayal, 2009; Witte, 2016; Franz, 2007; Liggesmeyer, 2009; Spillner & Linz, 2012; Hoffmann, 2013).

Prinzipiell lassen sich White-Box Testverfahren in zwei Kategorien einteilen: Die *kontrollflussorientierten Tests* sowie die *datenflussorientierten Tests*. Erstere konstruieren ihre Testfälle auschließlich aus den internen Berechnungspfaden des Softwaresystems während zweitere sich zusätzlich mit der Beschaffenheit der Daten beschäftigen (Spillner, Winter & Pietschker, 2017; Bath & McKay, 2015; Hoffmann, 2013).

Zu den typischen Vertretern der kontrollflussorientierten Tests gehören die *Anweisungsüberdeckung*, die *Zweigüberdeckung*, die *Pfadüberdeckung*, die *Bedingungsüberdeckung* und die *McCabe-Überdeckung*. Vertreter der datenflussorientierten Tests sind die *Defs-Uses Überdeckung* sowie die *Required-k-Tupel-Überdeckung*. Letztere ist im direkten Vergleich mit der Defs-Uses Überdeckung ähnlich und wird im Folgenden nicht näher betrachtet (Bath & McKay, 2015; Hoffmann, 2013).

3.3.2.1. Der Anweisungs- oder C_0 - Test

Der *Anweisungstest*, *statement test* oder auch *C_0 - Test* genannt, gehört zu den *kontrollflussorientierten* White-Box Tests. Dieses Verfahren ist das schwächste der vorgestellten kontrollflussorientierten Verfahren und somit in folgenden Tests bzw. Überdeckungen meist als Teilmenge enthalten (Kleuker, 2019; Kribernegg, 2013; Spillner & Linz, 2012; Hoffmann, 2013).

Die Idee des Testverfahrens besteht darin, alle Knoten des *Kontrollflussgraphen*, wie anhand eines Beispiels in Abbildung 3.11 exemplarisch dargestellt, zu durchlaufen, damit alle Anweisungen des untersuchten (Teil)Programms mindestens einmal ausgeführt werden. Dies wird mit der sogenannten *Anweisungsüberdeckung*, ausgeführt in Gleichung 3.5, messbar gemacht (Witte, 2018; Kleuker, 2019; Kribernegg, 2013; Spillner & Linz, 2012; Hoffmann, 2013).

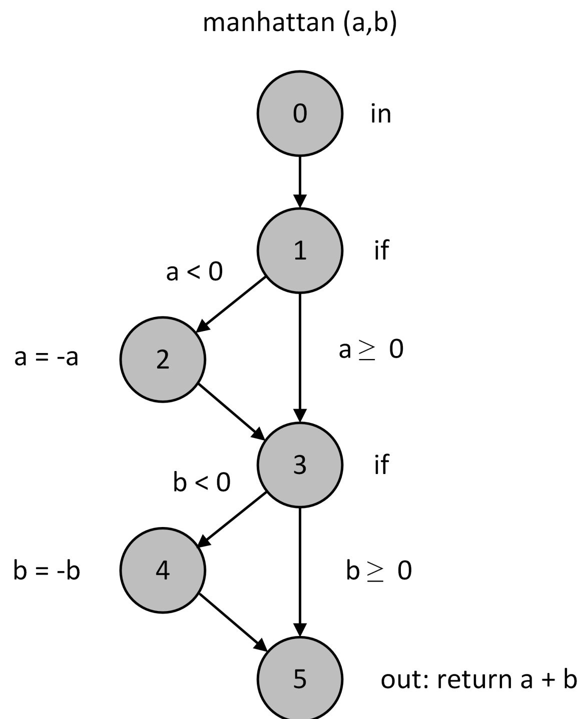


Abbildung 3.11.: Kontrollflussgraph der Funktion *manhattan.c* (vgl. Hoffmann, 2013).

$$Anweisungs\ddot{u}berdeckung = \frac{Anzahl\ der\ durchlaufenden\ Anweisungen}{Gesamtzahl\ der\ Anweisungen} \cdot 100\% \quad (3.5)$$

Um das grundsätzliche Ziel von 100 % Anweisungsüberdeckung zu erreichen, wird eine entsprechende Testmenge so gewählt, dass dies möglich ist. Die Existenz von sogenanntem *totem Code* soll aufgezeigt werden. Anhand eines Beispiels werden im Folgenden Eigenschaften des Verfahrens erörtert (Witte, 2018; Kleuker, 2019; Kribernegg, 2013; Spillner & Linz, 2012; Hoffmann, 2013).

Ein Programmstück soll die *Manhattan-Distanz* wie folgt berechnen:

$$d((x_1, y_1), (x_2, y_2) = |x_1 - x_2| + |y_1 - y_2| \quad (3.6)$$

Dies soll in C mittels der Funktion *manhattan.c*, wie in Listing 3.1, umgesetzt werden. Wie auch in Gleichung 3.6 klar ersichtlich ist, müssen die Beträge entsprechend

dem Vorzeichen geprüft werden. Außerdem wurde bereits erwähnt, dass es für die Anweisungsüberdeckung volkommen ausreicht, wenn alle Knoten des Kontrollflussgraphen anhand einer entsprechenden Testmenge durchlaufen werden (Hoffmann, 2013).

Listing 3.1: manhattan.c: Berechnung der Manhattan-Distanz in der Entwicklungssprache C (vgl. Hoffmann, 2013)

```
1 // Eingabe: a:int, b:int
2 // Ergebnis: |a| + |b|
3
4 int manhattan(int a, int b)
5 {
6         if (a < 0)
7                 a = -a;
8         if (b < 0)
9                 b= -b;
10        return a+b;
11 }
```

Im Beispiel der Manhattan-Distanz bedeutet dies konkret, dass der Aufruf *manhattan (-1, -1)* alle Fälle des Kontrollflussgraphen abdeckt. Positive Werte oder die Kombinationen von positiven und negativen Werten müssen somit nicht überprüft werden. Dies deutet auf eine Schwäche des Verfahrens hin. Anders formuliert bedeutet dies, dass das Verfahren die vorhandenen Strukturen in Bezug auf die Pfade überprüft, den eigentlichen Programmzustand jedoch nicht betrachtet. Dies ist ein typisches Verhalten von White-Box Tests (Kleuker, 2019; Hoffmann, 2013).

In der Praxis erweist sich diese Unzulänglichkeit unverhofft als Vorteil. Oft existieren im Code Pfade, die nicht einfach durch ein entsprechendes Set an Inputparametern ausführbar sind, sondern aufwendig mit eigens dafür erstellten Funktionen simuliert werden müssten. Weiters kann, wie bereits erwähnt toter Code vorhanden sein. Diese Sachverhalte werden mit dem Verfahren aufgedeckt. Daher ist der Anweisungs- bzw. C_0 - Test in vielen Bereichen ein Standardtest (Witte, 2018; Kribernegg, 2013; Spillner & Linz, 2012; Hoffmann, 2013).

3.3.2.2. Der Zweig- bzw. C_1 - Test

Der *Zweigtest*, *branch coverage test*, oder auch *C_1 - Test* genannt, gehört wie der Anweisungs- oder C_0 - Test, zu den *kontrollflussorientierten* White-Box Tests. Im Gegensatz zum Anweisungs- bzw. C_0 - Test ist dieser mächtiger und deckt ihn komplett ab (Khannur, 2014; Daigl & Glunz, 2016; Kribernegg, 2013; Spillner & Linz, 2012; Hoffmann, 2013).

Der Unterschied zum Anweisungs- oder C_0 - Test ist, dass hier jeder Zweig und nicht jede Anweisung getestet werden muss. Auf den Kontrollflussgraphen bezogen bedeutet dies, dass jede Kante anstatt jedes Knotens mindestens einmal erreicht werden muss. Für die Manhattan-Distanz, siehe dazu Abschnitt 3.3.2.1, bedeutet dies, dass die beiden Eingabemöglichkeiten *manhattan(-1, 1)* sowie *manhattan(1, -1)*, vergleiche dazu Abbildung 3.12, alle Zweige abdecken. Dies ist anhand des Listings 3.1 bzw. des Kontrollflussgraphens in Abbildung 3.11 ableitbar (Khannur, 2014; Daigl & Glunz, 2016; Kribernegg, 2013; Spillner & Linz, 2012; Hoffmann, 2013).

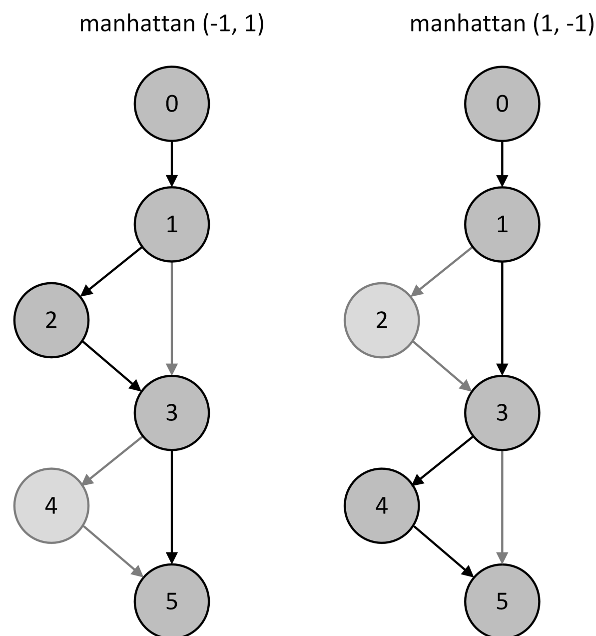


Abbildung 3.12.: Kontrollflussgraphen der Funktion *manhattan.c* in Bezg auf die Zweigüberdeckung (vgl. Hoffmann, 2013).

Wenig überraschend ist, dass sich die Zweigüberdeckung, wie in Gleichung 3.7, ähnlich bestimmen lässt wie die Anweisungsüberdeckung (Khannur, 2014; Daigl & Glunz, 2016; Kribernegg, 2013; Spillner & Linz, 2012; Hoffmann, 2013).

$$Zweigüberdeckung = \frac{Anzahl\ der\ durchlaufenden\ Zweige}{Gesamtzahl\ der\ Zweige} \cdot 100\% \tag{3.7}$$

Wie beim Anweisungs- bzw. C_0 - Test soll auch beim Zweig- bzw. C_1 - Test eine Abdeckung von 100% erreicht werden. Im Gegensatz zu ersterem zeigt der Zweig- bzw. C_1 - Test fehlende Anweisungen auf und kann so unterstützen, den Code zu verbessern. Daher ist der Zweig- bzw. C_1 - Test auch ein Standarttest in vielen Bereichen (Khannur, 2014; Kribernegg, 2013; Spillner & Linz, 2012; Hoffmann, 2013).

3.3.2.3. Der Pfad- bzw. C_2 - Test

Der *Pfadtest*, *path coverage test* bzw. C_2 *- Test* ist die mit Abstand mächtigste Technik der kontrollflussorientierten White-Box Verfahren. Dieser Test deckt, wie der Name C_2 vermuten lässt, die beiden Vorgänger, Anweisungs- oder C_0 - Test bzw. Zweig- bzw. C_1 - Test, volkommen ab (Irrgang, 1995; Kribernegg, 2013; Spillner & Linz, 2012; Hoffmann, 2013).

Die Idee des Pfad- bzw. C_2 - Tests ist es, jeden möglichen Pfad des Kontrollflussgraphen, ausgehend vom Ein- bzw. Ausgangsknoten, zu verbinden und einen separaten Testfall dafür zu erzeugen. Ein Beispiel dafür in Bezug auf die Manhattan-Distanz ist in Abbildung 3.13 dargestellt. Wie in der Grafik abgebildet, sind vier Testfälle nötig, um die Intention des Testverfahrens zu erfüllen. Dies ist durch das Listing 3.1 bzw. den Kontrollflussgraphen in Abbildung 3.11 herleitbar (Irrgang, 1995; Kribernegg, 2013; Spillner & Linz, 2012; Hoffmann, 2013).

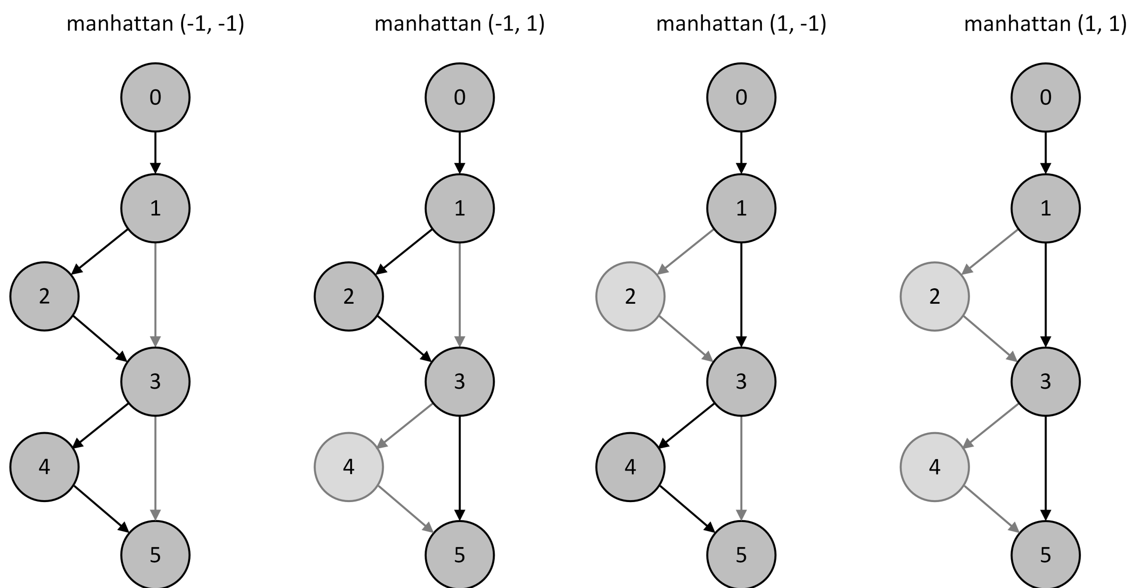


Abbildung 3.13.: Kontrollflussgraphen der Funktion *manhattan.c* in Bezg auf die Pfadüberdeckung (vgl. Hoffmann, 2013).

Die Pfadüberdeckung lässt sich ähnlich der Anweisungs- und Zweigüberdeckung ermitteln. Diese ist in Gleichung 3.8 dargelegt (Irrgang, 1995; Kribernegg, 2013; Spillner & Linz, 2012; Hoffmann, 2013).

$$Pfadüberdeckung = \frac{Anzahl\ der\ durchlaufenden\ Pfade}{Gesamtzahl\ der\ Pfade} \cdot 100\% \tag{3.8}$$

Die zu berücksichtigenden Testfälle wachsen, entsprechend den mathematischen Gesetzen der Kombinatorik, meist explosionsartig an. Bei einer Datenstruktur mit 512

Elementen, von denen jedes Element auf eine Bedingung überprüft werden soll, ergibt dies demnach 2^{512} Kombinationen. Das ist in der Praxis nicht machbar. Daher gibt es verfeinerte Varianten des Grundkonzepts, um die Anzahl der Fälle einzuschränken. Da dies für die Praxis den Aufwand oft nicht genug senkt, kommt das Verfahren meist nicht zum Einsatz. Für besonders sicherheitskritische Bereiche ist eine Pfadüberdeckung von 100% dennoch vonnöten (Irrgang, 1995; Kribernegg, 2013; Spillner & Linz, 2012; Hoffmann, 2013).

3.3.2.4. Der Bedingungstest

Unter den bisher genannten Verfahren ist der Zweig- bzw. C_1 - Test in den meisten Fällen der Kompromiss schlechthin. Dieser deckt mehr ab als der Anweisungs- bzw. C_0 - Test, ist aber in den meisten Fällen (mit vertretbarem Aufwand) umsetzbar, im Gegensatz zum Pfad- bzw. C_2 - Test.
Der Zweig- bzw. C_1 - Test unterscheidet jedoch nicht bezüglich der Codestruktur. Listing 3.2 und Listing 3.3 zeigen dies auf. Während der Zweig- bzw. C_1 - Test bei ersterem Codebeispiel für eine Zweigüberdeckung von 100% zwei Testfälle benötigt, sind es bei zweiterem bereits vier. Dies löst der *Bedingungstest*, auch *condition test* genannt, auf (Hass, 2008; Kleuker, 2019; Frühauf et al., 2007; Daigl & Glunz, 2016; Kribernegg, 2013; Spillner & Linz, 2012; Hoffmann, 2013).

Listing 3.2: Codefragment; Implementation der Logik nach Variante 1 (vgl. Hoffmann, 2013)

```
1 if ((A && !B) || (!A && B↩
    ))
2 {
3         foo();
4 }
```

Listing 3.3: Codefragment; Implementation der Logik nach Variante 2 (vgl. Hoffmann, 2013)

```
1 if (A)
2         if (!B)
3                 goto foo↩
                     ():
4 if (!A)
5         if (B)
6                 goto foo:
7 return;
8 foo:foo();
```

Um dieses Verhalten besser steuern zu können, nimmt der Bedingungstest zusätzlich zum Kontrollflussgraphen einen weiteren Parameter in die Testkonstruktion mit auf: Die Analyse der logischen Struktur von *Abfragen*, z.B. *if - Abfragen*. Dadurch ergeben sich drei Szenarien (Kleuker, 2019; Frühauf et al., 2007; Daigl & Glunz, 2016; Kribernegg, 2013; Spillner & Linz, 2012; Hoffmann, 2013):

- **Die einfache Bedingungsüberdeckung**
  Alle sogenannten *atomaren Prädikate*, im Beispiel des Listings 3.2 sind das die

Abfragen `(A && !B)` und `(!A && B)`, müssen mindestens einmal beide Wahrheitswerte annehmen. Somit ergeben sich zwei Testfälle. Die einfache Bedingungsüberdeckung ergibt sich demnach wie in Gleichung 3.9 abgebildet:

$$\frac{Anzahl\ der\ nach\ wahr\ und\ falsch\ ausgewerteten\ atomaren\ Prädikate}{2\ *\ Gesamtzahl\ der\ atomaren\ Prädikate} \cdot 100\% \tag{3.9}$$

- **Die minimale Mehrfachbedingungsüberdeckung**
  Alle sogenannten *atomaren und zusammengesetzten Prädikate* auch *Teilbedingungen* genannt, im Listings 3.2 sind das die Abfragen `(A && !B)`, `(!A && B)` und `((A && !B) || (!A && B))`, müssen mindestens einmal beide Wahrheitswerte annehmen. Daraus ergeben sich drei Testfälle. Die minimale Mehrfachbedingungsüberdeckung ergibt sich demnach wie in Gleichung 3.10 abgebildet:

$$\frac{Anzahl\ der\ nach\ wahr\ und\ falsch\ ausgewerteten\ Teilbedingungen}{2\ *\ Gesamtzahl\ der\ Teilbedingungen} \cdot 100\% \tag{3.10}$$

- **Die Mehrfachbedingungsüberdeckung**
  Alle sogenannten Möglichkeiten der Werte, im Beispiel des Listings 3.2 sind das die Abfragen `(A && !B)` und `(!A && B)`, müssen mindestens einmal beide Wahrheitswerte annehmen. Daraus ergeben sich bei zwei möglichen Wahrheitswerten sowie zwei Bedingungen vier Testfälle. Die Mehrfachbedingungsüberdeckung ergibt sich demnach wie in Gleichung 3.11 abgebildet:

$$\frac{Anzahl\ aller\ ausgewerteten\ Kombinationen\ der\ Teilbedingungen}{Gesamtzahl\ aller\ Kombinationen\ der\ Teilbedingungen} \cdot 100\% \tag{3.11}$$

3.3.2.5. Die McCabe Überdeckung bzw. the basic path test

Die *McCabe Überdeckung* bzw. der *basic path test* ist eine weitere kontrollflussorientierte White-Box Technik. Sie beruht auf den Arbeiten von Thomas J. McCabe, die in den 70er Jahren publiziert wurden (Ainapure, 2009; Jorgensen, 2014; Hoffmann, 2013).

Diese Methode nutzt für die Konstruktion der Testfälle die Erkenntnisse aus der Graphentheorie und der linearen Algebra. Zuerst wird der Kontrollflussgraph in einen zusammenhängenden Graphen verwandelt, indem die Knoten *in* und *out* über eine zusätzliche (Hilfs)Kante verbunden werden. Im Anschluss legt die Namenskonvention die Knoten als *nodes* bzw. *N*, sowie die Kanten als *edges* bzw. *E* fest. Die Anzahl der Elementarpfade, die nötig sind, werden mit der Variable $|B|$ abgebildet (Ainapure, 2009; Jorgensen, 2014; Hoffmann, 2013).

Somit ergibt sich durch die Erkenntnisse aus der Graphentheorie und der linearen Algebra ein Zusammenhang, der in Gleichung 3.12 dargestellt ist.

$$|B| = |E| - |N| + 2 \tag{3.12}$$

Mit diesem Zusammenhang ist es möglich, eine Anzahl von Elementarpfaden zu erhalten. Diese sind nötig, um alle weiteren Pfade aus diesen generieren zu können. Dies ist wie in Gleichung 3.13 dargelegt möglich.

$$\mathbf{p} = k_1 \cdot \mathbf{p}_1 + k_2 \cdot \mathbf{p}_2 + \ldots + k_\mathrm{n} \cdot \mathbf{p}_\mathrm{n} \tag{3.13}$$

Dabei stellt $\mathbf{p}$ den Vektor dar, der sich aus den linear unabhängigen Elementarpfaden $\mathbf{p_1} \ldots \mathbf{p_n}$ unter Zuhilfenahme der Konstanten $\mathrm{k}_1 \ldots \mathrm{k}_\mathrm{n}$ ergibt (Ainapure, 2009; Jorgensen, 2014; Hoffmann, 2013).

Die Erstellung der Tests erfolgt, indem ein Kontrollflussgraph und die daraus folgenden Elementarpfade generiert werden. Die Anzahl dieser ist, wie in Gleichung 3.12 dargestellt fix, die Wahl der Elementarpfade ist jedoch nicht eindeutig und kann daher variieren. Daher können auch die Testsets variieren. Jede McCabe Testmenge deckt den Anweisungs- bzw. C_0 - Test sowie den Zweig- bzw. C_1 - Test vollständig ab (Ainapure, 2009; Jorgensen, 2014; Hoffmann, 2013).

3.3.2.6. Die Defs-Uses Überdeckung

Die bisher vorgestellten kontrollflussorientierten White-Box Testverfahren setzten sich ausschließlich mit den internen Berechnungspfaden des jeweiligen Codes auseinander. Die Defs-Uses Kriterien, datenflussorientierte Testverfahren, betrachten nun die Beschaffenheit der Daten. Dies soll Variablenprobleme aufdecken (Bommer et al., 2008; Liggesmeyer, 2009; Roitzsch, 2005; Kribernegg, 2013; Hoffmann, 2013).

Betrachtet werden dabei die Pfade der Variablen, angefangen bei ihrer Definition bis hin zu deren Nutzung. Dabei gibt es folgende Kategorisierung (Bommer et al., 2008, 2008; Liggesmeyer, 2009; Roitzsch, 2005; Hoffmann, 2013):

- **Definitorische Nutzung, auch def-use genannt:** Diese Nutzung findet dann statt, wenn im Ablauf des Programms *Zuweisungen* vorkommen. Typischerweise geschieht dies durch Funktionsaufrufe mit Parametern oder bei der Zuweisung von konkreten Werten durch Definitionen oder Berechnungen an eine Variable.
- **Referenzierende Nutzung, auch r-use genannt:** Wenn eine Variable nicht verändert, aber genutzt wird, findet eine *Benutzung*, ein *r-use*, statt. Zwei Möglichkeiten lassen sich dabei unterscheiden:
  - **Berechnende Nutzung, auch c-use genannt:** Sind alle Nutzungen der Variablen, bei denen *Berechnungen*, auch *computational uses* genannt, stattfinden.
  - **Prädikative Nutzung, auch p-use genannt:** Bezeichnet alle Nutzungen der Variablen, bei denen *Vergleiche* in booleschen Ausdrücken stattfinden.

Ausgehend vom Code kann ein Kontrollflussgraph und eine Tabelle erstellt werden, um den Datenfluss übersichtlicher darzustellen. Im Kontrollflussgraphen werden dabei die Knoten mit den c-use, die Kanten mit den p-use Elementen beschriftet. In der Tabelle werden Definition und c-use den Knoten, p-use den Kanten zugeordnet (Hoffmann, 2013).

Ein wichtiger Begriff im Rahmen der Defs-Uses Überdeckung ist der sogenannte *definitionsfreie Pfad*. Dieser ist so definiert, dass die Variable x in einem Knoten s_0 vorerst festgelegt wird. Im Pfad $s_0, s_1, \ldots, s_\mathrm{n}$ wird diese Definition der Variable x an keiner weiteren Stelle $s_1, s_2, \ldots, s_\mathrm{n}$ erneut definiert (Hoffmann, 2013).

Anhand der genannten Faktoren ist es möglich, die Basis für die verschiedenen Defs-Uses Überdeckungen, meist *Kriterien* genannt, aufzuzählen (Liggesmeyer, 2009; Bommer et al., 2008; Roitzsch, 2005; Hoffmann, 2013):

- **All definitions (all defs) Kriterium:** Für jede Definition einer Variable durchlaufen die Testfälle einen definitionsfreien Pfad zu mindestens einem p- oder c-use. Dieses Kriterium ist vergleichsweise schwach und wird von vielen anderen mit abgedeckt.
- **All c-uses Kriterium:** Für jede Definition einer Variable durchlaufen die Testfälle einen definitionsfreien Pfad zu allen erreichbaren c-uses. Das Kriterium ist mächtiger als das all definitions Kriterium, kann dieses aber nicht vollständig abdecken.
- **All p-uses Kriterium:** Für jede Definition einer Variable durchlaufen die Testfälle einen definitionsfreien Pfad zu allen erreichbaren p-uses. Wie das c-uses Kriterium ist auch das p-uses Kriterium mächtiger als das all definitions Kriterium. Es kann dieses aber nicht vollständig abdecken.
- **All uses Kriterium:** Für jede Definition einer Variable durchlaufen die Testfälle einen definitionsfreien Pfad zu allen erreichbaren p- und c-uses.
- **All c, some p Kriterium:** Die Grundlage ist das all c-use Kriterium. Sollte zu einer Definition kein c-use, also keine berechnende Nutzung, existieren, wird zusätzlich mindestens ein definitionsfreier Pfad eines p-use, also eine prädikative Nutzung, hinzugenommen. Das Kriterium ist mächtiger als das all definitions Kriterium und kann dieses vollständig abdecken.
- **All p, some c Kriterium:** Die Grundlage ist das all p-use Kriterium. Sollte zu einer Definition kein p-use, also keine prädikative Nutzung, existieren, wird zusätzlich mindestens ein definitionsfreier Pfad eines c-use, also eine berechnende Nutzung, hinzugenommen. Das Kriterium ist mächtiger als das all definitions Kriterium und kann dieses vollständig abdecken.

In der Praxis ist eine algorithmische Berechnung der komplizierten Defs-Uses Überdeckungen Standard (Bommer et al., 2008).

3.3.3. Erfahrungsbasierte Testverfahren

Die *erfahrungsbasierten Testverfahren* zeichnen sich dadurch aus, dass es keine systematische und methodische Vorgehensweisen gibt. Für das Verfahren wird auf das Wissen, die Erfahrung und die Übernahme von Verantwortung in Bezug auf den Testprozess der Stakeholder, die das System testen sollen, zurückgegriffen. Ein weiteres Kennzeichen ist, dass diese Testverfahren schnell einsetzbar sind und in kurzer Zeit auch Ergebnisse liefern können (Bath & McKay, 2015; Kribernegg, 2013; Spillner & Linz, 2012).

Erfahrungsbasierte Tests lassen sich nicht exakt White- oder Black-Box Tests zuordnen. Sie können für beide Arten von Tests angewandt werden. Eine diesen Eigenschaften folgende Charakteristik ist, dass Testüberdeckungen nicht ermittelt werden können (Bath & McKay, 2015; Kribernegg, 2013; Spillner & Linz, 2012).

3.3.3.1. Intuitive Testfallermittlung bzw. Error Guessing

Die *intuitive Testfallermittlung* auch *error guessing* oder *special values testing* genannt, scheint auf den ersten Blick keinem Schema zu folgen. Das Testverfahren arbeitet weder *stochastisch* noch *deterministisch* und folgt einem *Ad-hoc Ansatz* (Pol, Teunissen & Veenendaal, 2002; Naik & Tripathy, 2011; Myers, Sandler & Badgett, 2011; Bath & McKay, 2014; Kribernegg, 2013; Liggesmeyer, 2009).

Die Qualität des Verfahrens ist im Gegensatz zu anderen Testverfahren nicht durch Überdeckungsmetriken, Prozesse oder Vorgänge definiert, sondern wird maßgeblich von der zu testenden Person bzw. der Gruppe festgelegt. Die Stakeholder haben meist viel Erfahrung im Testen von Software und kennen dadurch die Schwachstellen und Fallstricke der konkreten Software, der eingesetzten Methodik(en) oder generell der Softwareentwicklung gut oder sehr gut. Intuitive Testfallermittlung bzw. error guessing kann daher nicht erlernt werden (Pol et al., 2002; Naik & Tripathy, 2011; Myers et al., 2011; Bath & McKay, 2014; Kribernegg, 2013; Liggesmeyer, 2009).

Das Testverfahren eigent sich sehr gut als Ergänzung der systematisch agierenden Testverfahren, um dort die entsprechenden Freiheitsgrade intelligent zu nutzen. So ist es möglich, dass die Stakeholder ad-hoc Testfälle erwähnen und/oder erstellen können, die Fehler zu Tage treten lassen (Pol et al., 2002; Liggesmeyer, 2009).

Bei der Äquivalenzklassenbildung kommt das Verfahren bei der Auswahl der konkreten Testdaten zum Tragen, siehe dazu Abschnitt 3.3.1.1. Diese gibt das Verfahren nicht vor, sondern überlässt dies den testenden Stakeholdern. Durch intuitive Testfallermittlung bzw. Error guessing können die konkreten Werte intelligent und sinnvoll ausgewählt werden (Liggesmeyer, 2009).

3.3.3.2. Exploratives Testen

Das *explorative Testen* ist ein weiteres erfahrungsbasiertes Testverfahren. Ein Grundsatz dieses Verfahrens ist es, dass die Vorgänge des Lernens, der Testerstellung, Testausführung, Testauswertung und der Steuerung des Testens gleichzeitig stattfinden (Baumgartner, Klonk, Pichler, Seidl & Tanczos, 2018; Witte, 2016; Kribernegg, 2013; Schlich, 2019; Spillner, Roßner et al., 2014; Hendrickson, 2014; Spillner & Linz, 2012).

Das Hauptaugenmerk ist dabei die *Erfahrung* und die *Neugier* der ForscherInnen bzw. der testenden Stakeholder. Eine Vorbereitung auf das Testen ist daher nicht nötig und oft auch nicht möglich. Ziel ist es vielmehr, das Testobjekt zu *erforschen*, im Englischen *to explore it*, um daraus Tests und weitere Vorgehensweisen ableiten zu können (Baumgartner et al., 2018; Witte, 2016; Kribernegg, 2013; Schlich, 2019; Spillner, Roßner et al., 2014; Hendrickson, 2014; Spillner & Linz, 2012).

Dieser Vorgang ist an sich *iterativ*. Dadurch verbessert sich mit jedem Durchlauf bzw. Zyklus das Wissen um die zu testende Software und es erhöht sich die Qualität der Testerstellung, Testausführung, Testauswertung und natürlich auch der Steuerung des Ganzen (Baumgartner et al., 2018; Witte, 2016; Kribernegg, 2013; Schlich, 2019; Spillner, Roßner et al., 2014; Hendrickson, 2014; Spillner & Linz, 2012).

Das Testverfahren kann dann zum Einsatz kommen, wenn aus diversen Gründen wenig oder gar keine Informationen zum Testobjekt vorhanden sind. Das ist z.B. dann der Fall, wenn die Dokumentation zum Testobjekt unvollständig ist oder nur der Programmcode vorliegt (Witte, 2016; Kribernegg, 2013; Schlich, 2019; Spillner, Roßner et al., 2014; Hendrickson, 2014; Spillner & Linz, 2012).

Ein Ende des Testens kann dann erreicht sein, wenn das gewonnene Wissen rund um die Software ausreicht, um diese im Kontext bewerten zu können, andere Testverfahren basierend auf den gewonnenen Erkenntnissen darauf angesetzt werden, die definierte Zeit zu Ende ist oder andere definierte Ziele erreicht wurden. Das Testende ist somit nicht scharf definiert und muss von Fall zu Fall abgewogen werden. Dies obliegt hier, ebenso wie der gesamte Vorgang rund um das explorative Testing, stark der Einschätzung der testenden Stakeholder (Witte, 2016; Spillner, Roßner et al., 2014; Hendrickson, 2014; Spillner & Linz, 2012).

4. Metriken für die Testverfahren

In diesem Abschnitt werden die sogenannten *Metriken* entwickelt und vorgestellt. Diese sollen es ermöglichen, Testverfahren aus dem Abschnitt 3.3 vergleichbar zu machen.

4.1. Methodik zur Ermittlung der Metriken

Um Softwaretests vergleichbar machen zu können, ist es wichtig, sich über die Eigenschaften von Softwaretests, seien es Gemeinsamkeiten oder Unterschiede, Gedanken zu machen. Diese Eigenschaften können zum einen durch die Methode der *Literaturrecherche*, zum anderen durch die Praxis und deren Anforderungen mittels *ExpertInneninterviews* gewonnen werden.

In der Literatur gibt es über alle Testverfahren hinweg keine einheitlichen Metriken, die einen Vergleich möglich machen würden. Dennoch gibt es pro Kategorie wie z.B. bei den White-Box Testverfahren, siehe dazu Abschnitt 3.3.2, einige Hinweise auf Gemeinsamkeiten der dargelegten Verfahren. Im genannten Beispiel der White-Box Testverfahren sind dies die Überdeckungsmetriken, die jedes Verfahren auf unterschiedliche Art und Weise implementiert. Durch den direkten Vergleich der Testverfahren können weitere Eigenschaften sichtbar werden.

Anhand der Extraktion von Eigenschaften der Testverfahren durch Literaturrecherche, siehe dazu Abschnitt 3.3, war es möglich, einen Grundstock an Metriken zu erstellen. Anschließend wurden, die gewonnenen Erkenntnisse sortiert und kategorisiert. Der Schritt der Extraktion, der Sortierung und Kategorisierung wurde dabei solange wiederholt, bis kein nennenswerter Wissensgewinn mehr erzielt werden konnte.

Im Anschluss war es ratsam, die gewonnenen Erkenntnisse mittels ExpertInneninterviews bzw. Diskussionen mit den entsprechenden Stakeholdern zu erörtern. Dadurch war es möglich, diese zuvor gewonnenen Zwischenergebnisse weiter zu verfeinern bzw. die Tauglichkeit der potenziellen Metriken zu überprüfen. Ein laufender Sortier- und Kategorisierungsvorgang während dieser Tätigkeit war implizit gegeben. Durch diesen Austausch konnte es durchaus sein, dass sich kleine bis große Änderungen am Konzept ergaben.

Nach den notwendigen Gesprächen mit den ExpertInnen lag es nahe, den Vorgang der Sortierung, Kategorisierung sowie der Literaturrecherche so oft zu wiederholen, bis kein weiterer Erkenntnisgewinn mehr stattfand.

Im Anschluss stellte sich die Frage, ob die gewonnenen Metriken für ihre Aufgabe geeignet sind. Dies konnte mittels konkreten Bewertungen von ausgewählten Testverfahren eruiert werden. Auch hier war es entscheidend, die ExpertInnen einzubinden. War die Bewertung für die Stakeholder möglich, erfolgte die Einstufung der Metriken als tauglich. War dies allerdings nicht oder nur teilweise möglich, konnte dies an folgenden Faktoren liegen:

- Es wurden weitere Anforderungen an die Metriken entdeckt. Daher mussten die oben genannten Tätigkeiten wie Literaturrecherchen und/oder Diskussionen bzw. ExpertInneninterviews aufgrund der neuen Faktenlage wiederholt werden.
- Es war entsprechend der Anforderung nicht möglich, Metriken zu entwickeln. Eine adäquate Dokumentation der Fakten war hier von essenzieller Bedeutung.

4.2. Die ExpertInneninterviews

Für die ExpertInneninterviews war es äußerst hilfreich, ein methodisches Vorgehen zu definieren. Dieses Methodik ist zum einen in der Literatur, siehe Bogner, Littig und Menz (2013), beschrieben, zum anderen ergab sie sich aus der praktischen Notwendigkeit, Resultate und Ergebnisse zu bekommen, um diese im Anschluss verwenden zu können. Zu den ExpertInneninterviews gehörten folgende Aspekte:

- **1. Definition des Zieles:** Was soll durch ExpertInneninterviews erreicht werden?
- **2. Ableitung des Leitfadens:** Erstellung eines Leitfadens für die gesamten Gespräche.
- **3. Suche nach ExpertInnen:** Die Suche der ExpertInnen erfolgte nach gewissen Kriterien, um die erwarteten Informationen optimal verwenden zu können.
- **4. Durchführung der ExpertInneninterviews und Informationsmanagement:** Erfassung und Auswertung der Informationen, die durch die ExpertInneninterviews gewonnen wurden.

4.2.1. Definition des Zieles

Die *Definition des Zieles* war von großer Wichtigkeit. Die ExpertInneninterviews hatten grob gesehen zwei Ziele: Zum einen sollten sie es ermöglichen, Metriken, die im

ersten Schritt aus der Literatur aggregiert wurden, zu diskutieren, zu schärfen, zu erörtern und weitere darauf basierende Elemente wie z.B. (Teil)Metriken zu entdecken. Zum anderen war es wichtig, nach Abschluss der Entwicklung dieser Metriken diese sehr konkret zu bewerten, um weitere Aussagen darüber treffen zu können und die Metriken daher auch indirekt bestätigen zu können.

4.2.2. Ableitung des Leitfadens

Die *Ableitung des Leitfadens* erfolgte nach Festlegung der Ziele. Hier wurde mittels Beispielen aus der Literatur, siehe Bogner et al. (2013), ein Leitfaden erstellt. Dieser enthält wichtige organisatorische Elemente und führte durch die Interviews bzw. die Diskussionen. Die konkreten Leitfäden sind im Anhang A dargestellt.

4.2.3. Suche nach ExpertInnen

Die *Suche* der ExpertInnen erfolgte anhand einiger Kriterien. Dazu gehören unter anderem folgende Faktoren:

- 1. Fachwissen zum Thema Testing
- 2. Höchste abgeschlossene Ausbildung
- 3. Vorhandene facheinschlägige Weiterbildungen
- 4. Berufserfahrung
- 5. Alters- und Firmenverteilung der ExpertInnen
- 6. Regionale Einschränkung der ExpertInnen auf die Steiermark

Das *Fachwissen zum Thema Testing* ließ sich anhand eines Fragebogens evaluieren. Hier ergab sich, wie in Anhang B ersichtlich, dass die ExpertInnen ein sehr gutes Fachwissen zu ausgewählten Bereichen der Testverfahren aus dem Abschnitt 3.3 vorweisen konnten. Durch Lesen der Literatur wurde gewährleistet, dass Testverfahren, die unter einem anderen Begriff bekannt waren, richtig zugeordnet werden konnten.

Die *höchste abgeschlossene Ausbildung* ließ sich ebenfalls über einen Fragebogen ermitteln. Dieser Faktor war wichtig, um die Bildungsbandbreite der ExpertInnen aufzeigen zu können. Diese Bandbreite ist in Abbildung 4.1 ersichtlich.

Facheinschlägige Weiterbildungen waren bei vier von sieben ExpertInnen vorhanden. Diese gaben an, sich abseits ihrer Ausbildung bereits mindestens einmal in Bezug auf das Softwaretesting weitergebildet zu haben.

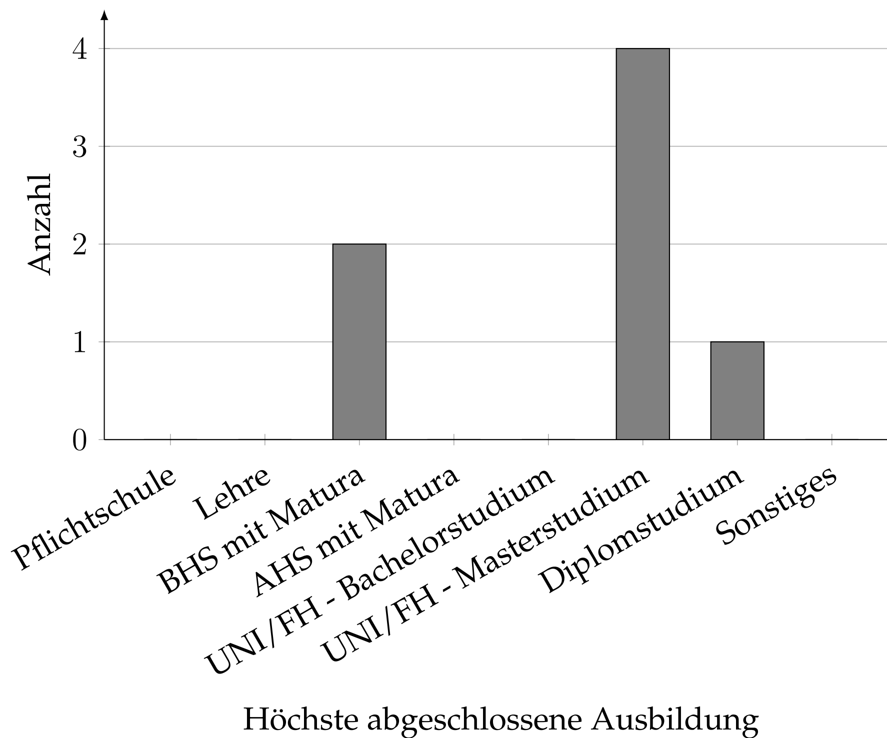


Abbildung 4.1.: Höchste abgeschlossene Ausbildung der ExpertInnen.

Einer der wichtigsten Faktoren bei der Auswahl der ExpertInnen war die *Berufserfahrung*. Diese ist in Abbildung 4.2 dargestellt. Je länger die ExpertInnen im Bereich der Softwareentwickelung bzw. des Softwaretestings arbeiten, desto wahrscheinlicher ist ein Zuwachs an praxisnahem Wissen in Bezug auf das Softwaretesting.

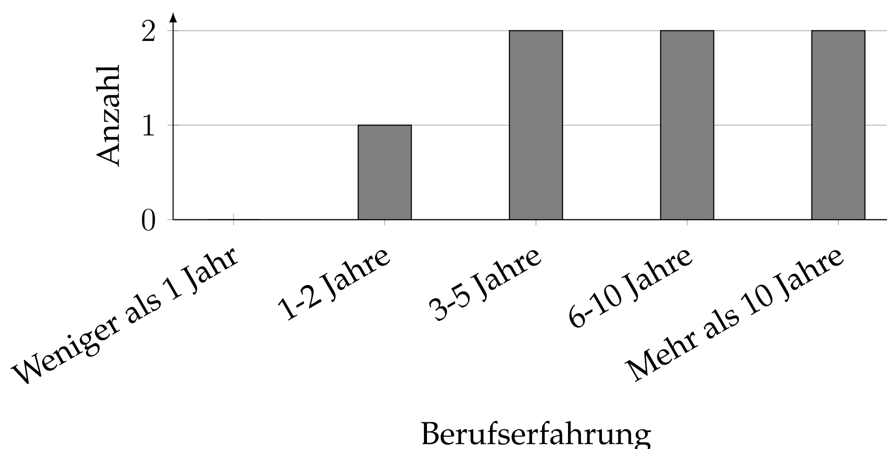


Abbildung 4.2.: Berufserfahrung der ExpertInnen.

Ein weiteres Kriterium bei der Auswahl war die *Alters- und Firmenverteilung der ExpertInnen*. Diese Faktoren rundeten das Bild weiter ab und gewährleisteten, dass nicht ausschließlich eine Alterskategorie an den Interviews teilnahm. Dies ist in Abbildung 4.3 dargestellt. Die meisten ExpertInnen arbeiten in unterschiedlichen Fir-

men in Österreich. Auch die Branchenvielfalt war gegeben. Daher traten hier keine nennenswerten Beeinflussungen aufgrund desselben Firmen-, Branchen- oder Abteilungsumfelds auf.

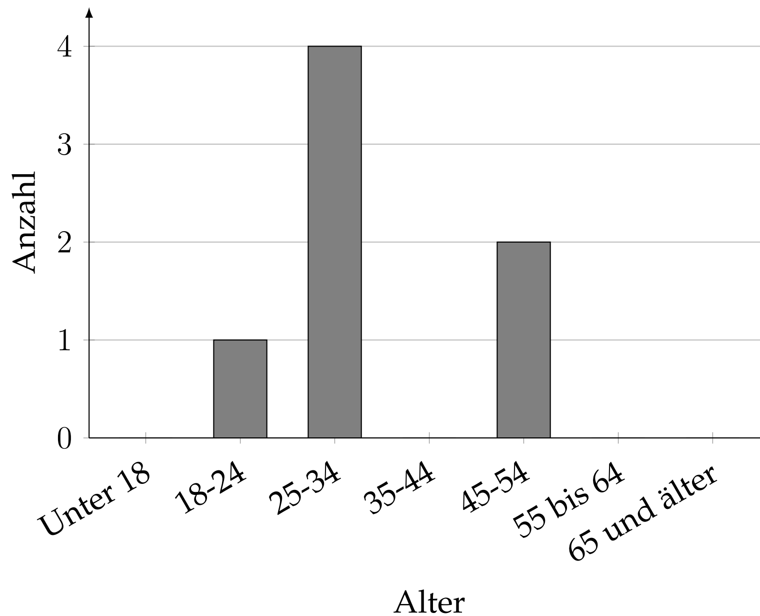


Abbildung 4.3.: Altersverteilung der ExpertInnen.

4.2.4. Durchführung und Informationsmanagement

Ein wesentlicher Faktor für die optimale Durchführung der ExpertInneninterviews war das *Informationsmanagement*. Ein Teil davon war die Vorbereitung und Organisation der Termine und Gespräche mit den ExpertInnen. Ein weiterer Teil war das Aufnehmen der Interviews sowie das Befragen der ExpertInnen anhand verschiedener Fragebögen bei der Durchführung der ExpertInneninterviews. Anschließend erfolgte das Auswerten und Bewerten der Informationen. Einige der so gewonnenen Erkenntnisse fließen in dieser Arbeit an verschiedenen Stellen ein und sind als solche gekennzeichnet.

4.3. Aufstellung und Beschreibung der Metriken

Wie bereits in Abschnitt 4.1 erwähnt, ist es Teil dieser Arbeit, die Metriken zu entwickeln und durch Vergabe von konkreten Werten pro Metrik und Testverfahren die Tauglichkeit der Metrik an sich festzustellen. Dies wird im Abschnitt 5 dargelegt.

Anhand der beschriebenen Methodik und unter Berücksichtung der erwähnten Blickwinkel wird im Folgenden die Aufstellung und Beschreibung der Metriken dargelegt.

4.3.1. Kategorie: Anwendbarkeit

In der Kategorie *Anwendbarkeit* ergeben sich zwei Metriken, die in Tabelle 4.1 aufgelistet sind. Die erste der beiden beschreibt dabei, ob das Testverfahren für neuen Code bzw. Features anwendbar ist, die zweite bezieht dies auf Fehlerkorrekturen. Bewertet werden die Testverfahren mit einer Likert Skala. Diese ermöglicht es je nach Testverfahren, Bewertungen in den Abstufungen von "Stimme völlig zu" bis "Stimme überhaupt nicht zu" zu vergeben.

Nr.	Metrik	Bewertungsskala	Wertebereich
1	Ist das Verfahren in Bezug auf neu entwickelten Code bzw. Features anwendbar?	Likert Skala	1: Stimme völlig zu 2: Stimme zu 3: Stimme weder zu noch nicht zu 4: Stimme nicht zu 5: Stimme überhaupt nicht zu
2	Ist das Verfahren in Bezug auf Fehlerkorrekturen anwendbar?		

Tabelle 4.1.: Metriken der Kategorie Anwendbarkeit mit Bewertungsskala und dem jeweils zulässigen Wertebereich.

4.3.2. Kategorie: Komplexität

Die Kategorie *Komplexität* enthält zwei Metriken, die in Tabelle 4.2 aufgelistet sind. Die Metriken beschreiben, wie intuitiv oder komplex das jeweilige Testverfahren per se ist, und ob ohne spezielle Werkzeuge wie z.B. Softwaresuiten das Verfahren durchgeführt werden kann. Die Bewertung erfolgt wie bereits in Abschnitt 4.3.1 beschrieben.

<table>
<tr><th>Nr.</th><th>Metrik</th><th>Bewertungsskala</th><th>Wertebereich</th></tr>
<tr><td>3</td><td>Ist das Testprinzip intuitiv und ohne spezielle Schulungen verständlich?</td><td rowspan="2">Likert Skala</td><td rowspan="2">1: Stimme völlig zu
2: Stimme zu
3: Stimme weder zu noch nicht zu
4: Stimme nicht zu
5: Stimme überhaupt nicht zu</td></tr>
<tr><td>4</td><td>Ist das Testverfahren simpel und ohne spezielle Tools anwendbar?</td></tr>
</table>

Tabelle 4.2.: Metriken der Kategorie Komplexität mit Bewertungsskala und dem jeweils zulässigen Wertebereich.

4.3.3. Kategorie: Abhängigkeit

Die Kategorie *Abhängigkeit* enthält eine Metrik. Diese ist in Tabelle 4.3 aufgelistet. Die Metrik bewertet, inwieweit Abhängigkeiten zwischen Testverfahren vorkommen.

Nr.	Metrik	Bewertungsskala	Wertebereich
5	Es gibt keine Abhängigkeiten zu einem anderen Verfahren, welches zuerst (teilweise) ausgeführt werden muss.	Likert Skala	1: Stimme völlig zu 2: Stimme zu 3: Stimme weder zu noch nicht zu 4: Stimme nicht zu 5: Stimme überhaupt nicht zu

Tabelle 4.3.: Die Metrik der Kategorie Abhängigkeit mit Bewertungsskala und dem jeweils zulässigen Wertebereich.

Dies kann sich dadurch äußern, dass Bedingungen für die Durchführung eines Testverfahrens B von Testverfahren A erledigt werden oder Vorgehensweisen und/-oder Zwischenergebnisse eines Testverfahren A für die Durchführung von Testverfahren B benötigt werden.

Dem Testverfahren, dem diese Eigenschaft mehr innewohnt bzw. als Basis für das andere gilt, wird diese Eigenschaft als *inhärent* angerechnet. Das jeweilige andere Testverfahren ist demnach von ersterem *abhängig*. Die Quantifizierung erfolgt erneut anhand der Likert-Skala.

Ein einfaches Beispiel dafür ist die Grenzwertbetrachtung: Da hierfür die Äquivalenzklassen benötigt werden, die auch beim Äquivalenzklassentest erstellt werden und diesem innewohnen, besitzt die Grenzwertbetrachtung eine Abhängigkeit von der Erstellung der Äquivalenzklassen. Siehe dazu Abschnitte 3.3.1.1 und 3.3.1.2.

4.3.4. Kategorie: Automatisierbarkeit

In der Kategorie *Automatisierbarkeit* kommen drei Metriken vor. Diese sind in der Tabelle 4.4 aufgelistet. Die Hauptaussage dieser Metriken bezieht sich darauf, ob das Testverfahren per se automatisierbar ist. Der maximal mögliche Automatisierungsgrad soll dabei als Prozentangabe in Relation zum Testverfahren ohne Automatisierung angegeben werden. Dieser Wert lässt sich für die Erstellung bzw. Wartung, für die Durchführung sowie die Auswertung von Tests, basierend auf dem jeweiligen Testverfahren, ermitteln.

Nr.	**Metrik**	**Bewertungsskala**	**Wertebereich**
6	In welchem Ausmaß ist das Verfahren in Bezug auf die Testerstellung bzw. Wartung automatisierbar?	Skala in %	0 bis 100%; Kleinstes Intervall: 10%
7	In welchem Ausmaß ist das Verfahren in Bezug auf die Testdurchführung automatisierbar?		
8	In welchem Ausmaß ist das Verfahren in Bezug auf die Testauswertung automatisierbar?		

Tabelle 4.4.: Die Metrik der Kategorie Automatisierbarkeit mit Bewertungsskala und dem jeweils zulässigen Wertebereich.

Besonders spannend dabei ist die Metrik 6 aus der Tabelle 4.4. Diese untersucht dabei nicht nur die Automatisierbarkeit in Bezug auf die einmalige Erstellung von Tests, basierend auf dem jeweiligen Testverfahren, sondern auch die Wartung desselben. Dies bedeutet, dass Testverfahren, die prinzipiell bei der Wartung und Erstellung von Tests einen hohen Automatisierungsgrad aufweisen, anders abschneiden als Testverfahren, die sich bei der Erstellung von Tests sehr gut, bei der Wartung von Tests aber nicht sehr gut automatisieren lassen.

4.3.5. Kategorie: Aufwand des Testings

Die Kategorie *Aufwand des Testings* ist umfangreich und beinhaltet acht Metriken. Diese sind in der Tabelle 4.5 aufgelistet.

Nr.	Metrik	Bewertungs-skala	Wertebereich
9	Welcher durchschnittliche Aufwand ist für das Erstellen bzw. die Wartung von Tests ohne Automatisierung im Vergleich zum Entwickeln des Codes notwendig?	Skala in %	Werte in %; Kleinstes Intervall: 10%
10	Welcher durchschnittliche Aufwand ist für das Erstellen bzw. die Wartung von Tests mit Automatisierung im Vergleich zum Entwickeln des Codes notwendig?		
11	Welcher durchschnittliche Aufwand ist für die Durchführung von Tests ohne Automatisierung im Vergleich zum Entwickeln des Codes notwendig?		
12	Welcher durchschnittliche Aufwand ist für die Durchführung von Tests mit Automatisierung im Vergleich zum Entwickeln des Codes notwendig?		
13	Welcher durchschnittliche Aufwand ist für das Auswerten der Testergebnisse ohne Automatisierung im Vergleich zum Entwickeln des Codes notwendig?		
14	Welcher durchschnittliche Aufwand ist für das Auswerten der Testergebnisse mit Automatisierung im Vergleich zum Entwickeln des Codes notwendig?		

Tabelle 4.5.: Darstellung der Metriken aus der Kategorie Aufwand des Testings mit Bewertungsskala und dem jeweils zulässigen Wertebereich.

Die Metriken, die zur Kategorie Aufwand des Testings gezählt werden, sind zentrale Elemente innerhalb aller verfügbaren Metriken. Metrik 9 und 10 beschreiben dabei die Erstellung und Wartung, 11 und 12 die Durchführung und 13 und 14 die Auswertung von Tests. Dabei bewerten die Metriken 9, 11 und 13 die Erfassung des Aufwands ohne Automatisierung, die Metriken 10, 12 und 14 die Erfassung des Aufwands mit Automatisierung. Die Bewertung des Aufwands erfolgt in Prozent, und

zwar die benötigter Zeit für die Testtätigkeit in Relation zur benötigten Entwicklungszeit.

Unter dem Begriff Testtätigkeiten fallen dabei das Erstellen, die Wartung, die Durchführung sowie die Auswertung von Tests, basierend auf dem jeweiligen Testverfahren. Zur *Entwicklung* gehören neben der klassischen Neuentwicklung von Features bzw. Komponenten auch die Wartung von Code oder fällige Fehlerkorrekturen.

Für das Verhältnis ist es dabei unerheblich, ob die Zeiteinheiten ZE in Minuten, Tagen oder sonstigen Maßeinheiten betrachtet werden, da sich diese wegkürzen lassen. Wichtig dabei ist es, im Nenner und Zähler dieselbe Maßeinheit zu verwenden. Dies wird in Gleichung 4.1 ersichtlich.

$$Aufwand\ in\ \% = \frac{Aufwand\ in\ ZE\ für\ die\ Testtätigkeit}{Aufwand\ in\ ZE\ für\ die\ Entwicklung} \quad (4.1)$$

Um den durchschnittlichen Aufwand entsprechend abschätzen zu können, wird ein Mittelwert über genügend viele Werte der Gleichung 4.1 gelegt. Diese Mittelwertbildung liegt im Auge des Betrachters. Das bedeutet, dass die ExpertInnen, die diese Werte liefern, dies selbst abschätzen. Damit wird erreicht, dass äußere Einflüsse weitgehend vermieden, die Werte belastbarer sowie vergleichbarer werden.

Im Folgenden sind einige Faktoren bzw. äußere Einflüsse in der Softwareentwicklung exemplarisch aufgelistet, die den Aufwand der Testtätigkeiten erheblich beeinflussen können:

- Die Bekanntheit und das Wissen rund um das Testverfahren
- Die aktuelle Anwendung oder Nichtanwendung des Testverfahrens
- Die Häufigkeit der Anwendung des Testverfahrens
- Die korrekte Anwendung des Testverfahrens laut Fachliteratur
- Der Grad an Toolunterstützung in Bezug auf das Testen
- Der Grad an Automatisierung in Bezug auf das Testen
- Die (integrierte) Entwicklungsumgebung, in der entwickelt wird
- Die Entwicklungssprache(n)

Schlussendlich ergibt sich somit eine Formel, die den durchschnittlichen Aufwand darlegt. Dieser in Gleichung 4.2 dargestellte Sachverhalt ermöglicht es, für alle sechs Metriken konkrete Werte zu ermitteln.

$$\varnothing\ Aufwand\ in\ \% = \frac{\varnothing\ Aufwand\ in\ ZE\ für\ die\ Testtätigkeit}{\varnothing\ Aufwand\ in\ ZE\ für\ die\ Entwicklung} \quad (4.2)$$

Um einen schnellen Vergleich zwischen den Testverfahren zu ermöglichen, ist es sinnvoll, die Werte der Metriken zu summieren. Dies erledigen die Metriken 15 und 16, dargestellt in Tabelle 4.6. Dabei summiert die Metrik 15 die Werte aus 9, 11 und 13, die Metrik 16 die Werte aus 10, 12 und 14.

Nr.	Metrik	Bewertungs-skala	Wertebereich
15	Welcher durchschnittliche Gesamtaufwand ist für den Vorgang des Testens ohne Automatisierung im Vergleich zum Entwickeln des Codes notwendig?	Skala in %	Werte in %; Kleinstes Intervall: 10%
16	Welcher durchschnittliche Gesamtaufwand ist für den Vorgang des Testens mit Automatisierung im Vergleich zum Entwickeln des Codes notwendig?		

Tabelle 4.6.: Summierte Metriken der Kategorie Aufwand des Testings mit Bewertungsskala und dem jeweils zulässigen Wertebereich.

4.3.6. Kategorie: Nutzen des Testings

Die letzte Kategorie, *Nutzen des Testings*, ist die umfangreichste unter allen und beinhaltet zehn Metriken. Diese sind in der Tabelle 4.7 aufgelistet.

Diese Metriken bewerten die jeweiligen Testverfahren anhand der Softwarequalitätskriterien, vorgestellt in Abschnitt 2.4. Die Bewertungsskala ist dabei eine Likert-Skala, die von "1: Äußerst hilfreich" bis hin zu "5: Überhaupt nicht hilfreich" reicht. So ist es möglich, je nach Qualitätskriterium die optimalen dafür geschaffenen Testverfahren zu erkennen.

Die ersten vier Qualitätskriterien beziehen sich dabei auf die kundInnenorientierten, die weiteren vier auf die herstellerInnenorientierten Qualitätskriterien, siehe dazu Abschnitt 2.4.

Die in Tabelle 4.7 vorgestellten Summen für die Metriken 25 und 26 ergeben sich aus den Mittelwerten von 17, 18, 19 und 20 bzw. 21, 22, 23 und 24. Die Zahl für die Berechnung der Mittelwerte ergibt sich aus der Zahl, die dem jeweiligen Likert-Wert zugeordnet ist. So ist dies z.B. beim Wert "2: Sehr hilfreich" die Zahl zwei.

Damit die Werte exakt sind, wird auf ein Runden der Werte und ein Rückführen in die Likert-Skala verzichtet. Somit sind z.B. Werte von 3,50 bis 4,25 unterscheidbar und ergeben nicht alle den Wert 3,00. Ein Vergleich mit den Summen ist möglich. Als Faustregel gilt: Je kleiner die Summe ist (als Minimum ist der Wert 1,00 möglich), desto hilfreicher ist das Verfahren, je höher die Summer ist (als Maximum ist der Wert 5,00 möglich), desto weniger hilfreich ist das Verfahren für die Überprüfung der jeweiligen auf den Mittelwert zusammengefassten Qualitätskriterien.

Nr.	Metrik	Bewertungs-skala	Wertebereich
17	Wie hilfreich ist das Verfahren für die Überprüfung des Qualitätskriteriums Funktionalität?	Likert Skala	1: Äußerst hilfreich 2: Sehr hilfreich 3: Etwas hilfreich 4: Nur bedingt hilfreich 5: Überhaupt nicht hilfreich
18	Wie hilfreich ist das Verfahren für die Überprüfung des Qualitätskriteriums Zuverlässigkeit?		
19	Wie hilfreich ist das Verfahren für die Überprüfung des Qualitätskriteriums Effizienz bzw. Laufzeit?		
20	Wie hilfreich ist das Verfahren für die Überprüfung des Qualitätskriteriums Benutzbarkeit?		
21	Wie hilfreich ist das Verfahren für die Überprüfung des Qualitätskriteriums Übertragbarkeit?		
22	Wie hilfreich ist das Verfahren für die Überprüfung des Qualitätskriteriums Änderbarkeit bzw. Wartbarkeit?		
23	Wie hilfreich ist das Verfahren für die Überprüfung des Qualitätskriteriums Testbarkeit?		
24	Wie hilfreich ist das Verfahren für die Überprüfung des Qualitätskriteriums Transparenz?		
25	Wie hilfreich ist das Verfahren für die Überprüfung der Qualitätskriterien Funktionalität, Zuverlässigkeit, Effizient bzw. Laufzeit und Benutzbarkeit?	Zahlenwerte	1,00 bis 5,00; Kleinstes Intervall: 0,25 bzw. $\frac{1}{4}$
26	Wie hilfreich ist das Verfahren für die Überprüfung der Qualitätskriterien Übertragbarkeit, Änderbarkeit bzw. Wartbarkeit, Testbarkeit und Transparenz?		

Tabelle 4.7.: Metriken der Kategorie Nutzen des Testings mit Bewertungsskala und dem jeweils zulässigen Wertebereich.

5. Resultate und Empfehlungen

Zu Beginn dieser Arbeit wurde der Rahmen derselben auf einer abstrakten Ebene vorgestellt. Im Anschluss wurden Grundlagen zum Thema Software und Softwarequalität, verschiedene Klassifikationen der Softwaretests sowie verschiedene Testverfahren der Softwareentwicklung beschrieben. Diesen folgend ging es darum, Metriken für Testverfahren zu entwickeln, um diese anhand verschiedener Kriterien bewertbar und vergleichbar zu machen. Dieser Abschnitt fasst am Ende der Arbeit die Ergebnisse zusammen, zieht ein Fazit und zeigt weitere Vorgehensweisen in Bezug auf dieses Forschungsgebiet auf.

5.1. Ausgangslage

Um Aussagen in Bezug auf die Ergebnisse der Arbeit tätigen zu können, ist es wichtig, anhand der Forschungsfrage und der dazugehörigen Hypothesen die wesentlichen Punkte aufzuzeigen:

Inwieweit ist es möglich, konkrete Softwaretests mithilfe von Metriken vergleichbar zu machen bzw. zu optimieren?

> H1: Softwaretests lassen sich durch eine Mindestanzahl von Metriken vergleichbar machen.
> H0: Softwaretests lassen sich nicht durch eine Mindestanzahl von Metriken vergleichbar machen.

Ausgehend davon ergeben sich gewisse Fragen bzw. Sachverhalte, die zur Beantwortung der Forschungsfrage unabdingbar sind. Diese werden im Anschluss Stück für Stück analysiert.

1) **Was sind konkrete Softwaretests?**
   Konkrete Softwaretests sind Testverfahren, hinter denen eine konkrete Methode und/oder Idee zum Testen von Software steht, wie sie in Abschnitt 3.3 aufgelistet sind. Diese Testverfahren, wie z.B. der Äquivalenzklassentest, beschreiben ein methodisches Vorgehen beim Testen von Software auf abstrakter Ebene,

das heißt ohne konkreten Bezug auf ein Softwareprojekt und dessen Einschränkungen in Bezug auf die Entwicklungssprachen bzw. -Umgebungen oder die Projektsituation.

2) **Was sind Metriken in Bezug auf Softwaretests?**
Die *Metriken in Bezug auf Softwaretests* werden in dieser Arbeit im Abschnitt 4 vorgestellt. Die Metriken haben die Aufgabe, das Testen anhand verschiedener Gesichtspunkte und Betrachtungsweisen in der Softwareentwicklung kategorisierbar und klassifizierbar zu machen. Ein häufiges Beispiel in der Theorie sind die Überdeckungsmetriken der White-Box Testverfahren. Diese spiegeln sich z.B. in den Metriken, die den Nutzen des Testings abbilden, wieder. In der Praxis ist oft der Aufwand im Vergleich zum Nutzen eines Verfahrens von Bedeutung. Da die Metriken Nutzen und Aufwand des Testings berücksichtigen, können daraus Erkenntnisse abgeleitet werden.

3) **Wie können Metriken vergleichbar gestaltet werden?**
Das *Vergleichen von Metriken* ist ein wesentliches Element, um daraus weitere Erkenntnisse ableiten zu können. Ein Anwendungsfall kann z.B. das Treffen von Entscheidungen auf Basis der Metriken sein. Das Ziel ist es, einzelne Metriken oder Gruppen von Metriken untereinander vergleichbar zu machen! Das hat den Effekt, anhand von Anforderungen an das Testverfahren wie z.B. ein möglichst niedriger Testaufwand oder ein möglichst hoher Nutzen in Bezug auf die Qualitätskriterien des Kunden, die besten dafür geeigneten Testverfahren auswählen zu können!

   Nicht immer werden Vergleiche Unterschiede ergeben. Dies hängt sehr stark vom Testverfahren ab. Zwei White-Box Testverfahren, die sehr ähnliche Eigenschaften haben, siehe dazu Abschnitt 3, werden natürlich ähnlich bis gleiche Eigenschaften in Bezug auf die Metriken haben. Daher werden dort Unterschiede besonders sichtbar, wo diese auch in den Testverfahren methodisch verankert sind.

   Ein wichtiger Punkt ist die Unterscheidung, anhand ausgewählter Metriken, zwischen der Bewertung von konkreten Softwaretests bzw. Testverfahren, wie sie in Abschnitt 3.3 vorgestellt werden, und der Bewertung von Softwaretests, die in konkreten Softwareprojekten eingesetzt werden! Entscheidend dabei ist, dass die Testverfahren generell nur von der Methode bzw. der Idee abhängig sind, die Bewertung von Softwaretests, die in konkreten Softwareprojekten eingesetzt werden, jedoch von vielen weiteren Faktoren wie das Know-how der MitarbeitInnen, der Projektsituation, der Sparte usw. abhängig sind.

4) **Gibt es eine Mindestanzahl an Metriken?**
Die *Mindestanzahl an Metriken* wurde durch die Methoden Literaturrecherche und ExpertInneninterviews, siehe dazu Abschnitt 3.3 bzw. Abschnitt 4, ermittelt. Dies war wichtig, um feststellen zu können, ob und wenn ja welche Me-

triken ein Mindestset bilden, das nicht weiter reduziert werden kann. Dadurch steigt die Effizienz bzw. die Wirkung der Metriken und Redundanzen können vermieden werden.

Weiters ist es notwendig, die vorher genannten Punkte im Hinblick auf die Randbedingungen der Forschungsfrage bzw. deren Bedeutungen in Bezug auf die Conclusio genauer zu betrachten. Dabei werden folgende Aspekte klar:

- **Zu Punkt 1): Welche konkreten Softwaretests werden verwendet?**
  Konkrete Softwaretests oder auch Testverfahren gibt es viele. Der Fokus dieser Arbeit liegt auf jenen, die im Abschnitt 2 bzw. Abschnitt 3 anhand unterschiedlicher Klassifizierungen der verschiedenen Testverfahren mithilfe der Literatur dargelegt wurden. Als Auswahl für die Anwendung in Bezug auf die Metriken wurden aus den vorher genannten und im Abschnitt 3.3 beschriebenen Testverfahren jene ausgewählt, die in der Praxis laut Literatur und ExpertInnen eine Rolle spielen. Diese sind in Tabelle C.1 bzw. C.2 sichtbar. Die Anwendung der Metriken auf weitere Testverfahren ist jederzeit einfach und unkompliziert möglich!

- **Zu Punkt 2): Gibt es Randbedingungen und wenn ja welche, um die Metriken aus Abschnitt 4 verwenden zu können?**
  Das Erstellen der Metriken für die Testverfahren erfolgte, wie im Abschnitt 4 beschrieben, mithilfe von Literaturrecherche und ExpertInneninterviews. Das Angebot der Literatur zu diesem Thema ist vielfältig und durchaus international, sodass eine große Bandbreite des existierenden Wissens zur Erstellung der Metriken genutzt werden konnte. Wie bereits in Abschnitt 4 dargelegt, ist es notwendig, das Wissen aus der Literatur mit dem Wissen aus der Praxis zu verbinden. Hierfür waren die ExpertInnen unverzichtbar. Eine mögliche Einschränkung dieser besteht darin, dass die ExpertInnen alle aus Österreich kommen. Durch die Kombination aus internationaler Literatur sowie lokalen ExpertInnen wird diese Einschränkung sofort wieder relativiert.

- **Zu Punkt 3): Inwiefern sind die Metriken aus Abschnitt 4 vergleichbar? Gibt es für die Vergleichbarkeit Randbedingungen?**
  Laut den Ergebnissen aus den ExpertInneninterviews sind die entwickelten Metriken prinzipiell vergleichbar. Wie bereits in Abschnitt 5.1 erwähnt, ist es jedoch wichtig abzuklären, ob die Metrik oder die Metrikgruppe, die Vergleiche liefern soll, sinnvoll anwendbar ist. So sind z.B. zwei sehr ähnliche Black-Box Testverfahren nicht wirklich vergleichbar, da sie dieselben Werte in Bezug auf einige Metriken haben. Weiters sind manche Metriken erst in Kombination mit anderen Metriken sinnvoll auswertbar, da sie bei vielen Verfahren ähnliche Werte haben. Diese Eigenschaften und Randbedingungen, die für sinnvolle Aussagen berücksichtigt werden müssen, werden im weiteren Verlauf dieses Abschnitts

anhand von Beispielen erörtert.

- **Zu Punkt 4): Gibt es eine Mindestanzahl der Metriken aus Abschnitt 4? Können Metriken für Vergleiche zwischen den Testverfahren ignoriert werden?** Die sogenannte Mindestanzahl an Metriken ist im Abschnitt 4 dargelegt. Die Metriken, die während der Literaturrecherche und den ExpertInneninterviews aus unterschiedlichen Gründen weggefallen sind, sind demzufolge hier nicht aufgelistet.

  Einige der Metriken sind teilweise eng mit anderen Metriken verwandt. Dies betrifft die Metriken, die Eigenschaften anderer Metriken als Summe darstellen. Das sind die Metriken 15 und 16, die jeweils 9, 11 und 13 sowie 10, 12 und 14 summieren oder 25 und 26, die die Metriken 17 bis 20 sowie 21 bis 24 summieren. Das Entscheidende dabei ist, dass die daraus entstandenen Metriken neue, oftmals sehr wichtige Kenngrößen für verschiedene Fragestellungen abbilden, die intuitiv so nicht ableitbar sind. Daher gelten diese Metriken nicht als überflüssig oder redundant, sondern als essentiell.

  Je nach Fragestellung ist es möglich, dass nicht alle Metriken benötigt werden. Bei der Frage, welches der Testverfahren den größtmöglichen Nutzen für den Kunden bringt, sind de facto nur fünf Metriken notwendig.

5.2. Anwendungsszenarien

Anhand der dargelegten Fakten ist es möglich, mit Hilfe einiger Beispiele verschiedene *Anwendungsszenarien* für die Praxis schematisch aufzuzeigen. Dabei ist es wichtig, das Konzept grundlegend zu kennen, um aussagekräftige und sinnvolle Ergebnisse zu erhalten.

Vor dem Start des Vorgangs ist es wichtig, sich anhand der Metriken gezielte Fragestellungen, die beantwortet werden sollen, zu überlegen. Im Anschluss kann der Vorgang durchgeführt werden. Dieser besagt, dass die Testverfahren, die betrachtet werden sollen, ausgewählt oder alternativ die hier vorgestellten, breit aufgestellten Testverfahren genützt werden sollen. Bei der ersten Durchführung des Vorgangs ist die Nutzung der hier verwendeten Testverfahren ausdrücklich empfohlen. Im Anschluss ist die Bewertungsmatrix mit Werten zu befüllen. Anhand der Fragestellungen wird die Matrix ausgewertet und die Ergebnisse können präsentiert werden.

Die Basis für die Vorstellung einiger Szenarien bildet die Bewertungsmatrix in Tabelle C.1 und C.2, die mit den ExpertInnen anhand der Metriken und der vorgestellten Testverfahren erstellt wurde. Dabei wird angenommen, dass die Matrix für jedes

vorgestellte Szenario ident ist. Entsprechend dieser Ausgangslage werden einige Szenarien vorgestellt.

Anwendungsszenario 1: Optimierung der Testverfahren im EntwicklerInnenteam

Relevante Fragen, die mit Hilfe der Bewertungsmatrix beantwortet werden sollen:

- Welche drei Testverfahren sind für unsere KundInnen sehr wichtig?
- Welche drei Testverfahren sind für die Qualität der Softwareentwicklung von großer Bedeutung?

Anhand der Bewertungsmatrix ist klar ersichtlich, dass die wichtigsten drei Testverfahren für die KundInnen der Use-Case Test, der Regressionstest und das explorative Testen sind. Die wichtigsten drei Testverfahren für das EntwicklerInnenteam in Bezug auf die Qualität der Softwareentwicklung sind die Bedingungsüberdeckung, die Zweigüberdeckung und die Defs-Uses-Überdeckung.

Anwendungsszenario 2: Optimierung des Aufwands der genutzten Testverfahren

Relevante Fragen, die mit Hilfe der Bewertungsmatrix beantwortet werden sollen:

- Welche drei der insgesamt 13 genutzten Testverfahren können am meisten automatisiert werden?
- Welche drei der insgesamt 13 genutzten Testverfahren können am wenigsten automatisiert werden?

Anhand der Bewertungsmatrix ist klar ersichtlich, dass die drei Testverfahren, die am meisten automatisiert werden können, der entscheidungstabellenbasierte Test, die Grenzwertbetrachtung und die Zweigüberdeckung sind. Die drei Testverfahren, die am wenigsten automatisiert werden können, sind der Use-Case Test, das error guessing und der Äquivalenzklassentest.

Anwendungsszenario 3: Optimale Testverfahren in Bezug auf die Softwarequalitätskriterien für HerstellerInnen und KundInnen

Relevante Fragen, die mit Hilfe der Bewertungsmatrix beantwortet werden sollen:

- Welche drei der insgesamt 13 genutzten Testverfahren sind in Bezug auf die Softwarequalitätskriterien für die HerstellerInnen und KundInnen am meisten von Bedeutung?
- Welche drei der insgesamt 13 genutzten Testverfahren sind in Bezug auf die Softwarequalitätskriterien für die HerstellerInnen und KundInnen am wenigsten von Bedeutung?

Diese Ergebnisse sind nicht direkt in der Bewertungsmatrix sichtbar. Durch die Gleichwertigkeit der Qualitätskriterien für HerstellerInnen und KundInnen reicht eine Mittelwertbildung der Metriken 25 und 26 aus, um das gewünschte Ergebnis zu erhalten. Demzufolge sind die Testverfahren Bedingungsüberdeckung, Defs-Uses-Überdeckung und Regressionstest die bedeutendsten Testverfahren, der Äquivalenzklassentest, das error guessing und das paarweise Testen die unbedeutendsten Testverfahren für HerstellerInnen und KundInnen.

Aus den drei vorgestellten Anwendungsszenarien ist klar ersichtlich, dass viele weitere Fragestellungen möglich sind. Eine Einstellungsmöglichkeit ist dabei die Menge an Testverfahren, die in die Matrix Eingang findet. Wie bereits erwähnt, ist es hierbei sehr wichtig, dass die ausgewählten Testverfahren Unterschiede untereinander aufweisen, da ansonsten die Ergebnisse der Metriken zu ähnlich ausfallen.

5.3. Conclusio und Ausblick

Ausgehend von der Softwareentwicklung, den Grundlagen des Softwaretestings sowie der Darlegung veschiedener Kategorisierungen von Fehlern, wurden verschiedene Klassifikationen in Bezug auf Softwaretests behandelt. Im Anschluss wurden diverse konkrete Testverfahren vorgestellt und ihre Vorgehensweisen und Methoden dargelegt. Aufbauend darauf wurden durch die Methode der Literaturrecherche und des ExpertInneninterviews Metriken entwickelt, welche die verschiedenen Testverfahren anhand ihrer Eigenschaften vergleichbar machten.

Aufgrund dieser Erkenntnisse ist es möglich, in Bezug auf die Forschungsfrage eine klare Antwort abzugeben: Ja, konkrete Softwaretests sind anhand ausgewählter Metriken vergleichbar. Weiters lassen sich Softwaretests durch eine Mindestanzahl an Metriken vergleichen. Die H1 Hypothese kann somit bestätigt werden.

Zusätzlich zur Beantwortung der Forschungsfrage konnten die Erkenntnisse anhand der Bewertungsmatrix so aufbereitet werden, dass eine Verwendung der Resultate in der Praxis relativ einfach möglich ist. Dadurch steht der Softwareindustrie und der Forschung ein Werkzeug zur Verfügung, mit dem weitere Erkenntnisse in beiden Bereichen gefördert werden können.

Aufbauend auf den Resultaten dieser Arbeit ist es möglich, Untersuchungen in EntwicklerInnenteams durchzuführen, um dort das Delta zwischen dem Ist-Stand bei der Verwendung der aktuellen Softwaretests und dem Soll-Stand, ermittelt durch die Bewertungsmatrix, zu erkennen. Die sich daraus ergebenden Optimierungspotenziale können untersucht und für die jeweilige Teams umgesetzt werden. Das Feedback der Teams bei der Umsetzung der Optimierungen ist dabei ein weiterer möglicher Untersuchungsgegenstand.

Der beschriebene Einsatz der Bewertungsmatrix ist nicht auf bestehende Teams oder Softwareprojekte begrenzt. In der Anfangsphase von Softwareprojekten oder beim Aufbau neuen EntwicklerInnenteams kann dieses Tool sinnvoll eingesetzt werden, um entsprechend den Metriken, von Anfang an optimale Testverfahren für die EntwicklerInnenteams und die Softwareprojekte zu verwenden. Dies ist für die ProjektmanagerInnen, die EntwicklerInnen, die KundInnen und alle weiteren Stakeholder der Softwareprojekte interessant, um einen optimalen Verlauf der Softwareentwicklung beobachten zu können! Diese Thematik kann in weiteren Untersuchungen im Detail erörtert werden.

A. ExpertInneninterviews

Leitfaden für das ExpertInneninterview

Rahmenbedingungen

Anrede: Welche Form der Anrede ist erwünscht?

Aufnahme des Gesprächs: Eine Aufnahme soll stattfinden, um mehr Inhalte besprechen zu können. Bitte um Zustimmung! Wenn ja, wird die Aufnahme ab jetzt gestartet.

Zeitlicher Rahmen: Es ist geplant, den ersten Teil des Interviews in etwa einer Stunde abzuhandeln. Die Ergebnisse und die eventuell darauf aufbauenden Fragen bzw. Diskussionspunkte sollen in einer zweiten Einheit, die je nach Umfang eine halbe Stunde bis Stunde dauern wird, erörtert werden.

Verwendung der Informationen: Alle Informationen werden in elektronischer Form weiterverarbeitet. Die Verwendung von Zitaten für die Masterarbeit erfolgt nur, wenn die ausdrückliche Zustimmung des Experten / der Expertin dafür vorliegt. Bitte um mündliche Zustimmung.

Fragen und Anmerkungen: Diese sind erwünscht und jederzeit willkommen!

Danke für die Zeit und die Bereitschaft, das Interview bzw. das Gespräch zu führen!

Vorstellungsrunde

Vorstellung des Experten bzw. der Expertin

Ausbildungen in Bezug auf das Testen von Software

Der Experte / die Expertin listet seine / ihre Ausbildungen auf.

Beruflicher Werdegang in Bezug auf das Testen von Softwareentwicklung

Der Experte / die Expertin listet seinen / ihren beruflichen Werdegang auf.

Weitere relevante Erfahrung in Bezug auf das Testen von Software

Der Experte / die Expertin listet weitere relevante Erfahrung auf.

Vorstellung meiner Person

Ausbildungen in Bezug auf das Testen von Software

- Abgeschlossenes Studium Telematik an der TU Graz (BSc)

- Aktuelles Masterstudium der IT- und Wirtschaftsinformatik an der FH Campus02 in Graz

Beruflicher Werdegang in Bezug auf das Testen von Software

- Aktuell berufstätig bei der BearingPoint GmbH in Premstätten/Graz als Product Owner (PO) im Bereich der Softwareentwicklung für Business Kunden

Weitere relevante Erfahrung in Bezug auf das Testen von Software

- Bestandene Zertifizierung im Jahres 2019: ISTQB® Certified Tester Foundation Level 2011
- Softwaretests und Abnahmen mit dem Entwicklungsteam bzw. dem Kunden gehören zum Alltag meiner Rolle als PO

Einstieg in die Thematik

Vorstellung des Themas

Aus meinem Background ergibt sich das generelle Interesse an Softwaretests. Dies führte zum Titel der Masterarbeit

„Untersuchung von Optimierungsmöglichkeiten bei der Durch- oder Einführung von Softwaretests anhand ausgewählter Randbedingungen“.

Die **Forschungsfrage** dazu lautet:
„Inwieweit ist es möglich, konkrete Softwaretests mithilfe von Metriken vergleichbar zu machen bzw. zu optimieren?“

Die Hypothesen dazu lauten:

1. **Hypothese zur Vergleichbarkeit**
   - *H1: Softwaretests lassen sich durch eine Mindestanzahl von Metriken vergleichbar machen.*
   - *H0: Softwaretests lassen sich nicht durch eine Mindestanzahl von Metriken vergleichbar machen.*
2. **Hypothese zur Optimierung**
   - *H1: Konkrete Softwaretests können signifikant optimiert werden, wenn entsprechende Metriken für diesen Test entwickelt werden.*
   - *H0: Konkrete Softwaretests können trotz entsprechender Metriken für den jeweiligen Test nicht signifikant optimiert werden.*

Der Fokus dieses Interviews bzw. Gesprächs liegt inhaltlich auf der Besprechung der ersten Hypothese.

Softwaretests und deren Metriken

Ausgangslage

Ausgangspunkt für das Interview bzw. das Gespräch ist die oben genannte erste Hypothese zur Vergleichbarkeit von Softwaretests

Fragen an den Experten / die Expertin:

Fragen zum Thema Softwaretests:

1. Welche konkreten Softwaretests kennen Sie / kennst du?
2. Welche konkreten Softwaretests haben Sie / hast du schon verwendet?
3. Welche Softwaretests verwenden Sie / verwendest du ständig?

Fragen zur Auswahl von Softwaretests:

4. Hatten Sie / hattest du bereits einmal die Situation, zwischen zwei oder mehreren Softwaretests eine Auswahl treffen zu müssen?
    1. Wenn ja, schildern Sie / schildere kurz die Situation.
    2. Wenn nein, warum war das Ihrer / deiner Meinung nach (noch) nicht der Fall?
5. Haben Sie sich / hast du dir schon einmal gewünscht, anhand von Auswahlkriterien Softwaretests vergleichen zu können?
    1. Wenn ja, schildern Sie / schildere kurz die Situation.
    2. Wenn nein, warum war das Ihrer / deiner Meinung nach nicht der Fall?

Fragen zu vergleichenden Metriken hinsichtlich Softwaretests:

6. Welche Metriken sollten Ihrer / deiner Meinung nach existieren, um Softwaretests vergleichbar machen zu können? Formulieren Sie / formuliere wenn möglich zwischen 10 und 30 Metriken.
7. Sollten diese Metriken verschiedenen Kategorien zuordenbar sein?
    1. Wenn ja welchen? Ordnen Sie / ordne diese den jeweiligen Kategorien zu.
    2. Wenn nein: Warum?

Vorstellung der bis jetzt erstellten Metriken:

8. Sind die dargestellten Metriken Ihrer / deiner Meinung nach ausreichend, um Softwaretests vergleichen zu können?
    1. Wenn ja, warum ist das der Fall?
    2. Wenn nein, warum ist das nicht der Fall?
9. Sind die dargestellten Metriken in Kombination mit den Kategorien ausreichend, um Ihrer / deiner Meinung nach Softwaretests vergleichen zu können?
    1. Wenn ja, warum ist das der Fall?
    2. Wenn nein, warum ist das nicht der Fall?
10. Ist die dargestellte Bewertungsskala sowie der zulässige Wertebereich Ihrer /deiner Meinung nach ausreichend? Wenn nein, schildern Sie / schildere bitte warum dem nicht so ist.

Fragen zu den Hypothesen:

11. Kennen Sie / kennst du Literatur in Bezug auf die Verwendung / Existenz von Metriken im Rahmen von Softwaretests? Wenn ja lassen Sie uns / lass uns kurz diese Informationen austauschen!

12. Ihrer / deiner Erfahrung nach: Welche der beiden Hypothesen, H1 und H0, ist realistischer? Warum?

Mindestanzahl der Metriken (Für das zweite Interview):

13. Wie viele Metriken Sind Ihrer / deiner Meinung nach mindestens notwendig, um Softwaretests vergleichbar zu machen? Ordnen Sie diese absteigend, beginnend mit der Wichtigsten an!
14. Warum sind Ihrer /deiner Meinung nicht mehr Metriken notwendig?
15. Metriken Excel, Frage dazu...

Leitfaden für den zweiten Teil des ExpertInneninterviews

Rahmenbedingungen

Anrede: Welche Form der Anrede ist erwünscht?

Aufnahme des Gesprächs: Eine Aufnahme soll stattfinden, um mehr Inhalte besprechen zu können. Bitte um Zustimmung! Wenn ja, wird die Aufnahme ab jetzt gestartet.

Zeitlicher Rahmen: Es ist geplant, den ersten Teil des Interviews in etwa einer Stunde abzuhandeln. Die Ergebnisse und die eventuell darauf aufbauenden Fragen bzw. Diskussionspunkte sollen in einer zweiten Einheit, die je nach Umfang eine halbe Stunde bis Stunde dauern wird, erörtert werden.

Verwendung der Informationen: Alle Informationen werden in elektronischer Form weiterverarbeitet. Die Verwendung von Zitaten für die Masterarbeit erfolgt nur, wenn die ausdrückliche Zustimmung des Experten / der Expertin dafür vorliegt. Bitte um mündliche Zustimmung.

Fragen und Anmerkungen: Diese sind erwünscht und jederzeit willkommen!

Danke für die Zeit und die Bereitschaft, das Interview bzw. das Gespräch zu führen!

Vorstellung der aktuellen Metriken

- Aktuelle Metriken als Ergebnis der ExpertInneninterviews
  - Grobe Vorstellung der Beschreibung der Metriken; Details folgen bei der konkreten Bewertung
- Gibt es aus deiner/Ihrer Sicht noch Diskussionsbedarf? Wenn ja zu welchem Punkt?

Vorstellung der Testverfahren

- Vorstellung der zu bewertenden Testverfahren
  - Auswahl aufgrund von diversen Kriterien
- Ev. Rückfragen zu den Fragebögen
- Gibt es aus deiner/Ihrer Sicht noch Diskussionsbedarf? Wenn ja zu welchem Punkt?

Bewertung der Testverfahren anhand der Metriken

- Gemeinsamer Start mit einem Verfahren
- Erarbeitung der anderen Verfahren in Eigenregie
  - Im Anschluss gemeinsame Besprechung der Ergebnisse
- Gibt es aus deiner/Ihrer Sicht noch Diskussionsbedarf? Wenn ja zu welchem Punkt?

Forschungsfrage und Hypothese

„Untersuchung von Optimierungsmöglichkeiten bei der Durch- oder Einführung von Softwaretests anhand ausgewählter Randbedingungen“.

Die **Forschungsfrage** dazu lautet:
„Inwieweit ist es möglich, konkrete Softwaretests mithilfe von Metriken vergleichbar zu machen bzw. zu optimieren?“

Die Hypothesen dazu lauten:

1. **Hypothese zur Vergleichbarkeit**
    - *H1: Softwaretests lassen sich durch eine Mindestanzahl von Metriken vergleichbar machen.*
    - *H0: Softwaretests lassen sich nicht durch eine Mindestanzahl von Metriken vergleichbar machen.*
2. **Hypothese zur Optimierung**
    - *H1: Konkrete Softwaretests können signifikant optimiert werden, wenn entsprechende Metriken für diesen Test entwickelt werden.*
    - *H0: Konkrete Softwaretests können trotz entsprechender Metriken für den jeweiligen Test nicht signifikant optimiert werden.*

<u>Fragen dazu:</u>
1. Wie sehen Sie / wie siehst du jetzt nach der Bewertung die erste Hypothese?
2. Möchten Sie / möchtest du noch etwas zu der ersten Hypothese hinzufügen?
3. Lassen Sich diese Metriken auf ein Mindestset reduzieren, also sind z.B. Redundanzen vorhanden, oder ist das Ihrer / deiner Meinung nach bereits das Mindestset an Metriken? Begründen Sie Ihre / begründe deine Antwort!
4. Wie sehen Sie / wie siehst du jetzt nach der Bewertung die zweite Hypothese?
5. Möchten Sie / möchtest du noch etwas zu der zweiten Hypothese hinzufügen?

B. Fragebögen

In diesem Anhang B werden einige Daten der Fragebögen dargestellt. Diese sind als Rohdaten ohne Interpretation(en) abgebildet. Eine mögliche Auswertung der Daten erfolgt im Hauptteil der Arbeit.

Die ersten Fragebögen, siehe Tabelle B.1 und Tabelle B.2, listen die Bekanntheit der Black-Box, White-Box und erfahrungsbasierten Testverfahren unter den ExpertInnen auf. Abkürzungen in Bezug auf den Parameter *Bekanntheit des Testverfahrens* sind:

- 1.: Das Verfahren ist mir ohne Lesen der Literatur bekannt
- 2.: Das Verfahren ist mir ohne Lesen der Literatur nicht bekannt
- 3.: Das Verfahren ist mir durch Lesen der Literatur bekannt
- 4.: Das Verfahren ist mir durch Lesen der Literatur nicht bekannt

	Bekanntheit des Testverfahrens			
Das Testverfahren	**1.**	**2.**	**3.**	**4.**
Der Äquivalenzklassentest	6	0	1	0
Die Grenzwertbetrachtung	7	0	0	0
Der zustandsbasierte Test	5	1	1	0
Der Use-Case Test	6	0	1	0
Der entscheidungstabellenbasierte Test	6	0	1	0
Das paarweise Testen	2	0	5	0
Der Regressionstest	7	0	0	0

Tabelle B.1.: Fragebogen in Bezug auf die Bekanntheit der Black-Box Testverfahren, bei einer Anzahl von sieben ExpertInnen.

Die Fragebögen in Tabelle B.3 und Tabelle B.4 listen die Anwendung der Black-Box-, White-Box- und erfahrungsbasierten Testverfahren der ExpertInnen im Alltag auf.

	Bekanntheit des Testverfahrens			
Das Testverfahren	**1.**	**2.**	**3.**	**4.**
Der Anweisungs- oder C_0 - Test	6	0	1	0
Der Zweig- oder C_1 - Test	6	0	1	0
Der Bedingungstest (oder C_3 - Test)	4	0	3	0
Die Defs-Uses Überdeckungen	0	1	6	0
Intuitive Testfallermittlung/Error guessing	6	0	1	0
Das explorative Testen	7	0	0	0

Tabelle B.2.: Fragebogen in Bezug auf die Bekanntheit der White-Box- und erfahrungsbasierten Testverfahren, bei einer Anzahl von sieben ExpertInnen.

Abkürzungen in Bezug auf den Parameter *Zustimmung* sind:

- 1.: Stimme völlig zu
- 2.: Stimme zu
- 3.: Stimme weder zu noch nicht zu
- 4.: Stimme nicht zu
- 5.: Stimme überhaupt nicht zu
- 6.: Keine Aussage möglich

	Zustimmung					
Das Testen von Software mittels genanntem Testverfahren gehört zu meinem Alltag	**1.**	**2.**	**3.**	**4.**	**5.**	**6.**
Der Äquivalenzklassentest	3	2	1	1	0	0
Die Grenzwertbetrachtung	3	2	1	1	0	0
Der zustandsbasierte Test	0	3	2	2	0	0
Der Use-Case Test	4	1	1	1	0	0
Der entscheidungstabellenbasierte Test	1	1	1	3	1	0
Das paarweise Testen	1	1	0	3	1	1
Der Regressionstest	5	2	0	0	0	0

Tabelle B.3.: Fragebogen in Bezug auf die Anwendung der Black-Box Testverfahren im Alltag der sieben ExpertInnen.

	Zustimmung					
Das Testen von Software mittels genanntem Testverfahren gehört zu meinem Alltag	**1.**	**2.**	**3.**	**4.**	**5.**	**6.**
Der Anweisungs- oder C_0 - Test	2	3	0	1	1	0
Der Zweig- oder C_1 - Test	2	3	0	1	1	0
Der Bedingungstest (oder C_3 - Test)	1	2	0	3	1	0
Die Defs-Uses Überdeckungen	0	1	1	3	2	0
Intuitive Testfallermittlung/Error guessing	4	1	1	1	0	0
Das explorative Testen	3	1	2	1	0	0

Tabelle B.4.: Fragebogen in Bezug auf die Anwendung der White-Box- und erfahrungsbasierten Testverfahren im Alltag der sieben ExpertInnen.

C. Bewertungsmatrix

Die Bewertungsmatrix stellt die konkreten Bewertungen der Testverfahren anhand der Metriken dar. Diese sind in Tabelle C.1 und Tabelle C.2 dargestellt. Die Anzahl der Stichproben pro Testverfahren für alle Metriken ist in Tabelle C.2 abgebildet.

	Metrik Nr.															
Testverfahren	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**
Der Äquivalenzklassentest	1,3	2,0	1,7	2,0	2,0	63	93	97	57	37	23	10	23	7	103	53
Die Grenzwertbetrachtung	1,3	1,5	1,8	1,0	2,5	58	98	98	53	28	10	3	15	3	78	33
Der zustandsbasierte Softwaretest	2,0	2,8	2,5	2,0	1,8	48	98	98	68	43	13	5	15	5	95	53
Der Use-Case Test	1,3	2,3	1,3	1,5	1,0	35	53	48	50	48	20	23	33	25	103	95
Der entscheidungstabellenbasierte Test	1,5	2,3	1,8	2,0	1,8	30	98	90	40	25	13	3	15	0	68	28
Das paarweise Testen	3,0	2,6	3,2	3,8	2,6	44	96	90	44	32	22	10	18	10	84	52
Der Regressionstest	3,8	2,0	2,5	3,0	3,0	20	100	100	43	35	13	5	13	3	68	43
Die Anweisungsüberdeckung	2,3	2,8	2,0	3,0	1,3	63	98	100	58	28	30	8	23	5	110	40
Die Zweigüberdeckung	1,4	2,4	2,6	3,0	2,4	66	96	96	48	24	28	6	26	4	102	34
Die Bedingungsüberdeckung	1,8	2,3	3,3	3,3	2,5	50	95	90	45	25	33	8	23	3	100	35
Die Defs-Uses Überdeckung	2,2	3,2	3,0	3,4	1,8	56	94	92	62	34	28	10	26	6	116	50
Das error guessing	2,0	2,4	2,2	1,6	1,4	4	44	48	24	34	12	20	12	22	48	76
Das explorative Testen	1,6	2,2	1,2	1,4	1,2	6	14	24	18	16	18	16	18	12	54	44

Tabelle C.1.: Bewertungsmatrix: Bewertung der Testverfahren mithilfe von Metriken der Kategorien Anwendbarkeit, Komplexität, Abhängigkeit, Automatisierbarkeit und Aufwand des Testings.

	Metrik Nr.										Stichprobe
Testverfahren	**17**	**18**	**19**	**20**	**21**	**22**	**23**	**24**	**25**	**26**	**Anzahl**
Der Äquivalenzklassentest	1,67	1,67	4,00	4,33	4,67	4,67	3,33	4,33	2,92	4,25	3
Die Grenzwertbetrachtung	2,00	1,50	4,25	3,25	2,75	2,75	4,25	4,00	2,75	3,44	4
Der zustandsbasierte Softwaretest	1,50	1,50	4,25	3,75	4,50	3,00	2,25	3,50	2,75	3,31	4
Der Use-Case Test	1,00	2,00	3,75	2,25	3,50	3,75	3,00	3,50	2,25	3,44	4
Der entscheidungstabellenbasierte Test	1,25	2,25	4,00	4,25	2,75	2,25	2,75	3,25	2,94	2,75	4
Das paarweise Testen	2,80	2,60	3,60	2,80	4,00	3,20	2,60	3,40	2,95	3,30	5
Der Regressionstest	1,75	2,50	3,50	1,75	1,75	2,25	2,50	3,25	2,38	2,44	4
Die Anweisungsüberdeckung	2,50	1,75	3,75	3,25	3,00	2,00	2,00	2,00	2,81	2,25	4
Die Zweigüberdeckung	2,60	1,80	3,60	4,00	3,60	1,60	1,20	1,60	3,00	2,00	5
Die Bedingungsüberdeckung	2,00	1,75	3,25	3,75	2,75	1,75	1,75	1,50	2,69	1,94	4
Die Defs-Uses Überdeckung	2,00	1,60	3,80	3,60	3,60	1,60	1,40	1,40	2,75	2,00	5
Das error guessing	3,00	2,80	3,60	3,00	3,80	3,20	3,60	3,60	3,10	3,55	5
Das explorative Testen	1,80	2,80	3,20	2,20	3,40	3,60	3,60	3,80	2,50	3,60	5

Tabelle C.2.: Bewertungsmatrix: Bewertung der Testverfahren mithilfe von Metriken der Kategorie Nutzen des Testings mit Angabe der Stichproben pro bewerteten Testverfahren.

Abbildungsverzeichnis

2.1. Prioritäteninversion und mögliche Lösung durch Prioritätenvererbung. Dadurch, dass Task C die Priorität von Task A erbt, wird er vor Task B abgearbeitet und kann somit den kritischen Abschnitt schneller verlassen. Im Anschluss kann Task A, der diese kritische Ressource benötigt, abgearbeitet werden und wird nicht mehr von Task B ausgebremst (vgl. Quade und Kunst, 2016). 8
2.2. Qualitätsmerkmale eines Software-Produkts (vgl. Hoffmann, 2013). . . 15
2.3. Korrelationsmatrix der verschiedenen Qualitätskriterien. Ein Plus- bzw. ein Minuszeichen drückt jeweils eine positive oder negative Korrelation zwischen den zwei betrachteten Qualitätskriterien aus (vgl. Hoffmann, 2013). 22

3.1. Darstellung der Merkmalsräume der Testklassifikation (in Anlehnung an Hoffmann, 2013). 26
3.2. Die vier Prüfebenen im Kontext der Entwicklungsphase sowie Programmstruktur (vgl. Hoffmann, 2013). 27
3.3. Ein möglicher Überblick über die verschiedenen Integrationsstrategien. 28
3.4. Die verschiedenen strukturorientierten Integrationsstrategien im Vergleich (vgl. Hoffmann, 2013). 29
3.5. Mögliche Gliederung der Prüfkriterien nach Hoffmann (2013) sowie Spillner und Linz (2012). 35
3.6. Mögliche Zustände und Zeigerpositionen eines Ringpuffers unter der Annahme von acht Speicherelementen (vgl. Hoffmann, 2013). 44
3.7. Der Zustandsübergangsgraph des Rinpuffers (vgl. Hoffmann, 2013). . 44
3.8. Der Zustandsbaum des Rinpuffers (vgl. Hoffmann, 2013). 45
3.9. Use-Case Diagramm (vgl. Spillner und Linz, 2012). 47
3.10. Ursache-Wirkungs-Graph des Beispiels Buchausleihe (vgl. Hoffmann, 2013 bzw. F. Wolf, 2018). 48
3.11. Kontrollflussgraph der Funktion *manhattan.c* (vgl. Hoffmann, 2013). . . 55
3.12. Kontrollflussgraphen der Funktion *manhattan.c* in Bezg auf die Zweigüberdeckung (vgl. Hoffmann, 2013). 57
3.13. Kontrollflussgraphen der Funktion *manhattan.c* in Bezg auf die Pfadüberdeckung (vgl. Hoffmann, 2013). 58

4.1. Höchste abgeschlossene Ausbildung der ExpertInnen. 68
4.2. Berufserfahrung der ExpertInnen. 68
4.3. Altersverteilung der ExpertInnen. 69

Tabellenverzeichnis

2.1. Modell von Trauboth (1996): Nennung einiger Beispiele, wo Fehler ihre Wurzeln haben können. 12
2.2. Vergleichsmodell von Frühauf et al. (2007) in Bezug auf Fehler und deren Wurzeln. 12

3.1. Beispiel zur Aufteilung in entsprechende Äquivalenzklassen (vgl. Liggesmeyer, 2009). 41
3.2. Grenzwertbetrachtungen zur Äquivalenzklasse aus Tabelle 3.1 43
3.3. Auflistung der notwendigen Tests aufgrund der Ergebnisse des ausgerollten Zustandsübergangsgraphen (vgl. Hoffmann, 2013). 46
3.4. Auflistung der vollständigen Entscheidungstabelle zum Beispiel der Buchausleihe (vgl. Hoffmann, 2013). 49
3.5. Auflistung der reduzierten Entscheidungstabelle, zum Beispiel der Buchausleihe (vgl. Hoffmann, 2013 bzw. F. Wolf, 2018). 50

4.1. Metriken der Kategorie Anwendbarkeit mit Bewertungsskala und dem jeweils zulässigen Wertebereich. 70
4.2. Metriken der Kategorie Komplexität mit Bewertungsskala und dem jeweils zulässigen Wertebereich. 71
4.3. Die Metrik der Kategorie Abhängigkeit mit Bewertungsskala und dem jeweils zulässigen Wertebereich. 71
4.4. Die Metrik der Kategorie Automatisierbarkeit mit Bewertungsskala und dem jeweils zulässigen Wertebereich. 72
4.5. Darstellung der Metriken aus der Kategorie Aufwand des Testings mit Bewertungsskala und dem jeweils zulässigen Wertebereich. 73
4.6. Summierte Metriken der Kategorie Aufwand des Testings mit Bewertungsskala und dem jeweils zulässigen Wertebereich. 75
4.7. Metriken der Kategorie Nutzen des Testings mit Bewertungsskala und dem jeweils zulässigen Wertebereich. 76

B.1. Fragebogen in Bezug auf die Bekanntheit der Black-Box Testverfahren, bei einer Anzahl von sieben ExpertInnen. 91
B.2. Fragebogen in Bezug auf die Bekanntheit der White-Box- und erfahrungsbasierten Testverfahren, bei einer Anzahl von sieben ExpertInnen. 92
B.3. Fragebogen in Bezug auf die Anwendung der Black-Box Testverfahren im Alltag der sieben ExpertInnen. 92

B.4. Fragebogen in Bezug auf die Anwendung der White-Box- und erfahrungsbasierten Testverfahren im Alltag der sieben ExpertInnen. 93

C.1. Bewertungsmatrix: Bewertung der Testverfahren mithilfe von Metriken der Kategorien Anwendbarkeit, Komplexität, Abhängigkeit, Automatisierbarkeit und Aufwand des Testings. 95
C.2. Bewertungsmatrix: Bewertung der Testverfahren mithilfe von Metriken der Kategorie Nutzen des Testings mit Angabe der Stichproben pro bewerteten Testverfahren. 96

Listings

2.1. POSIX konformes Verhalten aller 32-bit Computersysteme im Jahr 2038 mit numerischem Überlauf (vgl. Hoffmann, 2013). 10

3.1. manhattan.c: Berechnung der Manhattan-Distanz in der Entwicklungssprache C (vgl. Hoffmann, 2013) 56
3.2. Codefragment; Implementation der Logik nach Variante 1 (vgl. Hoffmann, 2013) 59
3.3. Codefragment; Implementation der Logik nach Variante 2 (vgl. Hoffmann, 2013) 59

Literaturverzeichnis

Agarwal, B. B. & Tayal, S. P. (2009). *Software Engineering* (2. Aufl.). New Delhi: Laxmi Publications.

Agarwal, B. B., Tayal, S. P. & Gupta, M. (2010). *Software Engineering and Testing*. Sudbury, Massachuetts: Jones & Bartlett Learning.

Ainapure, B. S. (2009). *Software testing and quality assurance*. Pune, India: Technical Publications.

Ammann, P. & Offutt, J. (2008). *Introduction to Software Testing*. Cambridge: Cambridge University Press.

Basu, A. (2015). *Software quality assurance, testing and metrics*. New Delhi: PHI Learning Pvt. Ltd.

Bath, G. & McKay, J. (2014). *The Software Test Engineer's Handbook - A Study Guide for the ISTQB Test Analyst and Technical Test Analyst Advanced Level Certificates 2012* (2. Aufl.). Santa Barbara: Rocky Nook, Inc.

Bath, G. & McKay, J. (2015). *Praxiswissen Softwaretest - Test Analyst und Technical Test Analyst: Aus- und Weiterbildung zum Certified Tester - Advanced Level nach ISTQB-Standard* (3. Aufl.). Heidelberg: dpunkt.Verlag.

Baumgartner, M., Klonk, M., Pichler, H., Seidl, R. & Tanczos, S. (2018). *Agile Testing - Der agile Weg zur Qualität* (2. Aufl.). München: Carl Hanser Verlag.

Becker, G. (1989). *Softwarezuverlässigkeit - Quantitative Modelle und Nachweisverfahren*. Berlin, New York: de Gruyter.

Belli, F., Pfleger, S. & Seifert, M. (1984). *Software-Fehlertoleranz und -Zuverlässigkeit*. Berlin Heidelberg New York: Springer-Verlag.

Birkmann, J., Bach, C., Guhl, S., Witting, M., Welle, T. & Miron, S. (2010). *State of the Art der Forschung zur Verwundbarkeit kritischer Infrastrukturen am Beispiel Strom/Stromausfall*. Berlin: Forschungsforum Öffentliche Sicherheit.

Bogner, A., Littig, B. & Menz, W. (2013). *Das Experteninterview - Theorie, Methode, Anwendung*. Berlin Heidelberg New York: Springer-Verlag.

Bommer, C., Spindler, M. & Barr, V. (2008). *Softwarewartung - Grundlagen, Management und Wartungstechniken* (1. Aufl.). Heidelberg: dpunkt.Verlag.

Broy, M. & Kuhrmann, M. (2013). *Projektorganisation und Management im Software Engineering* (1. Aufl.). Berlin Heidelberg: Springer Vieweg.

Brunst, M., Raszczyk, R. & Schneider, H. (2004). *Web Site Engineering: Einführung, verschiedene Vorgehensmodelle und im Detail das WSE-Komponentenmodell* (1. Aufl.). München: GRIN Verlag.

Cleff, T. (2010). *Basiswissen Testen von Software - Vorbereitung zum Certified Tester (Foundation Level) nach ISTQB-Standard* (1. Aufl.). Witten: W3l GmbH.

Craig, R. D. & Jaskiel, S. P. (2002). *Systematic Software Testing*. Norwood: Artech House.

Daigl, M. & Glunz, R. (2016). *ISO 29119 - Die Softwaretest-Normen verstehen und anwenden*. Heidelberg: dpunkt.Verlag.

Desai, S. & Srivastava, A. (2016). *Software testing : A practical approach* (2. Aufl.). Delhi India: PHI Learning.

Dustin, E., Rashka, J. & Paul, J. (2001). *Software automatisch testen - Verfahren, Handhabung und Leistung*. Berlin Heidelberg New York: Springer-Verlag.

Ebert, C. & Dumke, R. (1996). *Software-Metriken in der Praxis - Einführung und Anwendung von Software-Metriken in der industriellen Praxis*. Berlin Heidelberg New York: Springer-Verlag.

Franz, K. (2007). *Handbuch zum Testen von Web-Applikationen: Testverfahren, Werkzeuge, Praxistipps*. Berlin Heidelberg: Springer-Verlag.

Franz, K. (2014). *Handbuch zum Testen von Web- und Mobile-Apps - Testverfahren, Werkzeuge, Praxistipps* (2. Aufl.). Berlin Heidelberg: Springer-Verlag.

Frühauf, K., Ludewig, J. & Sandmayr, H. (2007). *Software-Prüfung - eine Anleitung zum Test und zur Inspektion* (6. überarb. u. aktualis. Aufl.). Zürich: vdf Hochschulverlag AG.

Fuchs, C. & Hofkirchner, W. (2003). *Studienbuch Informatik und Gesellschaft* (1. Aufl.). Norderstedt: BoD – Books on Demand.

Gerlich, R. & Gerlich, R. (2005). *111 Thesen zur erfolgreichen Softwareentwicklung - Argumente und Entscheidungshilfen für Manager. Konzepte und Anleitungen für Praktiker*. Berlin Heidelberg New York: Springer-Verlag.

Goll, J. (2011). *Methoden und Architekturen der Softwaretechnik* (1. Aufl.). Wiesbaden: Vieweg + Teubner Verlag.

Grechenig, T., Bernhart, M., Breiteneder, R. & Kappel, K. (2010). *Softwaretechnik: Mit Fallbeispielen aus realen Entwicklungsprojekten*. München: Pearson Studium.

Grünfelder, S. (2017). *Software-Test für Embedded Systems: Ein Praxishandbuch für Entwickler, Tester und technische Projektleiter* (2. Aufl.). Heidelberg: dpunkt.Verlag.

Hass, A. M. J. (2008). *Guide to advanced software testing*. Norwood: Artech House.

Hehl, W. (2008). *Trends in der Informationstechnologie - von der Nanotechnologie zu virtuellen Welten* (1. Aufl.). Zürich: vdf Hochschulverlag AG.

Hendrickson, E. (2014). *Explore It! - Wie Softwareentwickler und Tester mit explorativem Testen Risiken reduzieren und Fehler aufdecken* (1. Aufl.). Heidelberg: dpunkt.Verlag.

Hoffmann, D. W. (2013). *Software-Qualität* (2. Aufl.). Berlin Heidelberg New York: Springer-Verlag.

IEEE. (1990, Dec). IEEE standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, 1-84. doi: 10.1109/IEEESTD.1990.101064

Irrgang, R. (1995). *Entscheidungstabellen-Technik - Entscheidungstabellen erstellen und analysieren*. Renningen: expert verlag.

ISO/IEC 25010. (2011). *ISO/IEC 25010, systems and software engineering — systems and software quality requirements and evaluation (square) — system and software quality models.*

ISO/IEC 9126. (2001). *ISO/IEC 9126. software engineering – product quality*. ISO/IEC.

Jorgensen, P. C. (2014). *Software testing - a craftsman's approach, fourth edition* (4. Aufl.). Boca Raton, Fla: CRC Press.

Khannur, A. (2014). *Structured software testing: The discipline of discovering*. Partridge Publishing India.

Kleuker, S. (2019). *Qualitätssicherung durch Softwaretests - Vorgehensweisen und Werkzeuge zum Testen von Java-Programmen* (2. Aufl.). Wiesbaden: Springer Vieweg.

Kribernegg, A. (2013). *Software - test it professionally! - Ausbildung für zertifizierte Softwaretester Foundation und Full Advanced Level nach ISTQB-Standards*. Wien: Cyberpublishing E-Verlag GmbH.

Kuhn, D. R., Kacker, R. N. & Lei, Y. (2016). *Introduction to combinatorial testing*. Boca Raton: CRC Press.

Lassmann, W., Schwarzer, J. & Rogge, R. (2006). *Wirtschaftsinformatik - Nachschlagewerk für Studium und Praxis* (1. Aufl.). Wiesbaden: Gabler.

Leach, R. (2016). *Introduction to software engineering* (2. Aufl.). Boca Raton: CRC Press, Taylor & Francis Group.

Lee, C. (2004). *A practitioner's guide to software test design*. Boston London: Artech House Publishers.

Lehmbach, J. (2012). *Vorgehensmodelle im Spannungsfeld traditioneller, agiler und Open-Source-Softwareentwicklung : Analyse, Vergleich, Bewertung*. Stuttgart: Ibidem Verlag.

Liggesmeyer, P. (2009). *Software-Qualität: Testen, Analysieren und Verifizieren von Software* (2. Aufl.). Heidelberg: Spektrum.

Linz, T. (2014). *Testing in scrum: A guide for software quality assurance in the agile world*. Santa Barbara: Rocky Nook.

Liu, H. (2009). *Software performance and scalability - a quantitative approach*. Hoboke: John Wiley and Sons.

Metzner, A. (2020). *Software Engineering - kompakt*. München: Carl Hanser Verlag GmbH Co KG.

Meyer, A. (2018). *Softwareentwicklung: Ein Kompass für die Praxis*. Berlin München Boston: Walter De Gruyter GmbH.

Myers, G. J., Sandler, C. & Badgett, T. (2011). *The art of software testing* (3. Aufl.). New Yersey: John Wiley and Sons.

Müller, T., Käser, H., Gübeli, R. & Klaus, R. (2005). *Technische Informatik I - Grundlagen der Informatik und Assemblerprogrammierung* (2. Aufl.). Zürich: vdf Hochschulverlag AG.

Naik, K. & Tripathy, P. (2011). *Software testing and quality assurance - theory and practice*. New Yersey: John Wiley and Sons.

Nyamsi, E. A. (2019). *Projektmanagement mit Scrum - Tools zur Entwicklung von Software* (1. Aufl.). Berlin Heidelberg New York: Springer Vieweg.

Nörenberg, R. (2012). *Effizienter Regressionstest von E/E-Systemen nach ISO 26262*. Karlsruhe: KIT Scientific Publishing.

O'Regan, G. (2019). *Concise guide to software testing*. Cham Switzerland: Springer Nature.

Pol, M., Teunissen, R. & Veenendaal, E. v. (2002). *Software testing - a guide to the tmap approach*. Amsterdam: Addison-Wesley.

Puntambekar, A. (2009). *Software engineering* (1. Aufl.). Pune, India: Technical Publications.

Quade, J. & Kunst, E.-K. (2016). *Linux-Treiber entwickeln - Eine systematische Einführung in die Gerätetreiber- und Kernelprogrammierung - jetzt auch für Raspberry Pi* (4. aktualisierte und erweiterte Aufl.). Heidelberg: dpunkt.Verlag.

Riedermann, E. H. (1997). *Testmethoden für sequentielle und nebenläufige Software-Systeme*. Wiesbaden: Springer Fachmedien.

Roitzsch, E. H. P. (2005). *Analytische Softwarequalitätssicherung in Theorie und Praxis* (1. Aufl.). Münster: MV-Verlag.

Saleh, K. A. (2009). *Software engineering*. Fort Lauderdale: J. Ross Publishing.

Sandhaus, G., Knott, P. & Berg, B. (2014). *Hybride Softwareentwicklung: Das Beste aus klassischen und agilen Methoden in einem Modell vereint*. Berlin Heidelberg: Springer Vieweg.

Schlich, M. (2019). *Softwaretesten nach ISTQB für Dummies*. Weinheim: Wiley VCH Verlag GmbH.

Schmidt, D. (1994). *Erfolgreich Programmieren mit Ada - Unter Berücksichtigung des objektorientierten Standards*. Berlin Heidelberg New York: Springer-Verlag.

Schmitz, P., Bons, H. & van Megen, R. (1983). *Software-Qualitätssicherung — Testen im Software-Lebenszyklus* (2. Aufl.). Braunschweig: Friedr. Vieweg & Sohn Verlagsgesellschaft mbH.

Schneider, K. (2012). *Abenteuer Softwarequalität - Grundlagen und Verfahren für Qualitätssicherung und Qualitätsmanagement* (2. Aufl.). Heidelberg: dpunkt.Verlag.

Schwarz, H.-R. & Köckler, N. (2011). *Numerische Mathematik* (8. aktualisierte Aufl.). Berlin Heidelberg New York: Springer-Verlag.

Spillner, A. & Linz, T. (2012). *Basiswissen Softwaretest - Aus- und Weiterbildung zum Certified Tester - Foundation Level nach ISTQB-Standard* (5. Aufl.). Heidelberg: dpunkt.Verlag.

Spillner, A., Linz, T. & Schaefer, H. (2014). *Software testing foundations: A study guide for the certified tester exam* (4. Aufl.). Santa Barbara: Rocky Nook.

Spillner, A., Roßner, T., Winter, M. & Linz, T. (2014). *Praxiswissen Softwaretest - Testmanagement - Aus- und Weiterbildung zum Certified Tester - Advanced Level nach ISTQB-Standard* (4. überarb. u. erw. Aufl.). Heidelberg: dpunkt.Verlag.

Spillner, A., Winter, M. & Pietschker, A. (2017). *Test, Analyse und Verifikation von Software – gestern, heute, morgen*. Heidelberg: dpunkt.Verlag.

Steinhorst, W. (1999). *Sicherheitstechnische systeme - zuverlässigkeit und sicherheit kontrollierter und unkontrollierter systeme*. Braunschweig Wiesbaden: Vieweg.

Steyer, R. (2004). *Java Script - in 21 Tagen* (1. Aufl.). München: Markt+Technik Verlag.

Sutherland, J. (2015). *Die SCRUM Revolution: Management mit der bahnbrechenden Methode der erfolgreichsten Unternehmen*. Frankfurt New York: Campus Verlag.

Tomohiko, T., Takeshi, U. & Zengo, F. (2013). *Back-to-back testing framework using a machine learning method* (R. Lee, Hrsg.). Berlin Heidelberg New York: Springer.

Trauboth, H. (1996). *Software-Qualitätssicherung - Konstruktive und analytische Maßnahmen* (2. aktualisierte Aufl.; A. Endres, Hrsg.). München: R. Oldenbourg Verlag.

Voges, U. (1989). *Software-Diversität und ihre Modellierung - Software-Fehlertoleranz und ihre Bewertung durch Fehler- und Kostenmodelle*. Berlin Heidelberg New York:

Springer-Verlag.
Wagner, S. (2013). *Software product quality control*. Berlin Heidelberg: Springer-Verlag.
Wang, L. & Tan, K. C. (2006). *Modern industrial automation software design*. New Yersey: John Wiley & Sons.
Witte, F. (2016). *Testmanagement und Softwaretest: Theoretische Grundlagen und praktische Umsetzung* (2. Aufl.). Wiesbaden: Springer Vieweg.
Witte, F. (2018). *Metriken für das Testreporting - Analyse und Reporting für wirkungsvolles Testmanagement*. Berlin Heidelberg New York: Springer-Verlag.
Wolf, F. (2018). *Fahrzeuginformatik: Eine Einführung in die Software- und Elektronikentwicklung aus der Praxis der Automobilindustrie*. Wiesbaden: Springer Vieweg.
Wolf, H., van Solingen, R. & Rustenburg, E. (2014). *Die Kraft von Scrum: Inspiration zur revolutionärsten Projektmanagementmethode*. Heidelberg: dpunkt.Verlag.