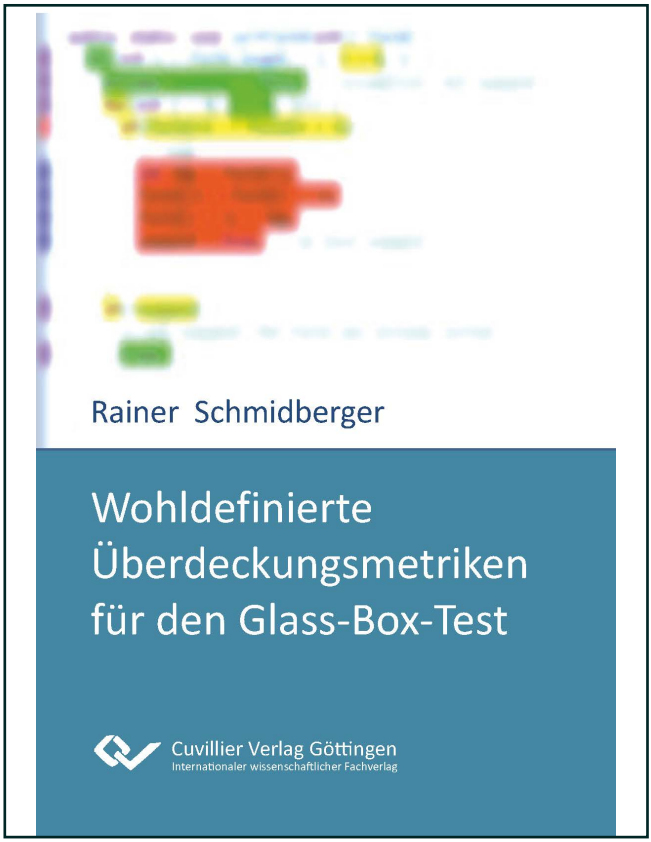
Rainer Schmidberger (Autor)

Wohldefinierte Überdeckungsmetriken für den Glass-Box-Test



https://cuvillier.de/de/shop/publications/6761

1
Einleitung und Überblick

Dieses Kapitel liefert einen Überblick über den Inhalt dieser Arbeit. Zunächst wird die Motivation vorgestellt, die der Arbeit zugrunde liegt und es werden die Zielsetzung sowie der Lösungsansatz der Arbeit beschrieben. Anschließend folgt die Inhaltsübersicht der einzelnen Kapitel.

1.1 Motivation

Der Programmtest hat sich in den letzten Jahren von einer „Randerscheinung“ zu einem zentralen und als erfolgskritisch wahrgenommenen Projektbestandteil entwickelt. Entsprechend verwenden die Unternehmen einen beträchtlichen Anteil der Projektbudgets für den Test, der in vielen Fällen dennoch nicht die angestrebte Güte erreicht – (zu)viele, auch schwerwiegende Fehler bleiben im Prüfling unentdeckt. Gleichzeitig spielt die Qualität der Software eine immer größere Rolle und die Zahl an „Post Release Defects“ entwickelt sich für die Unternehmen zu einem entscheidenden Wettbewerbsfaktor. Dieser gestiegenen Wahrnehmung und wirtschaftlichen Bedeutung des Tests folgend besteht in der Industrie großes Interesse daran, die vormals „intuitiv“ und „hemdsärmelig“ durchgeführten Testtechniken zu systematisieren.

Einen Beitrag zur Systematisierung des Tests kann der Glass-Box-Test (GBT) liefern, der den im Test ausgeführten (und damit auch den nicht ausgeführten) Programmcode anzeigt. Zwar erscheint uns heute der GBT als eine ausgereifte und etablierte Testtechnik, es zeigen sich jedoch bei genauer Betrachtung der hierzu genutzten Modelle und Metriken erhebliche Mängel: Wichtige GBT-relevante Programmmerkmale werden in den Modellen nicht berücksichtigt und für die Übertragung der Programme in das Modell gibt es keinen Standard.

Der Glass-Box-Test ist dabei keineswegs neu: Erste Arbeiten von 1963 gehen auf Miller und Maloney zurück [MM63], und bereits 1975 beschreibt Huang [Hu75] den Glass-Box-Test prinzipiell in der Form, wie er auch heute in den Lehrbüchern (wie z. B. [Li02, SL04]) behandelt wird[1]. Huang definiert Anweisungs- und Zweigüberdeckung auf Grundlage des Kontrollflussgraphen und beschreibt, wie Zähler in ein Programm eingefügt werden

[1] Soweit man es heute zurückverfolgen kann, wurde der GBT unabhängig von mehreren Autoren in der Zeit ab 1963 bis 1975 publiziert. Der Artikel von Huang hebt sich aber dadurch ab, dass er den GBT weitgehend in der heute bekannten Form beschreibt.

und wie die Auswertung des GBT erfolgt. Der Kontrollflussgraph ist seitdem im Wesentlichen unverändert das vorherrschende GBT-Modell. Dabei wird durch den Wegfall von (oder Verzicht auf) Goto-Anweisungen in den aktuellen Programmiersprachen der Kontrollfluss einfacher, Ausnahmen (Exceptions) wiederum machen den möglichen Kontrollfluss deutlich aufwändiger. Beides wird von der aktuellen Literatur zum GBT kaum berücksichtigt. Der CFG erlaubt zwar präzise, auf der Graphentheorie basierende Definitionen. Programme gängiger Programmiersprachen können aber nicht angemessen in den CFG abgebildet werden. Auch Ausnahmebehandlung oder die GBT-relevanten Ausdrücke lassen sich im CFG nicht zufriedenstellend abbilden.

Für das Verständnis einer im GBT ermittelten Überdeckung und der daraus resultierenden Schlussfolgerungen ist das zugrunde liegende Modell von großer Bedeutung. Wenn z. B. von einem Abnahmetest eine 80-prozentige Anweisungsüberdeckung gefordert wird, ist es bedeutsam, ob if-Anweisungen oder Schleifen selbst als Anweisung gewertet werden und damit zur Anweisungsüberdeckung beitragen oder nicht. Ein standardisiertes Referenzmodell des GBT, das diese Details umfassend definiert und auch strukturierte Programmiersprachen unterstützt, gibt es heute nicht. Dadurch sind viele, selbst grundlegende Überdeckungsmetriken bis heute für gängige Programmiersprachen nicht einheitlich definiert. Als logische Konsequenz folgen die Werkzeuge des GBT keinem gemeinsamen Standard und zeigen für die gleiche Programmausführung deutlich verschiedene Überdeckungswerte an. Zwar lassen sich diese Unterschiede durch die z. T. verschiedenen Techniken der GBT-Werkzeuge erklären, aber es gibt auch keine „Referenz“, an der sich die Werkzeughersteller orientieren könnten.

1.2 Zielsetzung und Lösungsansatz

Es ist das Ziel dieser Arbeit ein solches Referenzmodell zu liefern. Dieses Ziel lässt sich wie folgt formulieren:

> Ziel ist die Entwicklung eines programmiersprachenneutralen Modells, das eine einfache und präzise Abbildung der gängigen Programmiersprachen ermöglicht. Das Modell soll die weitgehend standardisierten Kontrollstrukturen wie z. B. Entscheidung oder Schleife enthalten. Ebenso soll die Ausnahmebehandlung berücksichtigt werden, sowie die Ausdrücke, die Auswirkung auf den GBT haben. Sowohl die populären kontrollflussbasierten Metriken, als auch Überdeckungsmetriken für logische und bedingte Ausdrücke sollen auf Grundlage des GBT-Modells definiert werden können.

Dieses Modell wird in zwei Schritten entwickelt: Erstens durch den Entwurf einer primitiven Sprache RPR (Reduced Program Representation), die die GBT-relevanten Aspekte der realen Programmiersprachen abstrahiert. Und zweitens der Definition der Ausführungssemantik mit Petri-Netzen, den sogenannten Modellnetzen. Die Modellnetze sollen auch für GBT-Werkzeugimplementierungen eine präzise Spezifikation liefern, die beschreibt, wie die zur Metrikenberechnung wichtigen Ausführungszähler in das Original-

Programm eingewoben werden sollen. Auf der Grundlage der Modellnetze erfolgt dann die präzise Definition der populären GBT-Metriken sowie weiterer neuer Metriken.

1.3 Übersicht und Gliederung

In Kapitel 2 werden zunächst die wesentlichen Grundbegriffe auf dem Gebiet des Tests definiert. Ausgehend von den Grundbegriffen wird der Test im Kontext des typischen Projektverlaufs für die verschiedenen Teststufen und der schrittweise steigenden Komplexität des Prüflings beschrieben. Hierbei wird insbesondere der Aspekt des GBT betrachtet, der in den verschiedenen Teststufen unterschiedlichen Rahmenbedingungen unterliegt und auch unterschiedliche Ziele verfolgt. Da sich die vorliegende Arbeit im Kern mit einem neuen Modell des GBT beschäftigt, folgt eine Einführung in die Modelltheorie, und es werden die GBT-Überdeckungsmetriken aus der Literatur zusammengefasst. Da die Relevanz der GBT-Metriken die Wirksamkeit des GBT voraussetzt, wird eine Reihe von Untersuchungen hierzu vorgestellt und zusammengefasst.

Kapitel 3 befasst sich mit der Literatur zu GBT-Modellen, den GBT-Werkzeugen und dem GBT-Einsatz in der Praxis. Zunächst wird eine Zusammenfassung der GBT-Modelle der Literatur geliefert, und es werden die Schwachstellen dieser Modelle herausgearbeitet. Es folgt eine Übersicht über die aktuelle GBT-Werkzeuglandschaft. Dies findet zum einen durch eine Zusammenfassung der aktuellen Literatur hierzu statt, zum anderen werden gängige Werkzeuge anhand eines Referenzprogramms und einer Referenzausführung untersucht. In beiden Fällen zeigt sich, dass die Werkzeuge keinem einheitlichen Standard folgen und durchweg verschiedene Ergebnisse liefern. Ergänzend zu diesen technischen Aspekten folgt eine Zusammenfassung der Literatur zum praktischen Projekteinsatz von GBT-Werkzeugen. Abschließend wird eine Übersicht über die Techniken zur GBT-Protokollierung geliefert, die eine zentrale Funktion innerhalb eines GBT-Werkzeugs bildet und über Möglichkeiten des GBT-Werkzeugs zur Metrikenerhebung entscheidet.

In den Kapiteln 4 und 5 wird ein neues GBT-Referenzmodell entwickelt, das deutliche Vorteile gegenüber den in Kapitel 3 beschriebenen GBT-Modellen der Literatur hat. Zunächst werden die Anforderungen an ein solches Modell zusammengefasst. Hierzu werden im Wesentlichen zwei Quellen genutzt: Die Überdeckungsmetriken, die auf Grundlage des Modells definiert werden sollen, sowie Richtlinien zur Zertifizierung sicherheitskritischer Software. Das GBT-Modell dieser Arbeit wird dann aus zwei Perspektiven entwickelt: Zum einen wird in Kapitel 4 eine Modellsprache, die Reduced Program Representation (RPR), definiert, die entsprechend den aufgestellten Anforderungen die GBT-relevanten Aspekte der realen Sprachen abbildet, die irrelevanten dagegen präteriert. Als zweite Perspektive wird in Kapitel 5 ein Ablaufmodell definiert, das präzise beschreibt, wie die genaue Ausführungssemantik eines GBT-Modellprogramms zu verstehen ist. Die Beschreibung einer Übertragung von Programmen der Programmiersprache Java in das GBT-Modell schließt dieses Kapitel ab.

Auf Grundlage des GBT-Referenzmodells werden dann in Kapitel 6 die populären GBT-Überdeckungsmetriken sowie weitere neu entwickelte Überdeckungsmetriken präzise definiert. Hierzu wird zunächst eine allgemeine Definition für Überdeckungsmetrik

und Überdeckung geliefert. Danach wird ein Regelwerk aufgestellt, mit dem Überdeckungsmetriken auf Plausibilität, Differenziertheit und Vergleichbarkeit geprüft werden können. Anschließend werden die verschiedenen Überdeckungsmetriken definiert und anhand des Regelwerks geprüft.

In Kapitel 7 werden prinzipielle Anforderungen an ein GBT-Werkzeug, die sich aus den Kapiteln 4, 5 und 6 ergeben, zusammengefasst. Anschließend wird das GBT-Werkzeug CodeCover vorgestellt, das in weiten Teilen eine Implementierung des Referenzmodells dieser Arbeit bildet. Es erfolgt auch ein Abgleich, in wie weit CodeCover die aufgestellten Anforderungen an ein GBT-Werkzeug erfüllt.

Ein neues, werkzeuggestütztes Verfahren zum Testfallentwurf wird in Kapitel 8 vorgestellt. Das Verfahren basiert auf dem in Kapitel 4 und 5 vorgestellten Modell und nutzt Erkenntnisse zur Fehlerprognose aus der Literatur. Als Resultat des Verfahrens werden dem Tester sogenannte Testfall-Hinweise offeriert, aus denen er systematisch neue und fehlersensitive Testfälle entwickeln kann. Einen wichtigen Aspekt dieses Verfahrens bilden auch Priorisierungen dieser vorgeschlagenen Testfall-Hinweise. Abschließend wird die Umsetzung der werkzeuggestützten Teile des Verfahrens im Werkzeug CodeCover beschrieben.

Abschließend werden in Kapitel 9 die Ergebnisse dieser Arbeit zusammengefasst, und es wird ein Ausblick auf weitere aufbauende Arbeiten geliefert.

2
Grundlagen und Begriffe

In diesem Kapitel werden die für diese Arbeit wichtigen Begriffe definiert. Beginnend mit den Grundbegriffen zur Qualitätssicherung und zum Test werden die Teststufen mit ihrem Bezug zum Glass-Box-Test beschrieben. Es folgen Grundlagen zu Modellen und den Überdeckungsmetriken

2.1 Grundbegriffe

2.1.1 Software-Qualitätssicherung

Nach [Li02] stellt die Software-Qualitätssicherung Techniken zur Erreichung gewünschter Ausprägungsgrade der Qualitätsmerkmale von Software-Systemen zur Verfügung, wobei die Qualitätsmerkmale sich nach [ISO 9126] in Funktionalität, Zuverlässigkeit, Benutzbarkeit, Effizienz, Änderbarkeit sowie Übertragbarkeit untergliedern lassen. Nach [LL10] hat die Software-Qualitätssicherung (QS) die Aufgabe, alle qualitätsrelevanten Aktivitäten und Prozesse zu gestalten, zu organisieren, abzustimmen und zu überwachen.

Def. **quality assurance**. *(1) A planned and systematic pattern of all actions necessary to provide adequate confidence that an item or product conforms to established technical requirements. (2) A set of activities designed to evaluate the process by which products are developed or manufactured. Contrast with: quality control (1). [IEEE610]*

In [LL10] werden neben den Prüfungen auch organisatorische und konstruktive Maßnahmen zur Software-Qualitätssicherung gerechnet. Diese organisatorischen und konstruktiven Maßnahmen finden im Gegensatz zu den Prüfungen präventiv statt und zielen auf allgemeine Prozessverbesserung und nicht auf die Prüfung eines einzelnen Artefakts.

Organisatorische Maßnahmen zur Qualitätssicherung sind beispielweise die Regelung von Zuständigkeiten, die Festlegung von Richtlinien, die Einführung von Standards, die Bereitstellung von Checklisten sowie die Qualifizierung der Mitarbeiter. Konstruktive Maßnahmen zielen nach [LL10] darauf ab, Probleme zu vermeiden. Der Einsatz geeigneter Methoden, Sprachen und Werkzeuge sowie die Schulung von Mitarbeitern und die Verwendung bestimmter Prozessmodelle werden als konstruktive Maßnahmen genannt. Die Einführung von Prozessreifegradmodellen wie z. B. [CMM, CMMI, ISO15504] ist in

aller Regel von organisatorischen und konstruktiven Maßnahmen zur QS geprägt. Bei der analytischen QS wird der Prüfling (Teil-, Zwischen- oder Endprodukt) auf Fehler hin untersucht, die Qualität des Prüfgegenstands wird bewertet oder es wird geprüft, ob ein Prüfgegenstand bestimmte vorgegebene Qualitätskriterien erfüllt. Die Prüfungen gliedern sich in die nicht automatisierbaren Prüfungen wie beispielweise Inspektionen oder Reviews und in automatisierbare Prüfungen am Rechner. Dabei haben die nichtmechanischen Prüfungen, die durch Menschen vorgenommen werden, trotz der Aufwände erhebliche Vorteile. So können diese Prüfungen auch für nicht formale Dokumente wie z. B. Anforderungsspezifikationen erfolgen. Hampp liefert in [Ham10] einen umfangreichen Überblick über diese nichtautomatisierbaren Prüfungen.

Abbildung 1: Übersicht über die Prüfverfahren nach [LL10]

Statische Prüfungen am Rechner werden in der Regel nur zur Prüfung formaler Dokumente eingesetzt. Bei Programmcode sind Architekturanalysen sowie Konformitätsanalysen hinsichtlich vorgegebener Programmierrichtlinien möglich. Ein in der Industrie verbreiteter Vertreter dieser Programmierrichtlinien ist der sogenannte MISRA-Standard [MISRA], der bei der Softwareentwicklung im Automobilbereich eine zentrale Rolle spielt. Die statische Prüfung kann Verstöße gegen diese Richtlinie anzeigen. Zudem kann ein Programm auf sogenannte Anomalien (wie z. B. Datenflussanomalien, [SL04]) hin untersucht werden. Mit der statischen Analyse können auch Code-Metriken erhoben und mit vorgegebenen Grenzwerten verglichen werden. Alle diese statischen Prüfungen haben gemeinsam, dass der Prüfgegenstand gelesen, aber nicht ausgeführt wird. Die Prüfungen, in denen der Prüfling ausgeführt wird, werden als dynamische Prüfung oder als Test bezeichnet.

2.1.2 Testen

Nach [My79] und [LL10] ist Testen die – auch mehrfache – Ausführung eines Programms auf einem Rechner mit dem Ziel, Fehler zu finden. Ergänzend ist nach [LL10] ein systematischer Test ein Test, bei dem die Randbedingungen definiert und präzise erfasst sind, die Eingabedaten systematisch ausgewählt werden und die Ergebnisse dokumentiert und nach Kriterien beurteilt werden, die vor dem Test festgelegt wurden.

Def. **test**. *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, [IEEE610].*

Grundsätzlich ist Testen immer eine Stichprobenprüfung, da der vollständige verifizierende Test bei Programmen der Industrie völlig ausgeschlossen ist. Bereits bei kleinen Programmen mit wenigen Integer-Variablen wären für einen vollständigen Test so viele Wertekombinationen zu testen, dass er eine astronomische Anzahl an Testfällen und damit eine nicht praktikable Testdauer erfordern würde. Damit müssen in der Praxis einige wenige Testfälle – ein verschwindend geringer Teil der theoretisch möglichen Testfälle – geschickt gewählt werden, um dennoch möglichst viele Fehler des Programms aufzudecken. Dijkstra hat dies in den berühmten Satz zusammengefasst:

Program testing can be used to show the presence of bugs, but never show their absence!
E.W. Dijkstra (1970)

So wird „to show the presence of bugs“ das zentrale Thema beim Testen, das in dem Satz von Myers zusammengefasst wird:

Testing is the process of executing a program or system with the intent of finding errors.
G. Myers (1979)

Misslingt nun dieses „Fehler entdecken“ trotz großer Anstrengungen, begründet sich nach Hetzel das Vertrauen darin, dass der Prüfling das leistet, was er leisten soll. So definiert Hetzel Testen als

Testing is the process of establishing confidence that a program or system does what it is supposed to.
B. Hetzel (1973)

Und Grimm stellt in [Gr95] dazu fest, dass Testen in der Praxis das einzige Verfahren ist, mit dem die realen Einsatzbedingungen eines Software-Systems angemessen berücksichtigt und die dynamischen Eigenschaften geprüft werden können. So liegt Testen in einem Spannungsfeld, weil einerseits durch den Test kein Nachweis der Korrektheit des Prüflings möglich ist, andererseits aber keine andere Prüftechnik in der Lage ist, das Testen auch nur annähernd zu ersetzen.

2.1.3 Black-Box-Test

Beim sogenannten Black-Box-Test (BBT) betrachtet man nach [LL10] das Programm als Monolithen, über dessen innere Beschaffenheit man nichts weiß und nichts wissen muss. Man prüft, ob das Programm das tut, was die Spezifikation verlangt. In der Literatur wird übereinstimmend festgestellt, dass der Black-Box-Test die wichtigste Form des Tests ist. Auch sind für den Black-Box-Test in der Literatur viele Techniken zum Testfallentwurf und zur Testfalldokumentation beschrieben. Eine Zusammenfassung hierzu folgt in Kapitel 2.6.1. Neben dem Begriff *Black-Box-Test* werden in der Literatur auch die Begriffe *Funktionstest* (functional testing) [IEEE610] oder *Spezifikationsbasierter Test* [ISTQB] verwendet.

Def. **functional testing**. *(1) Testing that ignores the internal mechanism of a system or component and focuses solely on the outputs generated in response to selected inputs and execution conditions. Syn: black box testing. Contrast with: structural testing. (2) Testing conducted to evaluate the compliance of a system or component with specified functional requirements. See also: performance testing. [IEEE610]*

2.1.4 Glass-Box-Test

Nach [LL10] lässt das Wort „Glass Box“ erkennen, dass der Programmcode bei dieser Art des Tests sichtbar ist. Als Glass-Box-Test (GBT) wird demnach ein Test bezeichnet, wenn beim Testen zusätzlich beobachtet wird, wie weit das Programm ausgeführt wird. Diesen Anteil des ausgeführten Programms, bezogen auf das gesamte Programm, bezeichnet man dabei als GBT-Überdeckung. Nach [LL10] geht es in jedem Glass-Box-Test darum, eine bestimmte auf den Programmcode bezogene Überdeckung zu erreichen. Das Kalkül des GBT ist einfach: Ein Defekt, der sich in einer Anweisung oder in einem Ausdruck befindet, kann nur gefunden werden, wenn die Anweisung oder der Ausdruck ausgeführt wird. Die einzige Chance, einen Defekt beim Testen zu finden, besteht somit darin, Eingabedaten zu wählen, die dazu führen, dass die Anweisung oder der Ausdruck ausgeführt wird. Damit führt eine höhere Überdeckung statistisch zur Entdeckung von mehr Defekten, und, wenn diese behoben sind, zu besserer Produktqualität. Andererseits gibt es natürlich (von sehr wenigen Defekttypen abgesehen) keinerlei Sicherheit, dass mit Ausführen einer Anweisung oder eines Ausdrucks der Fehler tatsächlich angezeigt, d. h. eine Abweichung zum Soll-Resultat erkannt wird. Ein Beispiel hierfür ist ein Ausdruck, der bei bestimmten Eingaben zu einer Division durch null (d. h. zu einem Fehlverhalten) führt. Solange genau diese Eingaben nicht gewählt werden, wird der Prüfling die Fehlersymptome des Defekts nicht zeigen, und es wird kein Fehlverhalten vorliegen. Ntafos fasst dies in [Na88] wie folgt zusammen:

Another problem is that most structural strategies do not provide any guidelines for selecting test data from within a path domain, and many errors along a path can be detected only if the path is executed with values from a small subset of its domain.

Gleichwohl ist sich die Literatur einig, dass der GBT eine wirksame und wirtschaftliche Ergänzung zum gründlichen Funktionstest liefert.

Def. **structural testing.** *Testing that takes into account the internal mechanism of a system or component. Types include branch testing, path testing, statement testing. Syn: glassbox testing; white-box testing. Contrast with: functional testing. [IEEE610]*

Für den Glass-Box-Test ist in jedem Fall ein Werkzeug erforderlich, das während der Testausführung die Ausführung der einzelnen Programmelemente protokolliert und daraus die Überdeckung berechnet. In der Regel werden in den Prüfling zusätzliche Anweisungen als Messpunkte eingewoben, die die Ausführungen einzelner Elemente des Programms zählen. In Abschnitt 3.3 wird dies ausführlich behandelt. Die Darstellung der GBT-Resultate kann für den Tester schließlich in anschaulicher Form erfolgen: So werden beispielsweise die Teile des Programms, die nicht ausgeführt sind, in einer speziellen Farbe hervorgehoben.

Beim Glass-Box-Test lassen sich zwei prinzipielle Verwendungs-Strategien abgrenzen: Erstens die Verwendung der Überdeckung als Testendekriterim. Nach [LL10] kann beispielsweise das Testziel sein, eine Anweisungsüberdeckung von 80 % zu erreichen. Dann wir getestet, bis 80 % der ausführbaren Anweisungen im Programm mindestens einmal ausgeführt sind. Auch die Vorgaben wichtiger Industriestandards zur Zertifizierung sicherheitskritischer Software sind als Testendekriterium zu verstehen (z. B. [RTCA92, IEC61508]). Sie verlangen in der Regel den Nachweis einer vollständigen (oder sehr hohen) Überdeckung.

Zweitens kann der GBT als Testfallentwurfsstrategie verwendet werden. So wird beispielsweise im Buch von Myers [My79] der GBT auch im Kapitel „Testfallentwurf“ behandelt. Aus Programmverzweigungen und logischen Ausdrücken leitet er Eingabedaten für Testfälle ab, die zu einer vollständigen GBT-Überdeckung führen. Eine ausführliche Beschreibung des GBT als Testfallentwurfsstrategie folgt in Abschnitt 2.6.2 auf Seite 29.

2.1.5 Fehler

Der Begriff „Fehler“ wird im allgemeinen Sprachgebrauch (aber auch in der Softwaretechnik) für prinzipiell verschiedene Sachverhalte genutzt. Die folgende Definition benennt diese verschiedenen Bedeutungen:

Def. **error**. *(1) The difference between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition. For example, a difference of 30 meters between a computed result and the correct result. (2) An incorrect step, process, or data definition. For example, an incorrect instruction in a computer program. (3) An incorrect result. For example, a computed result of 12 when the correct result is 10. (4) A human action that produces an incorrect result. For example, an incorrect action on the part of a programmer or operator. Note: While all four definitions are commonly used, one distinction assigns definition 1 to the word “error,” definition 2 to the word “fault,” definition 3 to the word “failure,” and definition 4 to the word “mistake.” See also: dynamic error; fatal error; indigenous error; semantic error; syntactic error; static error; transient error. [IEEE610]*

Oft ist aus dem Kontext heraus klar, welche Bedeutung das Wort „Fehler“ jeweils haben soll. Um Missverständnisse zu vermeiden, ist es ratsam, konsistent die eindeutigen Begriffe gemäß der Definition zu verwenden. Die deutschen Begriffe nach [ISTQB] sind in Tabelle 1 angegeben. Die drei Verwendungsmöglichkeiten des Begriffs Fehler lassen sich damit auch übersichtlich auf die drei Bedeutungen zusammenfassen:

- eine Person macht einen Fehler (Definition (4), mistake),
- das Programm enthält einen Fehler (Definition (2), fault) und
- die Programmausführung zeigt einen Fehler (Definition (3), failure)

Die drei Bedeutungen von „Fehler“ nach Tabelle 1 erfordern jeweils völlig unterschiedliche Maßnahmen um den „Fehler“ zu erkennen bzw. zu vermeiden. Organisatorische und konstruktive QS-Maßnahmen zielen auf die Vermeidung von Fehlhandlungen, die nichtmechanischen Prüfungen zielen auf das Erkennen der Defekte, und der Test zielt auf die Erkennung und Vermeidung des Programmfehlverhaltens.

Begriff	Definition
Mistake (**Fehlhandlung**, Versagen, Irrtum) Definition (4) nach [IEEE610]	**Eine Person macht einen Fehler**: Fehlerhafte Aktion einer Person (Irrtum), die zu einer fehlerhaften Programmstelle führt.
Fault, Defect, Bug (**Defekt**, Fehlerursache) Definition (2) nach [IEEE610]	**Der Programmcode enthält einen Fehler:** Fehlerhafte Stelle (z. B. eine Zeile) eines Programms, die eine Fehlwirkung auslösen kann.
Failure (Fehlerwirkung, Mangel) Definition (3) nach [IEEE610]	**Die Programmausführung zeigt einen Fehler:** Fehlverhalten eines Programms gegenüber der Spezifikation, das während seiner Ausführung (tatsächlich) auftritt.

Tabelle 1: Verwendungsbereiche des Worts „Fehler“

2.1.6 Testfall

Testfälle spezifizieren für einen Prüfling Eingabedaten mit den zugehörigen erwarteten Ausgaben – den Soll-Resultaten. Die Eingabedaten steuern die Programmausführung, und ein Fehler (oder eine Fehlerwirkung) wird dann angezeigt, wenn die tatsächlichen Ausgaben – die Ist-Resultate – von den spezifizierten Soll-Resultaten abweichen. Nach [IEEE610] umfasst ein Testfall die Eingabedaten, die für die Ausführung notwendigen Vorbedingungen, die Soll-Resultate sowie weitere erwartete Nachbedingungen. Während zur Auswahl fehlersensitiver Eingabedaten Kreativität und Intuition erforderlich ist, müssen die Soll-Resultate für die ausgewählten Eingabedaten sorgfältig aus der Anforderungsspezifikation abgeleitet werden.

Def. **test case.** *A set of input values, execution preconditions, expected results and execution postconditions, developed for a particular objective or test condition, such as to exercise a particular program path or to verify compliance with a specific requirement. [IEEE610]*

Def. ***test suite.*** *A set of several test cases for a component or system under test, where the post condition of one test is often used as the precondition for the next one. [ISTQB]*

Def. Die **Testausführung** *ist die Ausführung aller Testfälle der Testsuite eines Programms.*

Def. Die **Testfallausführung** *ist die Ausführung eines einzelnen Testfalls.*

Nach [LL10] ist ein Testfall gut, wenn er mit hoher Wahrscheinlichkeit einen noch nicht entdeckten Fehler anzeigt. Ein idealer Testfall ist demnach

- repräsentativ, d. h., er steht stellvertretend für viele mögliche Testfälle,
- fehlersensitiv, d. h., er hat nach der Fehlertheorie eine hohe Wahrscheinlichkeit, einen Fehler anzuzeigen,
- redundanzarm, d. h., er prüft nicht, was auch andere Testfälle schon prüfen.

Ausführliche Empfehlungen zur Struktur, in der die Testfälle beschrieben werden, sind in [IEEE829] enthalten. Dort wird auch die Zusammenfassung der Testfälle in eine Testsuite empfohlen.

2.2 Testprozess

2.2.1 Das V-Modell

Viele einschlägige Industriestandards (wie z. B. [ISO15504, IEC61508, VMXT04]) beschreiben den Test im Rahmen eines V-förmig angelegten Entwicklungsmodells, das auf Arbeiten von Boehm [Boe79] zurückgeht. Obwohl sich die Modelle alle geringfügig unterscheiden, sind sie in der Grundstruktur gleich: Die linke Flanke des „V“ enthält die verfeinernden Aktivitäten wie Anforderungsanalyse, Entwurf und Implementierung.

Die rechte Flanke des „V“ enthält die integrierenden Aktivitäten mit den sogenannten Teststufen. Je Teststufe werden die im korrespondierenden verfeinernden Prozessabschnitt erstellten Artefakte gegen die dem Artefakt zugrundeliegende Spezifikation validiert. Damit wird z. B. im Komponententest die implementierte Komponente gegen die der Komponente zugrundeliegende Spezifikation geprüft. Entsprechendes gilt für den Integrationstest, bis schließlich beim System- und Abnahmetest das System gegen die Anforderungsspezifikation des Systems geprüft wird. Neben den genannten V-Modellen sind zahlreiche Varianten in der Literatur und auch als Unternehmensstandards zu finden. In der Test-Literatur (z. B. [SL04]) spielt das V-Modell nach [ISTQB] eine große Rolle. Dieses Modell ist in Abbildung 2 dargestellt. Die gestrichelt dargestellten Pfeile entsprechen Prüfungen, ob die Ergebnisse eines Entwicklungsschritts die Vorgaben der Eingangsdokumente erfüllen. So wird z. B. nach Fertigstellung der Anforderungsdefinition diese daraufhin geprüft, ob sie die vorab festgelegten Vorgaben erfüllt. Diese Prüfungen

werden auch als Verifikation bezeichnet. Die Verifikation kann bei formalen Dokumenten mit formalen Eingangsdokumenten als formale Verifikation und damit als Korrektheitsnachweis stattfinden. Testen validiert für spezielle Datenpunkte den Prüfling gegen die Spezifikation, kann aber praktisch nie einen Korrektheitsnachweis erbringen.

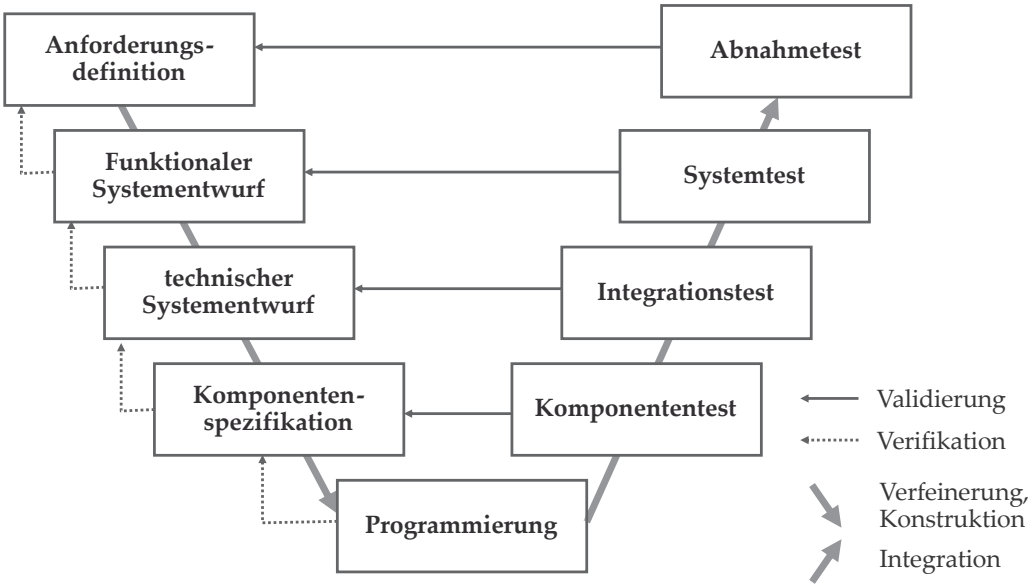


Abbildung 2: V-Modell nach ISTQB [ISTQB, SL04]

Def. **verification**: *(1) The process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase. (2) Formal proof of program correctness. („Are we doing the thing right?") [IEEE610]*

Def. **validation**: *The process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements. („Are we doing the right thing?") [IEEE610]*

2.2.2 Testaktivitäten

Der Testprozess ist nicht eine große gekapselte Aktivität, sondern besteht aus vielen Einzelaktivitäten. Die Aktivitäten im Programmtest bestehen nach [LL10, FLS07, IEEE829, Gri95, SL04] im Wesentlichen aus den nacheinander stattfindenden Teilaktivitäten Testvorbereitung (Testspezifikation), Testdurchführung und Testauswertung. Abbildung 3 zeigt diese Aktivitäten und ihr Zusammenwirken.

Def. **test process**. *The fundamental test process comprises planning, specification, execution, recording, checking for completion and test closure activities. [ISTQB]*

Begleitet werden die Testaktivitäten von einem Testmanagement (oder nach [ISTQB] der Testplanung), das Ziele, Budget, Zeitrahmen und Organisationsstruktur festlegt. Detail-

liert sind diese Management-Aktivitäten in [IEE829] sowie [SL04] beschrieben. Der Testprozess endet mit der Dokumentation der festgestellten Fehler; die Fehleranalyse und Fehlerbehebung zählen nicht mehr zum Testprozess hinzu. In der Praxis ist es auch üblich, Fehler in einem sogenannten Fehlerverfolgungssystem (Bug-Tracking-System) zu erfassen. Neben der genauen Beschreibung der Fehlersymptome werden die genaue Testkonfiguration sowie der Testfall erfasst, der den Fehler angezeigt hat. [IEEE829] und [SL04] enthalten eine ausführliche Übersicht über die Angaben einer solchen Fehlererfassung.

In [LL10] wird der Testfallentwurf als die eigentliche intellektuelle Leistung des Testens bezeichnet. Als wichtigster und umfangreichster Teil der Vorbereitung wird die Auswahl und Spezifikation der Testfälle genannt. Ohne Frage ist für diese Tätigkeit ein besonderes Maß an Kenntnis der Spezifikation und der Anwendungsdomäne erforderlich. Unterstützend liefert die Literatur eine Vielzahl an Verfahren, um den Testfallentwurf zu systematisieren. In Kapitel 2.6 werden diese Verfahren ausführlich beschrieben.

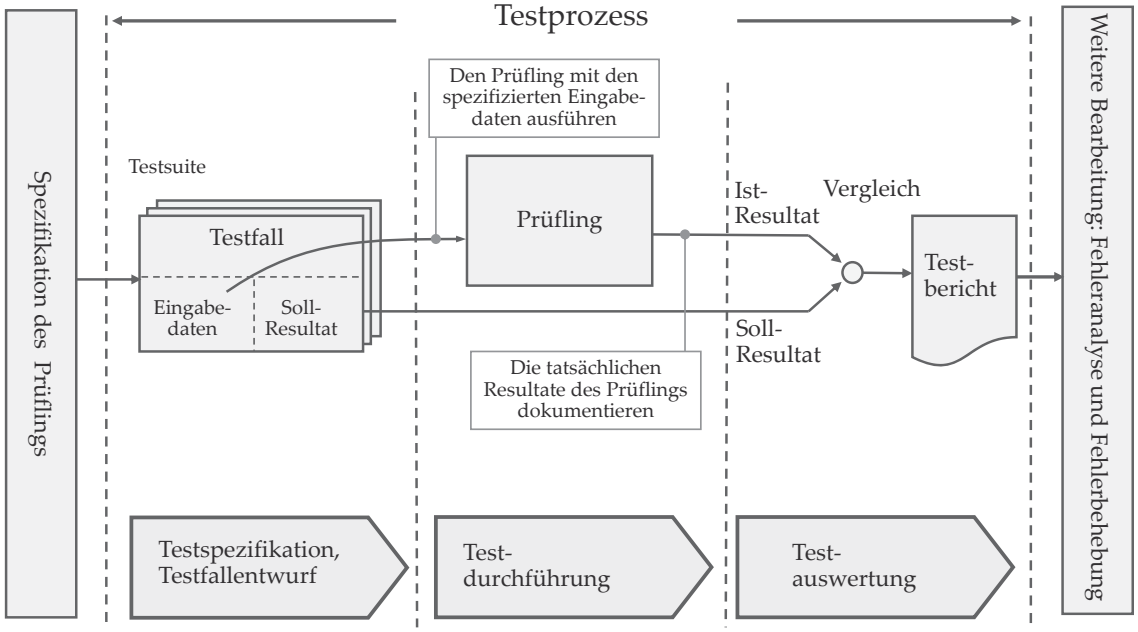


Abbildung 3: Testprozess

2.2.3 Wirtschaftlichkeitsaspekte

Die Wirtschaftlichkeit des Testens wird durch zwei Faktoren bestimmt: Die Kosten, um einen Fehler im Test zu finden (die Testkosten), und die Fehlerkosten, die dadurch entstehen, dass der Fehler unentdeckt bleibt und so lange Schaden anrichtet, bis er in einer späteren Phase, im Betrieb oder gar nicht entdeckt wird. In [LL10] wird daher empfohlen, eine Kostengrenze zu definieren, die das Testende steuert. Überschreiten die durchschnittlichen Testkosten für die Entdeckung eines Fehlers eine vorab festgelegte Kostengrenze, wird der Test beendet. Die Kostengrenze wird dabei nach [SL04] abhängig von der Kritikalität des Projekts und weiteren Faktoren (wie z. B. festgesetzte Vertragsstrafen oder spezielle Fehlerfolgekosten) festgelegt, die die Fehlerkosten beeinflussen.

Amland behandelt in [Am00] ausführlich solche Risikofaktoren und empfiehlt, den Test an diesen Risiken auszurichten. In der Praxis zeigt sich, dass die Testkosten pro entdecktem Fehler über längere Zeit nahezu konstant bleiben und danach deutlich anwachsen. Dementsprechend werden zu Beginn des Tests deutlich mehr Fehler gefunden als in einem späteren Testabschnitt.

Einen ausführlichen Bericht zur Wirtschaftlichkeitsbetrachtung des GBT befindet sich in [EBI06]. Die Autoren geben für ein untersuchtes Projekt auch die Testkosten für Testfälle des logikbasierten GBT mit 8,4 Aufwandstagen pro Fehler an. Die Kostengrenze des Projekts liegt nach Aussage der Autoren deutlich über diesem Wert; das getestete Programm stammt aus dem Bereich der Raumfahrt.

2.3 Klassifikation nach Teststufen

Die Aktivitäten des Tests lassen sich nach ganz unterschiedlichen Merkmalen klassifizieren. [LL10] nennt die folgenden Klassifikationen:

- Klassifikation nach den Grundlagen zur Testfallerstellung: Im Wesentlichen erfolgt hier eine Trennung in den Black-Box- und in den Glass-Box-Test (vgl. Kapitel 2.6). Diese Klassifikation spielt auch in dieser Arbeit eine große Rolle.
- Klassifikation nach dem Aufwand für Vorbereitung und Archivierung: Es erfolgt hier eine Untergliederung in Laufversuch, Wegwerftest und systematischen Test. Da in dieser Arbeit ausschließlich auf den systematischen Test Bezug genommen wird, wird diese Klassifikation nicht weiter behandelt.
- Klassifikation nach der Komplexität des Prüflings. Hier handelt es sich um die am häufigsten in der Literatur verwendete Klassifikation [ISTQB, SL04]. Der Prüfling wird schrittweise integriert und nimmt so eine immer höhere Komplexität an. [LL10] nennt Einzeltest, Modultest, Integrationstest und Systemtest. Diese Klassifikation wird nach [ISTQB] auch als Klassifikation nach Teststufen bezeichnet und wird im Folgenden ausführlich behandelt.
- Klassifikation nach den getesteten Eigenschaften. Im Einzelnen wird Funktionstest, Installationstest, Wiederinbetriebnahmetest, Verfügbarkeitstest, Last- und Stresstest, Regressionstest genannt. Damit wird dem Rechnung getragen, dass jede testbare Anforderung – und damit nicht nur die funktionalen Anforderungen – durch mindestens einen Testfall abgedeckt werden.
- Klassifikation nach beteiligten Rollen: Alpha- und Beta-Test und Abnahmetest.

In der Literatur (z. B. [Ost07]) findet man häufiger noch die Klassifikation nach Entwicklungsphasen. Faktisch wird diese Klassifikation mit der Klassifikation nach Teststufe oder Komplexität des Prüflings in weiten Teilen übereinstimmen. Die Klassifikation nach Teststufen (oder Komplexität des Prüflings) hat, wie im Folgenden noch dargestellt wird, wegen der praktikableren Abgrenzung aber Vorteile.

Durchläuft man den Entwicklungsprozess nach Abbildung 2 in chronologischer Reihenfolge, beginnt der Test nach der Implementierung mit dem Modultest, es folgen der Integrationstest und der Test des Gesamtsystems. Natürlich ist die Testdurchführung immer erst möglich, wenn der Prüfling zur Verfügung steht. So kann beispielsweise der

Test eines Moduls erst dann durchgeführt werden, wenn das Modul von der Entwicklung abgeschlossen und zum Test freigegeben ist. Dagegen können viele andere Arbeiten des Tests wie z. B. die Testplanung, der Testfallentwurf oder die Bereitstellung des Testgeschirrs bereits früher begonnen werden. Der Testfallentwurf kann beispielsweise unmittelbar mit Abschluss der Spezifikation des Prüflings beginnen. In der Literatur wird auch übereinstimmend empfohlen, diese vorbereitenden Arbeiten so früh wie möglich zu beginnen. So bildet der Testfallentwurf auch eine exzellente Prüfung der Spezifikation – sowohl der Anforderungsspezifikation wie auch der Modulspezifikation. Diese vorbereitenden oder nachbereitenden Arbeiten, die zeitlich nicht in der jeweiligen Testphase des Entwicklungsprozess liegen, werden in der sogenannten Teststufe mit den Aktivitäten der Testphase zusammengefasst. Da eine vollständige Darstellung der Teststufen den Rahmen dieser Arbeit bei Weitem sprengen würde, werden im Wesentlichen nur die für den GBT relevanten Eigenschaften der Teststufen im Folgenden behandelt. Eine ausführliche Behandlung der Teststufen befindet sich in [SL04, LL10].

2.3.1 Modultest

Die Begriffe Modul und Modultest werden in der Literatur nicht einheitlich definiert. Liggesmeyer versteht in [Li02] unter einem Modul die kleinste sinnvoll unabhängig testbare Einheit eines Programms. In [LL10] bildet ein Modul einen Verbund mehrerer zusammenhängender Einheiten, die gemeinsam eine Funktion bilden. Ausdrücklich wird der Modultest vom sogenannten Unit-Test abgegrenzt, wobei die Unit im technischen Sinne die kleinste testbare Einheit bildet. Eine Unit ist demnach eine Funktion, ein Unterprogramm oder auch eine Klasse. Ein Modul umfasst nach [LL10] mehrere dieser Units. Der Modultest ist in jedem Fall stark vom Begriff des Unit-Tests beeinflusst, und beide Begriffe werden oft auch als Synonym benutzt. Im IEEE Standard [IEEE610] werden Modul- und Unit-Test zwar nicht als Synonyme genannt, haben aber genau die gleiche Definition. Für diese Arbeit ist der genaue Zuschnitt des Moduls nicht wichtig, wichtig ist lediglich ein für den Test vom Gesamtsystem isolierbarer Betrieb des Moduls. Der Begriff Unit wird in dieser Arbeit im Folgenden als Synonym für Modul genutzt.

Def. **Modul**. *Abgrenzbare Einheit des Gesamtsystems, die isoliert betrieben und getestet werden kann. Synonyme: Unit, Subsystem, Komponente.*

Def. **Modultest**. *Test eines vom restlichen Gesamtsystem isolierten Moduls unter Verwendung von Testgeschirr, das die Eingabedaten liefert und die Ausgabedaten prüft [EBI06]. Synonyme: Unit-Test, Komponententest*

Faktisch ist der Begriff Modul- oder Unit-Test heute mit dem Einsatz der Unit-Test-Werkzeuge wie z. B. JUnit [Be03, JUnit] gleichzusetzen. Unit-Test-Werkzeuge unterstützen den Test einzelner unabhängiger Methoden einer Klasse, die selbst möglichst keine oder sehr wenige Abhängigkeiten von anderen Teilen des Gesamtsystems besitzen. Liggesmeyer bezeichnet den Modultest nach seinem Verständnis auch als „Testen im Kleinen“. Damit impliziert er einige prägnante Merkmale des Modul- oder Unit-Tests: Der