1 Einleitung

Fast jeder fortgeschrittene Programmierer hat ein oder mehrere Bücher über Programmiersprachen und Software-Entwicklung gelesen. Bücher über das Testen von Software stehen aber weit weniger oft in den Regalen. Das liegt vermutlich unter anderem am hartnäckigen Gerücht, dass Testen von Software langweilig sei. Um diesem Gerücht entschieden entgegenzutreten, ist dieses Kapitel geschrieben worden. Es soll ein Appetitanreger auf die im Buch behandelten Testthemen sein, beschreibt die Eingliederung des Tests in den Software-Entwicklungsprozess und analysiert zunächst einmal den Feind: den Software-Fehler. Doch zuvor noch ein paar Worte zur Motivation des Software-Tests, zur Eingliederung dieses Buchs in andere Literatur und ein paar Definitionen.

1.1 Motivation

Fehler in der Software eingebetteter Systeme können teure Rückholaktionen zur Folge haben. Selten ist es den Anbietern von eingebetteten Systemen möglich, einfach einen Bugfix per E-Mail zu verschicken und wieder dem Tagesgeschäft nachzugehen. Daher sollte man in Branchen mit hoher Anforderung an die Software-Integrität erstens besonders bedacht sein, Software-Fehler zu vermeiden, und zweitens, gemachte Fehler zu erkennen. Punkt eins zielt auf die Prozesslandschaft, die Qualifikation der Mitarbeiter und die Unternehmensstruktur ab. Er betrachtet also, wie sehr das Talent oder – viel wichtiger – das fehlende Talent eines Mitarbeiters die Qualität des Produkts beeinflussen kann. Punkt zwei heißt, die Software und begleitende Dokumente sorgfältig zu verifizieren. Solche Dokumente können zum Beispiel die Anforderungsdefinition, Analysen der zu erwartenden CPU-Last oder das Design festhalten.

1.2 Abgrenzung des Buchs zu ISTQB-Lehrplänen

Leider wurde lange Zeit nicht nur im deutschsprachigen Raum das Thema Verifikation von Software, wozu auch Testen gehört, wenig an Hochschulen gelehrt. Eine Konsequenz daraus ist, dass Vertreter der Industrie das International Soft-

ware Testing Qualifications Board (ISTQB) ins Leben gerufen haben, das eine »Certified Tester«-Ausbildung definiert. Der Inhalt des vorliegenden Buchs deckt sich *nicht* mit den Lehrplänen des ISTQB. In diesem Buch werden Themen genauer als durch das ISTQB behandelt, wenn sie für Embedded-Software besonders wichtig sind, und es werden Methoden präsentiert, die für eingebettete Software wichtig sein können, sich aber zurzeit nicht in den ISTQB-Lehrplänen befinden. Ebenso werden Themen der ISTQB-Lehrpläne hier nicht behandelt, wenn sie nicht technischer Natur sind oder nur Multisysteme oder reine Business-Anwendungen betreffen.

Das Buch ist für Personen *ohne* Vorwissen aus dem Bereich Software-Testing geschrieben. Absolventen der ISTQB-Certified-Tester-Lehrgänge werden im vorliegenden Buch daher viel Bekanntes wiederfinden und aus oben beschriebenen Gründen trotzdem ebenso viel Neues erfahren.

1.3 Zur Gliederung dieses Buchs

Die technischen Grundlagenkapitel dieses Buchs orientieren sich dabei am zeitlichen Verlauf eines Projekts. Das heißt, auch wenn das wichtigste Thema dieses Buchs der Test ist, handeln die ersten der folgenden Kapitel noch nicht vom Test, denn am Anfang eines Projekts gibt es zunächst noch keine Software, die man testen könnte. Wohl aber gibt es schon Dokumente (bei phasenorientierten Projekten) oder Vorgänge (bei agilen Vorgehensweisen), für die ein Tester einen Beitrag zur Software-Qualität leisten kann.

Kapitel 2 beschreibt, wie dieser Beitrag bei der Review von Anforderungstexten aussehen kann. Kapitel 3 beschreibt kurz die möglichen Verifikationsschritte beim Design. In den Kapiteln 4 und 5 wird beschrieben, wie Code-Reviews durchgeführt werden können und wie Werkzeuge funktionieren, die teilweise oder gänzlich automatisch Review-Aufgaben übernehmen können.

Dann erst geht es mit dem Testen los. In Kapitel 6 werden Unit-Tests, in Kapitel 7 Integrationstests und in Kapitel 8 Systemtests beschrieben. Bei Tests von Middleware können diese Teststufen alle stark verschwimmen, wie Kapitel 9 zeigt. Im Anschluss daran beschäftigt sich Kapitel 10 mit einer Art von Fehler, die nur durch Zufall in den zuvor beschriebenen Tests gefunden werden kann: Race Conditions. Das Kapitel zeigt, wie sich Data Races zuverlässig auffinden lassen, gefolgt von einem verwandten Thema: Kapitel 11 zeigt, wie sich Deadlocks automatisch auffinden lassen.

Das darauf folgende Kapitel 12 beschreibt Verfahren zur Bestimmung der maximalen Ausführungszeit von Code. Das Ergebnis kann ein wichtiger Beitrag für die Schedulability-Analyse sein, die in Kapitel 13 behandelt wird.

Spätestens nach diesem Kapitel hält sich die Reihung der weiteren Kapitel nicht mehr an den zeitlichen Verlauf eines Projekts. Die nun folgenden Kapitel sind auch weit weniger umfangreich. In Kapitel 14 wird die Hardware/Software-

Interaktionsanalyse vorgestellt, die auf Produkt/System-Ebene stattfindet. Mit einem kurzen Kapitel 15 über modellbasierten Test und einem technischen Ausblick in Kapitel 16 verlässt das Buch das technische Terrain.

In Kapitel 17 werden Hinweise für das Testmanagement gegeben und Kapitel 18 beschreibt die Aufgaben und möglichen Vorgehensweisen des Qualitätsbeauftragten. Auch diese beiden Kapitel sind bewusst kurz gehalten und geben eher nützliche Tipps, als dass man lange Abhandlungen darin findet, denn zu diesen Themen gibt es jede Menge Spezialliteratur. Das Kapitel 19 widmet sich dem Thema Haftung: In welchem Maß ist ein Programmierer oder ein Tester für Fehler haftbar? Dort erfahren Sie es.

Die meisten Kapitel schließen mit einem Fragenkatalog und Übungsaufgaben zur Kontrolle des Lernziels. Am Ende des Buchs finden Sie Lösungen dazu und Quellenverzeichnisse für referenzierte Literatur.

1.4 Die wichtigsten Begriffe kurz erklärt

1.4.1 Definition von Fachbegriffen

Die ISO 29119-1 und das ISTQB-Glossary of Terms definieren eine ganze Reihe von Fachwörtern rund ums Testen. In Einzelfällen sind sie allerdings nicht übereinstimmend. Dabei ist zu erwarten, dass das ISTQB-Glossar früher oder später an die ISO 29119-1 angepasst wird. Wenn man in einem Unternehmen Dinge rund um das Thema Software-Qualität benennen möchte, ist man gut beraten, sich an diesen Definitionen zu orientieren. Das ISTQB-Glossar kann man im WWW nachschlagen, siehe [URL: ISTQB] für die englische Fassung und [URL: ISTBQ/D] für die deutschsprachige, die ISO-Norm ist kostenpflichtig.

Es ergibt nicht viel Sinn, diese Definitionen hier zu kopieren. Stattdessen enthält die folgende Aufstellung die wichtigsten Begriffe, die im vorliegenden Buch Verwendung finden, unmittelbar gefolgt von der Definition aus dem ISTQB-Glossar und einer kurzen Ergänzung:

> *Agile Software-Entwicklung* ist eine auf iterativer und inkrementeller Entwicklung basierende Gruppe von Software-Entwicklungsmethoden, wobei sich Anforderungen und Lösungen durch die Zusammenarbeit von selbstorganisierenden funktionsübergreifenden Teams entwickeln.

Der mit großem Abstand bedeutendste Vertreter agiler Entwicklungsmodelle ist Scrum, gefolgt von Extreme Programming. Speziell in Projekten mit begrenzter Größe und nur vage definierten Anforderungen haben diese Entwicklungsmodelle ihre Berechtigung. Diese »agilen Methoden« kommen in Reinform ohne Projektleiter aus. Die Teams sind selbstorganisierend.

Audit: Ein unabhängiges Prüfen von Software-Produkten und -prozessen, um die Konformität mit Standards, Richtlinien, Spezifikationen und/oder Prozeduren basierend auf objektiven Kriterien zu bestimmen, einschließlich der Dokumente, die (1) die Gestaltung oder den Inhalt der zu erstellenden Produkte festlegen, (2) den Prozess der Erstellung der Produkte beschreiben (3) und spezifizieren, wie die Übereinstimmung mit den Standards und Richtlinien nachgewiesen beziehungsweise gemessen werden kann.

Ein Audit führt also ein externer Fachmann durch, der sich nicht nur das Produkt ansieht, sondern auch und vor allem den Entstehungsprozess des Produkts. Ein Auditor prüft, ob ein Unternehmen ein Prozessumfeld schafft, das die Wahrscheinlichkeit von guter (beziehungsweise angepasster) Software-Qualität maximiert.

Validierung ist die Bestätigung durch Bereitstellung eines objektiven Nachweises, dass die Anforderungen für einen spezifischen beabsichtigten Gebrauch oder eine spezifische beabsichtigte Anwendung erfüllt worden sind.

Wer diese Definition aus der deutschen Version des ISTQB-Software-Testing-Glossars beim ersten Durchlesen versteht, gewinnt einen Preis. Weiter auf Deutsch übersetzt könnte man sagen: Validierung bestätigt, dass das Produkt, so wie es vorliegt, seinen beabsichtigten Verwendungszweck erfüllen kann. Validierung stellt also sicher, dass »das richtige Ding erzeugt wird«. Etwas weniger streng definierte die nun abgelöste IEEE 829-2008 diesen Begriff und verstand Validierung einfach als Nachweis der Erfüllung der Anforderungen.

Verifikation ist die Bestätigung durch Bereitstellung eines objektiven Nachweises, dass festgelegte Anforderungen erfüllt worden sind.

Das IEEE Glossary of Software Engineering Terminology [IEEE 610.12] beschränkt den Begriff nicht auf Anforderungen, sondern bezieht ihn auf Entwicklungsphasen: »Verifikation ist die Prüfung eines Systems oder einer Komponente, mit dem Ziel zu bestimmen, ob die Produkte einer Entwicklungsphase die Vorgaben erfüllen, die zum Start der Phase auferlegt wurden.« Verifikation subsumiert gemäß IEEE also alle Techniken, die sicherstellen, dass man »das Ding richtig erzeugt«. Zu diesen Techniken gehören der Test des Produkts, Reviews, die Sicherstellung eines Zusammenhangs von Design, Anforderungen und Tests (Traceability) sowie Audits [PSS-05-0].

Im allgemeinen Sprachgebrauch kann man ein Ding nicht testen, das es noch nicht gibt. Das ISTQB-Glossar hat eine Definition des Begriffs Testen, der dem allgemeinen Sprachgebrauch widerspricht:

Test: Der Prozess, der aus allen Aktivitäten des Lebenszyklus besteht (sowohl statisch als auch dynamisch), die sich mit der Planung, Vorbereitung und Bewertung eines Software-Produkts und dazugehöriger Arbeitsergebnisse befassen. Ziel des Prozesses ist sicherzustellen, dass diese allen festgelegten Anforderungen genügen, dass sie ihren Zweck erfüllen, und etwaige Fehlerzustände zu finden.

Bei dieser Definition subsumiert der Begriff Test Aktivitäten, die in der älteren Definition von IEEE der Verifikation zugeordnet werden, und enthält auch Planungsschritte für das Software-Produkt. So weit gefasst ist dann fast alle Projektarbeit, außer der Produktentwicklung selbst, ein Test.

In diesem Buch wird, so wie bei der Definition von IEEE, dann vom *Test* gesprochen, wenn das Produkt Software, zumindest in Teilen, Gegenstand des Tests ist. Alle Schritte zur »Bewertung eines Software-Produkts und dazugehöriger Arbeitsergebnisse« davor werden aber nicht »Test« genannt, damit dem allgemeinen Sprachgebrauch und den IEEE-Standards nicht widersprochen wird.

Ansonsten entspricht die Verwendung von Fachwörtern in diesem Buch den Definitionen des ISTQB-Glossars und der ISO 29119-1. Und Hand aufs Herz: Ob man die Review dem Test (wie ISTQB) oder der Verifikation (wie IEEE) zuordnet, ist wohl weniger wichtig, als dass man sie ordentlich macht.

1.4.2 Zu Definitionen und TesterInnen

Mit der Niederschrift der Definition der verwendeten Fachbegriffe *zu Beginn* eines Dokuments sehen Sie eine Praxis, die auch im Umfeld der Software-Entwicklung gut aufgehoben ist und in einschlägigen Standards so empfohlen wird, zum Beispiel [IEEE 829]. Welche Zeitverschwendung, wenn Sie als Leser einen Begriff anders verstehen als der Autor, aber erst im Laufe des Lesens dahinter kommen und deshalb zum besseren Verständnis die ersten Passagen des Buchs noch einmal lesen müssen!

Apropos Missverständnis: Wenn in diesem Buch vom Beruf des Testers oder von anderen Rollen in der Software-Entwicklung gesprochen wird, so wird immer die männliche Form verwendet. Dies soll nur der Leserlichkeit dienen, aber in keiner Weise die Frauen diskriminieren. Ein Trost für alle Leserinnen: Wahre Top-Experten nennt man Koryphäen, auch wenn es Männer sind. Und zum Wort »Koryphäe« gibt es keine männliche Form.

1.5 Ein Überblick über das Umfeld des Software-Testing

In diesem Teil der Einleitung wird das Umfeld des Software-Tests erörtert und ein Überblick über Möglichkeiten und Grenzen des Testens gegeben. Für das Verstehen der späteren Kapitel ist das Lesen *nicht* notwendig, weil große Teile von Abschnitt 1.5 redundant zum Inhalt der vertiefenden Kapitel sind. Lesern, die

sich schon mit dem Thema Test auseinandergesetzt haben und die wissen, wie sie eine Brücke vom V-Modell zu agilen Methoden schlagen können, wird daher empfohlen, bei Kapitel 2 weiterzulesen.

Ohne Zweifel helfen die folgenden Seiten aber Neueinsteigern, die später vorgestellten Methoden im Software-Entwicklungsprozess besser einzuordnen. Und – wie gesagt – diese Seiten sind ein Appetitanreger auf das, was später noch im Buch behandelt wird.

Der Protagonist schlechthin im Umfeld des Software-Tests ist der Software-Fehler. Als Einstieg in das Thema Testen werden wir daher zunächst seine möglichen Ursachen untersuchen.

1.5.1 Ursachen von Software-Fehlern

Warum hat Software überhaupt Fehler? Die einfachste Antwort auf diese Frage ist, dass Software von Menschen geschrieben wird, und Menschen machen nun einmal Fehler. Wenn die Antwort auf eine komplexe Frage sehr einfach ist, dann übersieht sie meistens viele Facetten des zugrunde liegenden Problems. So auch in diesem Fall. Viele Software-Fehler, die sehr viel Geld kosteten, waren alles andere als einfache Programmierfehler. Die folgende Aufzählung skizziert einige typische, aber sehr unterschiedliche Ursachen für Fehler in Software.

- *Fehlerhafte Kommunikation oder keine Kommunikation bei der Anforderungsspezifikation.* In einem Projekt würde das bedeuten, dass nie ganz genau klar war, was die Software tun soll und was nicht. Oder, noch tückischer: Auftraggeber und Entwickler verstehen die Niederschrift der Anforderung unterschiedlich und sind sich der Möglichkeit einer anderen Interpretation gar nicht bewusst.
- *Psychologische oder kulturelle Gründe* können Auslöser für Kommunikationsprobleme sein: Manchen Personen fällt es schwer zuzugeben, wenn sie die Aufgabenstellung nicht verstehen. In manchen Kulturkreisen ist es üblich, aus Höflichkeit so zu tun, als verstünde man alles. Nicht selten hört man von euphorischen Verkäufern, die dem Kunden Produkteigenschaften zusichern, deren Realisierung nicht oder nur teilweise möglich ist. Mit Recht reklamiert der Kunde dann die entstehende Abweichung als Fehler.
- *Sich ändernde Anforderungen.* In welcher Softwareversion ist welche Version welcher Anforderung realisiert? Welche Tests müssen aufgrund einer Änderung modifiziert, welche Tests neu durchlaufen werden?
- *Softwarekomplexität.* Applikationen benutzen Module von Zulieferern oder anderen Projekten, bedienen sich komplizierter Algorithmen, unterstützen eine Vielzahl von Plattformen und so weiter. In großen Projekten können wenige Entwickler behaupten, die von ihnen erzeugte Software in allen Details zu verstehen.

- *Zeitdruck*. Entwicklungszeitpläne beruhen alle auf Aufwandsschätzungen. Wenn man hier sehr daneben lag und der Abgabetermin näher rückt, dann ist Eile angesagt. Zeitdruck ist ein perfekter Nährboden für Fehler.
- *Schlecht dokumentierter Code*. Der Programmierer, dessen Code im neuen Projekt wiederverwendet werden muss, war kein Freund der Dokumentation. Das Programm war für ihn schwer zu schreiben. Aus seiner Sicht ist es daher zumutbar, wenn auch die Dokumentation schwer zu lesen ist.
- *Programmierfehler.* Programmierer sind auch nur Menschen und machen Fehler im Software-Design oder bei der Umsetzung des Designs.

Ein Software-Fehler hat also viele verschiedene Ursachen. Tests sind nur geeignet, klassische Programmier- und Designfehler zu finden und sind bei allen, bis auf die letztgenannte Fehlerursache ein vergleichsweise wirkungsloses Instrument. Wenn zum Beispiel ein Kommunikationsproblem vorliegt und daher gegen eine falsche Anforderungsspezifikation getestet wird, dann nützt der gewissenhafteste Test nichts. Bis auf den Programmierfehler kann man aber allen anderen genannten Fehlerursachen durch Methoden des Managements begegnen.

Mit dem Wissen, dass Testen nur eine von vielen Maßnahmen im Kampf gegen Software-Fehler ist, nehmen wir nun den Programmierfehler, den Software-»Bug«, genauer ins Visier und erörtern, warum es so schwierig ist, alle Bugs zu finden. Die Bezeichnung Bug stammt übrigens laut Gerüchteküche von einer Motte, die auf einer Speicherplatine des Computers eines Zerstörers der US-Navy landete. Der resultierende Kurzschluss grillte nicht nur das unglückliche Insekt, sondern führte auch zu einer Fehlfunktion des Computers. Seitdem werden Software-Fehler nach dem Insekt benannt.

1.5.2 Warum Programmfehler nicht entdeckt werden

Vielen Software-Entwicklern ist Folgendes nicht unbekannt: Der Kunde meldet einen Programmierfehler, obwohl unzählige Stunden mit gewissenhafter Code-Review verbracht wurden, obwohl tagelang getestet wurde. Wie konnte dieser Fehler unbemerkt das Haus verlassen?

Unter der Annahme, dass sich der Kunde nicht irrt, kann einer der folgenden Punkte diese Frage beantworten:

- *Der Kunde führte Programmteile aus, die noch nie getestet wurden.* Entweder absichtlich (bei Zeitdruck oder Kostendruck) oder durch Unachtsamkeit wurden Programmteile nicht oder unzureichend getestet.
- *Die Reihenfolge, in der der Kunde Programmanweisungen ausführte, ist anders als die Reihenfolge, in der die Anweisungen getestet wurden*. Die Reihenfolge kann aber über Funktion und Fehler entscheiden.

- *Der Kunde verwendete eine Kombination von Eingangsgrößen, die nie getestet wurden.* Software wird in den seltensten Fällen mit allen möglichen Eingangsgrößen getestet. Der Tester muss daher eine geringe Zahl von Kombinationen selektieren und daraus schließen, dass die anderen Kombinationen auch funktionieren. Wenn dieser Schluss falsch ist, rutscht der Fehler durch.
- *Die operative Umgebung der Software ist beim Kunden eine andere.* Der Kunde verwendet eine andere Betriebssystemversion oder eine andere Hardware. Vielleicht stand dem Testteam die Anwendungsumgebung überhaupt nicht zur Verfügung und man musste diese simulieren oder von Annahmen ausgehen.

Software wird fast nie zu 100 % getestet. Das gilt auch für sicherheitskritische Anwendungen. [Hayhurst 01] beschreibt zum Beispiel, dass Flugkontrollsoftware bis zu 36 verschiedene Eingangsgrößen verarbeitet. Wollten wir alle möglichen Eingangskombinationen durchtesten und damit beweisen, dass keine ungewollten Wechselbeziehungen zwischen den Eingängen bestehen, so müssten wir 21 Jahre lang testen, selbst wenn wir pro Sekunde 100 Testfälle erstellen und durchführen könnten. In Kapitel 6 werden Techniken vorgestellt, die durch Analyse des Quellcodes den Testaufwand dramatisch reduzieren und trotzdem die Testschärfe nur gering beschneiden.

1.5.3 Angebrachter Testaufwand

Die Aufgabe des Testers ist es nun, mit den vorhandenen Ressourcen so zu testen, dass er mit größter Wahrscheinlichkeit alle Fehler findet, die in der Software stecken. Kapitel 8 wird dazu viele Techniken vorstellen. Die Aufgabe des Managements ist, dem Tester Ressourcen so zu genehmigen, dass der maximale Projektgewinn resultiert. Abbildung 1–1 zeigt dazu die anzustellende, wirtschaftliche Überlegung. Wird zu wenig gestestet und drohen dem Unternehmen daher Schadenersatzzahlungen oder Imageverlust, dann ist der Testaufwand zu erhöhen. Wird andererseits zu viel getestet, dann ist das Software-Produkt zwar vielleicht von guter Qualität, aber nicht mehr wirtschaftlich zu erzeugen.

Die Entscheidung, wie viel Aufwand in den Test zu stecken ist, ist eine rein wirtschaftliche und stützt sich im Idealfall auf sogenannte Prozessmetriken. Derartige Metriken erfassen beispielsweise wie viel Zeit (und damit Geld) es kostet, einen Fehler zu finden, und wie viel es kostet, einen Fehler nicht zu finden. Prozessmetriken sind ein wichtiges Werkzeug der Wirtschaftlichkeitsanalyse und des Software-Qualitätsmanagements.

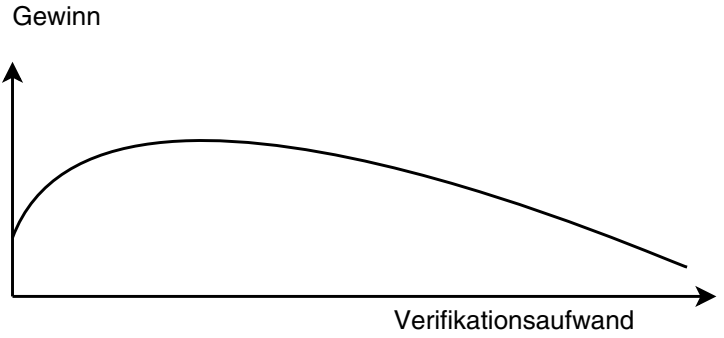

Abb. 1–1 *Der Testaufwand sollte sich am Betriebsgewinn orientieren.*

1.5.4 Der Tester und der Testprozess

Bei der Erstellung von Testfällen muss der Tester die Software, ihre Funktionalität, die Eingangsgrößen, die möglichen Kombinationen der Eingangsgrößen und die Testumgebung berücksichtigen. Dieser Prozess ist kein Kinderspiel, benötigt Zeit, Erfahrung und Planung. In der Zeitskala dieser Planung muss auch Zeit vorgesehen werden, die Testfälle zu revidieren, gefundene Bugs zu entfernen und die Software von Neuem zu testen. Im Zeitplan nur den Wonnefall vorzusehen – alles funktioniert klaglos, das Testfalldesign ist fehlerfrei –, ist ein Fehler, der Anfängern vorbehalten bleiben soll.

Vom Tester wird destruktive Kreativität verlangt, wenn seine Tests auch trickreiche Bugs finden sollen. Je mehr Freude der Tester am Finden von Fehlern hat, desto mehr wird er sich anstrengen, auch tatsächlich welche zu finden. Wenig Freude bereitet es, seine eigenen Fehler zu finden, und so ist eine der Grundregeln des Testens, dass Programmautor und -tester verschiedene Personen sein müssen. Glenford Myers, der erste nennenswerte Autor von Literatur über Software-Tests, schreibt dazu folgenden Absatz [Myers 78]:

> *»Testen ist ein destruktiver Prozess. Man versucht die Software zu brechen, ihre Schwachpunkte zu finden, Fehler aufzudecken. Es ist sehr schwierig für einen Programmierer, nachdem er während des Designs und des Programmierens konstruktiv war, plötzlich seine Perspektive zu ändern und eine destruktive Haltung gegenüber dem Programm einzunehmen. Aus diesem Grund können die meisten Programmierer ihre eigene Software nicht effektiv testen, weil sie es nicht schaffen, diese destruktive Haltung einzunehmen.«*

Um den Testprozess genauer zu beschreiben, unterteilt ihn James Whittaker in einem sehr gelungenen Überblicksartikel in vier Phasen [Whittaker 00]:

- Modellieren der Software-Umgebung
- Erstellen von Testfällen
- Ausführen und Evaluieren der Tests
- Messen des Testfortschritts

Diese Phasen und die damit verbundenen Probleme und Lösungsansätze wollen wir uns nun genauer ansehen.

1.5.5 Modellieren der Software-Umgebung

Eine der Aufgaben des Testers ist es, die Interaktion der Software mit ihrer Umgebung zu prüfen und dabei diese Umgebung zu simulieren. Das kann eine enorme Aufgabe sein, wenn man die Vielfalt von Schnittstellen bedenkt:

- Die *klassische Mensch/Maschine-Schnittstelle*: Tastatur, Bildschirm, Maus. Der Tester muss sich überlegen, wie er alle erwarteten und unerwarteten Mausklicks, Tastatureingaben und Bildschirminhalte in den Tests organisiert. Dabei können ihn Capture/Replay-Tools unterstützen, die diese Eingaben simulieren und die resultierende Bildschirmdarstellung mit einer gespeicherten Soll-Darstellung vergleichen. Der Einsatz solcher Tools ist mit einer nicht zu unterschätzenden Lernkurve verbunden. Billige Tools haben obendrein das Problem, dass man bei einem Wechsel der GUI-Version oder des Betriebssystems oft auch gleich viele Tool-Aufzeichnungen wiederholen muss.
- Das Testen der *Schnittstellen zur Hardware* kann eine Herausforderung für sich sein. Beim Test von eingebetteten Systemen muss oft erst ein Testgerät entwickelt beziehungsweise gekauft werden, damit die zu testende Software in ihrer Zielhardware auch getestet werden kann (*Hardware in the Loop*). Für die auf diesen Schnittstellen basierenden Kommunikationsprotokolle muss der Tester erwartete und unerwartete, gültige und ungültige Daten oder Kommandos versenden. Kapitel 8 zeigt Testtechniken dazu und Kapitel 14 stellt eine ergänzende Analysetechnik vor.
- Die *Schnittstelle zum Betriebssystem* sollte ebenfalls Gegenstand von Tests sein. Wenn die zu testende Software einige Dienste des Betriebssystems in Anspruch nimmt, dann muss auch geprüft werden, was passiert, wenn das Betriebssystem diese Dienste verweigert: Was ist, wenn das Speichermedium voll ist oder der Zugriff darauf misslingt? Um das ohne Kenntnis des Codes zu testen, könnte man beispielsweise Werkzeuge des Security-Testings verwenden, die die zu testende Software in einer Betriebssystem-Simulation laufen lassen [Whittaker 03]. In dieser Simulationsumgebung kann man dann zum Beispiel per Knopfdruck eine volle Festplatte simulieren. Manche sprechen in diesem Zusammenhang auch von *Sandbox Testing*.

- Auch *Dateisystem-Schnittstellen* findet man bei einigen eingebetteten Systemen. Der Tester muss Dateien mit erlaubtem und unerlaubtem Inhalt und Format bereitstellen.

Es ist unschwer zu erkennen, dass die Modellierung der Systemschnittstellen ein sehr aufwändiges Projekt werden kann. Stellen wir uns zum Beispiel Software für ein Motorsteuergerät in einem Kraftfahrzeug vor: Manche Prozesse im Motor und Interaktionen mit anderen Steuergeräten (wie ABS, ESP, ...) sind so komplex, dass sie nur unter großem Aufwand modelliert werden können. Es ist in der Testplanung abzuwägen, wie exakt das Modell der Umgebung sein soll. So muss etwa bei einem Steuergeräte-Test für Fahrzeuge geplant werden, was in einer Umgebungssimulation geprüft wird (siehe Abb. 1–2) und was etwa an einem Motorprüfstand oder in einem Testfahrzeug getestet wird (siehe Abb. 1–3).

Das Modell der Umgebung, beziehungsweise die Testausrüstung für Tests am Zielsystem, muss auch außergewöhnliche Situationen vorsehen. Etwa den Neustart der Hardware während der Kommunikation mit einem externen Gerät, die Parallelnutzung eines Betriebssystemservices durch andere Applikationen, einen Fehler im Speichermedium und so weiter.

Abb. 1–2 *Beispiel einer Testumgebung. Diese Ausrüstung wird verwendet, um Software für Steuergeräte automatisiert zu testen. Produziert die Testumgebung, wie hier, auch alle Stimuli für Sensoren und prüft sie die Funktion von Aktoren, so spricht man vom Hardware-In-The-Loop-Test. Foto © dSPACE GmbH.*

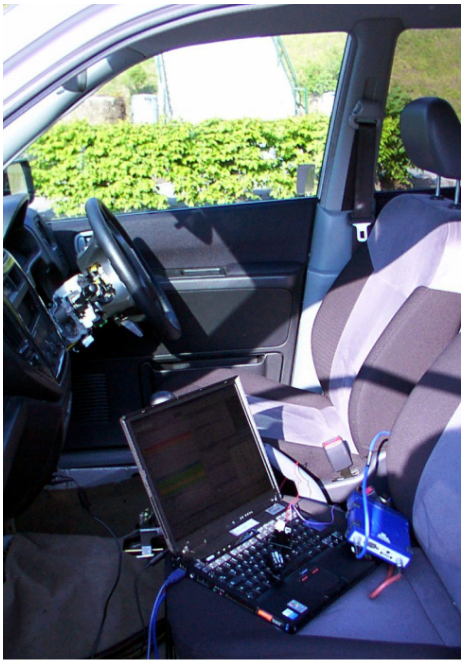

Abb. 1–3 *Eine Testschnittstelle ermöglicht während der Fahrt, Interna der Steuergeräte-Software per Laptop zu überwachen. Die Testfälle sind in diesem Testfahrzeug per Definition real. Ihr Design ist aber im Vergleich zur Simulation eingeschränkter, zum Beispiel weil Unfallszenarien nicht einfach zu durchlaufen sind.*

1.5.6 Erstellen von Testfällen

Software kann mitunter eine unendlich große Anzahl von verschiedenen Eingangsdaten verarbeiten. Denken wir nur an ein Textverarbeitungsprogramm oder an eine Signalverarbeitungssoftware, wo die Reihenfolge der Eingabe von Daten relevant ist. Die schwierige Aufgabe des Testers ist es nun, einige wenige Testszenarien zu entwerfen, die ein Programm prüfen sollen, das eigentlich unendlich viele Szenarien verarbeiten kann.

Bei der Durchführung der Tests wird man nach der strukturellen Testabdeckung (*Structural Test Coverage*) fragen: Welche Teile des Codes sind noch ungetestet? Um diese Frage zu beantworten, bedient man sich in den meisten Fällen eines Werkzeugs. Kapitel 6 wird die Funktionsweise solcher Werkzeuge erklären.

Abbildung 1–4 zeigt, wie so ein Werkzeug verschiedene Arten von Testabdeckungen nach Messung anzeigt und so den Testfortschritt visualisiert.

Mit dem Ziel, die gewünschte Testabdeckung am Quellcode zu erreichen wird der Tester daher Szenarien auswählen, die

- typisch auch in der Feldanwendung auftreten,
- möglichst »bösartig« sind und damit eher Fehler provozieren als »Schönwettertests«,
- Grenzfälle ausprobieren.

Für die Testabdeckung gibt es eine Reihe von Kriterien unterschiedlicher Schärfe. Im einfachsten Fall könnte man sich etwa zufriedengeben, wenn jedes Statement der Programmiersprache C einmal ausgeführt wurde. In sicherheitskritischen Anwendungen könnte man fordern, dass jedes Assembler-Statement mindestens einmal ausgeführt wird und dass zusätzlich bewiesen werden muss, dass jeder Einfluss auf eine Programmverzweigung auch unabhängig von anderen Einflüssen wirken kann. Abschnitt 6.6 wird einige Testabdeckungsmetriken für Modultests vorstellen.

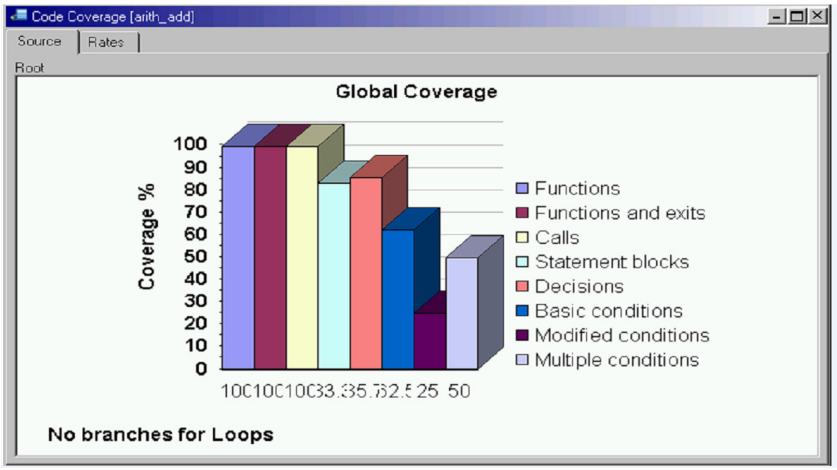

Abb. 1–4 *Anzeige der erreichten Testabdeckung in einem Werkzeug für Unit-Tests*

Die Testabdeckung aber nur am Quellcode zu messen, wäre keine gute Idee. Der Code zeigt nicht, was vergessen wurde zu implementieren, und er zeigt auch nicht, ob die Anforderungen richtig umgesetzt wurden. Dazu ist zumindest ein Blick in die Anforderungsspezifikation nötig. Je formaler diese verfasst ist, desto besser für den Zweck des Testens.

Auf Basis der Spezifikation lassen sich auch für viele Black-Box-Testtechniken Maße definieren, die die Schärfe/Überdeckung eines Tests völlig unabhängig vom Code bewerten. Die anzustrebende Überdeckung für einen Black-Box-Test nennt die Normenreihe ISO 29119 *Testüberdeckungselemente* (*test coverage items*). Kapitel 8 wird solche Testüberdeckungselemente vorstellen.

1.5.7 Ausführen und Evaluieren der Tests

Nachdem geeignete Testfälle ausgewählt/entworfen worden sind, kann zur Ausführung derselben geschritten werden. Die kann manuell erfolgen, halb automatisch sein oder voll automatisch ablaufen. Die Wahl des Automatisierungsgrads ist vor allem von zwei Faktoren abhängig: der Haftung bei Software-Fehlern und der Wiederholungsrate der Tests. Bei Anwendungen mit Sicherheitsrelevanz wird die Wahl eher auf automatische Tests fallen. Die Testskripts erlauben im Idealfall eine exakte Wiederholbarkeit der Tests.

Bei manuell durchgeführten Tests ist exakte Wiederholbarkeit schlecht möglich. Ganz speziell bei Echtzeitsystemen. Dafür ist man mit manuellem Testen schneller am Ziel: Bis automatische Tests fehlerfrei laufen, ist im Schnitt der zehnfache Aufwand wie für einen manuellen Test notwendig. Also rentiert sich Testautomation vor allem dann, wenn Tests oft wiederholt werden. Solche Wiederholungen können zum Beispiel notwendig sein, wenn man zu bestehender Software neue Funktionalität hinzufügt oder einen Fehler behebt. Bringt die Fehlerbehebung an anderer Stelle einen neuen Fehler? Um sicherzugehen, kann man dann alle existierenden Tests automatisch nochmals durchführen. Diese Testwiederholungen, »nur um sicherzugehen«, werden *Regressionstests* genannt. Automation ist besonders für Regressionstests geeignet.

Mit der Ausführung der Tests alleine ist es aber noch nicht getan. Es muss auch noch beurteilt werden, ob sie einen Fehler gefunden haben oder nicht. Das ist nicht immer einfach und gelegentlich eine Hürde für die volle Automation von Tests. Jedenfalls müssen die Kriterien für den Testerfolg genau spezifiziert sein. Ohne diese Festlegung würde der Software-Test allenfalls ein Herumprobieren sein, aber den Namen »Test« nicht verdienen.

Trotz des Determinismus und der Wiederholbarkeit ersparen vollautomatische Tests nicht die sorgfältige Dokumentation und Abstimmung des Testdesigns mit sich ändernden Versionen der Anforderungsspezifikation. Sonst würde die Testkollektion, wie es in Kritiken der Testautomation heißt, ein gewisses Eigenleben entwickeln: »Niemand getraut sich einen Test zu löschen, weil die Bedeutung nicht genau bekannt ist; kein Mitarbeiter weiß, was die Tests eigentlich tun, aber sie zu bestehen ist essenziell; mit neuen Anforderungen kommen aber neue Tests hinzu, und so wird die Testkollektion ein zunehmend mysteriöseres Orakel, ähnlich einer sprechenden Eiche aus einem Disney-Film« [Bach 96].

Bleibt trotz allem noch die Frage des Vertrauens in die automatischen oder manuellen Tests. Wenn das Testsystem immer »alles okay« meldet, sollten wir uns dann nicht irgendwann mal Sorgen machen? Um diese Sorgen in Grenzen zu halten, ist es empfehlenswert, bei der Entwicklung von Testskripts zum Debugging absichtlich Fehler im Testobjekt einzubauen oder das Testobjekt so zu konfigurieren, dass das Testskript einen Fehler erkennen sollte. Beispiel: beim Test

eines Video-Übertragungssystems auf einem eingebetteten Audio-Kanal kurz den Ton abdrehen. Wird dieser eingebrachte »Fehler« nicht in den automatischen Tests entdeckt, dann stimmt etwas mit der Testumgebung nicht.

1.5.8 Messen des Testfortschritts

Wie jedes andere Projekt sollte auch ein Testprojekt genau geplant werden. Teil dieses Plans ist die Festlegung des Projektziels, zum Beispiel wie viele unentdeckte Fehler welcher Kategorie die zu testende Software nach dem Test noch haben darf. Die Art und der Umfang der Tests werden sich nach dieser Größe richten, siehe Diskussion zu Abbildung 1–1.

Bei der Erstellung eines Zeitplans für den Software-Test können Code-Metriken helfen. Code-Metriken sind Maßzahlen, die aus dem Quellcode automatisch erhoben werden, siehe Abschnitt 4.5. Zum Beispiel die Anzahl der linear unabhängigen Pfade, die in Abschnitt 6.7 behandelt werden. Ist aus Aufzeichnungen bekannt, wie lange man im Schnitt pro Pfad testet, dann ist die Aufwandsschätzung schnell gemacht, wenn man sich vornimmt, alle linear unabhängigen Pfade zu testen.

Für die Aufwandsschätzung beim Test gegen die Anforderungsspezifikation, dem Systemtest, können die gleichen Verfahren herangezogen werden, die zur Schätzung der Programmerstellung dienen. Auch hier kann man eine Testabdeckung definieren. Etwa: Alle Software-Anforderungen einer gewissen Mindestpriorität sollen getestet werden.

Um den Zeitplan regelmäßig zu aktualisieren, kann man Messungen einer zuvor ausgewählten Testabdeckung verwenden. Die Erfassung des bisher in den Test gesteckten Aufwands und der bisher erreichten Testabdeckung erlaubt eine einfache Extrapolation und damit die Vorhersage des Erreichens von 100 % der gewünschten Testabdeckung. Dabei ist allerdings eine Warnung angebracht: Der Aufwand pro Prozent Abdeckung steigt typischerweise gegen Ende des Projekts stark an.

1.5.9 Testdesign und Testdokumentation im Software-Entwicklungsprozess

Der Software-Entwicklungsprozess beginnt meist mit der Analyse des Problems und dem Schreiben der Anforderungsspezifikation. Die Spezifikation bildet die Grundlage für funktionale Software-Systemtests, die die Einhaltung der Spezifikation testen und somit Designfehler und Programmierfehler aufdecken.

Mit dem Design der Systemtests kann begonnen werden, sobald die Anforderungen feststehen, also oft schon lange, bevor die Software überhaupt existiert. Ein früher Beginn des Testdesigns, parallel zum Software-Design, zahlt sich aus mehreren Gründen aus:

- Die Testphase verkürzt sich, denn wenn die zu testende Software fertig für den Test ist, muss man nicht erst beginnen, sich gute Tests auszudenken.
- Untestbare Anforderungen werden identifiziert. Es gibt kaum eine kritischere Review von Anforderungen, als wenn man sich Tests dafür ausdenkt.
- Gegebenenfalls wird die Notwendigkeit von spezieller Testhardware erkannt und diese kann rechtzeitig geschaffen oder beschafft werden. Solche Testhardware kann ein Testzugang am zu testenden Gerät sein, etwa ein Messpunkt auf einer Platine oder ein Messmittel, etwa ein Oszilloskop.

Mit dem Testen selbst kann man aber natürlich erst beginnen, wenn die Software fertig ist. Wenn die Anforderungen dem Entwickler-Team gewisse Freiheiten lassen, dann kann es sein, dass es beim Testdesign nicht möglich ist, den einen oder anderen Test ins letzte Detail auszuformulieren. Das ändert nichts an der Tatsache, dass es sinnvoll ist, mit dem Testdesign möglichst früh zu beginnen. Es bleiben aber durch offene Fragen Lücken im Testdesign. Der Testplan, in dem das Systemtestdesign meist dokumentiert ist, muss also ein »lebendes Dokument« sein. Es erfährt regelmäßige Aktualisierungen und muss daher unter Versionskontrolle stehen. Nicht nur das Füllen der genannten Lücken sind Gründe für Aktualisierungen. Natürlich kommt es vor, dass ein im frühen Projektstadium ausgedachter Testfall später nicht so umzusetzen ist, wie gedacht. Wird die Dokumentation nicht angepasst, so laufen reale Tests und Dokumentation auseinander und man entwertet das Test-Set. Im schlimmsten Fall manövriert man die Tests in einen unwartbaren Zustand und landet bei der in Abschnitt 1.5.7 erwähnten sprechenden Eiche aus einem Disney-Film.

In vielen Projekten ist aber der häufigste Grund für Updates von Tests eine Revision der Anforderungsspezifikation. Um bei einer solchen Revision der Anforderungen zu wissen, welche der schon bestehenden Systemtests überarbeitet werden müssen, wird eine *Traceability*-Tabelle erzeugt und stets auf dem aktuellsten Stand gehalten. Diese Tabelle verfolgt, welche Anforderung in welchem Testfall getestet wird. Ein Beispiel ist in Abbildung 1–5 zu sehen. Diese Tabellen können entweder manuell erstellt und gewartet werden oder sind ein Produkt von Requirements Engineering/Management Tools.

1.5.10 Verschiedene Teststufen und deren Zusammenspiel

Normalerweise werden im Software-Designprozess die Aufgaben der Software auf verschiedene Module (beziehungsweise Klassen) aufgeteilt. Es ist daher naheliegend, nach dem Programmieren zuerst jedes dieser Module einzeln zu testen, bevor sie zu einem Ganzen zusammengefügt werden. Diese in Kapitel 6 behandelten Tests werden Modultests genannt. Mit dem Design der Modultests kann begonnen werden, sobald das Software-Design genau die Aufgaben jedes Software-Moduls spezifiziert. Der Modultest testet das Software-Modul gegen diese

Modulspezifikation. Beim Modultest leistet man sich den Luxus, genau zu prüfen, welche Teile des Quellcodes tatsächlich durchlaufen wurden. Je nach Anspruch an die Schärfe dieser *White-Box-Tests* existieren verschiedene Maße für das Messen der Testabdeckung, wie Abschnitt 6.6 zeigen wird.

Ob beim Zusammenfügen der Module Fehler auftreten, prüft ein *Integrationstest*. So wie dem Thema Modultest widmet sich ein eigenes Kapitel dieses Buchs dieser in der Literatur wenig beschriebenen Teststufe. Integrationstests können geplant werden, sobald feststeht, welche Daten- und Kontrollflüsse zwischen den Modulen stattfinden. Integrationstests prüfen genau dieses Zusammenspiel und decken also Schnittstellenprobleme auf. Sie sind also ein Test der Software gegen das Design der Software-Architektur.

Nach bestandenem Integrationstest beginnt man mit der Durchführung der *Systemtests*. Systemtests prüfen die Einhaltung der Anforderungsspezifikation und testen die Software möglichst in der Zielumgebung. Auch bei gewissenhaftesten Unit-Tests und Integrationstests werden hier typischerweise Fehler gefunden: Design-Fehler. Die vorhergehenden Testschritte testen immer gegen das Design der Software. Ist das Design aber fehlerhaft oder lückenhaft und wird dadurch die Spezifikation nicht erfüllt, so ist die Software dennoch fehlerhaft und diese Fehler werden am ehesten im Systemtest in der Zielumgebung gefunden.

Ist die Zielumgebung nicht verfügbar und das Risiko vertretbar, so kann man sich mit der Simulation der Zielumgebung zufriedengeben. Bei sicherheitsrelevanter Software und Software mit hohen Ansprüchen an die Integrität ist ein Test in der Zielumgebung jedoch unverzichtbar. Abschnitt 8.12 auf Seite 183 gibt ein eindrucksvolles Beispiel aus der Praxis zu diesem Thema.

Systemtests werden, wie erwähnt, am besten schon lange vor den Integrationstests entworfen. Da zum Zeitpunkt des Entwurfs der Code noch nicht existiert, sind die Tests idealerweise unabhängig vom Design. Man spricht von *Black-Box-Tests*. Techniken dazu werden in Kapitel 8 vorgestellt.

Requirement No	Requirement Short Text	Tested in
R/SRD/INIT/10	Self Test	ST-IOPS-FT01
R/SRD/INIT/20	Self Test Fail Silence	ST-IOPS-FT01
R/SRD/PP/05	Ready Message	ST-IOPS-FT01
R/SRD/PP/20	PP-bus Performance	ST-IOPS-FT02
R/SRD/PP/30	Reply Message Format	ST-IOPS-FT02
R/SRD/PP/40	SetPosDebTime Cmd	ST-IOPS-FT03
R/SRD/PP/50	SetNegDebTime Cmd	ST-IOPS-FT03
R/SRD/PP/60	SetRepIntv Cmd	ST-IOPS-FT02
R/SRD/PP/70	Start Command	ST-IOPS-FT02
R/SRD/PP/80	Report Message Format	ST-IOPS-FT02
R/SRD/SI/10	Position Sampling Rate	ST-IOPS-FT03
R/SRD/SI/30	Debouncing	ST-IOPS-FT04, ST-IOPS-FT03
R/SRD/RA/10	Missing Parameters	ST-IOPS-FT05
R/SRD/RA/20	PP-bus Error	ST-IOPS-FT05
R/SRD/RA/30	Watchdog Kicking	ST-IOPS-FT05
R/SRD/SB/10	Maximum CPU Load	ST-IOPS-FT06
R/SRD/SB/20	CPU Idle Time	ST-IOPS-FT07
R/SRD/SB/30	Memory Footprint	ST-IOPS-FT08
R/SRD/SD/10	Software Coding Standard	verified in code reviews

Abb. 1–5 *Die Traceability-Tabelle für Systemtests eines fiktiven Projekts*

Mit hohen Ansprüchen an die Integrität der Software sollte auch die weitgehende Automation der Tests einhergehen. Bei Updates kann somit schnell der gesamte Funktionsumfang der Software getestet werden und die Wiederholbarkeit der Tests ist im Idealfall zu 100 % gesichert. Bei eingebetteten Systemen kann so ein automatischer Test in der Zielumgebung großen Aufwand bedeuten. Ein Beispiel ist in Abbildung 1–6 zu sehen. Die zu testende Software steuert eine Anzeige am Aktor Armaturenbrett. Dem Aktor muss im Testsystem ein entsprechender Sensor gegenüberstehen, der die Anzeige erfasst.

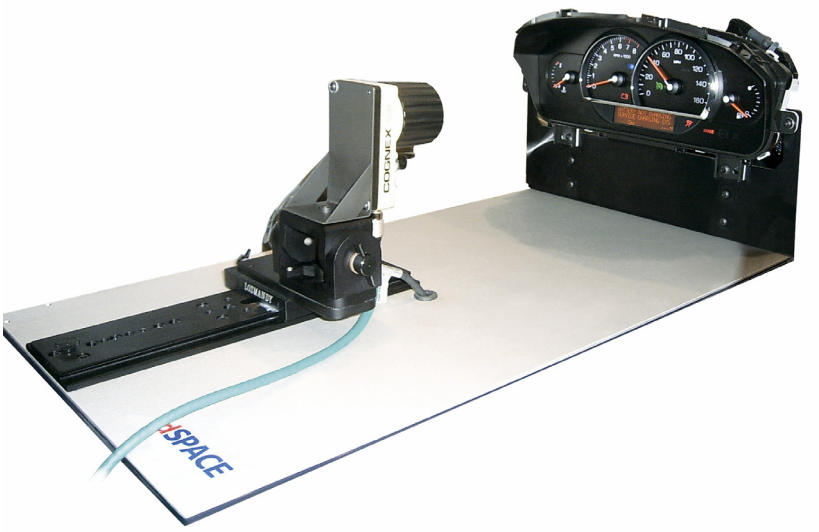

Abb. 1–6 *Automatische Systemtests können aufwändig werden. Ist der von der Software bediente Aktor die Tacho-Nadel, so sollte der Sensor des Testaufbaus die Position der Tacho-Nadel erfassen können. Foto © dSPACE GmbH.*

Die Durchführung der Testschritte erfolgt also in einem Bilderbuchprojekt in umgekehrter Reihenfolge wie die Planung und das Design der Tests. Dieses traditionelle Software-Entwicklungsmodell (*Software Life Cycle Model*) wird oft mit einem »V« visualisiert. Abbildung 1–7 zeigt solch ein Modell: das inzwischen abgelöste Entwicklungsmodell PSS-05 der europäischen Raumfahrtorganisation [PSS-05-0]. Der zeitliche Verlauf in diesem »V« ist von links oben nach unten und dann nach rechts oben. Und das Modell definiert für jeden Prozessschritt bestimmte Artefakte als zu erwartendes Ergebnis. Am linken Ast im »V« des PSS-05 sind dies nur Dokumente. Und zwar URD für das User Requirements Document, SRD für das Software Requirements Document, ADD für das Architectural Design Document und DDD für das Detailed Design Document.

Ob alle beschriebenen Testschritte in einem Projekt auch tatsächlich zum Einsatz kommen und alle Dokumente wirklich geschrieben werden, ist eine Frage der Wirtschaftlichkeit.

1.5.11 Andere Verifikationsmethoden als Ergänzung zum Test

Abbildung 1–7 zeigt am linken Ast des Modells Pfeile in Aufwärtsrichtung, also »gegen die Zeit«. Diese mit SVVP/SR, SVVP/AD und SVVP/DD beschrifteten Pfeile stehen für kleine Iterationen. Solche kurzen Zyklen sind Ergebnisse von Reviews. Welche Reviews durchgeführt werden, ist gemäß PSS-05 beim Projektstart zu definieren und in einem Software Verification and Validation Plan

(SVVP) zu dokumentieren. Dieses zentrale Planungsdokument hat leider viele verschiedene Namen. In älteren IEEE-Normen und bei ISTQB spricht man vom *Master Test Plan*. Die Normenreihe 29119 spricht vom *Test Plan* und lässt offen, ob es mehrere Bände dieses Dokuments gibt. »Test Plan« übersetzen Normenspezialisten auf Deutsch gerne als *Testkonzept* [Daigl16].

Alle vorgestellten Artefakte – User Requirements Document, Software Requirements Document, Architectural Design Document und das Detailed Design Document – werden im Idealfall einer Review unterzogen. Kommt der Reviewer zur Auffassung, dass zum Beispiel ein Anforderungsdokument Widersprüche enthält oder ein Design verbesserungswürdig ist, so wird der Widerspruch aufgelöst und das Design verbessert, ehe man die nächste Prozessphase betritt. Typischerweise finden solche Reviews innerhalb eines Teams statt. Dokumente, die Vertragsbestandteil sind oder die Kundenkommunikation unterstützen, wie Anforderungsdokumente, werden oft auch durch den Geschäftspartner inspiziert.

In Projekten mit höchster Integritätsanforderung ist der Abschluss einer Phase oft mit einer Technical Review verbunden. Ein Review Board, bestehend aus Kunden, Lieferanten und gegebenenfalls einer externen Firma, die ausschließlich zum Zweck der Review bestellt wurde, entscheidet, ob ein Zahlungsmeilenstein erreicht ist und es dem Lieferanten gestattet wird, in die nächste Projektphase einzutreten.

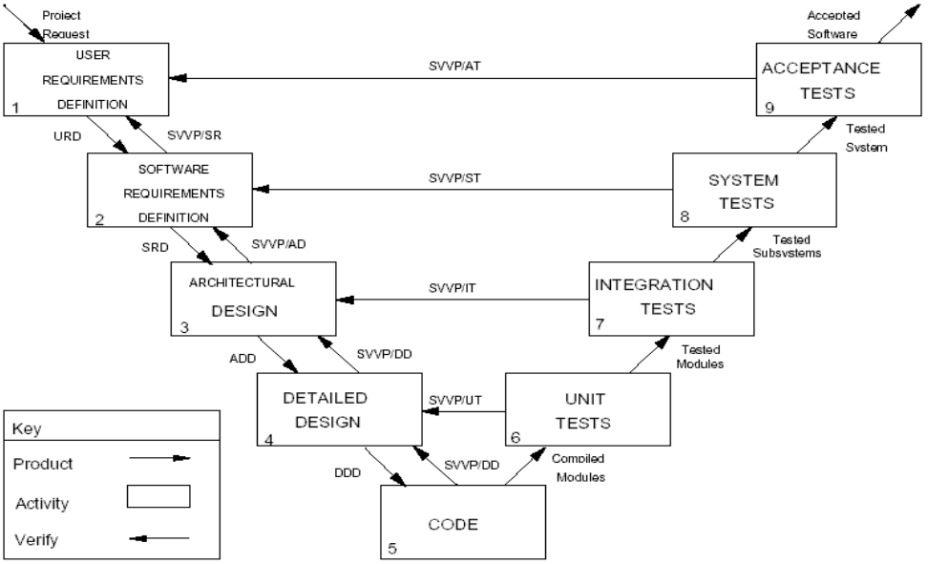

Abb. 1–7 *Das Software-Entwicklungsmodell nach PSS-05. Die horizontalen Pfeile zeigen, welche Testschritte welche Teile der Entwicklungsschritte abtesten. Sobald der Entwicklungsschritt an der Pfeilspitze fertig ist, kann mit dem Design der entsprechenden Tests begonnen werden.*

Auch der Quellcode wird einer Review unterzogen – am besten, bevor man Unit-Tests macht. Code-Reviews können zum Teil automatisiert werden. Man spricht von der Automatischen Statischen Analyse, Kapitel 4 beschäftigt sich damit.

Alle angesprochenen Reviews dienen dazu, die Fehlerkosten niedrig zu halten. Je früher im Entwicklungsprozess sich ein Fehler einschleicht, desto größer seine Tragweite. Denken wir zum Beispiel an eine ungeschickte Software-Architektur. Wenn der Code bereits implementiert ist und erst im Integrationstest erkannt wird, wie unpassend das Design ist, dann müssen die Architekturdokumentation geändert, der Code angepasst und die Unit-Tests aktualisiert werden. Wäre der Fehler gleich bei einer Review des Architectural-Design-Dokuments aufgefallen (in kleinen Projekten kann das auch eine Bleistiftskizze sein), wäre der Änderungsaufwand auf die Aktualisierung des Dokuments beschränkt gewesen.

Reviews und Tests reichen nicht aus, um Echtzeitsysteme mit höchsten Ansprüchen ausreichend abzusichern. Als weitere Verifikationsmethode kommt die Analyse hinzu. Im Speziellen ist dies die Analyse der Rechtzeitigkeit der Systemantwort, der sich die Kapitel 12 und 13 widmen. Bei Systemen mit Sicherheitsrelevanz kommt noch die Analyse des Systemverhaltens bei Teilausfällen hinzu, die in Kapitel 14 beschrieben wird, und man wird im Projektverlauf auch regelmäßig den Speicherverbrauch des Systems ermitteln, damit es nicht unerwartet zu bösen Überraschungen kommt.

1.5.12 Agile Prozessmodelle

Im realen Leben wird der in Abbildung 1–7 gezeigte Entwicklungszyklus mehrfach durchlaufen. Neue Iterationen können notwendig werden, wenn zum Beispiel neue Anforderungen hinzukommen. Zumindest teilweise sind Iterationen bei Re-Designs und bei Bugfixes notwendig. Entsprechend erfahren die begleitenden Projektdokumente mehrere Updates. Ohne solche Updates werden die Dokumente zunehmend wertloser!

Agile Methoden (Scrum, Extreme Programming …) machen diese Not zur Tugend und versuchen durch eine absichtlich hohe Anzahl von Iterationen und Lieferung von Teilen des Produkts, Dokumentation zu sparen und den Kunden verstärkt in den Entwicklungsprozess einzubinden. Anstelle von Dokumentation tritt verstärkte Kommunikation und hohe Bereitschaft zu enger Zusammenarbeit [Beck 01].

Bei vielen eingebetteten Systemen sind Teillieferungen von Software nur eingeschränkt oder gar nicht möglich. Eine frühe Kundeneinbindung ist ebenfalls nicht immer sinnvoll. Agile Projektmodelle werden daher oft adaptiert. Die Rolle des die Teillieferung abnehmenden Kunden wird zum Beispiel von einer anderen Abteilung übernommen, die die neu entwickelte Software und Hardware integriert.

1.5.13 Der Software-Test in agilen Vorgehensmodellen

Eine Spielart bei agilen Methoden ist die *Test-First*-Strategie (oder *Test Driven Development, TDD*). Das heißt, der Programmierer beginnt einen Unit-Test für den eigenen Code zu schreiben, bevor er mit dem Codieren beginnt. Zug um Zug werden in kleinen Iterationen zuerst der Test und dann das Testobjekt implementiert beziehungsweise erweitert. Sobald der Test keine Fehler mehr meldet, beginnt eine neue Iteration.

Bei maschinennaher Programmierung kann diese Strategie sehr ineffizient werden, wenn man beim Programmieren erst die neue Hardware »kennenlernt« und das detaillierte Design sich daher hochdynamisch ändern kann. Ebenso ist es bei der Entwicklung von eingebetteten Systemen oft der Fall, dass Hardware erst während der Software-Entwicklung entsteht. Das erschwert natürlich Test Driven Development mit Tests in der Zielumgebung und zwingt gegebenenfalls zur Nutzung von Simulatoren, aber es zwingt auch – als positiver Effekt – zu einem Design, bei dem Hardware-Abhängigkeiten sauber gekapselt sind. Es gibt ohne Zweifel Embedded-Projekte, die von TDD profitieren: Die kurzen Iterationen vom fertiggestellten Code zum unerwartet fehlgeschlagenen Testfall ersparen langwierige Debug-Sitzungen, wie man sie oft vorfindet, wenn ein Test erst lange nach dem Programmieren stattfindet; nur der gerade hinzugefügte Code kann für das Fehlschlagen des Tests verantwortlich sein [Grenning 11]. Von dieser Eigenschaft profitiert man aber auch, wenn man den Unit-Test (siehe Kapitel 6) *unmittelbar* nach der Implementierung eines Moduls macht und dabei strikt Black Box gegen die Spezifikation (das Design) des Moduls testet. Der Test muss dazu nicht vor oder während der Implementierung des Codes entstehen.

Beim Erstellen eines Unit-Tests für fertig implementierte Komponenten unterstützen aber leistungsfähige Unit-Test-Werkzeuge extrem und können die Arbeit deutlich beschleunigen. Abschnitt 6.5.2 wird die Funktionsweise dieser Werkzeuge vorstellen.

TDD ist keine Testmethode, sondern eine Entwicklungsmethode, die sicherstellt, dass der Code testbar und getestet ist. Das ist aber noch kein Garant für Qualität, wenn man an unvollständige, unstrukturierte und qualitativ schlechte Tests denkt. Allerdings zwingt TDD den Tester, sich intensiv mit der Spezifikation der zu testenden Komponente auseinanderzusetzen, anstatt sich fälschlicherweise nur am Code zu orientieren, wie der Erfahrungsbericht am Ende von Abschnitt 6.13.1 zeigen wird.

Ein Absatz des agilen Manifests schlägt folgende Wertschätzung vor: »Working software over comprehensive documentation«. Das bedeutet aber keinesfalls den Verzicht auf die Dokumentation von Anforderungen, die die Basis für Systemtests darstellen. Wenn der Dokumentation der Anforderungen keine Wertschätzung zukommt, dann kann das schon sehr zermürbend werden für einen Tester: Das, was gestern noch für Team-Mitglied 1 ein Bug war, kann heute für Mitglied 2 ein Feature sein. Daher ist für einen Tester eine gewissenhafte Doku-

mentation der Anforderungen Bedingung für saubere Arbeit. Ohne diese Dokumentation wird von Testteam-Ineffizienzen von 25 bis 30 % berichtet [Black 10]. Wie auch immer der Autor zu diesen Zahlen kommt.

Auch Test Driven Development macht die Dokumentation von Anforderungen nicht überflüssig. [Heusser 07] zitiert folgende frei übersetzte, verlockende Aussage zum Thema Anforderungsspezifikation und Test in agilen Projekten:

> *»Wir implementieren Projekte iterativ. Da ist ja ein großes Software-Projekt nur eine Aneinanderreihung von kleinen. Wenn wir keine klar testbaren Akzeptanzkriterien für das Arbeitspaket unserer nächsten zwei Wochen haben, dann haben wir ein großes Problem. Wenn wir es können, warum repräsentieren wir diese Kriterien nicht gleich direkt in Form von Testfällen, anstatt später aus den Anforderungen Testfälle abzuleiten?«*

Die Verlockung ist wahrlich groß: sich Anforderungsdokumentation zu ersparen und trotzdem Testfälle parat zu haben! Es gibt viele Gründe, dies *nicht* so zu tun. Abschnitt 2.4 wird die meisten davon nennen. Ein Beispiel ist die Wichtigkeit der Dokumentation von Hintergrundwissen rund um die Anforderungen. Also das Verstehen, *wieso* der Kunde etwas so wünscht und nicht anders. Ein Tester, der sich durch Lesen der Anforderungsdokumentation in die Kundensicht versetzen kann, macht die besten Akzeptanz-Testfälle. Ein Tester, der die folgenden Testfälle anstelle einer Anforderung von einem Kollegen zu übernehmen hat, versteht sicherlich nicht ohne nachzufragen, was der Kunde eigentlich benötigt:

Input	Erwarteter Output
32	0
100	38
212	100

Und was ist, wenn niemand da ist, den man fragen könnte? Dokumentation ist eine Art Sicherheitsgurt gegen Mitarbeiterfluktuation und Erinnerungslücken. Das gilt für den Test genauso wie für die Entwicklung. Wenn – im idealen agilen Projekt – immer jemand da ist, der Bescheid weiß, Zeit hat und kooperationswillig ist, dann darf man auch beim Test die Dokumentation einsparen. Solche Projekte gibt es aber nur im Märchen. Je komplexer und größer ein Projekt ist und je eher man Personen im Team hat, die nicht immer verfügbar sein werden, desto wichtiger wird die Dokumentation von Anforderungen *und* den daraus abgeleiteten Testfällen. Wenn man für das Produkt haftet, dann gilt ein nicht sauber dokumentierter Test als nicht gemacht, wie Kapitel 19 zeigen wird.

1.5.14 Wer testet die Tester?

Manche Leute stellen sich die Frage, ob man auch *unabhängige* Tests von projektspezifischer *Testsoftware* macht. Also unabhängige Tests der Tests. Diese Frage kann man generell mit »Nein« beantworten. Die zu testende Software ist der Prüfstein für den Test. Geht der Test nicht durch, so gibt es zwei Möglichkeiten (wenn wir davon ausgehen, dass die Prüfmittel fehlerfrei sind): Entweder der Test ist fehlerhaft oder die zu testende Software ist es.

Das heißt aber nicht, dass man sich nicht anderer Verifikationsmethoden bedient, bei denen ein weiteres Augenpaar mitwirkt. Bei hohen Integritätsanforderungen an die Software ist es durchaus üblich, Folgendes durchzuführen:

- Review der Unit-Tests, wenn diese vom Entwickler selbst gemacht wurden
- Review der Integrationstests, wenn diese von einem einzigen Entwickler entworfen/durchgeführt wurden
- Review des Systemtestdesigns gegen die Anforderungsspezifikation
- Review der Systemtestskripts gegen das Systemtestdesign (bei automatischen Tests)

In Abschnitt 1.5.7 wurde schon eine andere Methode erwähnt, um die Testqualität zu verifizieren: das absichtliche Einbauen von Fehlern in das Testobjekt und das anschließende Prüfen, ob der Test diese auch findet. Wenn man den Quellcode ändert, um Fehler einzubauen, spricht man von *Fault Seeding*. Wird das zu testende System im Betrieb manipuliert, um fehlerhaft zu sein, spricht man von *Fault Injection*. Die systematische Anwendung dieser Methoden ist in nur wenigen Projekten finanziell zu rechtfertigen und hat sich daher nicht wirklich durchgesetzt. Trotzdem ist es eine gute Idee, wenn man als Entwickler von Testskripts gelegentlich prüft, ob das entwickelte Testskript auch mit Fehlern des Testobjekts wie erwartet umgeht.

4 Automatische statische Code-Analyse

Die Software-Code-Review, also die Analyse des Quellcodes durch eine vom Programmautor verschiedene Person, ist eine der wichtigsten Verifikationsmaßnahmen in der Software-Entwicklung mit hohem Integritätsanspruch. Zwischen 30% und 50% aller Programmfehler werden vor der ersten Ausführung gefunden, zumindest bei »klassischen« Programmiersprachen [Fagan 86]. Leider benötigt eine gute Review aber viel Zeit. Einige Software-Werkzeuge versprechen, automatisch in wenigen Sekunden Reviews durchzuführen. Dieses Kapitel diskutiert, was von solchen Versprechungen zu halten ist, und erläutert (beispielhaft an der Programmiersprache C), wie diese Werkzeuge funktionieren.

4.1 Motivation zum Einsatz von Analysewerkzeugen

Kostet Sie ein Programmfehler viel Geld, so werden Sie nach dieser Erfahrung in zukünftigen Code-Reviews den Quellcode dahingehend prüfen, dass so ein Fehler nicht nochmals unbemerkt das Haus verlässt. Damit Kollegen auch von Ihrer Erfahrung profitieren, erfassen Sie vielleicht solche Fehler in einer Liste. In zukünftigen Code-Reviews wird dann die Liste vom jeweiligen Reviewer zur Hand genommen. Je länger die Liste, desto besser für die Qualität der Review, desto langweiliger für den Reviewer und desto teurer, weil zeitaufwändiger.

Die Idee, im Quellcode nach Fehlermustern mit einem Werkzeug automatisch zu suchen, liegt also fast auf der Hand und ist daher denkbar alt: *Lint*, ein Hilfsprogramm aus der Unix-Werkzeugkiste, tut genau das seit Ende der 1970er. Aus dem Lint-Handbuchtext: »Lint versucht Eigenheiten eines C-Programms zu entdecken, die aller Wahrscheinlichkeit nach Fehler sind, nicht portierbar sind oder unnötig. Es prüft die Typen von Variablen strikter als viele Compiler, findet unerreichbare Anweisungen, unbenutzte Variablen und logische Ausdrücke, die einer Konstanten entsprechen. Darüber hinaus prüft Lint, ob die Rückgabewerte von Funktionen auch tatsächlich verwendet werden.«

Basierend auf der Idee von Lint existieren heute Werkzeuge, die Quellcode nach verdächtigen Programmzeilen durchsuchen und den Programmierer warnen, wenn sie fündig werden. Diese Werkzeuge existieren längst nicht mehr für C

alleine, sondern für eine große Zahl von Programmiersprachen. Das Spektrum an Werkzeugen reicht von Freeware über etwas mächtigere kommerzielle Programme im Verkaufswert von 100 bis 1000 Euro bis hin zu High-End-Produkten, teilweise weit jenseits der 5000 Euro. Produkte im Hochpreissegment prüfen häufig auch einen, zumindest zum Teil, frei definierbaren Software-Coding-Standard.[1]

Was diese Tools für uns tun, wird *automatische statische Code-Analyse* genannt. Statisch deshalb, weil die zu prüfende Software dabei nicht ausgeführt wird. Zur statischen Analyse gehört auch das Erheben von Code-Metriken. Diese Maßzahlen versuchen, eine gewisse Aussage über die Qualität von Quellcode zu geben.

Die folgenden drei Unterkapitel widmen sich dem Thema Bugfinding durch Analysewerkzeuge, der darauf folgende Abschnitt 4.5 beschäftigt sich dann mit der Erhebung der besagten Metriken.

4.2 Techniken von Analysewerkzeugen im unteren Preissegment

Wie eingangs erwähnt, ist das Prüfen jeder Programmzeile nach einer Liste von potenziellen Fehlermustern eine ermüdende Angelegenheit. Speziell »kleine« Fehler, wie eine eventuell ungewollte implizite Typenkonversion werden leicht übersehen. Genau dies ist aber die Stärke einer automatischen Analyse. Die Nachfolger von Lint ermüden auch nach tausenden Codezeilen nicht und decken solche Gefahrenstellen auf. Listing 4–1 gibt eine Kostprobe. Das C-Programm ist voll von Fehlern, die automatisch erkannt werden können.

```
/* Programmiersprache C */

#include <string.h>

char* MyName();                                  /* 01 */
char cUartData = (char) 0xFA;                    /* 02 */


main()
{
   unsigned uiA = (unsigned) cUartData;          /* 03 */
   int i, a[10] = {0,0,0,0,0,0,0,0,0,0};         /* 04 */
```

1. Ein Software-Coding-Standard ist eine Vereinbarung zur Verwendung einer Programmiersprache. So ein Standard definiert zum Beispiel das Verbot der Verwendung bestimmter Sprach-Konstrukte (wie `goto` und `longjmp`) oder die Einhaltung bestimmter Richtlinien zur Namensgebung von Variablen. Die wohl bekanntesten Coding-Standards sind von der Motor Software Reliability Association (MISRA) für die Programmiersprachen C und C++. Nachdem diese Coding-Standards weit verbreitet sind, gibt es auch sehr kostengünstige Werkzeuge, die die Einhaltung der MISRA-Regeln überprüfen.

```
for (i = 0; i <= 10; i++)                    /* 05 */
   {
      a[i] += i;                             /* 06 */
   }
   for (i = 0; i <= 3; i++);                 /* 07 */
   {
      char szString[10];                     /* 08 */
      a[i] -= cUartData;                     /* 09 */
      strcpy(szString, MyName());            /* 10 */
   }
}


char* MyName(int i)
{
   char name[11] = "Joe Jakeson";            /* 11 */
   if (i) name[2] = 'i';                     /* 12 */
   return name;                              /* 13 */
}
```

Listing 4–1 *Code zur Demonstration eines einfachen Analysewerkzeugs*

Ein leistungsfähiges kommerzielles Programm zur statischen Code-Analyse hat am Programm in Listing 4–1 einiges auszusetzen: In Zeile 1 wird zu Recht gewarnt, dass die Deklaration des Prototyps der Funktion `MyName()` unvollständig ist. Für einen C++-Compiler ist in Zeile 1 das Parameterprofil für `MyName()` genau definiert: keine Parameter, also `void`. In C ist dem nicht so: `void` muss explizit in Klammern stehen, wenn die Funktion keine Parameter erwartet. Bei einer Deklaration wie in Zeile 1 sind die Parameter der Funktion undefiniert und der C-Compiler muss eine Annahme machen, sobald er einen Funktionsaufruf übersetzen muss. Nicht alle Compiler warnen den Benutzer, wenn sie so eine Annahme treffen müssen.

Auch Zeile 3 ist tückisch. Ein gutes Analysewerkzeug würde hier einen verdächtigen Type-Cast melden. Bei der Umwandlung des Typs `char` in den Typ `unsigned` wird nämlich zuerst implizit `char` in `int` gewandelt. Bei Architekturen, bei denen der Typ `char` vorzeichenbehaftet ist, wird dabei das in diesem Fall gesetzte Vorzeichenbit von `cUartData` auf die neue Bitbreite erweitert. Die Variable `uiA` enthält dann zum Beispiel den Wert `0xFFFFFFFA` und nicht `0xFA`, wie es der Programmierer vermutlich wollte.

Zählen Sie bei Reviews von Feld-Initialisierungen wie in Zeile 4 immer nach, ob tatsächlich die richtige Anzahl von Variablen initialisiert ist? Ein Analysewerkzeug tut das. In Zeile 4 fehlt eine Null. Das Werkzeug kann sogar Zugriffe mit unerlaubten Indizes erkennen, sofern dies durch Auswertung der Zahlenliterale und Konstanten möglich ist. In Listing 4–1 ist das der Fall. In Zeile 6 werden die Indexgrenzen einer Feldvariablen überschritten und diese Überschreitung kann automatisch erkannt werden.

In Zeile 7 sehen wir einen Fehler, dessen Erkennung für Lint-Derivate ein leichtes Spiel ist, der aber trotzdem häufig vorkommt und bei manueller Inspek-

tion nicht so schnell auffällt: Der Strichpunkt am Ende der Schleifenanweisung ist offensichtlich nicht gewollt.

Die Genauigkeit und Pedanterie einer programmunterstützten Analyse von Quellcode raubt manchen Programmierern viel Energie, wenn sie erstmals ein penibles Werkzeug an bislang ungeprüftem Quellcode ausprobieren. Auch in Listing 4–1 ist ein Fehler enthalten, der keine Auswirkung hat und dennoch von einem Analyseprogramm bemängelt würde: Der Rückgabetyp der Funktion `main()` ist nicht explizit definiert. Gemäß C-Standard muss der Compiler daher annehmen, dass der Rückgabewert vom Typ `int` ist. In `main()` ist aber keine `return`-Anweisung zu finden. Die Praxis, den Rückgabetyp des Hauptprogramms nicht explizit anzugeben, wenn `main()` keinen Wert an das Betriebssystem liefert, ist aber weit verbreitet. Schon beim berühmten »hello world«-Programm von den C-Veteranen Kernighan und Ritchie, in Listing 4–2 gezeigt, ist das so. Streng genommen hat dieses Programm nach heutigen C-Standards aber zwei Fehler [Stroustrup 02].

```
main()
{
   printf("Hello, world\n");
}
```

Listing 4–2 *Das Hello-World-Programm*

Analysewerkzeuge im unteren Preissegment neigen dazu, den Benutzer mit unnötigen Warnungen zu überhäufen. Als Abhilfe können in der Regel Warnungen Zeile für Zeile durch spezielle Kommentare unterdrückt oder global durch Setzen eines Parameters ausgeschaltet werden.

4.2.1 Sprachspezifische Fallstricke

Die Programmiersprache C spannt einige Fallstricke für den Programmierer, die die Portierbarkeit von Code beeinträchtigen. Da wären unter anderem die undefinierte Auswertungsreihenfolge bei arithmetischen Ausdrücken, compilerabhängige Ergebnisse beim Rechts-Shift von negativen ganzen Zahlen und die undefinierte Bitbreite des Datentyps `int`. Nach diesen Stolperfallen sucht eine automatische Analyse im Code. Auch die unsichtbare Nullterminierung von Strings hat schon so manchen Programmfehler ausgelöst. Über diesen Fallstrick stolpert übrigens Zeile 11 von Listing 4–1: Die Feldvariable `name` reserviert nicht genug Speicher, um die Endmarkierung `'\0'` zu speichern.

Der letzte, automatisch erkennbare Fehler von Listing 4–1 ist in Zeile 13 zu sehen. Es wird versucht, als Rückgabewert eine Variable zu verwenden, die mit dem Verlassen der Funktion nicht mehr existiert. Dies ist eine Art von Fehler, die auch von vielen Compilern gemeldet wird.

4.2.2 Kontrollflussanalyse

Die automatische Überprüfung von Quellcode macht es auch möglich, toten Code und unbenutzte Variablen auszuforschen. Speziell bei der Übernahme von Code aus anderen Projekten kann das sehr hilfreich sein. In Listing 4–1 gibt es übrigens zwei Variablen, die niemals referenziert werden.

Die *Kontrollflussanalyse* genannte Analysetechnik kann aber mehr, als nur toten Code aufzuspüren. Es werden die verschiedenen möglichen Ausführungspfade auf eine Reihe von potenziellen Unstimmigkeiten untersucht. Die meisten Tools erstellen für die Suche nach Kontrollflussanomalien einen Kontrollflussgraphen. Die Knoten des Kontrollflussgraphen sind Basisblöcke; das sind Code-Abschnitte, in die die Flusssteuerung an genau einem Einstiegspunkt eintritt und ohne möglichen Halt oder Verzweigung nur an genau einem Punkt wieder austreten kann [Pezzé 09]. Zwei typische Warnungen, die aufgrund der Analyse dieser Graphen ausgegeben werden, zeigt Listing 4–3.

```
/* Programmiersprache C */

#include <stdio.h>
extern int foo(void);

int sign(int x)
{
   if (x>=0)
   {
      return 1;
   }
   else
   {
      return -1;
   }
   return 0; /* Warnung: unerreichbarer Code */
}

int main(String args[])
{
   int i = foo();
   switch (i)
   {
      case 1: printf("Fall 1");
      break;
      case 2: printf("%d", sign(12));
      break;
      case 3: printf("%d", sign(-8));
      case 4: printf("Fall 4");
      break;
   }
   /* Warnung: ein case-Statement ohne break */
   return 0;
}
```

Listing 4–3 *Warnung bei Kontrollflussanomalien*

Während die erste Warnung definitiv auf Unsinn hinweist, kann die zweite Warnung durchaus ein Falschalarm sein und sollte in diesem Fall im Werkzeug *für diese spezielle Zeile* abgeschaltet werden, damit man nicht bei jedem neuerlichen Check damit belästigt wird. Es ist auch gute Praxis, im Code durch einen Kommentar darauf hinzuweisen, wenn man absichtlich einen `case`-Block ohne `break`-Statement beendet.

4.2.3 Datenflussanalyse, Initialisation Tracking

Das Beispiel in Listing 4–1 demonstriert eine Reihe von Fähigkeiten der automatischen Code-Analyse, bei Weitem aber noch nicht alle, zum Beispiel die Suche nach der Verwendung uninitialisierter Variablen. Im einfachsten Fall wird dabei geprüft, ob im Quellcode für jede Variable vor ihrer ersten Verwendung eine Zuweisung existiert. Listing 4–4 zeigt ein Programm, das mit dieser einfachen Technik nicht ausreichend geprüft wird: Den Variablen a, b und c wird zwar ein Wert zugewiesen, die Zuweisungen sind aber an Bedingungen geknüpft.

```
/* Programmiersprache C */

void oje(void)
{
   int a,b,c,m,n,p;

   scanf("%d", &a);

   if (a) b = 6; else c = b;
   a = c;

   while (n--)
   {
      /* ... */
      m = 6;
      /* ... */
   }
   p = m;
}
```

Listing 4–4 *Jede Menge uninitialisierte Variablen*

Auch wenn das Programm syntaktisch korrekt ist, warnen die meisten Compiler bei einer Verwendung von uninitialisierten Variablen. Doch häufig ist das oben beschriebene einfache Verfahren implementiert, das bei Listing 4–4 nicht greift. Während ein Compiler also in Listing 4–4 möglicherweise *keine* Fehlerquelle meldet, findet ein statisches Code-Analyseprogramm *vier* Stellen, wo auf eine uninitialisierte Variable zugegriffen wird: `b` wurde nicht initialisiert, `c` ist wahrscheinlich nicht initialisiert, `n` ist nicht initialisiert, `m` ist wahrscheinlich nicht initialisiert. Der Grund für die Überlegenheit der Spezialsoftware gegenüber der Überprüfung des Compilers liegt im intelligenteren Algorithmus, der Bedingungen verfolgt, und in

der höheren Anzahl der Durchläufe, mit der der Code analysiert wird. Das Prüfprogramm verfolgt, wie sich ein Fehler fortpflanzt.

Beim Schreiben dieses Texts hat der Buchautor zwei Compiler an Listing 4–4 versucht. Die Verwendung von `n` als nicht initialisierter Schleifenzähler wurde von beiden Compilern nicht erkannt. Der Grund dafür ist, dass nicht `n` selbst, sondern ein arithmetischer Ausdruck als Schleifenbedingung gewählt wurde. Ein gutes Analyseprogramm lässt sich davon nicht irritieren und erkennt auch, dass durch die Schleifenbedingung die Variable `m` möglicherweise nie initialisiert wird.

4.2.4 Datenflussanalyse, Value Tracking

Eine weitere Möglichkeit Fehler durch das Verfolgen von Zahlenwerten im Quellcode zu finden, ist die Analyse von Indizes von Feldvariablen sowie das Erkennen der Dereferenzierung eines Null-Zeigers und von Divisionen durch Null. Im Gegensatz zu den bisherigen Beispielen im vorherigen Abschnitt 4.2.3 beschränkt sich diese Analyse nicht auf Initialisierungswerte alleine. Listing 4–5 zeigt einige Beispiele. In der ersten Funktion kann ein intelligentes Analyseprogramm aus der Bedingung der `if`-Anweisung erkennen, dass die Variable `k` größer oder gleich 10 sein könnte. Die Verwendung als Index für eine Feldvariable mit der Dimension 10 führt daher zu einer Warnung.

In der zweiten Funktion passiert Ähnliches. Die Bedingung der `if`-Anweisung lässt den Schluss zu, dass die Zeigervariable auf Null zeigen kann und daher die beiden nachfolgenden Anweisungen Probleme bescheren können.

```
/* Programmiersprache C */

int a[10];

int eins(int k, int n)
{
   if (k >= 10) a[0] = n;
   return a[k];           /* Warnung: k könnte 10 sein */
}

int *zwei(int *p)
{
   if (p) printf("\n");   /* p könnte NULL sein ... */
   printf("%d", *p);      /* Warnung */
   return p + 2;          /* Warnung */
}

void drei(void)
{
   int a[10], *p, *q;
   p = a + 10;            /* OK, nicht unüblich */
   *p = 0;                /* Warnung */
   q = p + 1;             /* Warnung */
   q[0] = 0;              /* Warnung */
}
```

```
int x(int n)
{
   return n - 1;
}

int y(int n)
{
   return n / x(1);       /* Warnung nach 2. Durchlauf */
}

int z(int n)
{
   if (n > 0) n = 0;
   else if (n <= 0) n = -1;   /* Warnung */
   return n;
}
```

Listing 4–5 *Beispiele für automatisch erkennbare, datenabhängige Programmierfehler*

Eine gute Analyse wertet arithmetische Ausdrücke mit Zeigervariablen aus, sofern dies möglich ist. Somit wird in der dritten Funktion erkannt, dass bei den zwei in Listing 4–5 gezeigten Zuweisungen je eine Speicherzugriffsverletzung stattfindet beziehungsweise der Zeiger `q` nach der letzten Zuweisung auf eine undefinierte Stelle im Speicher zeigt.

Als vorletztes Beispiel zum Thema Value Tracking zeigt Listing 4–5 die beiden Funktionen `x()` und `y()`, wo zu sehen ist, warum ein Werkzeug zur automatischen Code-Analyse den Quelltext eines Programms in mehreren Durchläufen analysiert, wie zuvor erwähnt. Im ersten Durchlauf lernt das Analyseprogramm, dass `x()` mit dem Argument 1 aufgerufen wird. Im zweiten Durchlauf wird, wenn die Definition von `x()` erneut verarbeitet wird, gefolgert, dass dieses Argument einen Funktionsrückgabewert von Null zur Folge hat. Wenn im zweiten Durchlauf dann `y()` abgearbeitet wird, wird die Gefahr der Division durch Null erkannt und gemeldet.

Beachten Sie, dass das letzte Beispiel die Auswertung der Funktion erfordert und eine funktionsübergreifende Analyse darstellt. In manchen Analyseprogrammen kann sogar die Anzahl der Durchläufe eingestellt werden. Mit einer höheren Anzahl an Durchläufen arbeitet sich das Werkzeug tiefer in einen möglichen Aufrufbaum. Damit ist die Chance größer, auch bei Rekursionen noch Probleme zu erkennen.

Als kleine Zugabe erlaubt Value Tracking auch unsinnige Bedingungen zu erkennen, wie die Warnung zur zweiten `if`-Bedingung der Funktion `z()` in Listing 4–5 zeigt.

4.2.5 Semantische Analyse

Auch Analysewerkzeuge aus dem unteren Preissegment wissen über die Semantik von Funktionsparametern und Rückgabewerten von Funktionen der Standardbi-

bliothek Bescheid. Mit diesem Wissen kann die zuvor vorgestellte Technik erweitert werden, um Plausibilitätstests für die Parameterübergabe durchzuführen. Ein Beispiel dazu ist in Listing 4–6 zu sehen. Warnung 1 kann nur ausgegeben werden, weil das Analyseprogramm weiß, dass `fopen()` eine spezielle Funktion ist, deren erstes Argument nicht Null sein darf. Die Analyse des Ausdrucks vor dem Funktionsaufruf zeigt aber, dass sie Null sein könnte, sonst hätte die Abfrage auf Null keinen Sinn.

Warnung 2 wird gemeldet, weil über `fopen()` noch eine zweite semantische Eigenheit bekannt ist: Der Rückgabewert kann Null sein. Daher könnte `fclose()` auch mit einem ungültigen Parameterwert aufgerufen werden.

In Listing 4–6 ist im Kommentar auch der Prototyp der Funktion `fread()` zu sehen. Zusätzlich zur Prüfe-auf-Null-Semantik für das zweite und dritte Argument können Werkzeuge hier einen komplexeren Test ausführen: Wenn die Größe des zweiten Parameters multipliziert mit dem dritten Parameter die Größe des als ersten Parameter übergebenen Puffers übersteigt, dann wird ein Fehler gemeldet. Das ist im Listing als Warnung 3 markiert. Moderne Werkzeuge haben eine ganze Reihe von Semantiken für Standardfunktionen hinterlegt und gestatten auch die Definition von Semantiken für selbst geschriebene Funktionen.

Noch ein paar Beispiele für mögliche semantische Prüfungen für Funktionen der C-Standard-Bibliothek:

- Ein Aufruf der Funktionen `abort()` und `exit()` kehrt nicht mehr zurück. Danach stehender Code ist potenziell toter Code.
- Für `fgets()` wird geprüft, ob die Größe des zweiten Parameters die Puffergröße des ersten überschreitet.
- Der Zeiger, der der Funktion `free()` übergeben wird, ist nach dem Funktionsaufruf uninitialisiert und darf daher nicht dereferenziert werden.
- Nach dem Aufruf von `fclose()` ist das Argument der Funktion ein uninitialisierter Zeiger.

```
/* Beispiel semantischer Fehler in einem C-Programm */

void semantik(char *name)
{
   char buf[100];
   FILE *f;

   if (name == 0) printf("ok\n");
   f = fopen(name, "r"); /* Warnung 1: name könnte 0 sein */

   /* char *fread(char *, size_t, size_t, FILE *); */
   fread(buf, 100, 2, f);                  /* Warnung 3 */

   fclose(f);          /* Warnung 2: f könnte NULL sein */

}
```

Listing 4–6 *Beispiele für Fehler, die mithilfe semantischer Regeln entdeckt werden*

4.2.6 Starke Typenprüfung

Die Programmiersprache C erlaubt zwar die Definition von Typen als Basistypen (wie etwa Typ Strom als `float`, Typ Spannung als `double`), doch bei der Typenprüfung beschränkt sich der C-Compiler auf die Basistypen anstelle der Eigendefinitionen. Somit würden in diesem Beispiel Volt und Ampere ohne jede Warnung addiert werden. Deshalb werden die nur mit `typedef` definierten Typen auch schwache Typen genannt. Wären Strom und Spannung aus dem vorherigen Beispiel starke Typen, so wäre eine Addition der beiden Größen nur mit Typenkonversion möglich.

Für den Einsatz von Software in Geräten mit Sicherheitsrelevanz empfiehlt die Norm EN 61508 dringend die Verwendung einer Sprache mit starker Typenprüfung. C ist zwar für eingebettete Systeme häufig im Einsatz, doch C unterstützt die starke Typenprüfung nicht. Damit C dem Stand der Technik entsprechend für sicherheitsrelevante Geräte eingesetzt werden kann, muss daher ein Analyse-Tool verwendet werden, das dieses Manko kompensiert und eine starke Typenprüfung anbietet.

Erfahrungsbericht mit Statischer Code-Analyse

Ich habe für ein Unternehmen getestet, das Firmware in C für einen Mikrocontroller mit 16 KB ROM entworfen hat. Die Firmware war bereits in mehreren hundert Geräten am Markt und gehorchte keinem Coding-Standard, sondern war ein eher hässlicher Hack. Im Rahmen einer Produkterweiterung setzten wir PC-Lint ein, ein statisches Analysewerkzeug im unteren Preissegment. Wir erhielten mehr als 2000 Warnungen. Etwas mehr als eine Woche habe ich gebraucht, um alle Warnungen durchzusehen, den Code im Bedarfsfall zu korrigieren oder die Warnungen durch spezielle Kommentarzeilen zu unterdrücken. Es blieben nur 2 Warnungen, die tatsächlich Probleme hätten bereiten können. Zum einen ein Type-Cast, dessen Ergebnis Compiler-abhängig war. Wir hatten nicht vor, die Plattform zu ändern, also war das nicht relevant für uns. Das zweite Problem war ein nicht beachteter Rückgabewert der Schreibroutine eines EEPROM-Treibers. Hätte die Routine einen defekten Block erkannt, so hätte sie einen Fehlercode zurückgegeben und es wäre ein nochmaliger Aufruf notwendig gewesen. Dieser nochmalige Aufruf unterblieb aber im aufrufenden Modul, weil der Return-Wert nie geprüft wurde.

4.3 Techniken von Analysewerkzeugen im oberen Preissegment

In diesem Unterkapitel wird beschrieben, welche Fähigkeiten statische Analysewerkzeuge im oberen Preissegment anbieten können. Eine sehr gelungene Gegenüberstellung von drei Werkzeugen ist in [Emanuelsson 08] zu finden. Die in diesem Aufsatz genannten Fähigkeiten der Produkte werden hier komprimiert vorgestellt, mit den Eigenschaften eines vierten Produkts ergänzt und die Beschreibung um Produktfeatures aus dem Jahr 2012 erweitert.

4.3.1 Größerer Komfort für den Benutzer

Wenn man ein preisgünstiges Werkzeug zur statischen Code-Analyse erstmals für Code verwendet, so wird das Werkzeug typischerweise eine sehr große Anzahl von Warnungen ausgeben. Der größte Teil der Warnungen wird auf syntaktische Eigenheiten des Codes zurückzuführen sein. Gehorcht die analysierte Software einem Coding-Standard, so kann das Werkzeug im Regelfall so parametrisiert werden, dass es die Eigenheiten des Coding-Standards berücksichtigt und sich die Anzahl der gemeldeten Warnungen damit drastisch reduziert. Gehorcht der Code keinem einheitlichen Coding-Standard, so bedeutet es oft sehr viel Mühe, alle Warnungen zu prüfen und die mehrheitlichen Falsch-Alarme von den wenigen echten Fehlern zu unterscheiden. Es ist den Anwendern von statischer Code-Analyse daher ganz dringend zu empfehlen, von Anbeginn des Entwicklungsprojekts ein Analysewerkzeug einzusetzen und dieses so oft zu verwenden, wie den Compiler. Ein solches Werkzeug ist dann sinnvoll eingesetzt, wenn es keine Fehler meldet, bevor der Code irgendeinem weiteren Verifikationsschritt (Code-Review, Unit-Test …) unterzogen wird.

Wird in einem Projekt automatische statische Code-Analyse mit günstigen Werkzeugen zu spät eingeführt und existiert also schon eine Menge Code, der im schlimmsten Fall keinem Coding-Standard (keinen Programmier-Richtlinien) gehorcht, so sind tausende Falschwarnungen keine Seltenheit.

Ein Analysewerkzeug einzusetzen und seine Warnungen zu ignorieren, macht keinen Sinn. Die Durchsicht von vielen Meldungen (und vielen Falschwarnungen) kostet aber viel Zeit und ist keine besonders spannende Aufgabe. In solchen Situationen ist die Gefahr daher real, dass die Akzeptanz des Entwicklerteams für den Einsatz eines Code-Analysewerkzeugs gering ausfällt und die Einführung von automatischer statischer Code-Analyse nicht an technischer Umsetzbarkeit, sondern an »weichen Faktoren« scheitert.

Werkzeuge im oberen Preissegment haben daher Filter eingebaut, die aus den vielen Meldungen des Basis-Algorithmus die heraussuchen, die tatsächlich relevant sein dürften. Dieses Expertensystem kann natürlich nicht garantieren, dass dabei nicht auch echte Fehler weggefiltert werden. Durch die Filterung ist aber bei einem verspäteten Projekteinsatz mit deutlich weniger Akzeptanzproblemen beim Team zu rechnen als bei kostengünstigen Werkzeugen. Im Idealfall kann man die Filter konfigurieren.

Was ebenfalls zur verbesserten Akzeptanz beitragen kann, ist das Speichern von »bekannten Warnungen« in einer Datenbank und die Möglichkeit, diese wegzufiltern. Wenn alter Code-Bestand übernommen wird und dieser nicht verändert werden darf, ist es auf diese Art sehr einfach möglich, die Analyse nur auf neue Code-Teile zu beschränken.

Auch bieten hochpreisige Werkzeuge zum Teil eine Integration mit Konfigurationsmanagement-Werkzeugen an. Das Analysewerkzeug sagt zu jedem gefundenen Fehler in welchen Versionen der Software er auftritt. Diese Eigenschaft ist

für einen (zu) späten Einsatz des Analysewerkzeugs im Projekt komfortabel. In Projekten mit diszipliniertem Einsatz von automatischer statischer Analyse sollte es nie so weit kommen, dass ein automatisch erkennbarer Fehler überhaupt versioniert wird (also in das Konfigurationsmanagement-Werkzeug eingecheckt wird).

4.3.2 Concurrency Checks

Auch die Palette an durchgeführten Analysen vergrößert sich bei hochpreisigen Werkzeugen. Zum Beispiel wird untersucht, ob bei nebenläufigem (*multi-threaded*) Code doppelte Locks existieren oder Lock-Releases fehlen. Ebenso wird geprüft, ob bei multiplem Locking die Reihenfolge der Lock-Releases umgekehrt zu den Lock-Obtains ist. Durch diese Checks können einfache Dead-Locks abgefangen werden. Mehr zu Locks und Deadlocks in Kapitel 10 und in Kapitel 11.

Auch ist es mit statischer Analyse möglich, mit einer gewissen Treffsicherheit Data Races vorherzusagen. Dass das Erkennen von Data Races für den Benutzer bequem und zuverlässig funktioniert, ist auch im Interesse von Herstellern von Multi-Core-CPUs. Daher sind hier sehr komfortable Werkzeuge am Markt, die völlig gratis sind. Das passt nicht ganz zum Titel dieses Unterkapitels, »Techniken von Analysewerkzeugen im oberen Preissegment«. Wir beschränken uns daher hier auf die Erwähnung der Möglichkeit so einer Analyse und auf einen Querverweis auf Kapitel 10, das sich ausführlich mit Data Races beschäftigt.

4.3.3 Stack-Analyse und erweiterte Kontrollflussanalyse

Mit kleinen Einschränkungen kann durch Analyse des Quellcodes auch festgestellt werden, ob die Größe des Runtime-Stacks ausreicht. So eine Einschränkung ist zum Beispiel die Abwesenheit von Rekursionen. 100 % korrekte Ergebnisse so einer Analyse kann man aber nur erwarten, wenn die Analyse nicht auf Basis des Quellcodes, sondern auf Basis des Object-Codes erfolgt. Nur dann berücksichtigt die Analyse die Eigenheiten und den Optimierungsgrad des verwendeten Compilers. Werkzeuge, die Object-Code analysieren sind manchmal proprietäre Software. So zum Beispiel für den Steuerungsrechner der Ariane-5-Rakete. Es gibt aber auch kommerzielle Werkzeuge, die den Object-Code einer ganze Reihe von Prozessoren unterstützen, zum Beispiel [URL: StackAnalyzer], [URL: GNATstack] oder [URL: IAR].

Da es möglich ist, einen Baum aller Programmzustände zu erzeugen und zu untersuchen, ist es auch möglich, ein Set aller möglichen Kontrollflüsse durch ein Programm zu erzeugen. In dieser riesigen Pfadsammlung lässt sich zum Beispiel für C++-Code durch ein Werkzeug feststellen, ob es für jede Ausnahmesituation (*Exception*) auch eine Behandlung gibt. Ebenfalls gibt es Werkzeuge, die auf diese Art mit ziemlich hoher Treffsicherheit Memory Leaks detektieren.

4.3.4 Erschöpfende Analyse des Zustandsbaums

Wenn der Baum aller möglichen Programmzustände detailliert genug ist, dann lassen sich durch Traversion dieses Baums auch Laufzeitfehler erkennen und – schöner noch – es lässt sich de facto der *Beweis der Abwesenheit von bestimmten Fehlern* erbringen. Das Analyse-Tool durchläuft also das Laufzeitmodell des zu untersuchenden Programms bis in alle Blätter des Baums (untersucht also alle erreichbaren Programmzustände) und kann dann verlautbaren, dass in diesem Baum der Fehlerzustand soundso nicht vorkommt. Die Motivation, solch ein Werkzeug auf den Markt zu bringen, kommt vom missglückten Jungfernflug der Rakete Ariane 5. Dort hatte ein Überlauf bei der Konvertierung eines Gleitkommawerts in einen Integer-Wert mehrere hundert Millionen Euro gekostet. Kein Wunder, dass Werkzeuge zur erschöpfenden Analyse des Zustandsbaums also besonders Überläufe prüfen. Unter anderem wird in dem Baum in solchen Werkzeugen nach folgenden Fehlerzuständen gesucht:

- Overflows/Underflows bei skalaren Variablen
- Overflows/Underflows bei Gleitkommawerten
- Division durch Null
- Array Index ist größer als erlaubt
- Dereferenzierung von Nullpointern

Im Unterschied zur auf Seite 61 in Absatz 4.2.4 vorgestellten Datenflussanalyse von preiswerten Werkzeugen wird hier nicht mithilfe von Heuristiken geschlossen, dass ein Fehlertyp vorhanden ist oder nicht. Wenn das Werkzeug die Abwesenheit einer Nullpointer-Dereferenzierung meldet, dann ist *tatsächlich* keine *zur Laufzeit* im Code.

Natürlich ist so eine erschöpfende Analyse des Zustandsbaums sehr aufwändig. Es sollte auch nicht überraschen, dass Werkzeuge, die diese Analyse anbieten, meist nicht gerade billig sind.

4.4 Statische Security-Analyse (SSA)

Es gibt Werkzeuge für die statische Analyse, die speziell nach Sicherheitslücken im Code suchen. Dabei ist Sicherheit im Sinne von Sicherheit gegen Angriffe von Hackern gemeint (Security). Hacker benutzen oft Programmabstürze und Pufferüberläufe für Angriffe. Die einschlägigen Werkzeuge untersuchen den Quellcode daher nach Programmfehlern, so wie im Buch bisher beschrieben, und zusätzlich analysieren sie die Verwendung von Befehlen und Design-Konstrukten, die von Hackern ausgenutzt werden könnten. In der Programmiersprache C melden diese Werkzeuge zum Beispiel eine gefährliche Verwendung von `gets` und `strcpy`.

In C ist die Software bei einer fehlenden Prüfung auf eine Null-Terminierung von Strings nicht gegen String-Overflows geschützt und damit verwundbar.

Abbildung 4–1 zeigt ein Beispiel. Das Werkzeug meldet eine potenzielle Division durch Null ebenso wie einen möglichen Pufferüberlauf.

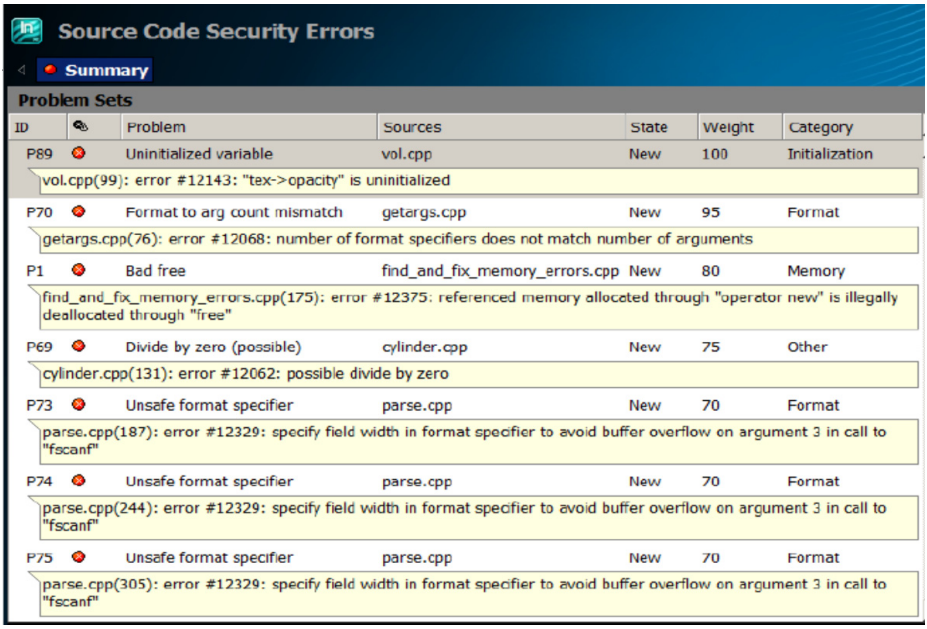

Abb. 4–1 *Ausschnitt eines Screenshots von Intel Inspector XE 2011*

In der letzten Revision des Coding-Standards MISRA-C wurden mehrere Regeln definiert, die es Hackern erschweren sollen, Schadsoftware in Steuergeräte mit Bus-Schnittstellen einzuschleusen. Die Motivation für die Definition dieser neuen Regeln war die berechtigte Sorge, dass Hacker die Personensicherheit gefährden können, wenn sie Steuergeräte in Fahrzeugen angreifen. Der Einsatz eines Werkzeugs, das die Einhaltung dieser Regeln automatisch prüft, ist daher auch ein gewisser Security-Check.

Damit schließt dieses Kapitel die Ausführungen zum Thema automatisches Finden von potenziellen Bugs und potenziellen Verwundbarkeiten und wendet sich einer automatischen Analyse zu, die verwendet wird, um die »innere Qualität« von Programmen zu bewerten.

4.5 Code-Metriken

Der ISTQB definiert *Metrik* als eine Mess-Skala plus das genutzte Messverfahren [ISTQB-D]. Das Wort Metrik entstammt dem Griechischen und bedeutet Kunst des Messens. Unter dem Begriff Code-Metrik versteht man dennoch nicht die Kunst, Quellcode zu vermessen, sondern die Maßzahl selbst. Die Motivation, diese Maßzahlen (werkzeugunterstützt) zu erheben, ist, in kurzer Zeit Eigenschaften des recht abstrakten Objekts Software quantifizieren zu können.

Eine denkbare einfache Software-Metrik ist die Zeilenanzahl einer Funktion oder einer Datei (*Lines of Code*, LOC).[2] Programme werden dank Metriken vergleichbar, zumindest dann, wenn sie in ähnlichen Programmiersprachen erstellt wurden. Aber, wie Peter Liggesmeyer in [Liggesmeyer 09] korrekterweise warnt, ist eine Aussage nur sinnvoll, wenn man mehrere Metriken gleichzeitig betrachtet. Der Blick auf eine Metrik alleine ist irreführend. So kann ein hochoptimierter FFT-Algorithmus mit 500 LOC um Kategorien aufwändiger zu schreiben und zu testen sein als 2000 LOC einer simplen Applikation.

Code-Metriken geben einen Hinweis auf die sogenannte *innere Qualität* von Software. Also die Wartbarkeit, Erweiterbarkeit, Analysierbarkeit, Testbarkeit von Code. Ohne Zweifel kann Code horrend schlecht programmiert sein, also von schlechter innerer Qualität und trotzdem alle Software-Anforderungen erfüllen. Also von guter *äußerer Qualität* sein. Es gibt aber zahlreiche Studien, die den Schluss zulassen, dass innere und äußere Qualität positiv korrelieren. Es ist daher nicht unüblich, dass Käufer von Software mit hohen Integritätsansprüchen den Nachweis des Erreichens von bestimmten Metrik-Schwellwerten einfordern (oder beim Kauf von Quellcode als ersten Schritt Code-Metriken erfassen und sie gegen Schwellwerte vergleichen). Ein solcher Vorschlag für Schranken von Metriken wird in Tabelle 4–1 vorgestellt. Die Metriken sind ein Subset der in [Kuder 08] ausgewählten Metriken. Die Schranken (beziehungsweise zur Akzeptanz vorgeschlagene Wertebereiche) stammen zum Teil ebenfalls aus [Kuder 08], zum Teil aus [ISO 26262] und zum Teil aus den Qualitätsanforderungen an die Software der GMES-Erdbeobachtungssatelliten der ESA. Die Metriken sind um wertvolle Kommentare ergänzt.

2. Einfach dann zumindest, wenn man sich darauf geeinigt hat, welche Kommentarzeilen man mitzählt oder nicht. Verschiedene Tools liefern gerade bei dieser einfachen Metrik verschiedene Ergebnisse.

Code-Metrik	Bereich	Kommentar
Anzahl der Exit-Points	1	Jede Funktion sollte genau einen Austrittspunkt haben. Bei Sprachen ohne Exception Handling kann man die Forderung auch aufweichen, um den Code einfacher zu halten: Zur Fehlerbehandlung sind mehrere Austrittspunkte gestattet, im fehlerfreien Fall gibt es nur ein einziges Function Return.
Anzahl der Sprunganweisungen	0	Die Verwendung von Goto erhöht die Anzahl der möglichen Pfade unnötig und reduziert die Testbarkeit und Wartbarkeit von Code.
Anzahl rekursiver Funktionen	0	Rekursionen haben in Software mit hohem Integritätsanspruch nichts verloren.
Anzahl der Funktionsparameter	0 bis 5	Eine hohe Anzahl von Funktionsparametern birgt ein höheres Risiko eines Integrationsproblems und verursacht großen Verwaltungsaufwand beim Aufruf der Funktion.
Verschachtelungstiefe	0 bis 4	Dies ist üblicherweise die Anzahl der Einrückungen bei Schleifen, if-Statements usw. pro Funktion. Je verschachtelter, desto fehleranfälliger.
Zyklomatische Komplexität von Funktionen	1 bis 12	Die zyklomatische Komplexität einer Funktion ist die Anzahl der Entscheidungen (Verzweigungen) plus eins. Sie ist ein Maß für den Testaufwand beim Basis Path Testing, siehe Abschnitt 6.7. Bei Funktionen mit großen switch/case-Anweisungen (zum Beispiel Nachrichten-Interpretern) ist es üblich, auch größere Werte zu akzeptieren.
Pfadanzahl pro Funktion	1 bis 80	Anzahl der nicht-zyklischen Ausführungspfade durch eine Funktion.
Anzahl dynamischer Objekte oder Variablen	0	Die [ISO 26262] gesteht ein, dass diese Forderung beim Einsatz von modellbasierter Entwicklung nicht erfüllbar sein kann.
Kommentardichte	> 0,2	Die Kommentardichte einer Funktion ist die Anzahl der Kommentarzeilen einer Funktion, dividiert durch die Anzahl der Codezeilen der Funktion.

Tab. 4–1 *Beispiele für elementare Code-Metriken*

Die Motivation für die angegebenen Schranken ist immer die Wahrscheinlichkeit für besser wartbaren beziehungsweise besser testbaren Code. Selbstverständlich kann Code, dessen Metriken alle vorgestellten Schwellwerte verletzen, fehlerfrei sein. Ebenso kann Code, dessen Metriken innerhalb aller angegebenen Schranken liegen, trotzdem schlecht zu testen und schlecht zu warten sein.

Erfahrungsbericht mit Code-Metriken

Ich habe in einem Unternehmen als Entwickler gearbeitet, das Telefonanlagen entwickelte. Das österreichische Tochterunternehmen lieferte die Firmware im C++-Quellcode an das deutsche Stammhaus, wo die Elektronik gefertigt wurde. Die erste (und vermutlich einzige) »qualitätssichernde Maßnahme«, die man im Stammhaus mit dem Firmware-Quellcode unternahm, war zu erheben, wie groß die Kommentardichte im Code war. Die österreichischen Programmierer waren alle lang gediente Mitarbeiter, die sich im Quellcode sehr gut auskannten. Sie hatten wenig Interesse daran, dass ihre Software auch von deutschen Kollegen gut verstanden wurde und der Entwicklungsstandort somit relativ leicht hätte verändert werden können. So fand ich im Code zahlreiche Kommentare der folgenden Form:

```
i = i + 1;  /* erhöhe i um eins */
```

Diese Kommentare ließen den Kommentardichte-Checker grünes Licht geben, haben aber natürlich nichts zur Wartbarkeit und Qualität des Codes beigetragen.

4.6 Werkzeuge für die Automatische Code-Analyse

Die Ausführungen in Abschnitt 4.2 basieren auf den Eigenschaften der Werkzeuge splint und PC-Lint. Vergleichbar damit ist das etwas mächtigere und modernere Werkzeug C-STAT. In Abschnitt 4.3 floss die Übermenge der Features von Klocwork-Produkten, GammaTech CodeSonar, QA-C, der Coverity Development Testing Platform, Polyspace Verifier und Astrée ein. Die letzten beiden Werkzeuge sind, wie die Namen vermuten lassen, Werkzeuge zur Exploration des Zustands[r|b]aums der Software. Alle diese Werkzeuge unterstützen zumindest die Analyse von Quellcode in C/C++.

Es gibt eine ganze Menge anderer Sprachen, für die automatische Code-Analysewerkzeuge am Markt sind. Unter anderem für Ada, Java, C#, Visual Basic, Python und sogar für IEC 61131-3 [Prähofer 12]. Erwähnenswert ist vielleicht, dass es für Java Werkzeuge gratis gibt, deren Funktionalität mit Werkzeugen im mittleren bis hohen Preissegment für C/C++ vergleichbar ist, zum Beispiel das von der NASA bereitgestellte Werkzeug »Pathfinder«.

Für manche Computersysteme ist das Thema Security, also Sicherheit vor unerlaubter und/oder böswilliger Benutzung von großer Wichtigkeit. In diesem Buch wird Security-Tests aber verhältnismäßig wenig Aufmerksamkeit gewidmet. Das bedeutet aber nicht, dass Security nicht auch im Embedded-Bereich wichtig sein kann. Alles ist Angriffsziel von Hackern und Verrückten. Wegfahrsperren für Kraftfahrzeuge werden ebenso geknackt wie die Service-Schnittstelle eines Herzschrittmachers [Halperin 08].

Werkzeuge für statische Security-Checks sind manchmal in Compilern integriert, siehe zum Beispiel Microsoft Visual Studio. An Stand-alone-Werkzeugen

wären zum Beispiel splint für C (Freeware), Fortify SPA für C, C++, C#, Java u.a., Code Assure für C, C++ und Java zu erwähnen.

Werkzeuge zur Erhebung von Code-Metriken machen im einfachsten Fall eine Auflistung von erhobenen Metriken, wie das Komplexitätsmesswerkzeug Testwell CMT++, können aber auch mit anderen Werkzeugen kombiniert sein. So erhebt zum Beispiel QA-C nicht den Anspruch, den gründlichsten Bugfinder am Markt zu haben, dafür vereint das Werkzeug eine grafische Benutzeroberfläche, Source-Code-Navigation, einen Metriken-Checker und eine passable statische Analyse zum Aufspüren von Bugs. Es versteht sich als ein Werkzeug zur Vorbereitung einer manuellen Code-Review. Die erhobenen Metriken und die grafische Aufbereitung derselben sollen helfen, bei der Review den Blick auf das Wesentliche zu konzentrieren. Auch mit freien Versionen von Microsofts Visual Studio können übrigens einige Code-Metriken berechnet werden.

Auch eine Integration von Metriken-Werkzeugen mit Unit-Test-Werkzeugen gibt es. So kann man in Cantata nicht nur Schranken für dynamische Tests definieren, sondern auch für Code-Metriken. Es ist also zum Beispiel definierbar, dass keine 100 % Unit-Tests-Finalisierung angezeigt wird, wenn die zyklomatische Komplexität von C-Funktionen einen definierbaren Schwellwert überschreitet.

Werkzeuge, die eine sehr große Anzahl von einfachen Code-Metriken erheben, diese zu komplexen Metriken verknüpfen und dann (heuristische) Aussagen über Wartbarkeit oder Testbarkeit erlauben, sind zum Beispiel McCabe IQ oder Logiscope. Einen kleinen Ausschnitt aus einem Analysebericht so eines Werkzeugs für Software sehr guter Qualität zeigt Abbildung 4–2.

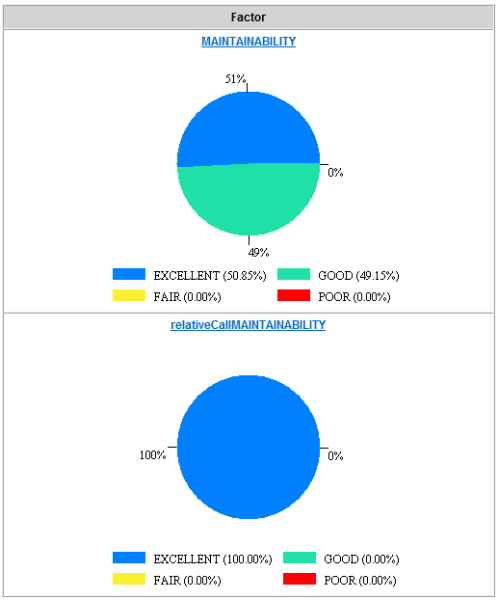

Abb. 4–2 *Ausschnitt aus einem Bericht in HTML-Form von Logiscope*

4.7 Diskussion

Wenn man vergleicht, wie schnell ein potenzieller Bug durch ein statisches Analysewerkzeug, wie in den Abschnitten 4.2 und 4.3 vorgestellt, gefunden wird, und wie lange man benötigt, solche Bugs durch Debugging-Sessions auf Systemebene zu identifizieren, dann ist man schnell davon überzeugt, solche Werkzeuge einzusetzen. Der Werkzeugeinsatz amortisiert sich relativ früh und ist bei Systemen mit Sicherheitsrelevanz nicht wegzudiskutieren. Je später der Einsatz im fortschreitenden Projekt erfolgt, desto mehr sinnlose Warnungen wird man aber beim Einsatz eines günstigen Werkzeugs prüfen müssen. Auch die Wartbarkeit von Code verbessert sich, wenn man gleich von Anbeginn des Projekts automatische statische Analyse einsetzt und danach trachtet, immer null Warnungen zu erhalten: Viele gefährliche Code-Konstrukte werden von vornherein vermieden und eine Analyse einer eventuell als gefährlich eingestuften Code-Passage zu einem späteren Zeitpunkt der Wartung kann unterbleiben.

Die Vielzahl unterschiedlicher Fehlertypen, die ein Low-Cost-Werkzeug findet, kann zwar einen kräftigen Dämpfer für einen späten Einsatz bedeuten, es können aber auch Fehler gemeldet werden, die von den Filtermechanismen der hochpreisigen Werkzeuge fälschlicherweise ausgeblendet/wegpriorisiert werden. In einer Studie [Zheng 06] wurde beschrieben, dass ein Low-Cost-Werkzeug etwa doppelt so viele Fehler wie ein hochpreisiges Produkt erkannte – sowohl hinsichtlich der Fehleranzahl wie auch in Bezug auf die Verschiedenartigkeit der Fehlertypen. Ein anderer Bericht bestätigt diese Studie. Wie groß die Schere der Anzahl der Fehlermeldungen in großen Projekten werden kann, berichtet [Emanuelsson 08]: 1,2 Millionen Defekte mit einem günstigen Tool und nur 40 Fehlermeldungen mit einem hochpreisigen Werkzeug.

Auch beim Einsatz von Werkzeugen der Art von Astrée und Polyspace muss man mit so manchem Bedienungsaufwand rechnen, wenn man den Code zum Beispiel nach potenziellen Überläufen und ähnlichen Laufzeitfehlern absucht. Der Lohn für diese Arbeit ist der Nachweis der Abwesenheit von bestimmten Fehlern. Einen Nachweis der Abwesenheit von Fehlertypen können Bugfinder-Werkzeuge nicht liefern.

Auch der gewissenhafte Einsatz von Werkzeugen kann aber eine Code-Review durch einen Menschen nicht ersetzen. Nur ein menschlicher Review-Partner kann unangepasstes Design oder die fehlerhafte Umsetzung von Anforderungen erkennen. Die Analyse des Codes durch einen Kollegen und die automatische Analyse durch ein Werkzeug sind daher großteils komplementäre Vorgänge zur Qualitätssicherung. Beide Verifikationsschritte sollten also eingesetzt werden.

4.8 Fragen und Übungsaufgaben

Frage 4.1: Die Kommentare im folgenden Listing schlagen sieben Warnungen vor. Welche dieser sieben Warnungen würde ein aggressives Werkzeug zur statischen Analyse tatsächlich melden? Welche nicht? Warum?

```
/* C-Programm */
int Hoo();                    /* 1, incomplete prototype */
char cUartData = (char) 0xFA;

main()
{
   unsigned uiA = 12;           /* 2, uiA is never used */
   int i, a[10] = {0,0,0,0,0,0,0,0,0,0};
                              /* 3, wrong number of init:s */

   for (i = 0; i <= 10; i++)
   a[i] += i;             /* 4, index exceeds buffer size */
   for (i = 0; i <= 3; i++); /* 5, suspicious semicolon */
   {
      a[i] = (int) Hoo;              /* 6, strange cast */
      a[i] -= cUartData;
      a[i] += Hoo();
   }
}                           /* 7, no return value of main */

int Hoo(int x)
{
   return x + 2;
}
```

Frage 4.2: Warum können Sie nicht mit einer Quellcode-Analyse eines Off-the-Shelf-Werkzeugs zuverlässig den Stack-Verbrauch bestimmen?

Frage 4.3: Können Sie mithilfe von Code-Metriken die Qualität einer Software-Bibliothek beurteilen, die Sie als Object-Datei (also nicht im Quellcode) erhalten haben?

Frage 4.4: Können Sie mithilfe von Code-Metriken eine hinreichend zuverlässige Aussage über die Qualität von Quellcode machen?