Experimenteller Vergleich statischer und dynamischer Softwareprüfung für eingebettete Systeme

Dietmar Ernst* Frank Houdek**
Wolfram Schulte* Thilo Schwinn**

Zusammenfassung

Im Entwicklungsprozeß von Software spielen Aktivitäten der Softwareprüfung eine wesentliche Rolle. Einerseits sind sie zur Überprüfung und Sicherstellung der geforderten Software–Qualität unerläßlich, andererseits sind mit den Prüfaktivitäten hohe Kosten verbunden. Eine sinnvolle Auswahl und Kombination der zu Verfügung stehenden Prüfverfahren kann somit gleichwohl zur Qualitätssteigerung und Kostenreduzierung beitragen.

Die Methoden der Softwareprüfung gliedern sich in dynamische und statische Verfahren. Diese Arbeit präsentiert einige Ergebnisse eines Experiments, das im Rahmen des Softwarelabor Ulm für die Daimler-Benz AG angefertigt wurde. In diesem Experiment vergleichen wir Tests und Reviews für das Umfeld eingebetteter Systeme. An der Durchführung des Experiments waren 13 Studenten beteiligt. Wichtige Ergebnisse waren, daß mit Reviews mehr Fehler entdeckt wurden als mit dem strukturellen oder funktionalen Test. Auch unter Berücksichtigung der dazu nötigen Aufwände schneidet die statische Prüftechnik deutlich besser als die dynamische ab. In Hinblick auf die bei den jeweiligen Prüfmethoden besser gefundenen Fehlerarten gab es keine signifikanten Unterschiede.

*Universität Ulm, Fakultät für Informatik, Abt. Programmiermethodik und Compilerbau, 89069 Ulm, e-mail: {dietmar, wolfram}@informatik.uni-ulm.de

**Daimler-Benz AG, Forschungszentrum Ulm, Abt. Softwaregestaltung, Postfach 23 60, 89013 Ulm, e-mail: {houdek, schwinn}@dbag.ulm.DaimlerBenz.com

Hinweis

Der Bericht enthält in der vorliegenden Form nur den Hauptteil. Der vollständige Anhang ist hier nicht abgedruckt, da er nur für eine Replikation des Experiments notwendig ist.

Falls Sie Interesse an einer Replikation des Experiments haben, sind wir gerne bereit, Ihnen die dazu notwendigen Unterlagen zu senden. Wenden Sie sich in diesem Fall bitte an Dietmar Ernst (`dietmar@informatik.uni-ulm.de`) oder Frank Houdek (`houdek@dbag.ulm.DaimlerBenz.com`).

Inhaltsverzeichnis

1 Einleitung

Software ist mittlerweile ein fast allgegenwärtiger Bestandteil unseres täglichen Lebens geworden. Waschmaschinen, Flugzeuge, Kraftwerke und vieles mehr enthalten Software als integralen Bestandteil. Von der Korrektheit der Software hängt zunehmend unser Leib und Leben ab. Somit ist klar, daß mit der Bedeutung von Software auch die Anforderungen an die Qualität derselbigen wachsen.

1.1 Experimentelles Software Engineering

Während in den klassischen Ingenieurdisziplinen wie z. B. Werkzeugbau gefestigtes Wissen existiert, wie bestimmte Eigenschaften in ein Produkt *hineinkonstruiert* werden können, ist dieses Wissen im Bereich der Softwareerstellung noch bruchstückhaft und unvollständig. Um beispielsweise einen korrosionsbeständigen, werkzeugtauglichen Stahl für Kaltarbeit zu erhalten, benötigt man eine Legierung mit 0.9 % Kohlenstoff, 0.4 % Silizium, 0.6 % Mangan und 0.4 % Chrom, der bei 1200 °C gehärtet und in Öl langsam abgekühlt werden muß. Durch Steigerung des Chromanteils läßt sich dieser Stahl hinsichtlich der Härte verbessern [Sta92].

Wesentlich unklarer ist jedoch, welche Vorgehensweise bei der Softwareerstellung angebracht ist, um ein zuverlässiges Banksystem in COBOL oder eine effiziente und fehlerfreie ABS–Steuerung in Assembler zu entwickeln.

Um dieses Wissen aufzubauen, bedient man sich eines experimentellen Ansatzes. Verschiedene Alternativen werden angewandt und hinsichtlich ihrer Eignung überprüft. Sämtliche Naturwissenschaften, aber auch viele andere Disziplinen bedienen sich dieser Vorgehensweise.

Auch im Bereich des Software Engineerings beginnt man, sich verstärkt dieser Herangehensweise zuzuweden. So teilen Rombach et. al. in [RBS92, S. V] die Forschungslandschaft des Software Engineering in drei große Bereiche ein:

Mathematisches oder theoretisches Software Engineering
: Dieser Teilbereich des Software Engineerings betrachtet den Softwareerstellungsprozeß als eine Abfolge von Transformationsschritten im weitesten Sinne. Formale Methoden, Transformationssysteme oder auch Programmverifikation finden sich in diesem Bereich.

Systembasiertes Software Engineering
: Während sich der Teilbereich des mathematischen Software Engineerings auch als „Softwareentwicklung im Kleinen“ charakterisieren läßt, beschäftigt sich das systembasierte Software Engineering eher mit „Softwareentwicklung im Großen“. Themengebiete sind hier Organisation und Abwicklung großer Softwareprojekte, Konfigurationsmanagement, Softwarearchitekturen oder Frameworks.

Experimentelles Software Engineering

Dieses Teilgebiet des Software Engineerings beschäftigt sich mit der experimentellen Erprobung der Verfahren und Ansätze, die in den anderen Teilgebieten erarbeitet wurden und werden. Ein Ziel bei der experimentellen Herangehensweise ist es, die Faktoren, die die Eignung eines Ansatzes bestimmen, zu ermitteln und so ein klareres Bild über das Zusammenspiel der einzelnen Methoden und Techniken und deren Auswirkung auf das Produkt, nämlich die Software und ihre Eigenschaften, zu gewinnen.

Die zunehmende Bedeutung dieses dritten Teilgebietes läßt sich auch an dem wachsenden Anteil von Beiträgen auf internationalen Konferenzen und Zeitschriften (vgl. [ZW97]) oder neuen Forschungsinitiativen[1] erkennen.

1.2 Experimenteller Vergleich statischer und dynamischer Softwareprüfung

Einen wesentlichen Anteil an der Software–Entwicklung nimmt die Softwareprüfung ein. So betragen die Aufwände im Bereich des Testes oft 40 % und mehr [Jon91, Boe81]. Die Bedeutung der Softwareprüfung wird auch am V–Modell [BD93, Som96] ersichtlich, in dem der gesamte rechte Ast den Prüfaktivitäten gewidmet ist.

Formale Ansätze, die eine Softwareprüfung im klassischen Sinne überflüssig machen, wie beispielsweise Programmverifikation (siehe z. B. [Rei91]) oder Programmtransformation (siehe z. B. [Par90]) sind in größeren Softwareprojekten bisher noch wenig vebreitet. Die Gründe hierfür sind sicher mannigfaltig: Probleme bei der Anwendung der Verfahren auf größere Projekte [HWW94] oder auch unzureichende Ausbildung der Entwickler in formalen Methoden [Fin96] sind zwei Beispiele hierfür.

Ein Mittelweg wird bei dem sogenannten Cleanroom–Ansatz [DM81] gegangen, bei dem während der eigentlichen Entwicklung kein Rechner verwendet wird, sondern die Entwicklung mit Hilfe (semi-)formaler Techniken und gegenseitiger Überprüfung durchgeführt wird. Testen wird im Sinne einer statistischen Kontrolle eingesetzt. Doch auch dieser Ansatz hat sich bisher in der Praxis wenig durchgesetzt.

Ein Ausweg aus diesem Dilemma ist, die anwendbaren Techniken der Softwareprüfung möglichst effizient zu kombinieren. Zu dieser Fragestellung soll unsere Untersuchung, die wir in diesem Bericht beschreiben, einen Beitrag leisten.

1.2.1 Methoden der Softwareprüfung

Die Aktivitäten der Softwareprüfung werden in dynamische und statische Verfahren aufgeteilt [FLS95]. Von dynamischer Prüfung spricht man, wenn der Prüfling bei seiner Ausführung beobachtet wird. Diese Art der Prüfung ist

[1] Als Beispiel sei die 1996 gegründete Fraunhofer–Gesellschaft für Experimentelles Software Engineering (IESE) in Kaiserslautern aufgeführt.

natürlich nur auf solche Objekte anwendbar, die eine Ausführung ermöglichen, wie beispielsweise Programmcode oder manche formale Spezifikationen. Im Falle von Programmcode wird diese Art von Softwareprüfung als Test bezeichnet.

Bei der statischen Prüfung wird der Prüfling in „ruhendem" Zustand betrachtet und analysiert. Dieses Prüfverfahren stellt eine andere Anforderung an den Prüfling, nämlich eine Lesbarkeit im weiteren Sinne. Analysedokumente, Entwurfsdokumente, Programmcode oder Dokumentationen sind typische Beispiele für geeignete Prüflinge. Weniger geeignet sind ausführbare Programme, die aus einer Abfolge von Maschinenbefehlen bestehen. Die Spanne der statischen Prüfaktivitäten ist dabei groß und reicht von der Durchsicht des Programmcodes am Bildschirm bis hin zu technischen Reviews und Inspektionen.

1.2.2 Zielsetzung unserer Untersuchung

Ausgangspunkt für unsere Untersuchung waren die Arbeiten von Basili und Selby [BS87] und Kamsties und Lott [KL95a], die statische und dynamische Softwareprüfaktivitäten in Hinblick auf den Programmcode einfacher Softwaretools[2] untersucht haben. In deren Experimenten konnte beobachtet werden, daß wenig Unterschied zwischen den Verfahren bestand, wobei die statischen Prüfverfahren ein wenig besser abschnitten.

Diese Beobachtung deckt sich auch mit vielen anderen Berichten (vgl. z. B. [GG93a]), die über positive Wirkung statischer Softwareprüfung berichten. Neben positiven Einschätzungen gibt es aber auch weniger euphorische Stimmen [PSV95].

Statische Prüfverfahren sind also kein Allheilmittel. Deren Wirksamkeit hängt, wie bei jedem anderem Verfahren auch, vor den jeweiligen Randbedingungen ab.

Im Rahmen der Produktentwicklung im Daimler–Benz Konzern spielen eingebettete Systeme eine wesentliche Rolle. Eingebettete Systeme sind integraler Bestandteil der meisten Produkte, wie Automobile, Luft- und Raumfahrtgeräte oder auch Kommunikationssysteme.

Diese Systeme zeichnen sich zumeist durch stringente Zeitbedingungen und hohe Nebenläufigkeit aus. Eigenschaften, von denen gemeinhin angenommen wird, daß sie die Lesbarkeit und Verständlichkeit erschweren. Da aber insbesondere auch bei Daimler–Benz ein hoher Anteil der Software–Entwicklungsaufwände sich auf Prüfaktivitäten konzentriert, ist es hier interessant, alternative Vorgehensweisen zu untersuchen.

Aus diesem Grund sollte im Rahmen unserer Softwarelabor–Kooperation[3] dieser Fragenkomplex im Rahmen einer empirischen Untersuchung evaluiert wer-

[2] Unter Softwaretools verstehen wir hier einfache Programme, die dem Benutzer bei der täglichen Arbeit unterstützen. Beispiele hierfür sind die UNIX–Werkzeuge `head` und `cat`.

[3] Im Rahmen von Softwarelabor–Kooperationen soll versucht werden, den Austausch von Industrie und Hochschule zu intensivieren und Studenten im Zuge ihrere Ausbildung stärker mit Fragestellungen aus der Praxis zu konfrontieren [HSS97].

den. Ziel unserer Untersuchung war es, einen quantitativen Vergleich von statischen und dynamischen Prüfverfahren in Hinblick auf Programmcode eingebetteter Systeme zu erhalten. In diesem Bericht beschreiben wir den Ablauf der Untersuchung und stellen unsere Ergebnisse vor.

1.3 Aufbau des Berichts

In Kapitel 2 gehen wir näher auf die eingesetzten Techniken der statischen und dynamischen Softwareprüfung ein. Werkzeuge, die sich bei der Anwendung dieser Techniken sinnvoll einsetzen lassen, werden wir dabei ebenfalls kurz beschreiben. Kapitel 3 enthält die Beschreibung der Experimentplanung, also unserer Hypothesen und Annahmen sowie deren Umsetzung in ein Experimentdesign. In Kapitel 4 erläutern wir die genaue Durchführung der einzelnen Abschnitte unseres Experiments, beschreiben die eingesetzten Materialien und gehen insbesondere auf unsere Beobachtungen bei der Ausführung und deren Interpretation ein. Eine kritische Betrachtung unserer Beobachtungen und Interpretationen ist Inhalt von Kapitel 5. Hier findet sich auch ein Vergleich unserer Ergebnisse mit anderen in der Literatur beschriebenen Experimenten. In Kapitel 6 fassen wir schließlich die Ergebnisse unseres Experiments kurz zusammen.

Der Anhang dieser Arbeit enthält sämtliche Materialien, die wir bei der Durchführung des Experiments eingesetzt haben. Hauptanliegen dieser Materialiensammlung ist die Replikation des Experiments an anderen Universitäten oder Forschungseinrichtungen, da eine hinreichende Verläßlichkeit von experimentellen Aussagen erst durch Replikation erreicht werden kann.

2 Grundlagen

In diesem Kapitel möchten wir die Problematik der Softwareprüfung vorstellen und die dabei relevanten Begriffe präzisieren.

2.1 Softwareprüfung

Die Methoden für die Softwareprüfung lassen sich in dynamische und statische Verfahren unterteilen, je nachdem, ob während ihrer Durchführung das Programm ausgeführt wird oder nicht. Hauptvertreter für die statischen Verfahren ist das Review, während der Test die bekannteste Methode für dynamische Verfahren ist. In Abbildung 1 sind bekannte Verfahren der Softwareprüfung in einer grafischen Übersicht angeordnet (siehe auch [FLS95]). In den folgenden Abschnitten werden wir die zwei Vertreter der Softwareprüfmethoden, die wir im Rahmen unserer Untersuchung betrachtet haben, genauer beschreiben.

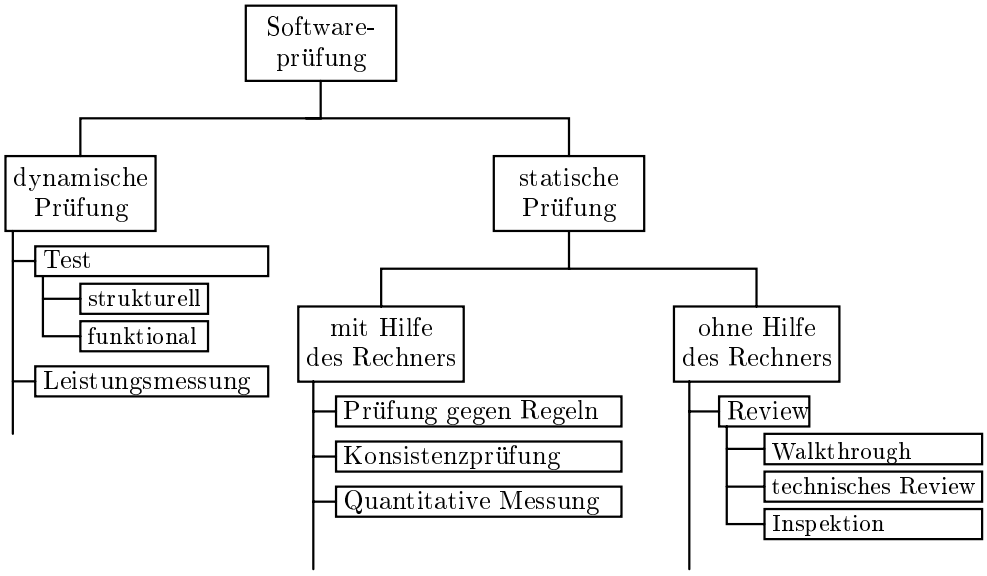


Abb. 1: Übersicht über Verfahren der Softwareprüfung.

2.2 Review

Dieser Abschnitt gibt einen kurzen Überblick über statische Prüfverfahren und dient der Einordnung und der Beschreibung der im Softwarelabor verwendeten Technik.

Im Gegensatz zur dynamischen Prüfung wird bei der statischen Prüfung das Programm nicht mit einem Rechner ausgeführt. Man kann zwar auch hier einen Rechner analytisch einsetzen, z. B. zu Konsistenzprüfungen oder für quantitative Untersuchungen. Im folgenden wollen wir uns jedoch denjenigen stati-

schen Prüfverfahren zuwenden, die ohne Hilfe eines Rechners auskommen. Das Grundprinzip dabei ist, daß eine oder mehrere Personen das zu prüfende Dokument bzw. Programm bewerten und Mängel ermitteln. Diese Verfahren werden hier unter dem Oberbegriff „Review“ zusammengefaßt.

2.2.1 Überblick

Zum Kreis der statischen Prüfverfahren zählen eine ganze Reihe von Reviewtechniken. Von den vielen möglichen Verfahren sollen hier die am häufigsten verwandten, nämlich Walkthrough, technisches Review und Inspektion kurz angeschnitten werden, um die Begriffe gegeneinander abzugrenzen und so die Einordnung der im Softwarelabor verwendeten Technik zu ermöglichen.

Von den genannten Verfahren ist der Walkthrough das einfachste Verfahren und erfordert den wenigsten Aufwand. Technisches Review und Inspektionen sind wohldefiniert und erfordern wesentlich mehr Aufwand. Insbesondere ist die Anzahl und der Umfang der Einzelschritte bei der Inspektion am größten. Allerdings verbessert sich die Wirksamkeit bezüglich Fehleraufdeckung gleichermaßen, so daß die Inspektion zwar am aufwendigsten ist, aber die größte Fehleraufdeckung aufweist.

Walkthrough

Ein Walkthrough besteht aus einer optionalen Vorbereitungsphase und der Walkthrough–Sitzung. Die Vorbereitungsphase dient dazu, daß die Gutachter das Werk vorab anschauen können, um sich Fragen zu überlegen. Dazu benötigen alle Gutachter Zeit. Um diesen Aufwand zu sparen, kann die Vorbereitungsphase auch ausgelassen werden. In diesem Fall erhalten die Gutachter das Werk erst zu Beginn der Sitzung. Der Autor moderiert die Walkthrough–Sitzung und erläutert sein Werk Stück für Stück. Die Gutachter können zum jeweiligen Abschnitt Fragen stellen und Probleme aufwerfen. In vielen Fällen identifiziert der Autor das Problem selbst, nachdem er von einem Gutachter auf einen Punkt aufmerksam gemacht wird. Identifizierte Probleme und aufgedeckte Fehler werden protokolliert.

Wie der Beschreibung zu entnehmen ist, hat der Autor eine ganz entscheidende Rolle für den Erfolg des Walkthrough–Prozesses. Er steuert den Ablauf der Sitzung und kann durch seine Präsentation fehlendes Wissen oder die eventuell ausgelassene Vorbereitung der Gutachter ausgleichen. Andererseits kann er die Gutachter an Problemfeldern vorbeilotsen, indem er Abschnitte besser darstellt als sie tatsächlich sind. Auch die Nacharbeit wird vom Autor vorgenommen und es erfolgt in der Regel keine weitere Prüfung bezüglich der Beseitigung der Probleme und Fehler. Damit liegt auch der Grad der Fehlerbeseitigung im Ermessen des Autors.

Nach [FLS95] ist ein Walkthrough eine „mildere Form des Reviews“, das sehr wenig Aufwand verursacht. Der geringe Aufwand und das informelle Verfahren führen aber auch zu einer geringeren Wirksamkeit bezüglich der Fehleraufdeckung.

deckung. Dafür sind Walkthroughs gut zur Schulung geeignet, da das Werk vom Autor erläutert wird und Probleme ausführlich diskutiert werden können. Generell eignet sich der Walkthrough für kleine Entwickler–Teams, die nicht die benötigten Kapazitäten für stärker strukturierte und damit aufwendigere Formen des Reviews aufbringen können.

Technisches Review

Das technische Review dient der Begutachtung eines Werks. Dabei wird durch gezielte Aufgabenstellung und Beschreibung der Aufträge an die Beteiligten versucht, die Schwächen von Walkthroughs zu vermeiden. Durch das informelle Vorgehen bei Walkthroughs sind nicht alle Aufwände transparent und können deshalb nicht geplant werden. Außerdem kann es passieren, daß das Werk während der Begutachtung verändert wird, da kein Einfrieren im informellen Prozeß vorgesehen ist und damit die Begutachtung ins Leere geht. Darüber hinaus ist nicht garantiert, daß eine sorgsame Nacharbeit erfolgt, da ein nicht definierter Prozeß auch nicht nachvollziehbar ist.

Mit dem technischen Review können unerschiedliche Ziele verfolgt werden. Neben der Identifikation von Fehlern kann beispielsweise ein Anliegen sein, neue Mitarbeiter zu schulen, das Wissen über neue Techniken zu verbreiten oder die Übergabe des Prüflings von der Entwicklung an die Wartung zu erleichtern.

Der Ablauf für das technische Review nach [FW82] und [FLS95] zeigt, wie die oben aufgeführten Probleme vermieden werden können. Der erste Schritt des technischen Reviews ist die Planungsphase. Hier werden vom Projektleiter ein Moderator und mehrere Gutachter ausgewählt, Begutachtungsaspekte festgelegt und Aufwände und Termine geplant. Als nächstes werden vom Moderator alle Beteiligten (der Autor und die Gutachter) zur Einführungssitzung eingeladen. Dabei erhalten die Gutachter die nötigen Unterlagen, insbesondere den Prüfling, und werden über Zweck und Ablauf des Reviews informiert. Optional kann der Autor einen Überblick über seine Arbeit geben. Die individuelle Vorbereitung der Gutachter erfolgt ähnlich der beim Walkthrough, wird jedoch immer durchgeführt. Dabei können unterstützende Materialien wie Programmierrichtlinien oder Checklisten eingesetzt werden. Die Review–Sitzung wird vom Moderator geleitet. Dieser führt seiten- bzw. abschnittsweise durch den Prüfling. Dabei muß er einen Konsens über die jeweiligen Anmerkungen herbeiführen. Einer der Gutachter übernimmt zusätzlich die Rolle des Protokollanten und notiert die Befunde. Der Autor selbst hat einen Beobachterstatus und wird nur zur Aufklärung grober Mißverständnisse und zur Beantwortung von Fragen hinzugezogen. Zum Abschluß der Review–Sitzung entsteht ein Review–Bericht, in dem alle gemeinsamen Anmerkungen zum Prüfling dokumentiert sind und eine Empfehlung ausgesprochen wird. Diese dient dem Projektleiter als Entscheidungsgrundlage, um Nacharbeit und eventuell ein erneutes Review anzusetzen oder die Freigabe des Prüflings zu erteilen. Die Nacharbeit wird vom Autor anhand des Protokolls vorgenommen und vom Moderator überprüft. Als letzter Schritt wird eine Analysephase empfohlen, um aus den im Review entdeckten Fehlern zu lernen.

Inspektion

Von den drei angesprochenen Review–Arten ist die Inspektion ist die formellste Art, eine statische Prüfung vorzunehmen. Alle Schritte sind wohldefiniert, und das Ziel ist jeweils beschrieben. Es besteht zwar große Ähnlichkeit zum technischen Review, dennoch gibt es einige entscheidende Unterschiede. Im Gegensatz zum technischen Review wird bei Inspektionen immer eine Überblickssitzung abgehalten. Außerdem sind das Überprüfen der Nacharbeit, die Erfassung und Analyse von Daten sowie das Ableiten von Kennzahlen wichtige Bestandteile des Inspektionsprozesses. Der Ablauf des Inspektionsprozesses (siehe Tabelle

Schritt	Beschreibung	Ziele
1. Planung	Zeitplan aufstellen Gutachter wählen Unterlagen ausgeben	Verfügbarkeit der Beteiligten und Verteilung der Unterlagen sicherstellen
2. Überblick	Präsentation des Prüflings durch den Autor	Fördern des Verständnisses für den Prüfling
3. Vorbereitung	Individuelle Begutachtung durch Gutachter	Mängel aufdecken
4. Inspektionssitzung	Formelle Beurteilung des Prüflings	Mängel abstimmen und protokollieren
5. Nacharbeit	Überarbeitung des Prüflings	Probleme und Mängel beseitigen
6. Follow–Up	Überprüfung der Nacharbeit Freigabe der Arbeit des Autors Abschließendes Inspektionsprotokoll Auswertung	Sicherstellen, daß die aufgedeckten Fehler und Mängel behoben werden Nachweis des Nutzens Gewinnung von Planungszahlen

Tab. 1: Phasen des Inspektionsprozesses.

1) ist immer gleich und aus den erhobenen Daten können wichtige Zahlen zur Planung und zum Nachweis des Nutzens gewonnen werden.

2.2.2 Einordnung und Beschreibung

Für das Experiment im Softwarelabor wird das technische Review verwendet, jedoch mit starker Fokussierung auf Fehlersuche ähnlich einer Code–Inspektion. Auch das Verfahren entspricht einem Teil des Inspektionsprozesses, deckt jedoch nicht alle Aspekte ab. Die folgende Beschreibung umfaßt Ziel, Verfahren, Rollen, Checklisten, Fehlerklassifikation der im hier beschriebenen Experiment eingesetzten Review–Ausprägung.

Ziel

Ziel des Reviews ist es, genau wie bei den eingesetzten dynamischen Prüfverfahren, Fehler im Prüfling aufzudecken, um später einen Vergleich zwischen statischen und dynamischen Prüfverfahren hinsichtlich ihrer Eignung, Fehler im Programmcode eingebetteter Systeme zu finden, durchzuführen.

Verfahren

Das Review besteht aus 5 Schritten (vgl. auch Tabelle 2). Diese sind Planung, Überblickssitzung, Vorbereitung, Inspektionssitzung und Analyse.

Schritt	Beschreibung	Ziele
1. Planung	Zeitplan aufstellen Gutachter wählen	Verfügbarkeit der Beteiligten sicherstellen
2. Überblick	Präsentation des Prüflings durch den Autor Unterlagen ausgeben	Fördern des Verständnisses für den Prüfling Verteilung der Unterlagen sicherstellen
3. Vorbereitung	Individuelle Begutachtung durch Gutachter	Mängel aufdecken
4. Inspektionssitzung	Formelle Beurteilung des Prüflings	Mängel abstimmen und protokollieren
5. Analyse	Untersuchungen	Effektivität und Effizienz bestimmen

Tab. 2: Ausprägung des Inspektionsprozesses beim Softwarelabor.

Im folgenden ist beschrieben, wie im Softwarelabor bei den einzelnen Schritten vorgegangen wurde:

Planung: Durch den experimentellen Aufbau steht fest, welche Inspektoren wann welches Programm zu prüfen haben. Die Checkliste und das Verfahren werden unverändert über alle Inspektionen beibehalten.

Überblick: Zur Initialisierung wird jeweils eine 1/2–stündige Veranstaltung abgehalten, in der die technischen Zusammenhänge und Hintergründe der Prüflinge erläutert werden.

Vorbereitung: Die Vorbereitung erfolgt individuell. Als Hilfsmittel wurde den Inspektoren eine Checkliste mit allgemeinen Programmierrichtlinien in die Hand gegeben. Alle Mängel wurden in ein Vorbereitungsprotokoll und anhand einer Fehlerklassifikation (s. u.) eingeordnet.

Sitzung: Um den Ablauf der Sitzung zu planen, fragt der Moderator am Anfang alle Inspektoren, wieviele Fragen und wieviele Fehler sie zu den einzelnen Klassen gefunden haben und wieviel Zeit sie für die Vorbereitung

aufgewendet haben. Zunächst werden die generellen Fragen vom Autor beantwortet. Dann wird Funktion um Funktion durch die verschiedenen Module gegangen. Die Inspektoren stellen Fragen zum jeweiligen Abschnitt und machen ihre Anmerkungen. Fehler, über die Konsens herrscht, werden in das Protokoll der Review–Sitzung aufgenommen.

Analyse: Es werden eine Reihe von Auswertungen vorgenommen, die im Abschnitt Bewertung der Reviews (4.2.3) nachzulesen sind. Gegenstände der Untersuchung waren Fehlerklassifikation, Effektivität, Effizienz, Fehlerüberdeckung, sowie das Verhältnis von Aufwand zu Größe des Prüflings.

Rollen

An den Reviews haben zwischen 4 und 6 Personen teilgenommen. Neben dem Moderator und dem Autor, der gleichzeitig als Protokollant fungierte, nahmen jeweils 2-4 Inspektoren teil. Dabei wurde die Rolle der Inspektoren von Studenten und die Rolle des Autors und des Moderators durch die Betreuer übernommen. Es sei hier darauf verwiesen, daß die Personalunion von Protokollant und Autor sowohl für technische Reviews als auch für Inspektionen unüblich ist.

Fehlerklassifikation

Zur Klassifikation der Fehler wurde ihre Kritikalität bestimmt. Die betreffende Definition für die Fehlerklassen sind an die Fehlerklassen der TekInspektion nach P. Johnson angelehnt.

Unbedeutend:	Entweder kosmetische Fehler oder andererseits Fehler, die sich in unschönem Code niederschlagen, für den Benutzer aber keine direkte Konsequenz haben.
Bedeutend:	Fehler, die Auswirkungen auf das funktionale Verhalten haben, aber den eigentlichen Systembetrieb nicht spürbar beeinträchtigen.
Schwerwiegend:	Ganze Funktionseinheiten arbeiten nicht zufriedenstellend.
Kritisch:	Diese Fehler beeinträchtigen das Systemverhalten so sehr, daß ein vernünftiger Betrieb keinesfalls möglich ist. Eine Gefährdung von Leib, Leben oder Material ist nicht ausgeschlossen.

Die Formulare und Checklisten, die bei der Durchführung der Reviews Anwendung fanden, sind in Anhang G wiedergegeben.

2.3 Test

Wie schon erwähnt, ist der Hauptvertreter der dynamischen Verfahren der systematische Test, bei dem Software geprüft wird, indem sie mit zuvor festgelegten Eingabewerten ausgeführt wird. Eine gute Einführung in die Thematik gibt [Mye79, Mye91].

Grundsätzlich ist der Test von Laufzeitversuchen und von der Abnahmeprüfung abzugrenzen: Beim Laufzeitversuch wird der Autor während der Implementierungsphase desöfteren Teile des Programms ablaufen lassen, um zu sehen, ob das Programm seinen eigenen Erwartungen entspricht. Ein solcher Laufzeitversuch läuft in der Regel unsystematisch ab. Bei der Abnahmeprüfung führt der Autor dem Kunden das Programm vor, um zu zeigen, daß es genau das tut, was der Kunde erwartet. Beim Test dagegen ist das Ziel das Gegenteil: Man versucht, Fehler zu finden.

Die Vorgehensweise beim Test läßt sich im Groben in drei Schritte untergliedern: Vorbereitungsphase, Durchführungsphase und Analysephase.

Vorbereitungsphase Während der Vorbereitungsphase werden all diejenigen Arbeiten ausgeführt, die für eine reibungslose Durchführung des Tests notwendig sind: In den meisten Fällen ist es nicht möglich, das System mit allen möglichen Abläufen zu testen. Daher bestimmt eine sinnvolle Auswahl der Testfälle maßgeblich den Erfolg eines Tests. Für jeden Testfall wird auch das erwartete Verhalten protokolliert. Auf die verschiedenen Möglichkeiten der Testfallauswahl gehen wir später genauer ein. Die zweite wichtige Tätigkeit in der Vorbereitungsphase ist die Bereitstellung einer Testumgebung, was gerade bei eingebetteten Systemen mit einem großen Aufwand verbunden sein kann. Hier ist es oft nur möglich, das System anstatt in der späteren realen Umgebung in einer Testumgebung auszuführen, die deshalb zuerst implementiert werden muß.

Durchführungsphase In dieser Phase wird das System mit den in der Vorbereitungsphase definierten Testfällen bearbeitet. Wichtig ist dabei, daß das Verhalten und die Ergebnisse protokolliert werden, um eine sinnvolle Analyse zu ermöglichen.

Analysephase Nach der Durchführung müssen die Ergebnisse (Protokolle, Ausgaben, Reaktionen) analysiert werden. Das tatsächliche Verhalten wird mit dem in der Vorbereitungsphase festgehaltenen, erwarteten Verhalten verglichen. Daraus ergeben sich bestimmte Fehlverhalten, die nun festgehalten werden. In vielen Fällen ist es noch sinnvoll, im Quelltext des Programms die Ursachen der Fehlverhalten zu suchen. Dies sind dann die gefundenen Fehler.

Im folgenden möchten wir möglichen Strategien für die Testfallauswahl in der Vorbereitungsphase vorstellen. Wir betrachten die zwei klassischen Ansätze: Beim *funktionalen Test* (Black–Box–Test) wählt man die Testfälle anhand der

Spezifikation aus, während beim *strukturellen Test* (White–Box–Test) die Testfälle mit Hilfe des Quelltextes bestimmt werden.

2.3.1 Der funktionale Test

Beim funktionalen Test geht man von den funktionalen Anforderungen an das Programm aus: Die Auswahl der Testfälle richtet sich nach den Ein- und Ausgaben sowie deren funktionellen Verknüpfung. Jede dort vorhandene Funktionalität sollte idealerweise mit allen möglichen Ein- und Ausgabewerten ausgeführt werden. Da ein vollständiger Test meist unmöglich ist, versucht man, die verschiedenen Ein- und Ausgabewerte in Gruppen aufzuteilen, so daß mit hoher Wahrscheinlichkeit gewährleistet ist, daß sich ein Programm mit Parametern derselben Gruppe gleich oder zumindest ähnlich verhält. In diesem Fall reicht es aus, wenn das Programm mit mindestens einem Element aus jeder Gruppe ausgeführt wird.

Für die Bestimmung der Gruppen gibt es wiederum verschiedene Ansatzmöglichkeiten. Im Rahmen der hier beschriebenen Untersuchung setzen wir die Klassifikationsbaummethode [GG93b] ein: Der Raum der Eingabewerte wird anhand verschiedener Aspekte (sogenannte *Klassifikationen*) betrachtet. Für jede solche Klassifikation werden disjunkte Klassen entworfen. Eine einzelne Klasse kann wiederum in mehrere Klassifikationen aufgeteilt werden. Ein Beispiel soll diese Vorgehensweise erläutern:

Beispiel 2.1 (head–Befehl)

Der UNIX–Befehl `head` soll getestet werden. Die Spezifikation für diesen Befehl lautet wie folgt:

> `head [-n` *anzahl*`]` `[`*datei* ...`]`
>
> Der `head`–Befehl kopiert die ersten *anzahl* Zeilen von jeder *datei* in die Standardausgabe. Falls kein Dateiname angegeben ist, kopiert `head` die ersten Zeilen von der Standardeingabe. Der Standardwert für *anzahl* ist 10.
>
> Falls mehr als eine Datei angegebeben ist, wird vor Beginn jeder Datei der Text `==>` *datei* `<==` ausgegeben.

Die möglichen Eingaben für den Befehl können offensichtlich nach der Anzahl der angegebenen Dateien und dem Vorhandensein der Option `-n` klassifiziert werden. Falls diese Option angegeben ist, empfiehlt sich eine weitere Unterscheidung danach, ob *anzahl* gleich 0, größer als die Länge der Datei, oder eine Zahl dazwischen ist. Außerdem klassifizieren wir die Eingaben noch nach der tatsächlichen Existenz der angegebenen Dateien. Der zugehörige Klassifikationsbaum ist in Abbildung 2 wiedergegeben. Die mit 1, 2 und 3 bezeichneten waagrechten Linien werden später erklärt. □

Dieses Beispiel verdeutlicht auch, daß eine genaue Spezifikation notwendig ist. Hier wurde nichts darüber ausgesagt, was bei einem Fehlerfall passieren soll,

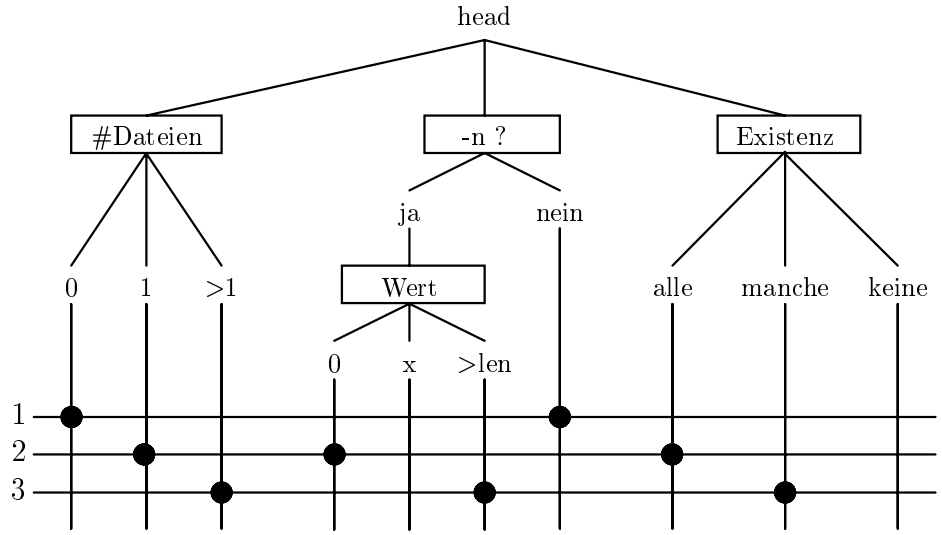


Abb. 2: Der Klassifikationsbaum für `head`.

beispielsweise wenn eine Datei nicht vorhanden ist, oder mehr Zeilen ausgegeben werden sollen wie tatsächlich vorhanden sind.

Erfahrungsgemäß ist es sinnvoll, einen Schwerpunkt bei der Auswahl der Klassen auf Grenzfälle und Fehler zu legen, da in diesen Bereichen die häufigsten Fehler zu finden sind.

Nachdem der Klassifikationsbaum erstellt wurde, müssen nun die Testfälle erstellt werden. Für jeden Testfall wird angegeben, zu welchen Klassen er bezüglich jeder Klassifikation gehört. Falls eine Klassifikation für einen Testfall unbedeutend ist, braucht dessen Klasse nicht angegeben zu werden. Jede Klasse sollte durch mindestens einen Testfall abgedeckt sein. Außerdem ist es auch oft sinnvoll, eine bestimmte Klasse öfters zu testen, indem bei anderen Klassifikationen variiert wird.

Beispiel 2.2 (head–Befehl)

In Abbildung 2 stellen die unteren drei waagrechten Linien drei Testfälle dar. Durch die schwarzen Kreise wird angegeben, welche Klassen durch einen Testfall abgedeckt werden sollen. Beispielsweise soll beim ersten Testfall die Anzahl der angegebenen Dateien gleich Null sein, und auch die Option `-n` soll nicht vorhanden sein. Da keine Dateien angegeben sind, macht auch eine Unterscheidung, ob die Dateien existieren oder nicht, keinen Sinn. Die zwei weiteren Testfälle sind ähnlich zu interpretieren. Weitere Testfälle sind hier nicht abgebildet. Sie können aus Abbildung 3 entnommen werden. □

Die entworfenen Testfälle sind bisher nur abstrakte Testfälle, da die genauen Eingabedaten noch nicht festgelegt sind. Dies muß in einem weiteren Schritt erfolgen. Für jeden abstrakten Testfall müssen also für das zu testende Programm die Eingabedaten so ausgewählt werden, daß sie zu den jeweils angegebenen

Klassen gehören.

Beispiel 2.3 (head–Befehl)
Für die in Abbildung 2 angegebenen abstrakten Testfälle sind die folgenden konkreten Testfälle denkbar:

1. `head`
2. `head -n 0 daten.txt`, wobei `daten.txt` eine lesbare Datei sein muß
3. `head -n 20 daten.txt daten2.txt`, wobei `daten.txt` weniger als 20 Zeilen hat und `daten2.txt` nicht existiert □

Der Klassifikationbaumeditor CTE

Der CTE [GW95] ist ein Werkzeug mit graphischer Benutzeroberfläche, das den Entwurf von Testfällen mit der Klassifikationsbaummethode unterstützt. Die Programmoberfläche (siehe Abbildung 3) besteht im wesentlichen aus zwei Teilen. In der oberen Hälfte wird der Klassifikationsbaum entworfen, während in der unteren Hälfte die Testfälle angegeben werden.

Der CTE besitzt einige Merkmale, die die Arbeit für den Programmtester erleichtert: Bei dem Entwurf des Klassifikationsbaum können keine inkorrekten Bäume erstellt werden; beispielsweise läßt der CTE nicht zu, daß eine Klasse als Nachfolger einer anderen Klasse angegeben wird, da eine Klasse immer Nachfolger einer Klassifikation sein muß.

Außerdem bietet der CTE die üblichen Editierfunktionen wie Kopieren, Einfügen und Verschieben von Knoten oder Teilbäumen.

Beim Erstellen der Testfälle wird darauf geachtet, daß zu keinem Zeitpunkt mehr als eine Klasse einer Klassifikation ausgewählt wird: Sobald eine Klasse selektiert wird, wird eine eventuell andere selektierte Klasse der Klassifikation deselektiert.

Für große Bäume bietet der CTE die Möglichkeit, Teilbäume in separaten Fenstern aufzuteilen. Außerdem können alle Komponenten (Knoten, Testfälle) mit Kommentaren annotiert werden.

Außer diesen Funktionen, die sofort beim Erstellen von Baum und Testfällen sichtbar werden, besitzt der CTE noch weitere Überprüfungsmöglichkeiten. Es können beispielsweise folgende Bedingungen überprüft werden:

- Kein Testfall darf doppelt auftreten.
- Jede Klasse muß mindestens in einem Testfall selektiert sein.

2.3.2 Der strukturelle Test

Während beim funktionalen Test der Quelltext für die Testfallauswahl unberücksichtigt blieb, werden beim strukturellen Test die Testfälle nur anhand

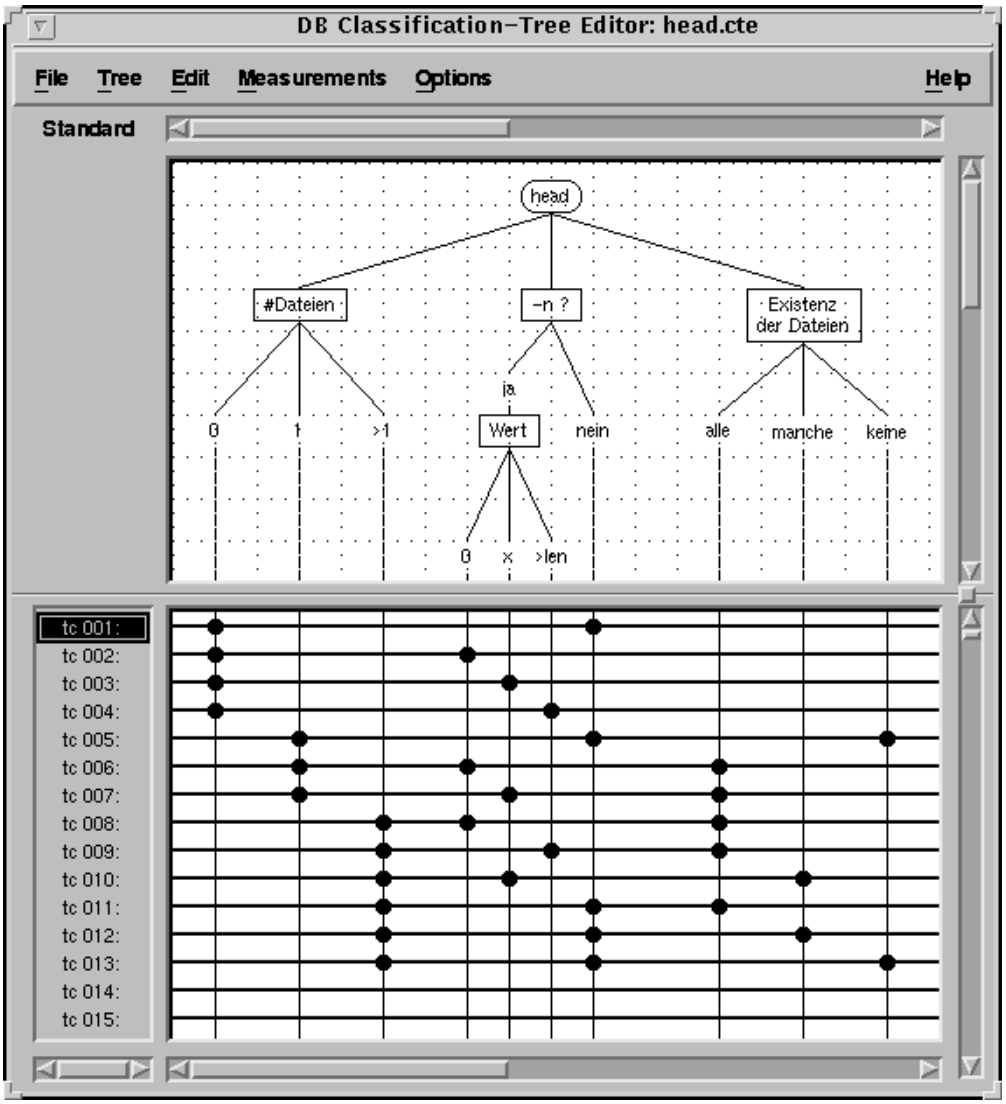


Abb. 3: Der Klassifikationsbaumeditor.

des Quelltextes ausgewählt; die Spezifikation wird nicht betrachtet. Ziel ist es, daß jeder mögliche Ablauf durch das Programm mindestens einmal getestet wird. Dies ist aber nicht mehr möglich, sobald eine Schleife mit nicht festgelegter Anzahl von Schleifendurchläufen vorhanden ist; es müßte eine unendliche Anzahl von möglichen Schleifendurchläufen getestet werden.

Aus diesem Grund beschränken wir uns beim strukturellen Test auf die *Zweigüberdeckung*. Dies bedeutet, daß bei jeder Verzweigung jede mögliche Alternative mindestens einmal ausgeführt werden muß. Somit ist auch als Minimalziel gewährleistet, daß jede Anweisung mindestens einmal ausgeführt wird.

Gerade für den strukturellen Test wurden in den letzten Jahren einige un-

terstützende Werkzeuge entwickelt. Dies war vor allem deshalb möglich, da im Gegensatz zum funktionalen Test, dem als Grundlage meistens eine informelle Spezifikation dient, der strukturelle Test auf dem Quelltext basiert, der in einer Programmiersprache mit einer festgelegten Semantik verfaßt ist.

Der GCT

Der GCT (*Generic Coverage Tool*, siehe [Mar91, Mar92]) ist ein Werkzeug, das in einen bestehenden C–Code verschiedene Kontrollstrukturen so einfügt, daß bei einer Programmausführung automatisch das Ablaufverhalten mitprotokolliert wird. Anschließend kann eine Statistik der erreichten Überdeckung ausgegeben werden. Außer der beschriebenen Zweigüberdeckung überprüft der GCT noch weitere Kriterien; eine optimale Überdeckung im Sinne des GCT ist erreicht, wenn folgende Bedingungen erfüllt sind:

Zweigüberdeckung: In einer Verzweigung (if, for, while, do–while, ?–Ausdruck) muß die steuernde Bedingung einmal wahr und einmal falsch sein.

Switch–Überdeckung: Bei einer switch–Anweisung muß jedes case mindestens einmal angesteuert werden. Dabei ist auch zu beachten, daß der GCT automatisch ein default–Label hinzufügt, falls dieses nicht vorhanden ist. Auch dieses default–Label muß mindestens einmal aufgerufen werden.

Bedingungsüberdeckung ist eine Erweiterung der Zweigüberdeckung. Falls in einer Bedingung mehrere Teilbedingungen auftreten, die beispielsweise mit `&&` oder `||` verknüpft sind, muß auch jede dieser Teilbedingungen mindestens einmal wahr und einmal falsch sein.

Schleifenüberdeckung: Bei einer Schleife (for, while, do–while) gibt es drei Möglichkeiten: Der Schleifenrumpf wird nie, genau einmal oder aber mehrmals ausgeführt. Jede Möglichkeiten muß mindestens einmal auftreten.

Nachdem der GCT den Quelltext analysiert hat, fügt er verschiedene Protokollierungsanweisungen ein. Anschließend wird der Quelltext compiliert. Wenn nun das Programm ausgeführt wird, dann wird durch diese Protokollierungsanweisungen beispielsweise ermittelt, wie oft eine Schleife ausgeführt oder wie oft eine Bedingung zu wahr bzw. zu falsch ausgewertet wurde.

Zur Auswertung stellt der GCT zwei Werkzeuge zur Verfügung: `gsummary` gibt einen Überblick über die erreichten Überdeckungsgrade, während `greport` eine genaue Auflistung ausgibt, wie oft beispielsweise welche Schleife ausgeführt wurde.

Zur näheren Erläuterung stellen wir die Arbeit mit dem GCT anhand eines Beispiels vor.

Beispiel 2.4 (Primfaktorenzerlegung)

Das folgende Programm für die Primfaktorenzerlegung soll getestet werden.

```
 1  #include <stdio.h>
 2
 3  void main(argc, argv)
 4    int argc;
 5    char **argv;
 6  {
 7    int akt_pos, zahl, i;
 8
 9    for (akt_pos=1; akt_pos<argc; akt_pos++) {
10      zahl = atoi(argv[akt_pos]);
11      printf("Primfaktorenzerlegung von %d : ",zahl);
12
13      i=2;
14      while (i <= zahl) {
15        if ((zahl % i) == 0) {
16          printf("%d ", i);
17          zahl = zahl / i;
18        }
19        else {
20          i++;
21        }
22      }
23      printf("\n");
24    }
25  }
```

Das instrumentierte und compilierte Programm heiße `primfak`. Es erhält als Argumente eine beliebig lange Liste von Zahlen, von denen jeweils die Primfaktorenzerlegung berechnet wird.

Zu Beginn wählen wir die drei Testfälle `primfak`, `primfak 2` und `primfak 3` aus und starten das Programm mit diesen Aufrufen. Diese Testfälle gewährleisten eine vollständige Zweigüberdeckung: Jede Anweisung wird mindestens einmal ausgeführt, und bei jeder der Verzweigungen in Zeile 9, 14 und 15 wird jede Alternative ebenfalls mindestens einmal ausgewählt.

Jedoch haben wir die vom GCT geforderten Überdeckungsbedingungen noch nicht vollständig erfüllt. Dies zeigt sich, wenn wir die zwei Analysewerkzeuge des GCT verwenden. Mit `gsummary` können wir eine Statistik ausgeben lassen:

```
BINARY BRANCH INSTRUMENTATION (6 conditions total)
0 ( 0.00%) not satisfied.
6 (100.00%) fully satisfied.

LOOP INSTRUMENTATION (6 conditions total)
2 (33.33%) not satisfied.
4 (66.67%) fully satisfied.

OPERATOR INSTRUMENTATION (6 conditions total)
0 ( 0.00%) not satisfied.
6 (100.00%) fully satisfied.

SUMMARY OF ALL CONDITION TYPES (18 total)
2 (11.11%) not satisfied.
16 (88.89%) fully satisfied.
```

Dies zeigt, daß bei der Schleifenüberdeckung zwei Kriterien nicht erfüllt waren, ansonsten aber die Überdeckung optimal ist. Da im Quelltext keine switch–Anweisungen und auch keine Mehrfachbedingungen vorkommen, entfallen bei `gsummary` auch diese Angaben.

Um nun zu erfahren, an welchen Stellen die Überdeckung noch verbessert werden kann, verwenden wir den Befehl `greport`. Dieser liefert konkrete Angaben über die nicht erreichten Überdeckungen. Für unsere Testfälle liefert der Aufruf von `greport` folgende Ausgabe:

```
"primfak.c", line 9:
             loop zero times: 1, one time: 2, many times: 0.
"primfak.c", line 14:
             loop zero times: 0, one time: 1, many times: 1.
```

Diese Ausgabe liefert eine genaue Statistik darüber, daß der Rumpf der for–Schleife in Zeile 9 nie mehrmals hintereinander ausgeführt wurde. Dies ist auch offensichtlich, denn mit dieser Schleife werden sukzessive die Parameter, mit dem das Program aufgerufen wurde, abgearbeitet, und es gab keinen Testfall mit mehreren Parametern.

Die zweite Zeile besagt, daß der Rumpf der while–Schleife in Zeile 14 immer mindestens einmal ausgeführt wurde; ein Fall, bei dem der Rumpf überhaupt nicht ausgeführt wird, fehlt.

Mit diesen Aussagen können wir nun zwei weitere Testfälle konstruieren, die zusammen mit den schon vorhandenen eine optimale Überdeckung bewirken. Der Testfall `primfak 34 46 242` hat zur Folge, daß die for–Schleife auch mehrmals ausgeführt wird, und bei dem Testfall `primfak 1` wird die while–Schleife überhaupt nicht ausgeführt.

Damit entdecken wir auch einen Mangel des Programms: Für den Testfall `primfak 1` wird keine Primfaktorenzerlegung ausgegeben. □

3 Aufbau des Experiments

In diesem Kapitel beschreiben wir, wie wir das in der Einleitung beschriebene Ziel — die vergleichende Untersuchung statischer und dynamischer Softwareprüfung — in ein Experiment umgesetzt haben. Dazu konkretisieren wir zuerst in Abschnitt 3.1 unsere Zielsetzung sowie unsere damit verbundenen Annahmen, d. h. wir verfeinern das *was* der Untersuchung. In Abschnitt 3.2 beschreiben wir die Umsetzung dieser Ziele in einen konkreten Experimentablauf und erläutern damit das *wie*.

3.1 Zielsetzung und Hypothesen

Die Ziele, die mit dem Experiment verfolgt werden sollten, lassen sich in zwei Bereich aufteilen: Ziele für die teilnehmenden Studenten und Ziele für das Softwarelabor.

Die Durchführung eines derartigen Experiments hängt in starkem Maße von einer genügend großen Anzahl von Teilnehmern ab. Eine entsprechende Motivation der Teilnehmer — in diesem Fall von Studenten — ist also unerläßlich. So wurden, neben der Möglichkeit, einen geforderten Leistungsnachweis (Schein) erwerben zu können, folgende Aspekte in den Vordergrund gestellt:

- *Kennenlernen und Anwenden verschiedener Prüfaktivitäten.*
  Dies ist für die Studenten attraktiv, da einerseits diese Themen in der sonstigen Ausbildung nur wenig beachtet werden,[4] in der praktischen Software–Entwicklung aber einen hohen Stellenwert besitzen.

- *Fähigkeit, die Stärken und Schwächen der verschiedenen Prüfaktivitäten einschätzen zu können.*
  Die Kenntnis der Stärken und Schwächen der verschiedenen Prüfaktivitäten ist Voraussetzung für deren sinnvolle und zielgerichtete Anwendung. Dieses Wissen kann aber erst durch eine wiederholte Anwendung verschiedener Techniken reifen.

- *Fähigkeit, eigenen Prüfkonzepte erstellen zu können.*
  Diese Aufgabe ist typischer Bestandteil der Aufgaben eines Projektleiters oder Qualitätsmanagers, also Rollen, die viele Studenten in ihrem Berufsleben wahrnehmen werden.

Unser Interesse als Vertreter des Softwarelabors lag mehr in dem wissenschaftlichen Vergleich der statischen und dynamischen Softwareprüfung. Im einzelnen waren wir an den folgenden Punkten interessiert, wobei für uns der Schwerpunkt auf der Wirkung der Prüfaktivitäten in Hinblick auf eingebettete Systeme lag.

[4] Im Rahmen eines Software–Grundpraktikums werden zwar die grundlegenden Mechanismen dynamischer Softwareprüfung behandelt, jedoch in der Regel aus Zeitgründen nur rudimentär umgesetzt.

- Der quantitative Vergleich von Review und Test in Hinblick auf Effektivität und Effizienz.
- Das Herausarbeiten von Stärken und Schwächen der einzelnen Techniken. Dabei interessierte uns vor allem, ob es Klassen von Fehlern gibt, die sich typischerweise mit einer der beiden Techniken besser finden lassen.
- Das Identifizieren von Schwachstellen der jeweiligen Prüfaktivitäten in Hinblick auf die spezifischen Aspekte eingebetteter Systeme, wie beispielsweise Zeitbedingungen und Nebenläufigkeit.

Zur Verfolgung dieser Ziele haben wir eine Reihe von Fragestellungen abgeleitet, die im folgenden aufgeführt sind.

(F1) Wie viele Fehler werden bei Test und Review gefunden? Dabei soll insbesondere die Klasse der Fehler betrachtet werden, die beiden Techniken zugänglich ist, nämlich die Klasse der Fehler mit beobachtbaren Fehlverhalten.

(F2) Wie hoch ist der zeitliche Aufwand für Review und Test und wie verteilt sich dieser auf die einzelnen Phasen der jeweiligen Technik?

(F3) Wie effizient sind die Techniken, d. h. wie wirken die Techniken unter Berücksichtigung wirtschaftlicher Randbedingungen?

(F4) Besteht ein Zusammenhang zwischen Programmgröße und der benötigten Zeit zur Softwareprüfung?

(F5) Ergeben sich Reifeeffekte bei mehrmaliger Anwendung derselben Technik und inwieweit bestimmt die Erfahrung der Teilnehmer den Erfolg der Technik?

(F6) Gibt es Fehlerklassen, die sich im Rahmen der jeweiligen Prüfaktivitäten besonders gut/schlecht finden lassen?

(F7) Welcher Anteil der Fehlermeldungen kommt erst im Rahmen einer gemeinsamen Reviewsitzung hinzu?

(F8) Welchen Einfluß haben Ausbildung und Motivation auf die Wirkung der jeweiligen Technik?

(F9) Wie gut kann die eigene Leistung in den einzelnen Techniken beurteilt werden?

Mit diesen Fragen verbanden wir eine Reihe von Annahmen und Hypothesen, die wir im folgenden kurz skizzieren wollen.

Basierend auf [KL95b] vermuteten wir zu (F1) eine im wesentlichen gleiche Effektivität der beiden Prüfverfahren. Zudem nahmen wir an, daß bei Fehlern, die typisch für eingebettete Systeme sind und sich auf Nebeläufigkeit oder harte Zeitbedingungen beziehen, die statische Prüfung schlechter abschneidet.

Bei (F2) und (F3) gingen wir, ebenfalls aufgrund der Beobachtungen in [KL95b], von nicht–signifikanten Unterschieden aus.

Zu (F4) vermuteten wir einen quasi–linearen Zusammenhang zwischen der Größe des beim Review betrachteten Programms und der damit verbundenen Aufwände. Für den Test hatten wir diesbezüglich keine Hypothese.

Bei (F5) vermuteten wir einen sublinearen Zusammenhang zwischen der Erfahrung und dem Erfolg der Prüfung. Während sich zu Beginn noch große Fortschritte in der Effektivität erzielen lassen, ist dies nach einer gewissen Erfahrungssammlung wohl kaum der Fall.

Bezüglich (F6) gingen wir davon aus, daß sich Fehler zu Nebenläufigkeit und Zeitbedingungen mit statischen Prüfaktivitäten schlechter identifizieren lassen als andere Fehler.

Den Nutzen der Reviewsitzungen (F7) sahen wir als gegeben an, ohne die damit verbundenen Synergieeffekte jedoch quantifizieren zu können.

Den Einfluß der Ausbildung und Motivation (F8) als auch die Einschätzbarkeit der eigenen Leistung (F9) nahmen wir als gering an.

3.2 Experimental Design

Nachdem im vorangegangenen Abschnitt das Ziel der geplanten Untersuchung beschrieben wurde, wollen wir in diesem Abschnitt den Augenmerk auf die konkrete Umsetzung im Rahmen eines Praktikums legen.

Rahmenbedingungen und Grobkonzept des Experiment–Aufbaus

Bedingt durch unser Vorhaben, die experimentelle Untersuchung im Rahmen eines Praktikums im Hauptstudium durchzuführen, mußten wir einige Restriktionen beachten: Zum besseren Verständnis der nachfolgenden Konzeption des Experiments wollen wir daher diese hier kurz beschreiben.

Faktor Zeit: Ein organisiertes Praktikum sollte sich in seiner Laufzeit an der Vorlesungszeit orientieren und daher zwischen 12 und 15 Wochen dauern.

Faktor Aufwand: Im Prüfungsplan werden Praktika mit 4 SWS Aufwand angegeben. Zusammen mit der häuslichen Vorbereitung sollte sich so ein wöchentlicher Aufwand von durchschnittlich 8 Stunden pro Student und Woche ergeben.

Faktor Teilnehmerzahl: Da die Fakultät für Informatik in Ulm relativ klein ist, gibt es dementsprechend wenig Studenten im Hauptstudium. Daher sind in einem derartigen Praktikum nur zwischen 8 und 12 Teilnehmer zu erwarten.

Da die Studenten typischerweise nur wenig Erfahrung mit Testen und keinerlei Erfahrung mit Reviews hatten, war es klar, daß vor einem aussagekräftigen Vergleich der Techniken eine längere Einübungsphase durchgeführt werden

mußte. Für diese Aufgabe konnten wir auf Erfahrungen an der Universität Kaiserslautern zurückgreifen [KL95b]. Dort wurde in den Jahren 1994 und 1995 im Rahmen einer Vorlesung über Software Engineering ein kleines Experiment durchgeführt, bei dem verschiedene Prüfaktivitäten verglichen wurden. Für die Replizierung dieses Experiments sprach auch, daß sämtliche dazu benötigten Materialien in einem technischen Bericht [KL95a] zusammengefaßt sind.

Der nächste Schritt sollte die Anwendung der erlernten Techniken auf eingebettete Systeme sein. Insbesondere in dieser Phase des Praktikums wollten wir unsere, in Abschnitt 3.1 vorgestellten Fragen untersuchen.

Inhalt des dritten und letzten Abschnitts sollte die selbständige Entwicklung eines kleinen eingebetteten Systems sein. Dazu sollte neben der eigentlichen Entwicklung mit Design und Implementierung auch das Planen und Durchführen von Prüfaktivitäten ein wesentlicher Bestandteil sein.

Feinkonzept des Experiment–Aufbaus

In Abbildung 4 findet sich eine an dem zeitlichen Ablauf orientierte graphische Darstellung des Praktikums. Die Inhalte der einzelnen Abschnitte wollen wir im folgenden etwas näher beleuchten. Ziel dieser Beschreibung ist, neben einer Darstellung der einzelnen Aktivitäten, das Zusammenspiel der einzelnen Teile zu verdeutlichen. Die Einzelheiten der Durchführung sowie die Ergebnisse sind Inhalt des nächsten Kapitels. Die Zeitachse verläuft in der Abbildung von links

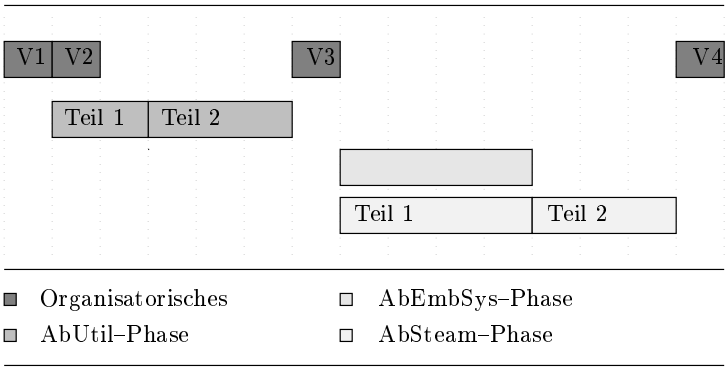


Abb. 4: Ablauf des Praktikums.

nach rechts. Die Zeitspanne zwischen zwei gestrichelten Linien beträgt eine Woche. Um die einzelnen Teile des Ablaufs möglichst klar zu benennen, haben wir die folgenden Kurzbezeichnungen eingeführt:

AbUtil–Phase Bezeichnet die Phase, bei der die Prüftechniken anhand kleiner UNIX–Werkzeuge (*Utilities*) eingeübt wurden.

AbEmbSys–Phase Bezeichnet die Phase, bei der die Prüftechniken auf vorgegebene eingebettete Systeme angewandt wurden.

AbSteam–Phase Bezeichnet die Phase, bei der zuerst die Steuerung eines Dampfdruckkessels (engl.: *steam boiler*) entwickelt und anschließend geprüft wurde.

Einführende Vorlesung (V1)
Am Anfang des Praktikums stand zuerst eine einführende Vorlesung über Prüfaktivitäten, die sich weitgehend an der Darstellung in [FLS95] orientierte. Ziel dieser dreistündigen Einführung war es, die Studenten auf das Praktikum einzustimmen und auf einen theoretisch einheitlichen Stand zu bringen.

Vertiefende Vorlesung (V2)
Nachdem die allgemeine Theorie über Softwareprüfung bekannt war, wurde diese im zweiten Schritt in Richtung der zunächst anzuwendenden Techniken konkretisiert. Außerdem wurde versucht, den Studenten die Thematik des experimentellen Software Engineerings näher zu bringen. Wichtig war uns an dieser Stelle, die Studenten für das Experimentieren zu begeistern, da dies unabdingbare Voraussetzung für verläßliche Aussagen ist. Falls dieser Schritt nicht gelingt und die Studenten die Veranstaltung als „Prüfung“ im weitesten Sinne verstehen, wären gegenseitige Unterstützung und damit ein Verfälschen der Ergebnisse unvermeidlich.

Einüben der Techniken (AbUtil–Phase, Teil 1)
Noch am selben Tag, an dem die Lesetechnik vorgestellt wurde, hatten die Studenten Gelegenheit, diese an einem kleinen Beispiel selbst praktisch anzuwenden. Die Techniken des Testens wurden als „Hausaufgabe“ und in der darauffolgenden Woche bearbeitet. Die Materialien zu diesen Beispielen stammten aus der bereits erwähnten Experimentdokumentation [KL95a]. Ziel dieses Schrittes war es, die Studenten mit der Anwendung der Techniken und der verwendeten Testumgebung vertraut zu machen. Die Daten, die dabei gesammelt wurden, haben wir keiner weiteren Analyse unterzogen.

Wechselweises Anwenden der Techniken (AbUtil–Phase, Teil 2)
Nach dem Einüben der Techniken erfolgte nun die erste experimentelle Untersuchung. Die Teilnehmer wurden in drei Gruppen eingeteilt, die jeweils eine der drei Prüfmethoden (Lesetechnik, funktionaler Test, struktureller Test) auf ein gemeinsames Prüfobjekt anwandten. In den beiden darauffolgenden Wochen wurden die Prüfmethoden getauscht. Am Ende dieses dreiwöchigen Abschnittes hatte jeder Teilnehmer jede Prüfmethode auf ein jeweils unterschiedliches Prüfobjekt angewandt. Da die Teilnehmer innerhalb der Gruppen für sich arbeiteten, hatten wir am Ende dieses Abschnittes 39 Prüfergebnisse (3 Wochen $\times$ 13 Teilnehmer).

Vorlesung über eingebettete Systeme (V3)
Während im ersten Teil des Praktikums die Hauptintention ein erstes praktisches Kennenlernen von Prüfaktivitäten war, ging es im zweiten Teil um deren Anwendung in Hinblick auf eingebettete Systeme. Dazu wurden zuerst einige typische Konzepte eingebetteter Systeme vorgestellt, um so den Grundstein für die weiteren Aktivitäten zu legen.

Prüfen eingebetteter Systeme (AbEmbSys–Phase)
In diesem Teil des Praktikums wurden die Teilnehmer wiederum in Gruppen eingeteilt, in denen aber echte Teamarbeit praktiziert wurde. Den vier Teams wurde in vier aufeinanderfolgenden Wochen jeweils ein eingebettetes System mitgegeben, das je zwei Teams mittels dynamischer bzw. statischer Prüfmethoden analysieren sollten. Am Ende der Woche stand dann die Abgabe des Prüfberichts bzw. eine moderierte Reviewsitzung. Die in diesem Abschnitt eingesetzten Programme, Prüfmethoden und Vorgehensweisen unterschieden sich deutlich von denen in der AbUtil–Phase und waren stärker an der industriellen Praxis orientiert. Ziel dieses Abschnittes war das Sammeln von Daten zur Wirkung der Prüfmethoden bei realistischen Randbedingungen.

Entwicklung eines eingebetteten Systems (AbSteam–Phase, Teil 1)
Parallel zu den Prüfaktivitäten sollten die Studenten auch Gelegenheit erhalten, selbständig ein eingebettetes System zu entwickeln. Hierbei handelte es sich um eine fehlertolerante Dampfkesselsteuerung. Neben der Implementierung waren auch Design und Testfallbestimmung ein Teil der Entwicklung.

Prüfen der selbstentwickelten Systeme (AbSteam–Phase, Teil 2)
Das Prüfen von Lösungen, die andere entwickelt haben, hat generell einen besonderen Reiz. In diesem letzten Abschnitt des Praktikums sollten die Studenten die Gelegenheit erhalten, die zuvor entwickelten Systeme gegenseitig zu testen. Für uns war in diesem Abschnitt vor allem von Interesse, wie die Studenten mit dem bisher gewonnenen Wissen zur Softwareprüfung umgehen. Die einzelnen Teams durften dazu ihre eigenen Prüfstrategien definieren und anwenden.

Abschlußbesprechung (V4)
Die Abschlußbesprechung soll einen runden Abschluß des Praktikums darstellen. Dabei wurden einerseits bereits gewonnene Erkenntnisse präsentiert, andererseits wurde die Meinung der Teilnehmer zum Verlauf des Praktikums erfragt, um diese bei künftigen Veranstaltungen zu berücksichtigen.

Nach diesem Überblick wollen wir nun im folgenden Kapitel die Umsetzung der drei Phasen AbUtil, AbEmbSys und AbSteam beschreiben und insbesondere die dabei gewonnenen Erkenntnisse präsentieren.

4 Ablauf, Analyse und Ergebnisse

In Kapitel 3 haben wir einen Überblick über den Aufbau des Experiments gegeben. Wir präsentieren nun in den folgenden Abschnitten den genauen Ablauf und die Ergebnisse, wobei wir uns an den drei großen Abschnitten des Praktikums orientieren, nämlich der AbUtil-, AbEmbSys- und AbSteam–Phase.

4.1 Einübungsphase (AbUtil–Phase)

Die erste Phase des Gesamtexperiments diente zur Einübung in die verschiedenen Techniken der Softwareprüfung. Hierzu wurde das sogenannte K&L–Experiment repliziert, das in dieser Form schon mehrmals an der Universität Kaiserslautern durchgeführt wurde (siehe [KL95b, KL95a]) und dessen Grundlagen ein Experiment war, das an der University of Maryland entstand (siehe [BS87]). Bei beiden Experimenten war das Ziel, die Effektivität und Effizienz von Software–Prüfmethoden zu vergleichen. Es handelte sich dabei um die Lesetechnik durch schrittweise Abstraktion, um den funktionalen Test und um den strukturellen Test.

Bevor wir die Ergebnisse unseres Experiments mit denen des K&L–Experiments vergleichen, möchten wir zunächst die verwendeten Programme und den Ablauf des Experiments zu beschreiben.

4.1.1 Die Prüfkandidaten

Die AbUtil–Phase (siehe Abschnitt 3.2) läßt sich wiederum in zwei Phasen unterteilen. Sowohl in der Einübungsphase als auch in der Durchführungsphase hat jeder Student jede Technik einmal angewandt. Da es natürlich unsinnig ist, ein Programm nacheinander mit zwei verschiedenen Techniken zu testen, waren dafür sechs verschiedene Programme notwendig. Diese wurden identisch aus dem K&L–Experiment übernommen. Als Programmiersprache wurde durchgehend C gewählt. Bei der Einübungsphase handelte sich um folgende Programme:

count: Zählt die Anzahl der Buchstaben, Wörter und Zeilen eines Textes

series: Erzeugt eine benutzerdefinierte Zahlenfolge

tokens: Berechnet die Anzahl der einzelnen Buchstaben eines benutzerdefinierten Textes

Für die Durchführungsphase wurden die folgenden Programme verwendet:

ntree: Ermöglicht den Aufbau und die Manipulation von Baumstrukturen, wobei ein Knoten beliebig viele Nachfolger haben kann. Es ist dabei möglich, einen neuen Baum zu erzeugen, einen neuen Nachfolgerknoten für einen bestimmten Vaterknoten anzulegen, den Baum nach einem Knoten zu

durchsuchen, zu testen, ob zwei Knoten denselben Vaterknoten besitzen, und den Baum auszugeben.

cmdline: Analysiert die Kommandozeile eines vorgegebenen Programms auf Korrektheit. Dafür müssen verschiedene Bedingungen erfüllt sein: Bestimmte Optionen müssen genau einmal angegeben sein, andere Optionen erfordern weitere Parameter eines bestimmten Typs, manche Optionen schließen sich gegenseitig aus.

nametbl: Verwaltet eine Symboltabelle. In diese Symboltabelle können Bezeichner mit einem bestimmten Typ eingetragen werden, der Typ eines Bezeichners kann geändert und abgefragt werden, und die Symboltabelle kann ausgegeben werden.

Sämtliche Programme enthielten verschiedenartige Fehler. Dabei wurde darauf geachtet, daß sämtliche Fehler auch beim Ausführen beobachtet werden können, daß also nicht durch das Vorhandensein eines Fehlers ein anderer Fehler entweder eliminiert wird oder gar nicht auftreten kann. Die Gesamtzahl der Fehler sowie die Länge des Quelltexts der einzelnen Programme kann aus Tabelle 3 entnommen werden.

Programm	Codelänge	Fehleranzahl
count	44	8
series	89	4
tokens	127	5
ntree	260	8
cmdline	279	9
nametbl	282	8

Tab. 3: Länge und Fehleranzahl der einzelnen Programme.

4.1.2 Ablauf der AbUtil–Phase

Wie schon in Abschnitt 3.2 beschrieben, wurden im Gegensatz zum K&L–Experiment bei uns die Programme im Wochenrhythmus geprüft. Die Einübungsphase diente dazu, daß die Studenten die verschiedenen Techniken kennenlernen und an einem ersten Beispiel erproben, um auch die Testumgebung kennenzulernen. Die Aufteilung der Techniken und der zu prüfenden Programme war dabei für alle Studenten gleich: Sie prüften das Programm „count“ mit der Lesetechnik, „series“ mit dem funktionalen Test und „tokens“ mit dem strukturellen Test. Am Ende der Einübungsphase gab es eine Diskussion, bei der noch vorhandene Unklarheiten besprochen wurden. Außerdem gaben die Studenten schon zu diesem Zeitpunkt erste Bewertungen der einzelnen Prüftechniken an.

Für die Durchführungsphase wurden die Studenten in 3 Gruppen eingeteilt. Die Zuordnung zwischen Gruppe, Methode und Prüfkandidat wurde nach dem

Prinzip des *fractional factorial* [BS87] vorgenommen und ist in Tabelle 4 ersichtlich. Durch diese Anordnung wird einerseits gewährleistet, daß jedes Pro-

	ntree	cmdline	nametbl
Gruppe 1	Lesetechnik	struktureller Test	funktionaler Test
Gruppe 2	funktionaler Test	Lesetechnik	struktureller Test
Gruppe 3	struktureller Test	funktionaler Test	Lesetechnik

Tab. 4: Angewandte Prüftechniken der einzelnen Gruppen.

gramm mit jeder Technik geprüft wird und daß jede Gruppe jede Technik einmal durchführt. Zu erwähnen ist, daß während der gesamten AbUtil–Phase jeder Teilnehmer individuell gearbeitet hat. Eine Gruppenarbeit während dieser Phase ist nicht sinnvoll.

Um allen Teilnehmern einen einheitlichen Rahmen vorzugeben, haben wir wie auch im K&L–Experiment Formblätter verwendet, die einerseits genau angeben, wie eine bestimmte Prüftechnik anzuwenden ist, und andererseits auch gewährleisten, daß das Ergebnis der Prüfung in einem einheitlichen Rahmen festgehalten wird. Gerade der letzte Punkt vereinfacht auch stark eine spätere Auswertung. Im Anhang A haben wir sämtliche Formulare wiedergegeben, die wir in der AbUtil–Phase verwendet haben (Formblätter F1 – F13). Auch sämtliche weitere programmspezifischen Unterlagen, wie die Spezifikation oder der Quelltext, wurden als Formblätter ausgegeben. Dabei handelt es sich um die Formblätter F14 – F35 (siehe Anhang A).

Diese Formblätter wurden auch schon im K&L–Experiment verwendet. Wir haben diese lediglich in deutscher Sprache verwendet und an manchen Stellen leicht modifiziert. Außerdem wurde das Formblatt F10a neu erstellt: Nach der Einübungsphase erschien es uns sinnvoll, auch die Gründe für nicht erreichbare 100–prozentige Überdeckung beim strukturellen Test formell festzuhalten.

4.1.3 Die Testumgebung

Damit ein Test reproduzierbar ist, sollte das zu testende Programm während des Programmablaufs keine Eingaben vom Benutzer abfragen. Statt dessen ist es besser, wenn alle Eingaben in einer Datei festgehalten werden, die dann sukzessive eingelesen werden. Um außerdem die Testdurchführung komfortabel zu halten, wurde eine Testumgebung entworfen, die in leicht vereinfachter Form schon beim K&L–Experiment verwendet wurde. Die Testumgebung ist in Abbildung 5 schematisch wiedergegeben. Sie wird mit dem Befehl `run-suite` gestartet. Dieses Shell–Skript (siehe Anhang I) bestimmt zuerst das zu prüfende Programm. Danach sucht es im Verzeichnis `test-dir` alle Dateien, die die Endung `.test` besitzen. Jede Zeile in einer solchen Datei stellt einen separaten Testfall dar; das Programm wird also für jede Zeile einmal aufgerufen und bekommt als Aufrufparameter den Inhalt dieser Datei. Die Ausgaben des Programms werden einerseits auf dem Bildschirm ausgegeben und andererseits in

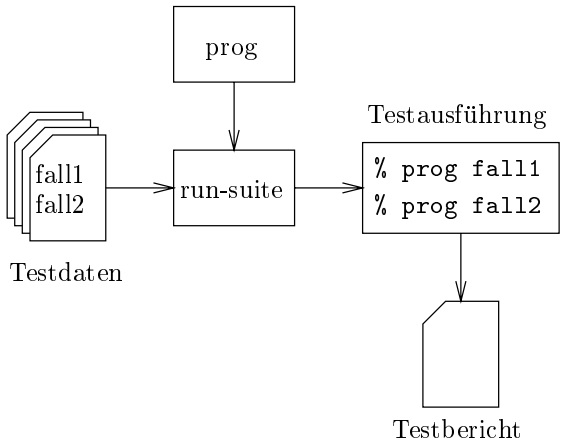


Abb. 5: Die Testumgebung.

der Datei `test-results.summary` protokolliert. Mit dem Befehl `make print` wird einerseits diese Datei auf dem Drucker ausgegeben, andererseits wird eine Kopie der Testfälle und der Ausgaben in ein Verzeichnis des Betreuers geschrieben.

4.1.4 Ergebnisse

Da in unserem Experiment das Hauptaugenmerk auf der AbEmbSys–Phase lag, möchten wir hier nur kurz die Ergebnisse der AbUtil–Phase vorstellen und auf die Unterschiede zu den Ergebnissen des K&L–Experiments eingehen. Wir beziehen uns hierbei nur auf die Durchführungsphase. Das Datenmaterial für die nachfolgenden Analysen umfaßt 39 Datenpunkte: Jeder der 13 Teilnehmer hat drei Programme geprüft.

Effektivität

Die Effektivität gibt an, wieviele Fehlverhalten bzw. Fehler gefunden wurden. In Abbildung 6 haben wir die durchschnittlichen Werte der gefundenen Fehlverhalten bzw. isolierten Fehler in Abhängigkeit der variierten Einflußfaktoren angegeben: Variiert wurden die Technik (CR=Lesetechnik, FT=funktionaler Test, ST=struktureller Test), das Programm (p_1=ntree, p_2=cmdline, p_3=nametbl) und Gruppenzugehörigkeit (Gruppe g_1-g_3). Neben den Mittelwerten finden sich zudem einige Kennzahlen, die wir im Zuge der einfaktoriellen Varianzanalyse (siehe [Bos94]) für unsere Untersuchung ermittelt haben.

Die Hypothese, daß die Einflußfaktoren keinen Einfluß auf die Anzahl der beobachteten Fehlverhalten bzw. isolierten Fehler haben, kann aufgrund des Datenmaterials nur für den Einflußfaktor Programm abgelehnt werden. Insbesondere ergibt sich in Hinblick auf die Techniken kein signifikanter Unterschied der Effektivität. Tendentiell schneidet aber der funktionale Test hier am besten ab.

Einfluß	Mittelwerte (Fehlverhalten)			SS_T	ν_T	SS_R	ν_R	F
Technik	CR=50 %	FT=60 %	ST=56 %	0.065	2	0.916	36	0.293
Programm	p_1=49 %	p_2=51 %	p_3=66 %	0.230	2	0.750	36	0.008
Gruppe	g_1=59 %	g_2=53 %	g_3=55 %	0.024	2	0.921	36	0.628
Teilnehmer	[13 Mittelwerte]			0.212	12	0.769	26	0.825

Einfluß	Mittelwerte der (Fehler)			SS_T	ν_T	SS_R	ν_R	F
Technik	CR=53 %	FT=67 %	ST=59 %	0.124	2	1.008	36	0.124
Programm	p_1=57 %	p_2=52 %	p_3=69 %	0.200	2	0.927	36	0.030
Gruppe	g_1=66 %	g_2=52 %	g_3=61 %	0.143	2	0.926	36	0.076
Teilnehmer	[13 Mittelwerte]			0.484	12	0.698	26	0.186

Abb. 6: Effektivität in Hinblick auf beobachtete Fehlverhalten und isolierte Fehler.

Aufwand

Ein weiterer Gesichtspunkt ist der Aufwand, der benötigt wird, um Fehlverhalten zu identifizieren oder die Fehler zu isolieren. In Abbildung 7 sind die Aufwände für die verschiedenen Schritte in Abhängigkeit der eingesetzten Technik wiedergegeben. Zudem finden sich die Niveaus aus der Varianzanalyse wieder. Man kann in beiden Fällen Abstufungen erkennen. Um Fehlverhalten zu identifizieren, braucht man beim strukturellen Test am längsten, während dies beim funktionalen Test am schnellsten durchführbar ist. Dies läßt sich damit begründen, daß beim funktionalen Test nur die Spezifikation für diesen Schritt verwendet wird, während bei den anderen beiden Techniken der Quelltext als Basis dient. Dies kann auch als Begründung dafür dienen, daß das Isolieren der Fehler beim funktionalen Test am längsten dauert, denn jetzt muß sich der Tester zuerst in den Quelltext einarbeiten, der bei den beiden anderen Techniken schon bekannt ist. In der Gesamtheit der beiden Testphasen lassen sich keine größeren Unterschiede mehr erkennen.

	CR	FT	ST	F
Fehlverhalten	117	98	143	0.010
isolierte Fehler	12	42	20	0.011
Gesamt	129	140	163	0.092

Abb. 7: Aufwand in Minuten.

Effizienz

Ein wichtiger Faktor zur Beurteilung der Eignung von Prüfmethoden ist die Effizienz, d.h. wieviele Fehler in einer bestimmten Zeit gefunden werden. Beim Beobachten der Fehlverhalten war der Funktionalen Test am effizientesten, während bei der Fehlerisolation dieser am schlechtesten abschneidet und Lese-

technik die beste Effizienz besitzt. Als Begründung lassen sich ähnliche Gründe wie die des Aufwands angeben.

	CR	FT	ST	F
Fehlverhalten	2.41	3.23	2.09	0.020
isolierte Fehler	27.31	8.30	18.64	0.000
Gesamt	2.41	2.40	2.00	0.370

Abb. 8: Effizienz in gefundene Fehler pro Stunde.

Zusammenfassung

Abschließend möchten wir die Ergebnisse nochmals zusammenfassen und mit den Ergebnissen des K&L–Experiments in Beziehung setzen. Die letzte Spalte in Tabelle 5 gibt dabei an, ob die Ergebnisse mit dem K&L–Experiment übereinstimmen.

		K&L	AbUtil	
Effektivität	Fehlverhalten	$CR \cong FT \cong ST$	$CR \cong FT \cong ST$	ja
	Fehler	$CR \cong FT \cong ST$	$CR \cong FT \cong ST$	ja
Aufwand	Fehlverhalten	$CR > ST > FT$	$ST > CR > FT$	nein
	Fehler	$FT > ST > CR$	$FT > ST > CR$	ja
	Gesamt	$CR \cong FT \cong ST$	$CR \cong FT \cong ST$	ja
Effizienz	Fehlverhalten	$FT > ST > CR$	$FT > ST \cong CR$	nein
	Fehler	$FT > ST \cong CR$	$CR > ST > FT$	nein
	Gesamt	nicht beschrieben	$CR \cong FT \cong ST$	
Einfluß von Motivation und Ausbildung		kein Zusammenhang	kein Zusammenhang	ja
Einschätzbarkeit der Leistung		nicht gegeben	kaum gegeben	ja

Tab. 5: Vergleich der Ergebnisse der AbUtil–Phase.

4.2 Prüfen eingebetteter Systeme (AbEmbSys–Phase)

Im folgenden beschreiben wir die Durchführung und insbesondere die Ergebnisse der AbEmbSys–Phase. Ziel dieses Experiment–Abschnittes war es, die Eignung statischer und dynamischer Prüfaktivitäten insbesondere in Hinblick auf eingebettete Systeme und deren Spezifika vergleichend zu betrachten.

Neben eines Wechsels der Betrachtungsgegenstände von einfachen Kommandozeilensystemen hin zu eingebetteten Systemen gab es auch Änderungen in Hinblick auf die eingesetzten Prüfaktivitäten. Wir gingen in diesem Teil des

Praktikums weg von den „reinen“ Techniken, wie sie in der AbUtil–Phase eingesetzt wurden, und setzten statt deren kombinierte Techniken ein, wie sie eher dem aktuellen Einsatz in der Industrie entsprechen (siehe Abschnitt 4.2.2). Die 13 Teilnehmer arbeiteten in diesem Experiment–Abschnitt auch nicht mehr einzeln, sondern in vier Gruppe mit je drei bzw. vier Teilnehmern.

In vier aufeinanderfolgenden Wochen wurde je ein System überprüft und zwar von zwei Teams mittels Review und von zwei Teams mittels Test. Dabei wurden die jeweils eingesetzten Prüfaktivitäten immer wieder getauscht (siehe auch Abbildung 9). Die dabei erwähnten Systeme (z. B. Mikrowelle) werden im nächsten Abschnitt näher erläutert.

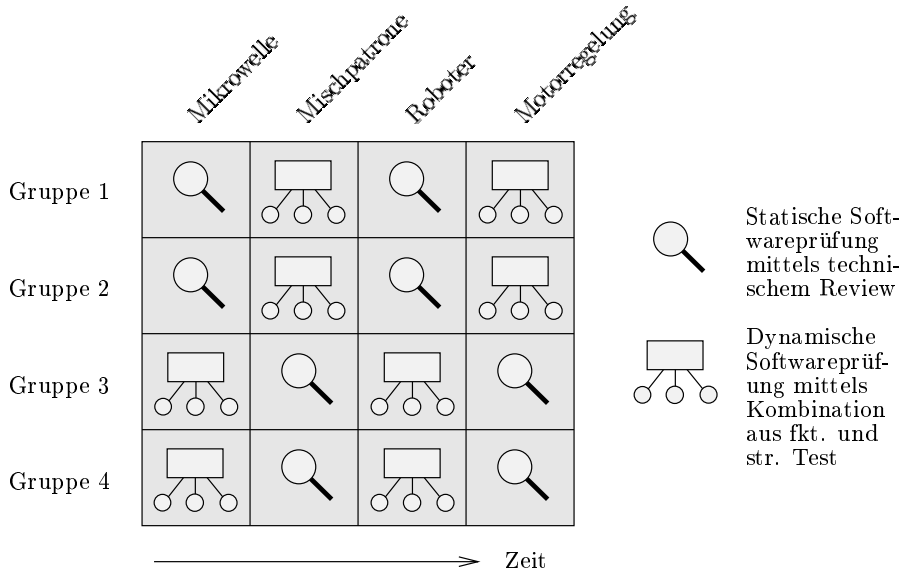


Abb. 9: Ablauf der AbEmbSys–Phase.

Die vertikale Replikation (d. h. mehrere Teams, die dieselbe Technik gleichzeitig anwenden) hatte dabei die Aufgabe, durch Verdopplung der Datenmenge Anomalien besser erkennen zu können. Die horizontale Replikation (d. h. mehrmaliges Anwenden derselben Technik durch dasselbe Team) ermöglichte neben einer höheren Datenmenge auch die Untersuchung von Lern- oder Reifeeffekten.

4.2.1 Verwendete Systeme

Bei den eingesetzten Systemen handelte es sich jeweils um C–Programme. Als Zielplattform wählten wir bewußt keinen Mikrocontroller sondern UNIX–Rechner. Dies geschah aus den folgenden Gründen:

- Den Studenten ist der Umgang mit UNIX–Rechnern vertraut. Sie mußten sich also in keine neue Umgebung einarbeiten. Ein Einführen neuer Rechnerstrukturen hätten neben den Prüfaktivitäten weitere neue Aspekte für

die Studenten gebracht, was wiederum Aussagen zum Einfluß der angewandten Prüftechniken auf die Ergebnisse erschwert hätte.

- Das Werkzeug zur Überprüfung der Codeüberdeckung im strukturellen Test (GCT, siehe Abschnitt 2.3.2) ist für UNIX–Rechner konzipiert. Ein Einsatz dieses oder eines vergleichbaren Werkzeugs auf Mikrocontrollern wäre mit immensem Zeit- und/oder Kostenaufwand verbunden gewesen.

Bei den Systemen haben wir jeweils einen einheitlichen Aufbau gewählt. Sämtliche externe Komponenten der Systeme wurden dabei nicht in Hardware, sondern durch C–Funktionen realisiert. Diese waren aber für die Teilnehmer nur durch eine Spezifikation in Form kommentierter C–Header–Dateien sichtbar. Die Ein- und Ausgabe der Systeme wurde über Ereignis- und Protokolldateien gesteuert. Dadurch war es möglich, die jeweilige Programmausführung trotz interaktiver Systeme reproduzierbar zu gestalten. In Abbildung 10 haben wir den prinzipiellen Aufbau der untersuchten Systeme graphisch illustriert.

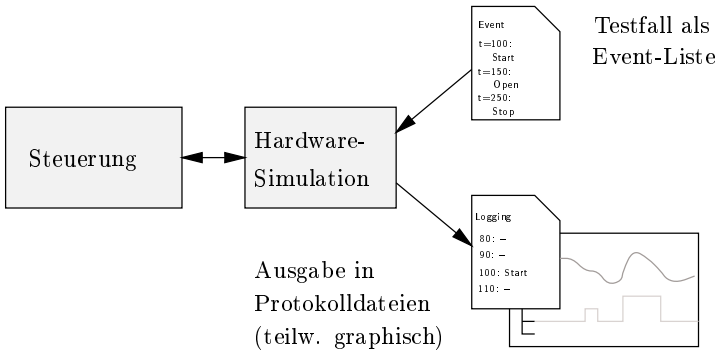


Abb. 10: Schematische Darstellung des Systemaufbaus.

In den Eingabedateien werden die Ereignisse, auf die das System reagieren soll, spezifiziert. Dazu werden sowohl das Ereignis selbst als auch der Zeitpunkt bzw. die Zeitspanne, zu der dieses Ereignis eintritt bzw. anliegt, angegeben. Die Ausgabe erfolgt über textuelle oder graphische Ausgaben. Bei den graphischen Ausgaben handelt es sich um Zeitliniendiagramme, die das Verhalten von bestimmter Hardwareparameter im Verlauf der Zeit anzeigen, oder Ort–Zeit–Diagrammen, die Bewegungen im Raum illustrieren.

In der Hardwaresimulation der Systeme werden sowohl die Ein- und Ausgabedateien gelesen bzw. geschrieben als auch die Veränderungen der Umgebung in Abhängigkeit von Eingaben und Anweisungen des Steuerteils berechnet. Der Steuerteil selbst repräsentiert die Softwarekomponente der betrachteten Systeme, die im Rahmen der Prüfaktivitäten untersucht wurden.

In den Abbildungen 11 bis 14 haben wir die eingesetzten Systeme kurz charakterisiert. Neben einer kurzen Beschreibung der Funktionalität finden sich dort auch Angaben zu den zugrundeliegenden Software–Architekturen und enthaltenen Fehlern.

Zur Klassifikation der Fehler haben wir die vier Fehlerklassen *Unbedeutend*, *Bedeutend*, *Schwerwiegend* und *Kritisch* eingesetzt. Deren Definition findet sich in Abschnitt 2.2.2 auf Seite 16. Im folgenden werden wir oft von *KS–Fehlern* sprechen. Wir meinen damit die Klasse der kritischen und schwerwiegenden Fehler, also die Art von Fehlern, die einen bestimmungsgemäßen Einsatz der Systeme verhindert.

System 1: Mikrowellensteuerung

Umfang:	308 Zeilen C–Code, davon 41 Zeilen mit Kommentar und 29 Zeilen in Header–Dateien
Architektur:	Zustandsautomat, Einprozeßsystem
Zeitbedingungen:	Hart
Fehler:	7 KS–Fehler

Die Mikrowellensteuerung hat die Aufgabe, den typischen Betrieb einer Mikrowelle zu gewährleisten. Dazu gehört das Entgegennehmen von Benutzeranweisungen (Laufzeit und Leistungsstufe) und der Ansteuern der Mikrowellenspule. Neben der zeitlichen Überwachung der Betriebszeit gibt es eine besonders scharfe Zeitbedingung, nämlich das Abschalten der Mikrowellenspule innerhalb weniger Millisekunden nach dem Öffnen der Gerätetür.

Abb. 11: Mikrowellensteuerung.

System 2: Elektronische Mischpatrone

Umfang:	322 Zeilen C–Code, davon 47 Zeilen mit Kommentar und 26 Zeilen in Header–Dateien
Architektur:	Drei parallele Prozesse, Fuzzy–Regler
Zeitbedingungen:	Weich
Fehler:	4 KS–Fehler

Eine elektronische Mischpatrone hat, ähnlich wie ihr mechanische Pendant auch, die Aufgabe, zwei verschiedene Flüssigkeiten in einem einstellbaren Verhältnis zu mischen. Durch das Messen des aktuellen Mischungsverhältnisses ist eine laufende Nachregelung möglich. Das Hauptaugenmerk bei diesem System lag insbesondere auf der Interaktion der parallelen Prozesse.

Abb. 12: Elektronische Mischpatrone.

System 3: Robotersteuerung

Umfang:	448 Zeilen C–Code, davon 58 Zeilen mit Kommentar und 14 Zeilen in Header–Dateien
Architektur:	Sequentialisiertes Mehrprozeßsystem, PI–Regler
Zeitbedingungen:	Weich
Fehler:	5 KS–Fehler

Hier handelt es sich um die Steuerungskomponente eines autonomen, mobilen Roboters. Sein Standardverhalten ist ein geradeaus fahren, wobei er auf Hindernisse reagiert (durch Ausweichen nach einer Kollision). Erkennt die Steuerung Pfeiftöne, so wird in Abhängigkeit der Tonhöhe eine Kurvenfahrt durchgeführt.

Abb. 13: Robotersteuerung.

System 4: Motorsteuerung

Umfang:	485 Zeilen C–Code, davon 92 Zeilen mit Kommentar und 83 Zeilen in Header–Dateien
Architektur:	Zwei parallele Prozesse, jeweils ein Zustandsautomat
Zeitbedingungen:	Hart
Fehler:	6 KS–Fehler

Die Aufgabe der Motorsteuerung ist es, den Motor eines schweren landwirtschaftlichen Gerätes zu regulieren. Ein Teil der Steuerung sitzt dabei im Fahrerstand und nimmt die Benutzereingaben (Start, Stop, Pedal) entgegen. Über ein Bussystem werden diese Eingaben an den zweiten Teil der Steuerung weitergegeben, der im Motorblock untergebracht ist. Dieses Teilsysten überwacht den Motorbetrieb, indem es Kraftstoffpumpe, Anlasser und Batterie ansteuert und kontrolliert.

Abb. 14: Motorsteuerung.

4.2.2 Ablauf und Datenerfassung

Der Ablauf einer einzelnen Untersuchung sah wie folgt aus: In einer einführenden Sitzung stellten wir den Studenten (Reviewer wie Tester) mit einigen wenigen Folien das zu betrachtende System vor. Neben einer kurzen Beschreibung des funktionalen Verhaltens gingen wir auch auf spezielle Systemaspekte wie Format der Eingabedatei oder Interpretation der graphischen Ausgaben ein. Im Anschluß an die Vorstellung und der Beantwortung aktueller Fragen wurden die Materialien (Spezifikation und Vorbereitungsformulare für die Inspektoren, siehe Anhang B–E) ausgeteilt.

In der folgenden Woche, in der der Test und die Review–Vorbereitung durch die

Studenten erfolgte, standen wir jeweils für Fragen zur Verfügung. Eine Woche nach Ausgabe der Materialien lieferten die Studenten dann ein Testprotokoll ab bzw. nahmen an den Review–Sitzungen teil.

Der Ablauf des Testens, das nun eine Kombination aus funktionalem und strukturellen Test war, wurde von uns durch ein Rahmen–Testprotokoll (siehe Anhang F) vorgegeben. Im folgenden findet sich eine Auflistung der Aktivitäten, die im Rahmen des Tests durchgeführt und protokolliert werden sollten. Näheren Angaben zur Vorgehensweise beim Testen finden sich in Abschnitt 2.3.

1. Erstellen des Klassifikationsbaums und Aufstellen von Äquivalenzklassen mit Hilfe des Werkzeuges CTE (siehe Abschnitt 2.3).
2. Auflistung der konkreten Testfälle mit den erwarteten Ergebnissen.
3. Erstellen eines Protokolls der Testläufe und Vergleich der erwarteten mit den tatsächlichen Ergebnissen. Zur Protokollierung konnten die Ausgabedateien, die durch die vorgegebene Testumgebung erzeugt wurden, verwendet werden.
4. Analyse der Ausgabe und Auflistung der Fehlverhalten aus dem funktionalen Test.
5. Ermitteln der Überdeckung durch Testfälle aus funktionalem Test.
6. Ableiten von zusätzlichen Testfällen, um eine maximale Überdeckung gemäß den Kriterien des eingesetzten Instrumentierers zu erreichen (siehe Abschnitt 2.3.2).
7. Ermitteln der Überdeckung durch den erweiterten Testfallsatz.
8. Begründen der Umstände, die eine 100 %ige Überdeckung nicht möglich machen.
9. Beschreiben der Fehlerverhalten, die bei Ausführung der zusätzlichen Testfällen beobachtet wurden.
10. Beschreiben der Fehler, die mit Hilfe der beobachteten Fehlverhalten isoliert wurden.

Den Anfang eines jeden Testprotokolls (siehe Anhang F) bildete ein Formblatt, in dem zusätzliche Informationen, die sich insbesondere auf Aufwände und Motivation bezogen, erhoben wurden.

Beim Review waren die Vorgaben bezüglich der Dokumentation weniger streng. In der Vorbereitungsphase notierten sich die Inspektoren auf ausgeteilten Formblättern (siehe Anhang G.1) die entdeckten Fehler und notierten sich Fragen, die sie in der Sitzung an den Autor stellen wollten. Eine spezielle Technik zur Vorbereitung, wie beispielsweise Szenarien oder Perspektiven [BGL+96], wurde nicht vorgegeben. Als Hilfestellung gab es eine Checkliste (siehe Anhang G.4), die vor allem die Einhaltung von Programmierrichtlinien abfragte. Unsere Anregung, die in der AbUtil–Phase vorgestellte Lesetechnik durch schrittweise Abstraktion zu verwenden, wurde von Seite der Studenten nicht berücksichtigt.

In der Review–Sitzung waren außer den Inspektoren noch der Autor des jeweiligen Systems sowie ein Moderator anwesend. Die Protokollierung wurde vom Autor übernommen. Bei der Sitzung wurden zuerst einige organisatorische Fragen geklärt (z. B. Aufwände und gefundene Fehler, siehe Anhang G.3).

Dann führte der Moderator in überschaubaren Einheiten durch die Listings des ausgeteilten Systems und holte dabei die Anmerkungen der Inspektoren zu diesem Abschnitt ein. Die Fragen an den Autor wurden, soweit sie sich auf Details bezogen, ebenfalls im Rahmen des schrittweisen Durchgehens gestellt und beantwortet.

4.2.3 Analyse

In diesem Abschnitt wollen wir das Ergebnis der Datenanalyse präsentieren. Die einzelnen Daten wurden dabei sowohl aus den von den Teilnehmern ausgefüllten Formblättern, als auch aus den Prüfungs- bzw. Sitzungsprotokollen gewonnen.

Da die Datenmenge, die dieser Analyse zugrundeliegt, sehr klein ist (jeweils acht Datensätze für Test und Review), haben wir meist auf tiefgehende statistische Analyseverfahren verzichtet und statt dessen vor allem Techniken der beschreibenden Statistik eingesetzt.

Da wir für Test und Review nicht überall dieselben Daten erhoben haben, wollen wir die einzelnen Ergebnisse nicht sofort einander gegenüberstellen, sondern zuerst das Datenmaterial je Technik präsentieren. Am Ende dieses Abschnitts stellen wir dann Ergebnisse — soweit sinnvoll — einander gegenüber.

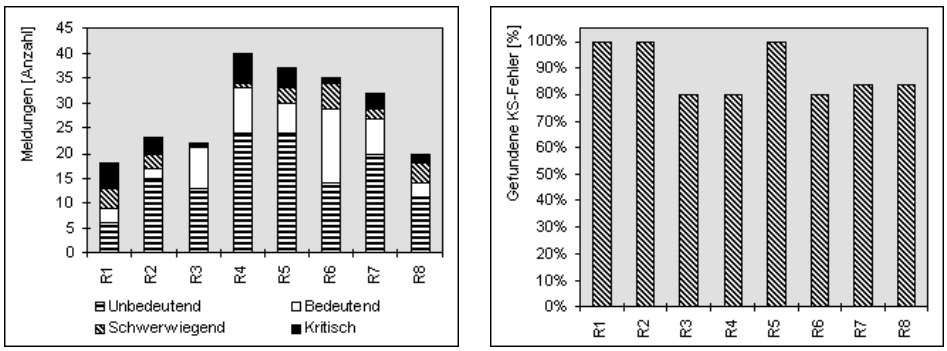
In Anhang H sind die durchgeführten Analysen zusammen mit den dazu notwendigen Metriken, Erhebungstechniken und erzielten Beobachtungen nochmals in komprimierter Form zusammengefaßt. Diese Angaben sollen bei ganzer oder teilweiser Replikation helfen, die Datenerfassung und -analyse durchzuführen.

Ergebnisse aus den Reviews

Effektivität

Zuerst wollen wir die Effektivität der Technik, d. h. die Anzahl der damit gefundenen Fehler, betrachten. In Abbildung 15(a) ist die Summe aller Meldungen, die in den acht Review–Sitzungen R1 bis R8 gemeldet wurden, aufgetragen. Die unterschiedliche Schraffur gibt dabei die Kritikalität der Meldungen an. Interessant ist die hier durchgängig hohe Anzahl der Meldungen. Beim Analysieren der Meldungen beobachteten wir aber, daß sich viele Meldungen nicht auf echte Fehler bezogen, sondern Vorschläge in Richtung Lesbarkeit, Verständlichkeit und Wartbarkeit waren. Viele dieser Verbesserungsmeldungen ergaben sich auf Basis der der eingesetzten Checkliste.

Beschränken wir uns nun bei der Betrachtung auf die Klasse der KS–Fehler. In Abbildung 15(b) haben wir die gefundenen KS–Fehler mit der Gesamtzahl der enthaltenen KS–Fehler in Beziehung gesetzt. Das Resultat ist erstaunlich: Die Trefferquote lag zwischen 80 % und 100 %. Das Resultat ist insbesondere auch deswegen verblüffend, da sich einige Fehler auch mit Zeitbedingungen oder Nebenläufigkeit beschäftigten, also Fehlerarten, von denen wir angenommen hatten, daß sie in einem Review wenig entdeckt werden.



(a) (b)

Abb. 15: Gesamtzahl der Meldungen beim Review (a) und Prozentsatz der gefundenen KS–Fehler.

Eine Betrachtung der nicht entdeckten Fehler ergab keine systematische Verteilung über die möglichen Fehlerarten. Gemeinsamkeiten konnte nicht abgeleitet werden.

Synergieeffekte bei der Review–Sitzung

Ein Kritikpunkt an Reviews ist oft der hohe zeitliche Aufwand, der mit einem vollständigen Review–Prozeß verbunden ist. Zur Untersuchung der mit dieser Kritik oft verbundenen Forderung, die Sitzungen wegfallen zu lassen, untersuchten wir die Quellen der einzelnen Fehlermeldungen. In Abbildung 16 haben wir diese Aufteilung zusammengetragen. Auffällig ist hier der hohe An-

	R1	R2	R3	R4	R5	R6	R7	R8	∅
Sitzung	28 %	40 %	25 %	43 %	51 %	50 %	22 %	15 %	34.2 %
Einzel	47 %	24 %	42 %	21 %	4 %	35 %	12 %	66 %	31.4 %
Mehrere	25 %	36 %	33 %	36 %	45 %	15 %	66 %	19 %	34.4 %

Sitzung: Prozentsatz der Fehler, die erst in der Sitzung entdeckt wurden.
Einzel: Prozentsatz der Fehler, die nur von einem einzelnen Inspektor entdeckt wurden.
Mehrere: Prozentsatz der Fehler, die von mehreren Inspektoren entdeckt wurden.

Abb. 16: Quellen der Fehlermeldungen.

teil an Fehlern (34 %), die erst im Rahmen der Reviewsitzung gefunden wurden. Die Sitzungen dienten also mitnichten nur zum Zusammentragen der einzelnen Fehlermeldungen, sondern stellten einen erheblichen Erfolgsfaktor im Rahmen dieser Prüfaktivitäten dar.

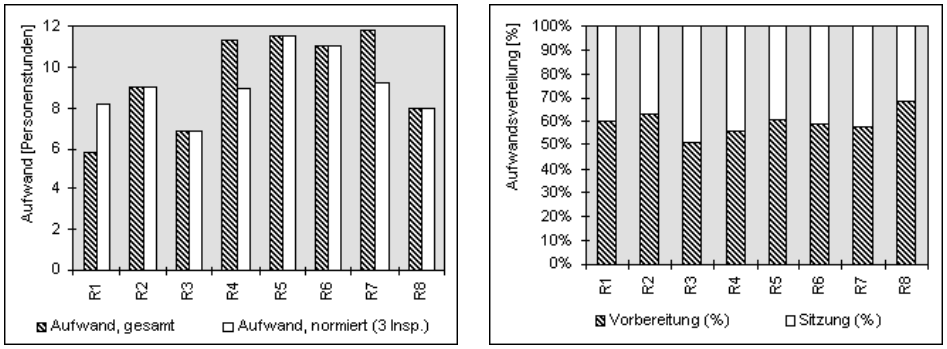
Analysiert man allerdings die Art der Fehler, die in den Reviewsitzungen zusätzlich gefunden wurden, so sind relativ wenig KS–Fehler darunter. Meist handelte

es sich um *Analogiefehler*, d. h. Fehler derselben Art, die mehrfach auftauchen: Ein Inspektor mahnt einen Fehler an, worauf ein anderer Inspektor feststellt, daß sich diese Situation auch noch an anderen Stellen im Code finden läßt.

Ein Punkt, den wir nicht weiter systematisch untersucht haben, sei hier aber dennoch erwähnt: Das Ablehnen von Fehlern in der Reviewsitzung. In einigen Fällen meldete ein Inspektor einen Fehler, der dann bei der gemeinsamen Betrachtung in der Sitzung von den anderen wieder (zurecht) verworfen wurde. Auch dieser Aspekt kann zur Rechtfertigung von Reviewsitzungen herangezogen werden.

Aufwände

Zu jeder Nutzenbetrachtung gehört auch die Berücksichtigung der zugehörigen Kosten. Beim Review schlagen sich Aufwände vor allem in den wiederkehrenden Aktivitäten wie Vorbereitung und Sitzung nieder. Aus diesem Grund haben wir uns bei der Kostenbetrachtung auf diese Aktivitäten konzentriert. In Abbildung 17(a) finden sich die Aufwände je Sitzung in Personenstunden als Absolutwerte. Da aufgrund der wechselnden Zahl von Inspektoren (jeweils zwischen zwei und vier) diese Werte schlecht vergleichbar sind, finden sich in dieser Abbildung auch die auf 3 Inspektoren normierten Werte.



(a) (b)

Abb. 17: Absolutaufwände beim Review (a) und relative Aufwandsverteilung auf Vorbereitung und Reviewsitzung (b).

Für sich betrachtet sind diese Werte allerdings relativ wenig aussagekräftig. Sie dienen aber als Basis für zwei weitere Untersuchungen: Die Beziehung Vorbereitung/Sitzung (1) und das Verhältnis von Programmgröße und Aufwand (2).

Zu (1): In Abbildung 17(b) haben wir die Verteilung von Aufwänden für die Vorbereitung und die Sitzung relativ zueinander aufgetragen. Auffällig ist hierbei, daß das Verhältnis zwischen Vorbereitung und Sitzung in jedem Review ungefähr 60:40 beträgt, unabhängig von der Programmgröße und der Anzahl der beteiligten Personen.

Zu (2): Die Beziehung zwischen Programmgröße und Aufwand haben wir in Abbildung 18 gegeneinander aufgetragen. Hier wollen wir nun die in Abschnitt

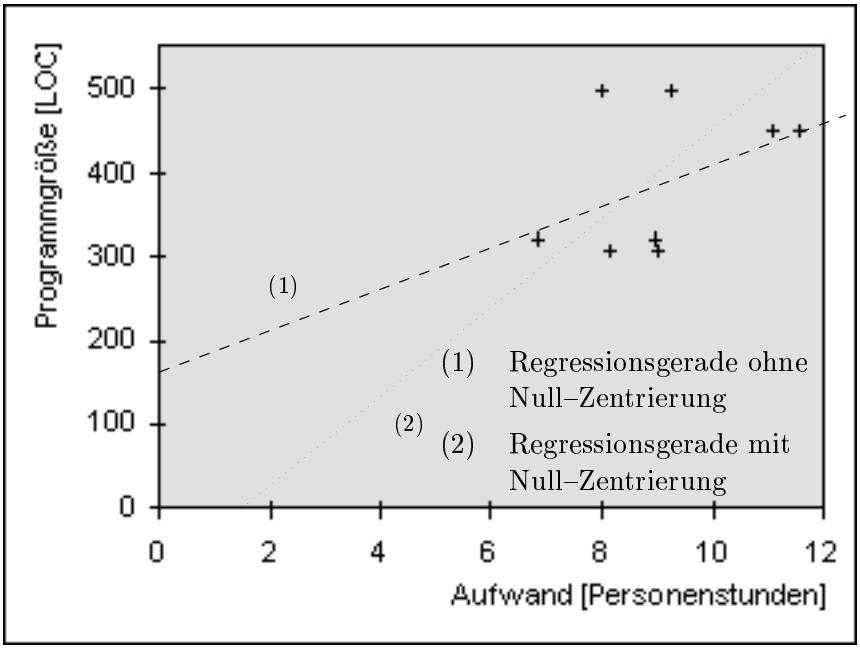


Abb. 18: Beziehung zwischen Aufwänden und Programmgröße.

3.1 geäußerte Vermutung eines quasi–linearen Zusammenhangs zwischen Aufwand und Programmgröße (zumindest in dem Bereich der Programmgröße, die im Rahmen einer Reviewsitzung sinnvoll bearbeitet werden kann, d. h. ca. 100–1000 Zeilen) näher untersuchen.

Der empirische Korrelationskoeffizient[5] für diese Datenmenge $r = 0.42$ stützt diese Annahme nur bedingt. Fügen wir der Datenmenge aber noch den Datenpunkt $(0, 0)$ hinzu[6], so erhalten wir $r = 0.86$. In Abbildung 18 haben wir auch die entspechenden Regressionsgeraden mit und ohne Null–Zentrierung eingezeichnet. Mit Null–Zentrierung erhalten wir darüber hinaus den empirischen Zusammenhang von

$$\text{Aufwand} = 0.018ph \cdot \text{Größe} + 1.47ph,$$

wobei ph für Personenstunde steht.

[5]Der empirische Korrelationskoeffizient r einer zweidimensionalen Stichprobe ist ein Maß dafür, wie sehr die Werte einer Beziehung $y = ax + b$ folgen. Es gilt dabei $-1 \leq r \leq 1$. Liegen alle Stichprobenwerte auf einer Geraden, so gilt $r = 1$ (bzw. $r = -1$ bei negativer Steigung). Je näher $|r|$ an 0 liegt, desto weniger ist eine solche Beziehung gegeben [Bos94].

[6]Dies ist gerechtfertigt, da durch diesen Datenpunkt eine wichtige zusätzliche Information über den zugrundeliegenden Sachverhalt modelliert wird. So ist es sicherlich sinnvoll, eine Null–Zentrierung der Beziehung Größe $\leftrightarrow$ Aufwand anzunehmen.

Effizienz

Nach der getrennten Betrachtung von Effektivität einerseits und Aufwand andererseits kann nun die Kombination beider Maße, die Effizienz, betrachtet werden. Da aber die daraus resultierenden Werte in isolierter Betrachtung wenig interessant sind, wollen wir diese Daten erst bei der vergleichenden Betrachtung von Review und Test auf Seite 51 präsentieren.

Reifeeffekte

Die horizontale Replikation, d. h. das mehrmalige Anwenden derselben Technik durch dasselbe Team, ermöglicht es, Reifeeffekte zu untersuchen. Dazu betrachten wir die beiden Hypothesen I_0 und J_0

- I_0 Der Prozentsatz der identifizierten KS–Fehler beim Review ist unabhängig von der Häufigkeit der Anwendung der Technik.
- J_0 Die Effizienz in Hinblick auf die KS–Fehler beim Review ist unabhängig von der Häufigkeit der Anwendung der Technik.

Wenden wir auf die entsprechenden Daten die einfaktorielle Varianzanalyse (F–Test) [Bos94] an, so ergibt sich, daß beide Hypothesen nicht abgelehnt werden können. Die entsprechenden Überschreitungswahrscheinlichkeiten liegen bei Betrachtung von I_0 bei 0.92 und im Falle von J_0 bei 0.47. Die Annahme von Reifeeffekten findet also keine Begründung in den gemessenen Werten.

Die dieser Auswertung zugrundeliegenden Daten sowie einige Werte der Analyse haben wir in Abbildung 19 zusammengetragen. Im Inneren der beiden Tabellen finden sich dabei die Meßwerte aus der ersten bzw. zweiten Anwendung der Technik. Die übrigen Werte ergeben sich im Zuge der Auswertung, die beispielsweise in [Bos94, S. 91ff] näher erläutert ist.

Untersuchung der Hypothese I_0:

	n_i	% KS–Fehler				$\frac{x_i^2}{n_i}$
1. Anw.	4	100	100	80	80	32400
2. Anw.	4	100	80	83	83	29929
Summe	8					62329

$v_{ber} = \frac{1/1}{672.5/6} = 0.0089$
$f_{(1,6)90\,\%} = 3.78$
$\Rightarrow I_0$ nicht abl.

Untersuchung der Hypothese J_0:

	n_i	% KS–Fehler pro Stunde				$\frac{x_i^2}{n_i}$
1. Anw.	4	1.2	0.77	0.58	0.35	1.67
2. Anw.	4	0.43	0.36	0.42	0.62	0.84
Summe	8					2.51

$v_{ber} = \frac{0.07/1}{0.86/6} = 0.59$
$f_{(1,6)90\,\%} = 3.78$
$\Rightarrow J_0$ nicht abl.

Abb. 19: Untersuchung von Reifeeffekten.

Ergebnisse aus den Tests

Effektivität

Beim Analysieren der Tests werden wir uns auf die Klasse der KS–Fehler beschränken. Andere Fehler werden aufgrund ihrer Natur typischerweise beim Testen wenig gefunden, da sie oft eher die Programmstruktur als konkrete Fehlverhalten betreffen. Eine Ausnahme bilden kosmetische Fehler, deren Auswirkungen dem Benutzer durch „unschöne“ Ausgaben im weitesten Sinne sichtbar werden.

In Abbildung 20 findet sich die relative Anzahl der in den einzelnen Tests gefundenen Fehler. Die unterschiedliche Schraffur gibt dabei an, in welcher Testphase — funktioneller oder struktureller Test — die jeweiligen Fehler entdeckt worden sind. Auffällig ist hier die relativ geringe Anzahl an KS–Fehlern: 25 % bis 75 %

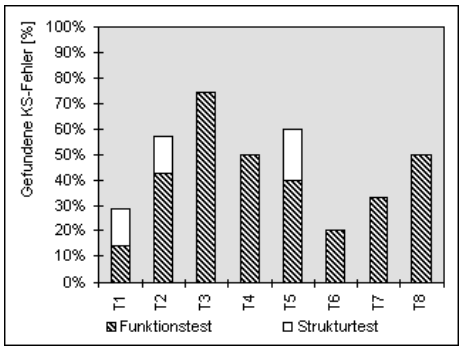


Abb. 20: Relativer Anteil der gefundenen KS–Fehler.

aller Fehler blieben unentdeckt.

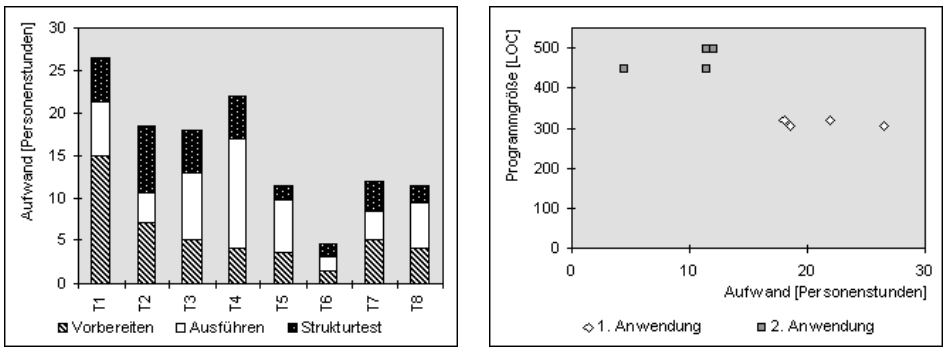
Die Tatsache, daß nur sehr wenig Fehler beim strukturellen Test gefunden wurden, ist dagegen wenig verwunderlich, da dieser als Ergänzung zum funktionalen Test durchgeführt wurde.

Aufwände

Zur Betrachtung der Aufwände wollen wir die Testvorgehensweise in drei große Abschnitte einteilen: (1) Aufstellen der Testfälle für den funktionalen Test, (2) Durchführung des funktionalen Tests mit Analyse der Ergebnisse und (3) Erweiterung der Überdeckung durch strukturellen Test.

In Abbildung 21(a) haben wir die jeweiligen Aufwände in den einzelnen Tests aufgetragen. Auffällig sind hier vor allem die sich ändernden Gesamtaufwände in Abhängigkeit von der ersten bzw. zweiten Anwendung dieser Teststrategie. Beim Punkt *Reifeeffekte* werden wir darauf näher eingehen.

Ein Zusammenhang zwischen Aufwand und der Programmgröße scheint es daher kaum zu geben. Abbildung 21(b) illustriert dies anschaulich. Auch die



(a) (b)

Abb. 21: Aufwände in Personenstunden in den einzelnen Testabschnitten (a) und Beziehung zwischen Aufwänden und Programmgröße (b).

Betrachtung des empirischen Korrelationskoeffizienten $r = 0.165$ stützt diese Vermutung.

Ein weiterer interessanter Punkt sind die hohen Aufwände in der Ausführungsphase. Dies lag — wie wir auch bei Gesprächen mit den Teilnehmern immer wieder hörten — an der nicht optimalen Testumgebung und insbesondere an einem nichtdeterministischen Zeitverhalten der Systeme. An dieser Stelle machte sich unsere Entscheidung für UNIX als Basisplattform negativ bemerkbar. Je nach Rechnerauslastung liefen die Systeme merklich unterschiedlich und erschwerten so eine zuverlässige und reproduzierbare Ausführung der Testkandidaten.

Effizienz

Auch an dieser Stelle wollen wir die Effizienz, also den Quotienten aus Anzahl der gefundenen Fehler und der benötigen Zeit, nicht näher betrachten und verweisen statt dessen auf die gegenüberstellende Betrachtung von Test und Review auf Seite 51.

Reifeeffekte

Analog zur Untersuchung der Reifeeffekte beim Review wollen wir auch hier die beiden Hypothesen:

I'_0 Der Prozentsatz der identifizierten KS–Fehler beim Test ist unabhängig von der Häufigkeit der Anwendung der Technik.

J'_0 Die Effizienz in Hinblick auf die KS–Fehler beim Test ist unabhängig von der Häufigkeit der Anwendung der Technik.

mit Hilfe der einfaktoriellen Varianzanalyse testen. In Abbildung 22 finden sich die zugrundeliegenden Daten sowie einige Werte der Analyse.

Untersuchung der Hypothese I_0':

	n_i	% KS–Fehler				$\frac{x_i^2}{n_i}$
1. Anw.	4	29	57	75	75	13924
2. Anw.	4	60	40	33	50	5402.3
Summe	8					19326.3

$v_{ber} = \frac{990.13/1}{4802.8/6} = 1.24$
$f_{(1,6)90\,\%} = 3.78$
$\Rightarrow I_0'$ nicht abl.

Untersuchung der Hypothese J_0':

	n_i	% KS–Fehler pro Stunde				$\frac{x_i^2}{n_i}$
1. Anw.	4	0.08	0.22	0.17	0.14	0.088
2. Anw.	4	0.26	0.44	0.17	0.26	0.321
Summe	8					0.409

$v_{ber} = \frac{0.0363/1}{0.0508/6} = 4.29$
$f_{(1,6)90\,\%} = 3.78$
$\Rightarrow J_0'$ **ablehnen**

Abb. 22: Untersuchung von Reifeeffekten.

Auch hier können wir im Fall der Effektivität die Hypothese nicht ablehnen. Die Überschreitungswahrscheinlichkeit für I_0' liegt hier bei 0.31. Für die Effizienz hingegen können wir die Hypothese J_0' zum Signifikantniveau von 10 % ablehnen. Bei der zweiten Anwendung der Testtechnik fanden die Studenten also in etwa dieselbe Anzahl an Fehlern in deutlich kürzerer Zeit.

Einschätzung der eigenen Leistung

Beim Testen haben wir Angaben zur eigenen Einschätzung der Teilnehmer in Hinblick auf ihre jeweilige Leistung beim Testen abgefragt. In Abbildung 23 sind diese Angaben zusammen mit der jeweils tatsächlich beobachteten Leistung aufgetragen. Ein Zusammenhang zwischen subjektiver Einschätzung und ob-

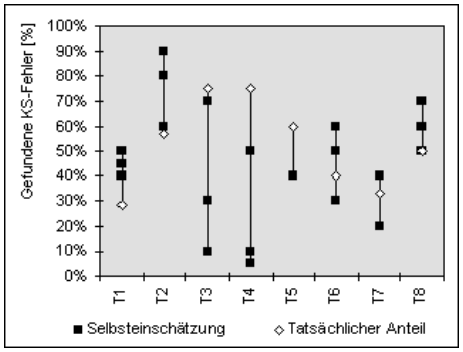


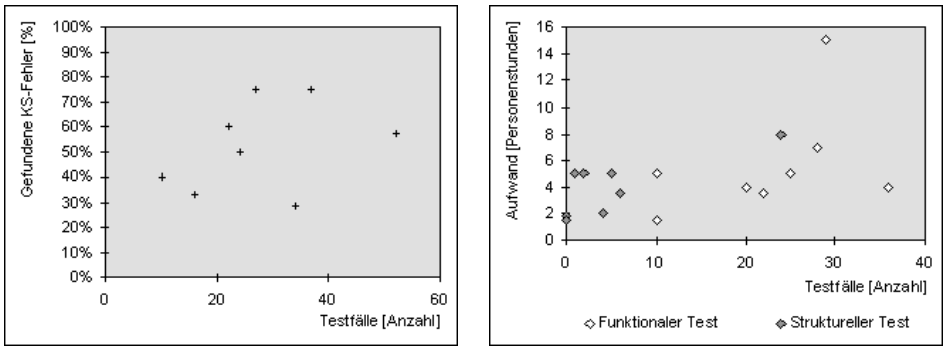
Abb. 23: Einschätzung der eigenen Leistung.

jektiver Leistung ist hier nicht zu erkennen. Diese Beobachtung stützt andere Ergebnisse (z. B. [BS87, S. 1288]), die ebenfalls davon sprechen, daß eine Eigenbewertung beim Test kaum zutreffend ist. Auffällig ist hier jedoch, daß sich die Teilnehmer nicht generell als relativ erfolgreich einstufen.

Einfluß der Anzahl der Testfälle

Ein wichtiges Element beim Testen sind die Testfälle. Jedoch ist es äußerst schwierig, vorab zu entscheiden, ob ein Satz von Testfällen gut oder weniger gut ist. Ein relativ einfaches Maß in diesem Zusammenhang ist die Anzahl der Testfälle. An dieser Stelle wollen wir daher unsere Beobachtungen zu Einfluß der Testfallanzahl und Aufwand zur Testfallerstellung diskutieren.

In Abbildung 24(a) haben wir die Anzahl der gefundenen KS–Fehler und die Anzahl der dabei eingesetzten Testfälle einander gegenübergestellt. Es scheint



(a) (b)

Abb. 24: Beziehung Anzahl der Testfälle $\leftrightarrow$ gefundener Fehler (a) und Aufwand zur Erstellung der Testfälle.

zwar einen gewissen (linearen) Zusammenhang zu geben, allerdings ist dieser nicht besonders ausgeprägt ($r = 0.35$). Ähnliche Ergebnisse erhält man für die verwandten Zusammenhänge (siehe Abbildung 25).

Zusammenhang	r
Testfälle für den funktionalen Test und Anteil der im funktionalen Test gefundenen Fehler	0.31
Testfälle für den strukturellen Test und Anteil der im strukturellen Test gefundenen Fehler	0.33
Größe des Klassifikationsbaums und Anteil der im Test gefundenen Fehler	0.16

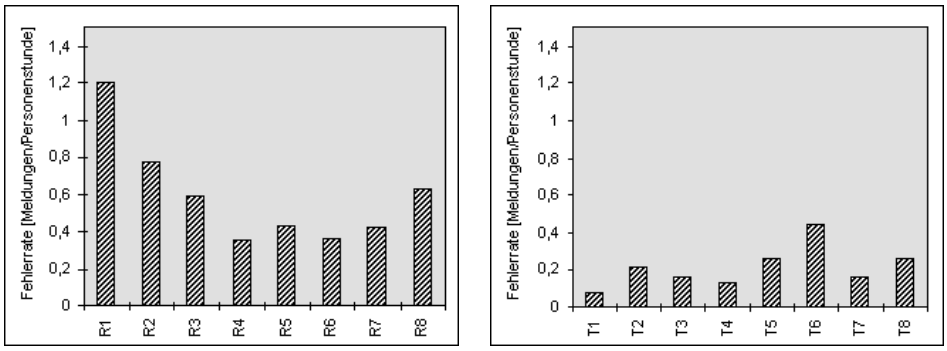
Abb. 25: Untersuchung Testfallanzahl $\leftrightarrow$ Anteil der gefundenen Fehler.

Neben dem Einfluß der Anzahl der Testfälle auf das Ergebnis, wollen wir auch die Aufwände zur Testfallerstellung betrachten. In Abbildung 24(b) haben wir die Aufwände zur Erstellung der Testfälle für den funktionalen und strukturellen Test aufgetragen.

Gegenüberstellung der Ergebnisse

Das wohl wichtigste Ergebnis, die Effizienz der Techniken, haben wir bereits vorweggenommen: Beim Review wurde eine Trefferrate von 80 % bis 100 % erzielt, während beim Test diese nur zwischen 25 % und 75 % lag. Unsere Vermutung zu (F1), nämlich eine annähernd gleiche Effektivität von statischer und dynamische Prüfung, müssen wir also revidieren.

Auch bei Berücksichtigung der dazu nötigen Aufwände schneidet die statische Prüftechnik deutlich besser als die dynamische ab (siehe Abbildung 26).



(a) (b)

Abb. 26: Effizienz, d. h. Anzahl der gefundenen KS–Fehler pro Zeit für Review (a) und Test (b).

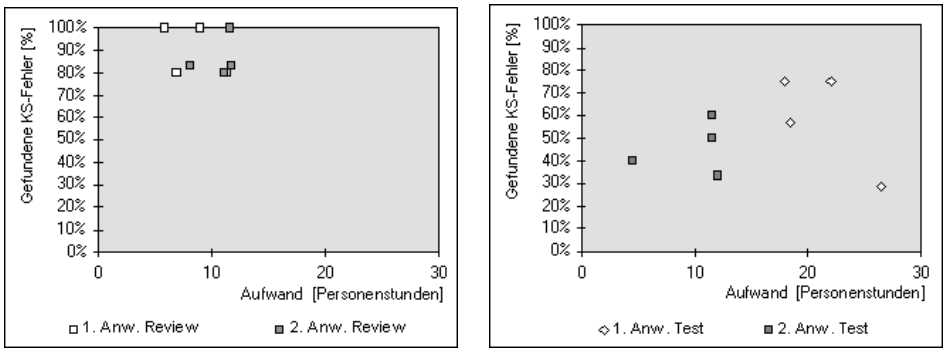
Der Zusammenhang zwischen den eingesetzten Aufwänden und den erzielten Resultaten, d. h. die Effizienz, ist in Abbildung 27 aufgetragen. Im Mittel werden beim Review 0.59 Meldungen zu KS–Fehler pro Personenstunde abgegeben im Vergleich zu nur 0.22 Meldungen beim Test. Unsere Vermutungen zu (F2) und (F3) haben sich damit ebenfalls als nicht zutreffend erwiesen.

Bezüglich eines Zusammenhangs zwischen Größe des betrachteten Prüfobjekts und den Prüfaufwänden (F4) ergab sich ein differenziertes Bild: Bei statischen Methoden ist ein quasi–linearer Zusammenhang erkennbar; bei dynamischen Methoden jedoch nicht.

Bezüglich der Reifeeffekte (F5) ergab sich ebenfalls ein eher differenziertes Bild: Bei den Reviews schien es wenig Unterschied zu machen, ob der Inspektor bereits Erfahrung mit der Technik gesammelt hat oder nicht. Beim Testen führt diese Erfahrung hingegen zu besseren Ergebnissen bzw. geringeren Aufwänden.

In Hinblick auf Fehlerarten, die bei der einen oder anderen Prüftechnik besser oder schlechter gefunden werden, konnten wir keine signifikanten Beobachtungen machen. Insbesondere wurden auch Fehler, die sich mit Zeitbedingungen oder Nebenläufigkeit beschäftigen, beim Review ebenso entdeckt wie auch beim Test. Unsere Annahme, daß sich diese Fehler beim Review schlechter finden lassen (F6), findet damit keinen Halt in den Beobachtungen.

Einflüsse von Motivation und Ausbildung auf die Ergebnisse waren bei beiden



(a) (b)

Abb. 27: Zusammenhang zwischen Aufwand und Anzahl der gefundenen KS–Fehler für Review (a) und Test (b).

Prüfverfahren nicht erkennbar. Unsere Hypothese zu (F8) hat sich als zutreffend erwiesen.

Freilich sind diese Ergebnisse nicht absolut zu sehen. Vielmehr sollen sie Trends aufzeigen und Hinweise auf mögliche Resultate beim Einsatz dieser Prüfaktivitäten.

4.3 Entwickeln und Prüfen von eingebetteten Systemen (AbSteam–Phase)

Ziel der AbSteamPhase war es, den Studenten die Möglichkeit zu geben, selbständig ein eingebettetes System zu entwerfen, zu implementieren und zu prüfen. Dabei wurde den Studenten möglichst freie Hand gelassen, denn unser Hauptinteresse war festzustellen, welche Prüfmethoden sie nach den gemachten Erfahrungen präferieren.

Die Aufgabenstellung basiert auf dem Steamboiler–Problem nach Abrial (vgl. [Abr96]) und wurde von uns für das Softwarelabor angepaßt. Die Aufgabe und der Ablauf der AbSteam–Phase wird im folgenden beschrieben (siehe Abschnitt 4.3.1) und abschließend bewertet (siehe Abschnitt 4.3.4).

4.3.1 Steamboiler–Problem

Die Aufgabe besteht darin, eine fehlertolerante Steuerung für eine Dampfmaschine zu erstellen. Die Steuerung soll den Wasserstand im Dampfkessel regulieren. Das korrekte Verhalten der Steuerung ist für die Lebensdauer des Dampfkessel sehr wichtig, denn falls der Wasserstand zu hoch oder zu niedrig ist, kann der Kessel beschädigt werden (z. B. wenn zu hoher Druck entsteht). Die Aufgabe der Steuerung ist, den Wasserstand im Kessel so zu regulieren, daß er stets innerhalb eines zuvor definierten Bereichs liegt. Neben dieser Hauptaufgabe muß die Steuerung bei Gefahr den Dampfkessel abschalten, um zu jeder

Zeit Sicherheit für das Bedienpersonal zu gewähren. Außerdem soll das System fehlertolerant sein, d. h. bei Ausfall einzelner Apparaturen, soll die Steuerung den Weiterbetrieb sicherstellen, bis die erforderliche Reparatur erfolgt.

Zum System gehören ein Reservoir und ein Kessel. Aus dem Reservoir wird Wasser in den Kessel gepumpt, das dort verdampft wird, um eine Maschine anzutreiben. Der Zustand des Systems wird durch einige Apparaturen erfaßt, die den Wasserstand im Kessel, den austretenden Dampf und das hineingepumpte Wasser messen. Für die Steuerung sind folgende Teile von Bedeutung:

- der Kessel
- der Wasserstandsmesser
- der Wasserdampfmesser an der Kesseldüse
- die Pumpe mit Durchlaufmesser
- das Kommunikationssystem
- das Steuerpult

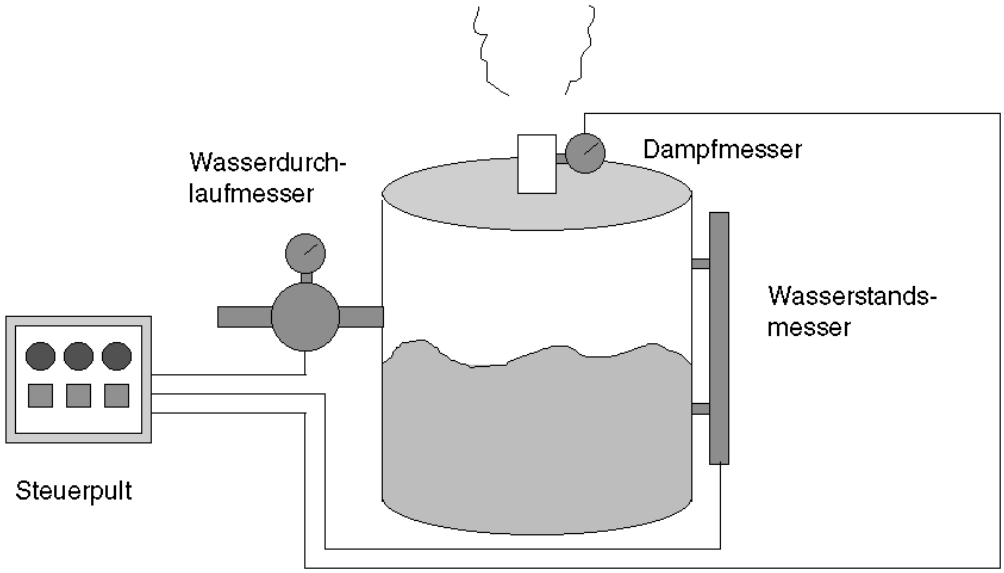


Abb. 28: Dampfkessel.

Das Programm zur Steuerung des Systems arbeitet zyklusorientiert. Jeder Zyklus besteht aus drei Phasen:

In der ersten Phase werden Botschaften der verschiedenen Bestandteile empfangen. Diese beinhalten unter anderem Meßdaten über den Wasserstand, den austretenden Wasserdampf und das hineingepumpte Wasser.

In der zweiten Phase werden die Meßdaten analysiert und entsprechende Entscheidungen getroffen: Wenn der Wasserstand zu niedrig ist, wird die Pumpe geöffnet. Analog dazu wird bei zu hohem Wasserstand die Entscheidung „Pumpe schließen“ getroffen. In allen anderen Fällen (wenn der Wasserstand innerhalb des vorgesehenen Bereiches liegt) macht das Programm nichts (d. h. der Zustand der Pumpe wird beibehalten).

In der dritten Phase werden Botschaften zurück an die Bestandteile gesendet. Dazu gehören unter anderem Anweisungen zum Öffnen oder Schließen der Pumpe.

Vorab sei bemerkt, daß die Wasserstandsmessung ausreicht, um das System zu steuern. Weitere Messungen wie der Wasserdampf und die Menge des gepumpten Wassers werden nicht benötigt, um zu entscheiden, ob die Pumpe geöffnet oder geschlossen werden muß. Dies hängt nur vom Vergleich des Wasserstandes mit den vordefinierten Grenzwerten ab (zu hoch, zu niedrig). Dennoch werden sowohl der Wasserdampf als auch die Menge des gepumpten Wassers gemessen und zwar zu folgenden Zwecken: Am Ende jedes Zyklus kann mit Hilfe dieser Messungen und der Anweisung an die Pumpe der als nächstes erwartete Wasserstand berechnet werden. Zu Beginn jedes Zyklus wird der vom Wasserstandsmesser gesendete Meßwert mit dem erwarteten Wasserstand verglichen. Bei gravierenden Unterschied der beiden Größen, darf man annehmen, daß der Wasserstandsmesser defekt ist. Der Meßwert wird daher korrigiert (bzw. durch den berechneten Wasserstand ersetzt), um den Weiterbetrieb zu ermöglichen.

Das Messen weiterer Größen bringt allerdings auch weitere Fehlerquellen mit sich. Insbesondere können die betreffende Meßgeräte ausfallen. Die folgenden möglichen Defekte werden in Betracht gezogen:

- Defekt des Wasserstandsmessers
- Defekt des Wasserdampfmessers
- Defekt des Durchlaufmessers in der Pumpe

Die Entscheidung, ob ein Meßgerät defekt ist, trifft die Steuerung durch Vergleichen des eingehenden Meßwert mit einem erwarteten Wert, der im Zyklus davor berechnet wird. Sobald eine Meßapparatur für defekt befunden ist, sendet die Steuerung eine entsprechende Fehler–Botschaft an das Steuerpult, auf dem dann eine Warnlampe aufleuchtet. Daraufhin repariert der Maschinist die betreffende Meßapparatur. Sobald die Reparatur abgeschlossen ist, signalisiert der Maschinist dies durch Drücken eines Knopfes am Steuerpult und eine Reparatur–Botschaft wird an die Steuerung übertragen.

Zwischen Fehler–Botschaft und Reparatur–Botschaft betrachtet die Steuerung die betroffene Apparatur als defekt und ignoriert ihre Messungen. Der fehlende Meßwert wird durch eine Schätzung ersetzt, die auf dem physikalischen Verhalten des Systems basiert.

In der Aufgabe von Abrial war zusätzlich ein Defekt des Kommunikationssystems vorgesehen. Dies wurde im Rahmen des Praktikums ignoriert, um die

Aufgabe in einem bestimmten Rahmen zu halten. Ohnehin ist das Problem durch die anderen möglichen Ausfälle aufwendig genug. So muß beispielsweise für jede Apparatur das Intervall der möglichen Meßwerte für den nächsten Zyklus vorausberechnet werden, um die Entscheidung über einen möglichen Defekt zuverlässig zu treffen. Darüber hinaus kann ein Defekt mehrere Phasen anhalten und es wird eine Fortschreibung der Werte notwendig.

4.3.2 Ablauf der AbSteam–Phase

Für die AbSteam–Phase wurden die Studenten erneut in Gruppen eingeteilt. Jeder der 4 Gruppen wurden die gleichen Aufgaben gestellt; diese sind in Abbildung 29 beschrieben. Die gestrichelten, horizontalen Linien stellen dabei den Zeitraum von einer Woche dar.

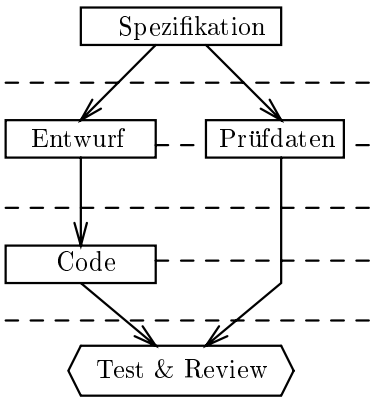


Abb. 29: Die Teilphasen der AbSteam–Phase.

Zunächst erstellte jede Gruppe an Hand der vorgegebenen Spezifikation einen Entwurf für das Steam–Boiler–Problem und Prüfdaten, um das zu erstellende System zu prüfen. Nach Abgabe dieser Teile erhielten die Gruppen die Entwicklungsumgebung und konnten mit der Implementierung beginnen. Als nächstes stellten sie ihre selbst entwickelten Prüfstrategien vor. Die gegenseitige Prüfung erfolgte anschließend.

Entwurf

Für den Entwurf von Echtzeitsystemen gibt es eine Reihe von gängigen Methoden und Notationen (z. B. SA/RT). Der Einsatz einer solchen Methode wäre an dieser Stelle wünschenswert, war aber im Softwarelabor aufgrund der zeitlichen Rahmenbedingungen nicht möglich, da die Vermittlung der nötigen Grundkenntnisse wenigstens eine weitere Vorlesung erfordert hätte. Daher wurden für den Entwurf keine besonderen Methoden oder Techniken vorgegeben. Jede Gruppe konnte selbst eine Methode wählen oder intuitiv vorgehen.

Es wurden lediglich Hinweise für die Problemlösung gegeben. Kernstück der Problemlösung ist die Vorausberechnung der möglichen Werte, die eine Apparatur im nächsten Zyklus liefert. Sie liegen in einem Intervall, das sich aus den aktuellen Werten und physikalischen Gesetzmäßigkeiten ergibt. Liefert ein Meßgerät einen Wert außerhalb des Intervalls, so ist es defekt. Für alle Meßgeräte muß eine entsprechende Berechnung erfolgen. Für defekte Geräte wird das Intervall fortgeschrieben.

Prüfdaten

Ein sinnvolles Vorgehen bei der Software–Entwicklung ist es, bereits zu einem frühen Zeitpunkt Testfälle abzuleiten, mit denen später die Softwaremodule getestet werden. Dadurch wird insbesondere sichergestellt, daß sich spätere Prüfaktivitäten nicht an der entwickelten Software orientieren, sondern an der Spezifikation.

Deshalb waren unmittelbar aus der Spezifikation Testfälle für den funktionalen Test abzuleiten und mit Hilfe des Klassifikationsbaum–Editors CTE zu dokumentieren.

Diese Überlegung der frühzeitigen Festlegung läßt sich mit großer Wahrscheinlichkeit auch auf Reviews übertragen. Deshalb waren neben Testfällen auch Fragen abzuleiten, die sich bei der späteren Review–Vorbereitung als nützliche Hilfestellungen erweisen können.

Implementierung

In dieser Phase war ein C–Modul für die Steuerfunktionen zu erstellen. Als Grundlage dient der zuvor erstellte Entwurf. Für den Ablauf steht eine Entwicklungsumgebung bereit, die den Dampfkessel simuliert. Die Entwicklungsumgebung bindet das Modul wie folgt ein: System und Steuerung bilden jeweils eine eigene Einheit und kommunizieren über einen Kommunikationssystem, das durch zwei Puffer realisiert ist, aus denen die Steuerung Botschaften abholt und in die sie Botschaften schreibt. Dadurch kann die Entwicklungsumgebung als Testumgebung verwendet werden, indem über eine Remote–Datei Botschaften in die Puffer eingespeist werden. Abbildung 30 verdeutlicht diese Vorgehensweise.

Prüfstrategien

Bevor man beginnt, eine Prüfung vorzunehmen, überlegt man zunächst, welche Kriterien anzulegen sind und mit welchem Verfahren die betreffenden Fehler am besten gefunden werden können. Eine solche Prüfstrategie zu überlegen und den zugehörigen Prüfprozeß niederzuschreiben, war nun die Aufgabe jeder Gruppe. Dabei wurden keine Vorgaben gemacht, denn uns interessierte, ob die einzelnen Gruppen nach den bereits gemachten Erfahrungen statische oder dynamische Prüfmethoden oder Mischverfahren präferieren.

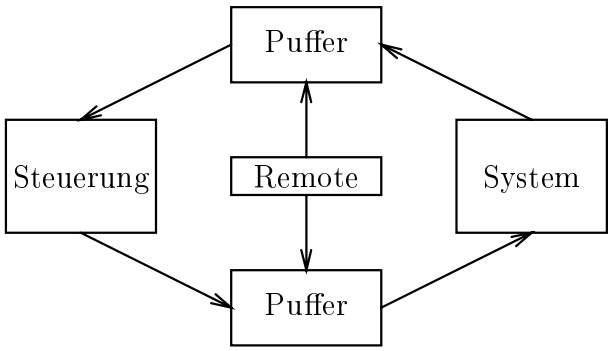


Abb. 30: Das Kommunikationssystem.

Prüfung

Nach Abgabe aller Implementierungen wurde jede Implementierung jeweils an zwei andere Gruppe zur Überprüfung übergeben. Die Verteilung erfolgte so, daß jedes Programm zweimal geprüft wurde und daß jede Gruppe zwei Programme prüfen mußte. Jeder Gruppe verwendete ihre zuvor beschriebene Prüfstrategie.

4.3.3 Die Testumgebung

Die Entwicklungsumgebung ist gleichzeitig als Testumgebung konzipiert. In diese Umgebung wird der Programmteil für die Steuerung eingebunden und das Verhalten des Dampfkessels wird auf dem Bildschirm angezeigt. Zusätzlich wird ein Protokoll mit den Kesseldaten in die Datei `steam.log` geschrieben. Zum Erstellen der Testfälle und zum Beeinflussen des Systems können in einer Remote–Datei externe Ereignisse für das System angegeben werden, die in die beiden Kommunikationskanäle eingeschleust werden.

In der Logdatei `steam.log` (siehe Abbildung 32) werden für jeden Zyklus die tatsächlichen Werte für Wasserstand, Wasserzulauf und Wasserverdampfung sowie die versendeten Botschaften protokolliert.

Mit der Remote–Datei werden die externen Einflüsse auf das System simuliert. Dazu konnten zu bestimmten Zeitpunkten Defekte hervorgerufen und Reparaturen signalisiert werden. Außerdem konnten die Anfangsbedingungen gesetzt werden.

Dies wird durch das Beispiel in Abbildung 33 demonstriert: Zu Beginn der Simulation, d. h. zum Zeitpunkt 0 wird der Wasserstand auf 50 und die Wasserrate auf 0 gesetzt. Nach 200 Zyklen gehen der Wasserstandsmesser und der Wasserdurchlaufmesser kaputt. Weitere 100 Zyklen später ist der Wasserstandsmesser wieder funktionsfähig.

Die Simulation zeigt den Kessel mit aktuellem Füllstand sowie die Meßwerte an den Apparaturen an (siehe Abbildung 31). Die Bewegung der Pumpe wird animiert. Im Fall, daß der Kessel überbeansprucht wird, wird eine entsprechende Meldung ausgegeben, z. B. „Der Kessel ist explodiert“.

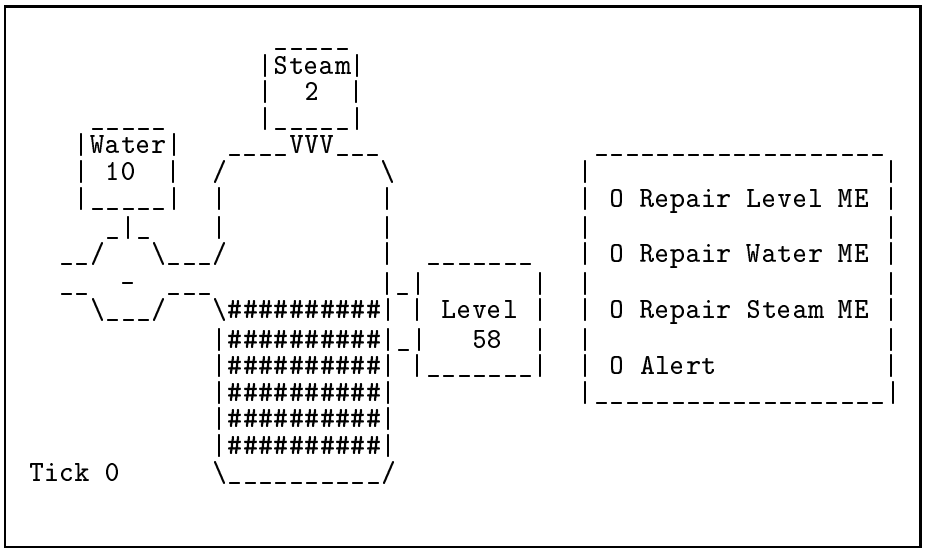


Abb. 31: Screenshot der Simulationsumgebung des Dampfkessels.

```
================================================
Phase: 0

A) Extern an Steuerung und System:
B) System an Steuerung:
   M_Wasserrate  10
   M_Wasserstand  58
   M_Dampfrate  2
C) System: Wasserstand 58, Wasserrate 10, Dampfrate 2
D) User Data:
Zustand: 0
Pumpe  : 0
Ews    : 0 0
Edr    : 0 0
E) Steuerung an System:
   M_Nothalt
```

Abb. 32: Beispiel einer Log–Datei.

```
TIME IS 0
SET Wasserstand 50
SET Wasserrate 0
TIME IS 200
CRASH Wasserstandsmesser
CRASH Wasserdurchlaufmesser
TIME IS 300
REPAIR Wasserstandsmesser
```

Abb. 33: Beispiel für eine Remote–Datei.

4.3.4 Ergebnisse

Dieser Abschnitt stellt die Ergebnisse der verschiedenen Gruppen dar.

Entwurf

Für den Entwurf waren keine besonderen Methoden oder Notationen vorgeschrieben. Dennoch haben die Mehrzahl der Gruppen allgemein anerkannte Notationen wie z. B. Zustandsdiagramme als Beschreibungsformen gewählt. Tabelle 6 zeigt, welche Teile die Entwürfe der einzelnen Gruppen enthielten. Ein X bedeutet, daß eine Gruppe das betreffende Dokument erstellt hat. Alle Gruppen haben den Ablauf der Steuerung genau beschrieben. Die Vorausberechnung für die Meßergebnisse war nur bei zwei Gruppen enthalten. Die meisten Gruppen haben zusätzlich die Zustände, in die das System kommen kann, durch Zustandsdiagramme angegeben. Eine Gruppe hat darüber hinaus die Zusammenhänge zwischen Ereignissen und Aktionen in einer Ereignistabelle beschrieben.

Gruppe	1	2	3	4
Ablauf der Steuerung	X	X	X	
Vorausberechnung der Meßergebnisse	X			X
Zustandsdiagramm	X		X	X
Ereignistabelle	X			
Annahmen zur Spezifikation	X		X	X

Tab. 6: Komponenten des Entwurfs.

Testfälle und Review–Fragen

Zusammen mit den Entwürfen wurden auch die Testfälle und die Fragen für das Review abgegeben. Dabei wurden von drei Gruppen Klassifikationsbäume mit dem Tool CTE (siehe auch 2.3) erstellt. Die Tabelle 7 zeigt die Anzahl der

erstellten Testfälle und abgegebenen Reviewfragen. Wie zu sehen ist, hat keine Gruppe Ergänzungen zur Checkliste gemacht.

Gruppe	Testfälle	Fragen für Checklisten
1	30	0
2	0	0
3	42	0
4	64	0

Tab. 7: Anzahl der Testfälle und Ergänzungen zur Checkliste.

Implementierung

Die Implementierung der Steamboiler–Steuerung erfolgte in ANSI C. Alle Programme hatten ungefähr vergleichbare Größe (ca. 250 Zeilen). Die genauen Programmgrößen sind in Tabelle 8 dargestellt. Alle Programme waren sehr gut lesbar und der Programmierstil entsprach den Standards, wie sie durch die Checkliste vorgegeben waren. Insgesamt haben die abgegebenen Programme einen ähnlichen Aufbau wie die in der AbEmbSys–Phase inspizierten Programme und sind wohldokumentiert.

Gruppe	LOC	Module
1	325	4
2	217	1
3	251	1
4	181	1

Tab. 8: Umfang des Quelltextes.

Prüfstrategien

Jede Gruppe präsentierte ihre eigene Prüfstrategie. Dabei wurden teilweise sehr interessante Strategien entwickelt. Diese sind in Abbildung 34 graphisch dargestellt und im folgenden beschrieben.

Gruppe 1 beginnt mit einem funktionalen Test. Danach ist eine Korrektur des Programms durch die Autoren vorgesehen, falls sich bei der Ausführung gravierende Probleme ergeben haben, die auch den später geplanten strukturellen Test zum Scheitern verurteilen würden. Es folgt eine Überprüfung des inzwischen nachbearbeiteten Programms durch Wiederholung der Tests und durch informelles Lesen. Abschließend wird ein struktureller Test durchgeführt und alles in einem Protokoll zusammengefaßt.

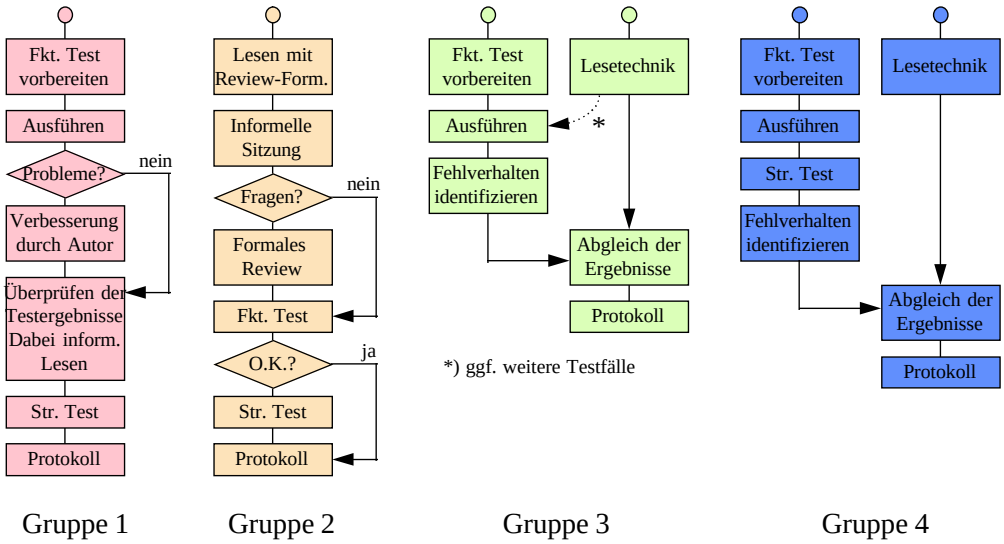


Abb. 34: Prüfstrategien der Gruppen.

Die Prüfstrategie von Gruppe 2 beginnt mit einem Leseschritt. Dieser entspricht der Vorbereitungsphase eines Reviews. Dabei werden Fehler in einer informellen Review–Sitzung zusammengetragen. Falls dabei Fragen oder Unstimmigkeiten aufkommen, wird eine formale Review–Sitzung durchgeführt. Im Anschluß wird das Programm einem funktionalen Test unterzogen. Sollte der Grad der Überdeckung nicht hoch genug sein, so werden weitere Testfälle für einen anschließenden strukturellen Test erzeugt. Zum Abschluß wird ein zusammenfassendes Protokoll erstellt.

Die Mitglieder der Gruppe 3 teilen sich in zwei Untergruppen auf. Eine Teilgruppe führt Testaktivitäten (funktionalen und strukturellen Test) durch, während die andere das Programm liest. Die Ergebnisse beider Teilgruppen werden in einem gemeinsamen Abschlußprotokoll zusammengefaßt.

Auch die Mitglieder der Gruppe 4 teilen sich in zwei Untergruppen auf. Eine Teilgruppe bereitet den funktionalen Test vor, während die andere Teilgruppe das Programm liest. Allerdings werden hier Erkenntnisse aus dem "Lesen" beim Durchführen des funktionalen Tests berücksichtigt. Dagegen entfällt der strukturelle Test. Die Ergebnisse beider Teilgruppen werden wie bei Gruppe 3 in einem gemeinsamen Abschlußprotokoll zusammengefaßt.

Prüfung

Alle Prüfverfahren waren effektiv, d. h. es wurden wichtige Fehler aufgedeckt.

Die Beurteilung der Programme fiel teilweise unterschiedlich aus. Tabelle 9 zeigt die Anzahl der Fehler, die in einem Prüfdurchgang von einer bestimmten Gruppe in einem bestimmten Programm aufgedeckt wurden: Pro Zeile kann

	Programm von			
	Gruppe 1	Gruppe 2	Gruppe 3	Gruppe 4
Gruppe 1		5	5	
Gruppe 2			12	7
Gruppe 3	4			0
Gruppe 4	8	4		

Tab. 9: Aufgedeckte Fehler.

abgelesen werden, welche zwei Programme eine bestimmte Gruppe geprüft hat und wieviele Fehler sie dabei aufgedeckt hat.

4.3.5 Bewertung

Eine statistische Bewertung, wie sie bei den anderen Phasen teilweise durchgeführt wurde, kann für die AbSteamPhase nicht gemacht werden, da die Anzahl der Freiheitsgrade und Variationen zu groß ist. So erfolgte beispielsweise keine Messung der Aufwände, weil die Verfahren beliebig durch die Gruppen bestimmt werden konnten und die Meßwerte nicht vergleichbar sind. Insbesondere machen vor diesem Hintergrund auch keine Untersuchungen bezüglich Korrelation der verschiedenen Größen (Programmgröße, Anzahl der Testfälle, Fehleranzahl, etc.) Sinn, und es ist nicht möglich, signifikante Aussagen zu machen.

Deshalb kann in dieser Phase auch keine quantitative Aussage über den Vergleich zwischen statischen und dynamischen Prüfverfahren gemacht werden. Stattdessen folgen einige subjektive Beobachtungen, die sich auf qualitative Aspekte beziehen.

Die Implementierungen wiesen ein ähnlichen Aufbau wie die in der Phase zuvor inspizierten Programme auf. Dies impliziert eine Art Lern- bzw. Schulungseffekt, der durch das Betrachten und Diskutieren anderer Sourcen entsteht. Dadurch ist ansatzweise ein einheitlicher und akzeptierter Stil entstanden.

Die Prüfstrategien geben keinen eindeutigen Aufschluß über die Präferenz von statischen oder dynamischen Püfmethoden, auch wenn hier ein leichter Ausschlag des Pendels in Richtung Test zu sehen ist. Bei allen Gruppen sind beide Elemente, jedoch mit unterschiedlichem Gewicht vorgesehen. Interessant ist, wie die Gruppen das Zusammenspiel der beiden Verfahren arrangiert haben. Trotz positiver Erfahrungen mit Reviews wollte aber keine Gruppe auf dynamische Prüfverfahren vollständig verzichten.

Gruppe 1 benutzt die Tatsache, daß man, nachdem man beim funktionalen Test ein Fehlerverhalten ausgemacht hat, sich auf die Fehlersuche begeben muß. Dabei wird auf jeden Fall der Code betrachtet, deshalb führt Gruppe 1 an dieser Stelle gleich einen informellen Leseschritt ein. Gruppe 2 macht das Review vor dem Test und kann dadurch mit großer Wahrscheinlichkeit einige Test-

durchläufe sparen, leider liegen uns dazu keine Daten vor. Gruppe 3 prüft Dinge, die sich im Review nicht entgültig klären lassen mit zusätzlichen Tests. Für Gruppe 4 stehen beide Methoden unabhängig nebeneinander und dienen beide dem Aufdecken von Fehlern. Der strukturelle Test wird von allen Gruppe als nachgelagertes Prüfverfahren oder gar als überflüssig angesehen.

Zusammenfassend läßt sich sagen, daß auf jeden Fall ein Bewußtsein für statische und dynamische Prüfverfahren und ein Verständnis für mögliche Anwendungen geschaffen wurde.

5 Bewertung des Experiments

5.1 Kritische Betrachtung der Aussagen

Neben theoretischem Wissen sind insbesondere experimentelle Aussagen wichtig, um Entscheidungen auf Basis einer vernünftigen Grundlage treffen zu können. In Hinblick auf diese weitreichende Bedeutung ist mit den experimentellen Ergebnissen auch eine gewisse Verantwortung verbunden. Eine genaue Betrachtung der Umstände, unter denen die experimentellen Aussagen gültig sind, ist also unerläßlich.

5.1.1 Betrachtung der internen Gültigkeit

Unter der internen Gültigkeit (*internal valitidy*) wird die Gültigkeit der experimentellen Aussagen in Hinblick auf ihren Entstehungsprozeß betrachtet.

Da dieses Experiment mit nur 13 Studenten durchgeführt wurde, wohnt den Ergebnisse wenig Verläßlichkeit im statistischen Sinne innen. Dazu sind unbedingt weitere Replikationen des Experiments notwendig. Die Annahme, daß die Ergebnisse der einzelnen Teilnehmer bzw. Teams unabhängig voneinander sind, spielt hier ebenfalls eine große Rolle. Wir glauben aber — und auch die Diskussionen mit unseren Studenten haben das immer wieder gezeigt — daß es uns gelungen ist, den experimentellen Gedanken zu vermitteln. Interaktion zwischen den Individuen bzw. Teams, soweit sie die Bearbeitung der Aufgaben betraf, war fast nicht gegeben.

Die Genauigkeit der Datenerfassung, insbesondere soweit sie die Erfassung von Zeiten betrifft, ist in jedem Fall mit einer gewissen Unschärfe behaftet. Obwohl dadurch auch die einzelnen Ergebnisse mit einem gewissen Abstand betrachtet werden müssen, glauben wir jedoch, daß diese Ungenauigkeit relativ wenig Einfluß auf die eigentlichen Aussagen dieses Experiments hat.

5.1.2 Betrachtung der externen Gültigkeit

Die externe Gültigkeitsuntersuchung (*external validity*) beschäftigen sich mit der Frage, inwieweit die Ergebnisse des Experiments auf andere Bereiche übertragbar sind. Insbesondere diese Frage ist aber im Sinne der oben beschriebenen Entscheidungsgrundlage relevant.

Die wesentlichen Punkte, die Zweifel die Übertragbarkeit der Ergebnisse wecken können, haben wir im folgenden zusammengefaßt und diskutiert.

- Die Softwareprüfungen wurde von Studenten und nicht von Entwicklern aus der industriellen Praxis durchgeführt.

  In jedem Fall besteht ein Unterschied darin, ob jemand eine Technik erst seit einiger Zeit oder aber schon viele Jahre lang anwendet. Dieser Effekt wird auch von unseren Untersuchungen zu Reifeeffekten im Bereich des Testens bestätigt. Allerdings berichten DeMarco und Lister in [DL87]

davon, daß diese Unterschiede oft überbewertet werden. Die individuelle Leistung variiert deutlich stärker, als dies aufgrund der Erfahrung gegeben wäre.

Alle Teilnehmer waren Studenten im Hauptstudium, die teilweise kurz vor Abschluß ihres Studiums standen, also Personen, die in Kürze ihr Berufsleben beginnen. Insbesondere auf Bereiche, die mit jungem oder unerfahrenem Personal arbeiten, sind die Ergebnisse sicher in gewisser Weise übertragbar.

- Die Beispielprogramme stammen nicht aus einem realen Entwicklungsprojekt.

  An dieser Stelle sehen wir die größten Probleme bei der Übertragbarkeit unserer Ergebnisse. Die in unserem Experiment eingesetzten Programme waren durchgängig gut kommentiert, die Problemstellung war einfach zu verstehen, und der gewählte Lösungsweg war leicht nachvollziehbar. Genau hierin sehen wir durchaus Abweichungen zu vielen Programmen aus der industriellen Praxis: Die Probleme sind wesentlich komplexer und die Sachverhalte oft wenig intuitiv. Dies sind Eigenschaften, die insbesondere das Lesen von Programmcode deutlich erschweren.

- Ideale Randbedingungen für die Softwareprüfung.

  Im Rahmen der Untersuchung gab es für die Teilnehmer wenig äußere Zwänge wie beispielsweise Termindruck. Sie konnten soviel Zeit in Anspruch nehmen, wie sie für nötig erachtet haben. In vielen realen Projekten spielt aber der Zeitdruck eine wesentliche Rolle, die insbesondere oft Auswirkungen auf die Vorbereitung eines Reviews hat. Es ist völlig unklar, wie die Ergebnisse unter restriktiveren Umgebungsbedingungen im Sinne von Zeitdruck ausgesehen hätten.

Trotz berechtigter Bedenken gegen die Übertragbarkeit der Ergebnisse im einzelnen glauben wir aber, daß die hier aufgedeckten Grundtendenzen auf andere Bereiche im Umfeld der Entwicklung eingebetteter Systeme übertragbar sind.

5.2 Vergleich mit anderen Experimentbeschreibungen

Das Themenfeld Softwareprüfung war bereits Bestandteil einer Reihe von experimentellen Untersuchungen [SBPM84, Cha95, PV94, PVB95]. Der Fokus dieser Untersuchungen liegt hier meist im Vergleich eng benachbarter Techniken (z. B. Vergleich verschiedener Lesetechniken [BGL+96]).

Die vergleichende Betrachtung statischer und dynamischer Softwareprüfung ist Gegenstand einer Reihe von Experimenten, die an der Universität von Maryland entwickelt wurden [BS87] und mittlerweile an mehreren Standorten repliziert worden sind [KL95b]. Wesentliche Beobachtung in dieser Experimentreihe war, daß sich die betrachteten Softwareprüfverfahren, nämlich Lesen des Programmcodes, funktionaler Test sowie struktureller Test relativ wenig in ihrer Effektivität und Effizienz unterschieden haben.

Myers stellt in [Mye78] ein Experiment vor, in dem er ebenfalls statische und dynamische Softwareprüfmethoden miteinander verglichen hat. Dort wurde beispielsweise auch die Verteilung der gefundenen Fehler untersucht mit dem Ergebnis, daß es relativ wenige Überdeckungen zwischen den unterschiedlichen Methoden gibt. Da deshalb in der Kombination von zwei unterschiedlichen Methoden fast doppelt so viele Fehler gefunden werden und dafür auch die doppelte Zeit benötigt wird, bleibt die Effizienz dieser kombinierten Methode ungefähr gleich. Dieser interessante Aspekt wurde in unserem Experiment bisher nicht untersucht. Die übrigen Ergebnisse aus [Mye78] stimmen weitgehend mit den unsrigen überein.

5.3 Erfahrungen

5.3.1 Aufwand

Generell ist zu sagen, daß die Durchführung des Experiments sehr aufwendig war. Vor allem gilt dies für die AbEmbSys- und die AbUtil–Phase, da hier keinerlei fertige Materialien zur Verfügung standen. Aber auch der Aufwand für die AbUtil–Phase ist nicht zu unterschätzen, obwohl das Material vom K&L–Experiment in Kaiserslautern übernommen werden konnte. Es mußte für unsere Bedürfnisse angepaßt und aufbereitet werden. Insbesondere mußten die Formulare und Programmspezifikationen ins Deutsche übersetzt werden.

In Tabelle 10 sind die ungefähren Aufwände in Personentagen (PT) aufgeführt, die wir in den einzelnen Phasen des Praktikums zur Vorbereitung und Durchführung aufgebracht haben. Dabei wollen wir einige Punkte etwas n"her erläutern: Die Vorbereitung der Infrastruktur in der AbUtil–Phase schließt die Beschaffung des in dieser Phase verwendeten Werkzeugs GCT mit ein. Die Installation des Werkzeugs und eine Einarbeitung sind ebenfalls mitgerechnet. Außerdem wurde eine lokale Testumgebung entworfen, die im wesentlichen aus dem UNIX–Shellskript `run-suite` (siehe Anhang I) besteht.

Zur Vorbereitung der Infrastruktur der AbEmbSys–Phase gehörte im wesentlichen die Beschaffung und Installation des CTE. Die Testumgebung konnte zu großen Teilen von der AbUtil–Phase übernommen werden. Allerdings hat sich herausgestellt, daß die Standardeinstellungen für den GCT nicht für Mehrprozeßsysteme funktioniert. Deshalb wurde für solche Systeme (wie beispielsweise die elektronische Mischpatrone, siehe Anhang C) eine Testumgebung entwickelt, die diese Probleme umgeht.

Ein wesentlicher Zeitfaktor bei der AbEmbSys–Phase war das Erstellen der Prüflinge, da die Systeme eigens für das Experiment implementiert wurden. Eine Ausnahme stellt der Roboter dar, der im Rahmen einer Diplomarbeit [Han96] entwickelt wurde. Allerdings mußte die Implementierung in vielen Details geändert werden, da die Originalimplementierung für einen Mikroprozessor und nicht für ein UNIX–System geschrieben wurde. Für jede der vier Implementierungen wurden ungefähr vier Personentage benötigt.

Die Vorstudie bei der AbSteam–Phase umfaßt zunächst die Auswahl des zu

Phase	Arbeitsschritt	Aufwand in PT	
AbUtil	Experimentplanung	3	
	Vorlesungsvorbereitung	3	
	Listings und Spezifikationen aufbereiten	2	
	Formblätter aufbereiten	2	
	Infrastruktur vorbereiten	5	
	Auswertung (ohne Dokumentation)	5	
	Experimentbetreuung	4	$\sum 24$
AbEmbSys	Planung	3	
	Infrastruktur vorbereiten	2	
	Vorlesungsvorbereitung	3	
	Listings erstellen	4×4	
	Formblätter erstellen	2	
	Experimentbetreuung	6	
	Auswertung (ohne Dokumentation)	8	$\sum 40$
AbSteam	Vorstudie	2	
	Umgebung vorbereiten	3	
	Spezifikation erstellen	1	
	Auswertung (ohne Dokumentation)	1	
	Experimentbetreuung	2	$\sum 9$

Tab. 10: Aufwände der einzelnen Arbeitsschritte.

verwendenden Systems. Außerdem wurde hier die Einarbeitung in die Funktionsweise des Steamboilers (siehe [Abr96]) mitgerechnet.

5.3.2 Erfahrungen zu den eingesetzten Werkzeugen

Im Rahmen des Experiments wurden unterschiedliche Werkzeuge eingesetzt. Im folgenden möchten wir kurz einige Erfahrungen, die wir mit ihrer Verwendung gemacht haben, beschreiben.

GCT Der strukturelle Test ist mit Werkzeugunterstützung deutlich einfacher durchführbar als ohne, da eine Protokollierung der Zweigüberdckung von Hand sehr viel länger dauert und auch fehleranfällig ist. Das Generic Coverage Tool (GCT) ist für diese Aufgabe sehr gut geeignet. Auch Mehrprozeßsysteme lassen sich auf leichte Art und Weise testen. Für unser Experiment war nur nachteilig, daß der GCT bisher nur für SunOS–Plattformen unterstützt wird und noch nicht für Solaris angeboten wird.

CTE Die Arbeit mit dem CTE wurde von den Studenten ziemlich kritisch beurteilt, da sie zumindest bei unseren relativ kleinen Systemen kaum Zeitersparnis gebracht hat. Erst durch einige Prüffunktionen, die das Werkzeug zur Verfügung stellt, wird der Einsatz des Werkzeugs sinnvoll.

Es ist jedoch denkbar, daß bei größeren Systemen der CTE hilfreich ist zur Verwaltung von Äquivalenzklassen und Testfällen.

Einige Funktionalitäten wurden vermißt. Beispielsweise ist nur über ein Kommentarfeld möglich, das erwartete Verhalten für einen bestimmten Testfall anzugeben. Hier wäre sicherlich mehr Unterstützung bis hin zu einer semiautomatischen Testfallgenerierung denkbar.

Generell hat die Verwendung der Werkzeuge gezeigt, daß diese einige organisatorische Arbeiten abnehmen. Sie verhindert in diesem Bereich auch eine Reihe von Fehler, die ein Prüfer ohne Werkzeugeinsatz machen würde. Andererseits wird der Prüfer durch die Verwendung eines Werkzeugs oft auch eingeengt, da er seine Arbeit dem Werkzeug anpassen muß. Somit erscheint ein Werkzeugeinsatz nur dann sinnvoll, wenn damit die Arbeitsweise des Prüfers nicht beschränkt, sondern vereinfacht und unterstützt wird.

6 Zusammenfassung

Softwareprüfung spielt innerhalb der Softwareentwicklung eine wichtige Rolle. Sie dient dazu, die Konformität der Software gegenüber mannigfaltigen Vorgaben zu überprüfen. Neben diesem zielgerichteten Anspruch zeichnet sich aber die Softwareprüfung auch durch hohe Aufwände aus, die mit den Prüfaktivitäten verbunden sind.

Um den Ansprüchen nach steigender Produktivität einerseits und gleichbleibender oder verbesserter Qualität andererseits nachzukommen, ist es nötig, die absoluten Anstrengungen in diesem Bereich zu reduzieren, ohne dabei deren Effektivität zu verlieren. Dies erfordert, daß die zur Verfügung stehenden Techniken zur Softwareprüfung in optimaler Weise kombiniert werden müssen. Dies wiederum setzt voraus, gesichertes Wissen über die Wirkungsweise der zur Verfügung stehenden Techniken in Hinblick auf z. B. Effektivität und Effizienz zu haben. Auf dem Weg zu diesen Informationen sind Experimente ein wichtiges Element. Sie ermöglichen es, gezielt Einflußfaktoren und Wirkungsweisen zu untersuchen.

In diesem Bericht beschreiben wir eine solche experimentelle Untersuchung von Techniken zu Softwareprüfung mit besonderer Beachtung von eingebetteten Systemen und ihren spezifischen Besonderheiten. Im Rahmen dieser Untersuchung wurden 4 kleinere Systeme von je vier Gruppen mit Verfahren der statischen Softwareprüfung (technisches Review) und dynamischen Softwareprüfung (Test) untersucht. Die wesentlichen Ergebnisse dieser Untersuchung lassen sich wie folgt zusammenfassen:

Effektivität: Bei dem technischen Review wurden jeweils zwischen 80 % und 100 % der enthaltenen kritischen bzw. schwerwiegenden Fehler entdeckt. Beim Testen wurde jedoch nur zwischen 25 % und 75 % der enthaltenen Fehler dieser Fehlerklasse gefunden.

Effizienz: Auch bei Betrachtung der zur Fehlerfindung nötigen Zeit schnitt das technische Review deutlich besser als der Test ab: 0.59 Fehler pro Personenstunde wurde durch statisches Analysieren des Programmcodes gefunden, währen die dynamische Überprüfung nur 0.22 Fehler pro Personenstunde ergab.

Fehlerklassen: Wir vermuteten, daß Fehler, die im Bereich von Parallelität oder Zeitbedingungen durch statische Analyse deutlich schlechter gefunden werden können als durch Ausführung des Prüflings. Unsere Beobachtungen konnten diese Hypothese jedoch nicht bestätigen.

Ausblick

Im Rahmen unserer Untersuchung haben wir eine Reihe von Fragestellungen identifiziert, die eine genauere Betrachtung wert sind:

- Welchen Einfluß haben die konkreten Prüfobjekte auf die Ergebnisse der Untersuchung?

  Innerhalb unserer Prüfobjekte gab es bereits gewissen Schwankungen hinsichtlich der Wirksamkeit der einzelnen Techniken. Trotz einer gewissen Varianz in der Architektur waren doch einige Aspekte ähnlich (z. B. Kommentierungsgrad, Art der verwendeten Kontrollstrukturen).

- Welche Auswirkung haben beschränkte Ressourcen auf die Wirkungsweise der Prüfverfahren?

  Für die Tests und die Vorbereitung zu den Reviews war es den Studenten freigestellt, wieviel Zeit sie für einzelne Schritte aufwenden. Denkbar wäre auch eine Strategie, die bestimmte Zeiten vorgibt. Zwar werden mit kürzeren Zeiten wohl weniger Fehler gefunden, aber es ist unklar, wie sich die Effizienz in diesem Fall verhält, d. h. ob im Mittel bei kürzeren Prüfzeiten mehr oder weniger Fehler pro Zeiteinheit gefunden werden.

- Welchen Einfluß haben die konkreten Ausprägungen der hier eingesetzten Prüfverfahren?

  Im Rahmen der Untersuchung wurden Verfahren eingesetzt, die in gewisser Weise den in vielen Unternehmen verbreiteten Stand der Technik widerspiegeln, der jedoch merklich vom aktuellen Stand der Kunst entfernt ist. Eine Möglichkeit der Variation wäre hier z. B. das Einbeziehen von anderen Lesetechniken als die in dieser Untersuchung praktizierte Vorbereitung mit Hilfe von Checklisten. Eine denkbare Variante wäre der Einsatz von perspektiven–basiertem Lesen [BGL+96] mit speziell auf das Umfeld der eingebetteten Systeme zugeschnittenen Perspektiven.

Neben einer Variation der Parameter des Experiments und weitergehenden Untersuchungen spielt aber auch die Replikation dieses Experiments eine wichtige Rolle auf dem Weg zu gesicherterem Wissen über die Wirkungsweise von Verfahren zur Softwareprüfung zu erhalten. Durch die Sammlung aller Materialen, die für die Replikation notwendig sind, hoffen wir, einen Beitrag auch in dieser Richtung geleistet zu haben.

Hinweis

Der Bericht enthält in der vorliegenden Form nur den Hauptteil. Der vollständige Anhang ist hier nicht abgedruckt, da er nur für eine Replikation des Experiments notwendig ist.

Falls Sie Interesse an einer Replikation des Experiments haben, sind wir gerne bereit, Ihnen die dazu notwendigen Unterlagen zu senden. Wenden Sie sich in diesem Fall bitte an Dietmar Ernst (`dietmar@informatik.uni-ulm.de`) oder Frank Houdek (`houdek@dbag.ulm.DaimlerBenz.com`).

Literatur

[ABL96] J.-R. Abrial, E. Börger und H. Langmaack (Herausgeber). *Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler Control*, Band 1165 der Reihe *Lecture Notes in Computer Science*. Springer–Verlag, 1996.

[Abr96] J.-R. Abrial. *The B–Book*. Cambridge University Press, 1996.

[BD93] A.-P. Bröhl und W. Dröschel (Herausgeber). *Das V–Modell: Der Standard für die Softwareentwicklung mit Praxisleitfaden*. R. Oldenbourg Verlag, 1993.

[BGL+96] V.R. Basili, S. Green, O. Laitenberger, F. Lanubile, F. Shull, S. Sørumgård und M.V. Zelkowitz. *The empirical investigation of perspective–based reading*. Technischer Bericht ISERN-96-06, Universität Kaiserslautern, 1996.

[Boe81] B.W. Boehm. *Software Engineering Economics*. Prentice Hall, Englewood Cliffs, New Jersey 07632, 1981.

[Bos94] K. Bosch. *Elementare Einführung in die angewandte Statistik*. Vieweg Verlag, 5. Auflage, 1994.

[BS87] V.R. Basili und R.W. Selby. *Comparing the Effectiveness of Software Testing Strategies*. IEEE Transactions of Software Engineering, SE-13(12):1278–1296, Dezember 1987.

[Cha95] J.K. Chaar. *On the evaluation of software inspections and tests*. In: *Proceedings of the 8th International Quality Week*. Software Research Inc., Juni 1995.

[DL87] T. DeMarco und T. Lister. *Peopleware: Productive Projects and Teams*. Dorset House Publishing, 1987.

[DM81] M. Dyer und H.D. Mills. *The Cleanroom approach to reliable software developement*. In: *Proceedings in Validation Methods Research for Fault–Tolerant Avionics and Control Systems Sub–Working–Group Meeting: Production of Reliable Flight–Cruical Software*, November 1981.

[Fin96] K. Finney. *Mathematical Notation in Formal Specification: Too Difficult for the Masses?* IEEE Transactions on Software Engineering, 22(2):158–159, Februar 1996.

[FLS95] K. Frühauf, J. Ludewig und H. Sandmayr. *Software–Prüfung: Eine Anleitung zum Test und zur Inspektion*. B.G. Teubner, 2. Auflage, 1995.

[FW82] D.P. Freedman und G.M. Weinberg. *Handbook of Walkthroughs, Inspections, and Technical Reviews — Evaluating Programs, Projects, and Products.* Little, Brown and Company, Boston and Toronto, 1982.

[GG93a] T. Gilb und D. Graham. *Software Inspections.* Addison–Wesley, Reading, Masachusetts, 1993.

[GG93b] M. Grochtmann und K. Grimm. *Classification Trees for Partition Testing.* Software Testing, Verification & Reliability, 3(2):63–82, Juni 1993.

[GW95] M. Grochtmann und J. Wegener. *Test Case Design Using Classification Trees and the Classification–Tree Editor CTE.* In: *Proceedings of the Eighth International Software Quality Week, San Franscisco*, 1995.

[Han96] R. Hanselmann. *Objektorientierte Softwareentwicklung für eingebettete Systeme.* Diplomarbeit, Fachhochschule Ulm, Juni 1996.

[HSS97] F. Houdek, F. Sazama und K. Schneider. *Risikominimierung bei der Einführung neuer Softwaretechnologien in die industrielle Praxis durch externe Experimentierfelder.* In: *Proceedings der GI–Jahrestagung 97*, 1997.

[HWW94] M. Heisel und D. Weber-Wulff. *Korrekte Software: Nur eine Illusion?* Informatik Forschung und Entwicklung, 9:192–200, 1994.

[Jon91] C.B. Jones. *Applied Software Measurement assuming Production and Quality.* MacGraw–Hill, 1991.

[KL95a] E. Kamsties und C.M. Lott. *An Empirical Evaluation of Three Defect–Detection Techniques.* Technischer Bericht ISERN-95-02, Universität Kaiserslautern, 1995.

[KL95b] E. Kamsties und C.M. Lott. *An Empirical Evaluation of Three Defect–Detection Techniques.* In: W. Schäfer und P. Botella (Herausgeber): *Software Engineering — ESEC '95: Proceedings of the 5th European Software Engineering Conference, Sitges, Spain, September 1995*, Band 989 der Reihe *Lecture Notes in Computer Science*, Seiten 362–383. Springer–Verlag, 1995.

[Mar91] B. Marick. *Experience with the Cost of different Coverage Goals for Testing.* Pacific Northwest Software Quality Conference, Oktober 1991. Available via anonymous ftp from cs.uiuc.edu:/pub/testing/experience.ps.Z.

[Mar92] B. Marick. *Generic Coverage Tool (GCT) User's Guide: Documentation for version 1.4 of GCT*, 1992. Available via anonymous ftp from cs.uiuc.edu:/pub/testing/gct.files/doc.ps.tar.Z.

[Mye78] G.J. Myers. *A Controlled Experiment in Program Testing and Code Walkthroughs/Inspections.* Communications of the ACM, 21(9):760–768, September 1978.

[Mye79] G.J. Myers. *The Art of Software Testing.* John Wiley & Sons, 1979.

[Mye91] G.J. Myers. *Methodisches Testen von Programmen.* R. Oldenbourg Verlag, 4. Auflage, 1991.

[Par90] H. Partsch. *Specification and Transformation of Programs: a formal approach to software development.* Springer–Verlag, 1990.

[PSV95] A.A. Porter, H. Siy und L.G. Votta. *A review of software inspections.* Technischer Bericht CS-TR-3552, University of Maryland, Dept. of Computer Science, College Park, MD, Oktober 1995.

[PV94] A.A. Porter und L.G. Votta. *An Experiment to Assess Different Defect Detection Methods for Software Requirements Inspections.* In: *Proceedings of the 16th International Conference on Software Engineering*, Seiten 103–112. IEEE Computer Society Press, Mai 1994.

[PVB95] A.A. Porter, L.G. Votta, Jr. und V.R. Basili. *Comparing Detection Methods for Software Requirements Inspections: A Replicated Experiment.* IEEE Transactions on Software Engineering, 21(6):563–575, 1995.

[RBS92] H.D. Rombach, V.R. Basili und R.W. Selby (Herausgeber). *Experimental Software Engineering Issues: Critical Assessment and Future Directions, International Workshop Dagstuhl Castle, Germany, September 1992, Proceedings*, Band 706 der Reihe *Lecture Notes in Computer Science.* Springer–Verlag, 1992.

[Rei91] W. Reif. *Korrektheit von Spezifikationen und generischen Moduln.* Doktorarbeit, Fakultät Informatik, Universität Karlsruhe, 1991.

[SBPM84] R.W. Selby, V.R. Basili, J. Page und F.E. McGarry. *Evaluating Software Testing Strategies.* In: *Proceedings of the Ninth Annual Software Engineering Workshop*, Seiten 42–53. NASA Goddard Space Flight Center, Greenbelt MD 20771, 1984.

[Som96] I. Sommerville. *Software Engineering.* Addison–Wesley, 5. Auflage, 1996.

[Sta92] Verlag Stahlschlüssel Wegst GmbH, Marbach. *Stahlschlüssel*, 16. Auflage, 1992.

[ZW97] M.V. Zelkowitz und D. Wallace. *Experimental validation in software engineering.* IEEE Computer, 1997. (to appear, available via *http://aaron.cs.umd.edu/pub/computer97.ps*).