Seminar „Simulation und Bildanalyse mit Java“ SS2004

Themenschwerpunkt: Tests in Informatik und Statistik

Arbeit von Christian Aich und Robert Reeb

Thema 2: Einführung in den Softwaretest

- was ist ein Testfall?
- Methoden der Testfallfindung
    - Black-Box-Test
    - White-Box-Test
- Testauswertung
- Software-Metriken
- Testende-Kriterien
- Alternativen zum Software-Test: „Manuelles“ Testen

Testfall

- **Was ist ein Testfall?**

  Der entscheidende Punkt, der auf den ersten Blick recht einfach erscheint, ist die Definition des Begriffs *Testen.* Viele Leute verwenden eine falsche Definition des Testens, was zu einem falschen Ansatz beim Programmtesten führt.
  Beispiele für solche falsche Definitionen sind Aussagen, wie

  „Testen ist der Prozess, der zeigen soll, dass keine Fehler vorhanden sind"
  „Der Zweck des Testens ist es zu zeigen, dass ein Programm die geforderten Funktionen korrekt ausführt.
  „Testen ist der Prozess, der das Vertrauen erzeugt, dass ein Programm das tut, was es soll.

  Diese Definitionen sind deshalb inkorrekt, da sie fast das Entgegengesetzte beschreiben, als was das Testen angesehen werden sollte.
  Da aber der Zweck des Testens ist, die Qualität zu verbessern oder die Zuverlässigkeit zu erhöhen, sollte man als Ziel das Finden von Fehlern haben und diese dann beheben.
  Eine bessere Definition ist daher:

  „Testen ist der Prozess, ein Programm mit der Absicht auszuführen, Fehler zu finden"

  Dies impliziert, dass das Testen ein destruktiver, ja geradezu ein sadistischer Prozess ist.

  Natürlich will man am Ende mit dem Test ein gewisses Vertrauen in das Programm etablieren, dass es das tut, was es tun soll. Aber dieses Ziel erreicht man am Besten durch eine sorgfältige Suche nach Fehlern.
  Dabei ist zu beachten, dass ein Testfall nicht nur aus Eingabedaten besteht. Beim Aufstellen der Testfälle sollten deshalb auch gleich erwartete Ausgaben sowie eventuelle Variablenänderungen erfasst werden.

  Es ist allerdings im Allgemeinen nicht machbar, oft sogar unmöglich, alle Fehler eines Programms zu finden. Dies führt direkt zu folgender Fragestellung:

Testfallentwicklung

Mit den gegebenen Beschränkungen der Zeit, Kosten, Computerzeit, etc. wird die folgende Frage zum Angelpunkt des Testens:

„Welche Untermenge aller denkbaren Testfälle bietet die größte Wahrscheinlichkeit möglichst viele Fehler zu finden?"

Zur Lösung dieser Frage werden in der Praxis die sog. Blackbox- und Whitebox-Tests verwendet.

- **Blackbox-Test**
  Dabei betrachtet der Tester das Programm als Blackbox, d.h. der Tester ist nicht an dem internen Verhalten und an der Struktur des Programms interessiert, sondern daran, Umstände zu entdecken, bei denen sich das Programm nicht gemäß den Anforderungen verhält.
  Die Testdaten werden nur aus der Spezifikation (Aus-/Eingabeparameter sowie die Funktion des Programms) abgeleitet, d.h. ohne spezielle Kenntnis der internen Struktur.

- **Whitebox-Test**
  Im Gegensatz hierzu wird beim Whitebox-Test die interne Struktur des Programms untersucht. Bei dieser Strategie definiert der Tester die Testdaten mit Kenntnis der Programmlogik.

Um diese Tests durchzuführen müssen wir uns zuerst auf sinnvolle **Testprinzipien** festlegen:

- Definition der erwarteten Werte
  sonst wäre es möglich, ein plausibles aber fehlerhaftes Ergebnis als korrekt zu betrachten
- eigene Programme testen ist ineffizient
  Dies ist ein psychologischer Effekt. Da Testen ein destruktiver Prozess ist, haben viele Programmierer Probleme, nach konstruktiver Arbeit an Design und Codierung „über Nacht“ die Einstellung zu ihrem Programm zu ändern.
- die Ergebnisse eines jeden Tests gründlich überprüfen
  Ein signifikanter Anteil an Fehlern, der schließlich gefunden wurde, wäre schon in früheren Testfällen erkennbar gewesen.
- Testfälle müssen für ungültige und unerwartete ebenso wie für gültige und erwartete Eingabedaten definiert werden
  Oft haben Testdaten mit unerwarteten oder unzulässigen Eingabewerten eine höhere Fehlererkennungsrate.
- Ein Programm zu untersuchen, um festzustellen, ob es nicht tut, was es tun sollte, ist nur die eine Hälfte der Aufgabe. Die andere Hälfte besteht darin, zu untersuchen, ob das Programm etwas tut, was es nicht tun soll
- Wegwerftestfälle vermeiden, d.h. die Tests sollten reproduzierbar sein
- Es sollte kein Testverfahren unter der stillschweigenden Annahme geplant werden, dass keine Fehler gefunden werden
- Testen ist eine kreative und herausfordernde Aufgabe

Daraus lässt sich ableiten, dass ein erfolgreicher Testfall dadurch gekennzeichnet ist, dass er einen bisher unbekannten Fehler entdeckt.
Als vernünftig erweist sich eine Teststrategie, die beide oben genannte Verfahren verwendet.

Vorab ein Überblick über die gängigen Methoden beider Verfahren:

Blackbox	**Whitebox**
Äquivalenzklassen	Erfassung (Ausführung, coverage) aller
Grenzwertanalyse	Befehle
Ursache-Wirkungsgraph	Entscheidungen
Fehlererwartung (error guessing)	Bedingungen
	Entscheidungen/Bedingungen
	Mehrfachbedingungen

Obwohl die Methoden alle getrennt behandelt werden, empfiehlt es sich, für einen wirksamen Test eines Programms die meisten, wenn nicht alle Methoden zum Testfallentwurf heranzuziehen, da jede Methode bestimmte Stärken und Schwächen zeigt.

Wir wollen zuerst mit den **Whitebox-Verfahren** beginnen:

Logic-coverage Testing (Testdeckungsgrad)

Beim Whitebox-Testen betrachtet man den Anteil des Programms (source code), der durch die Testfälle ausgeführt oder angesprochen wird.

- **Line coverage** (Ausführung aller Anweisungen)
  Hier soll jede Anweisung im Programm wenigstens einmal ausgeführt werden.
  Das Verhältnis von ausgeführten zu vorhandenen Anweisungen wird als C0-Überdeckung bezeichnet. Bei 100% C0-Überdeckung führen die Testfälle jede Anweisung mindestens einmal aus.

  Folgendem Beispielprogramm kann man entnehmen, dass dieses Kriterium äußerst schwach ist.

  *IF (A > 1) AND (*statt OR*) (B = 0)*

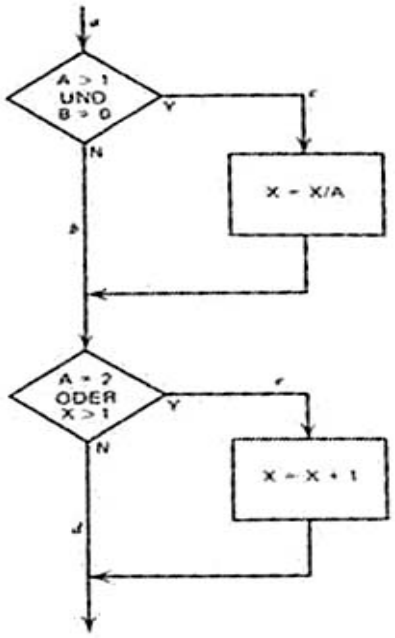
  THEN X := X/A;

  END;

  *IF (A = 2) OR (*AND*) (X > 1) (*X <= 1*)*

  THEN X := X - 1;

  END;



Wird der Pfad (ace) durchlaufen, werden zwar alle Anweisungen abgedeckt, allerdings können folgende Fehler unentdeckt bleiben: falls statt UND ein ODER stehen sollte, oder X > 0 statt > 1
Des weiteren gibt es einen Pfad (abd), durch das Programm, auf dem der Wert von X unverändert bleibt. Sollte das ein Fehler sein, so wird er nicht entdeckt. Mit anderen Worten, die Ausführung eines jeden Befehls ist so schwach, dass man diese Strategie in Allgemeinen als nutzlos ansieht.

- **Decision oder branch coverage**
  Dies ist eine strengere Strategie für die Erfassung der Logik bei der alle Entscheidungen oder Sprünge ausgeführt werden. Bei diesem Verfahren müssen hinreichend viele Testfälle entworfen werden, so dass bei jeder Entscheidung sowohl der THEN-Zweig als auch der ELSE-Zweig mindestens einmal durchlaufen wird.
  Das Verhältnis von ausgeführten zu vorhandenen Entscheidungspfaden (z.B. IF/ELSE-Zweige) wird als C1-Überdeckung bezeichnet. Bei 100% C1-Überdeckung liefert jede Entscheidung mindestens einmal TRUE und einmal FALSE.
  Man kann zeigen, dass diese Strategie auch die Forderung nach der Ausführung eines jeden Befehls erfüllt.

- **Condition coverage**
  Da bei der vorigen Methode nur Zweiwegentscheidungen oder Sprünge erfasst wurden, werden hier auch Mehrfachentscheidungen (CASE, SELECT, usw.) berücksichtigt. Hier wird verlangt, dass jeder mögliche Ausgang wenigstens einmal getestet wird und jeder Eingangspunkt des Programms oder der Unterroutine mindestens einmal angesprochen wird. In der obigen Grafik wird dies erreicht durch die Pfade (ace) & (abd) oder (acd) & (ace).
  Ein Nachteil dieses Verfahrens ist, dass der Pfad, auf dem X verändert wird, nur in der Hälfte der Fälle durchlaufen wird, also ein möglicher Fehler unentdeckt bleiben könnte.

- **Decision/condition coverage**
  Eine weitere Verbesserung der Teststrategie ist die Kombination der beiden vorangegangenen Strategien. Dazu sind genügend Testfälle erforderlich, so dass alle Pfade der Bedingungen in einer Entscheidung durchlaufen werden, dass alle Entscheidungen jeden möglichen Zweig benutzen und dass jeder Eingang mindestens einmal aufgerufen wird.
  Das Verhältnis von durchlaufenen zu vorhandenen Pfaden des Programms durch die Testfälle wird als C7-Überdeckung bezeichnet.
  Allerdings hat diese Strategie eine Schwäche. Obwohl es den Anschein hat, dass alle Bedingungen berücksichtigt werden, kommt es häufig vor, dass etliche Bedingungen andere Bedingungen maskieren, was durch den Compiler verursacht werden kann. Z.B. ziehen einige Compiler, die UND/ODER Bedingungen auseinander, was zur Folge hat, dass bei einer NEIN-Entscheidung von A > 0 die Bedingung B=0 überhaupt nicht mehr überprüft wird.

Art	*a*	*b*	*x*	*Soll*	*Ist*	*„Weg“*	*Bedingungen*
C0	2	0	1	-0.5	-0.5	ACE	
C1	2	0	1	-0.5	-0.5	ACE	
	1	1	1	1	1	ABD	
	2	0	1	-0.5	-0.5	ACE	TT:TF
	1	1	1	1	1	ABD	FF:FF
	1	0	2	2	1	ABE	FT:FT
	2	0	1	-0.5	-0.5	ACE	TT:TF
	1	1	1	1	1	ABD	FF:FF
	1	0	2	2	1	ABE	FT:FT
	2	2	2	0	1	ABE	TF:TT
C7	1	1	1	1	1	ABD	
	2	2	2	0	1	ABE	
	4	0	2	0.5	0.5	ACD	
	2	0	1	-0.5	-0.5	ACE	

- **Fazit**
  In Programmen mit Einfachentscheidungen ist als minimales Kriterium eine hinreichende Anzahl von Testfällen zu betrachten, für die gilt:

  1. Alle Zweige einer Entscheidung werden mindestens einmal durchlaufen
  2. Jeder Eingang wird mindestens einmal benutzt.

  In Programmen, die Mehrfachbedingungen verwenden, ist das Minimalkriterium eine hinreichende Anzahl von Testfällen, die alle möglichen Kombinationen von Bedingungen in einer Entscheidung berücksichtigen und alle Eingangspunkte in das Programm ansprechen.
  Es sollte beachtet werden, dass fehlende Statements unberücksichtigt bleiben, also Fehler, die durch vergessenen Code bedingt sind, bleiben unentdeckt.

Nun zu den **Blackbox-Verfahren**:

- **Äquivalenzklassen**
  Wie oben hergeleitet soll hier die Untermenge von Testfällen bestimmt werden, die mit höchster Wahrscheinlichkeit Fehler enthält. Zur Bestimmung dieser Untermenge sollte man wissen, dass ein gut ausgewählter Testfall zwei weitere Eigenschaften haben sollte:

  1. er reduziert die Anzahl anderer Testfälle, die man entwickeln müsste um ein vorgegebenes Ziel beim Testen zu erreichen
  2. er überdeckt eine große Menge anderer möglicher Testfälle, d.h. er sagt etwas über die An- oder Abwesenheit für Fehler für diesen speziellen Satz von Eingabedaten und darüber hinaus

Letzteres bedeutet, wenn ein Testfall in einer Äquivalenzklasse einen Fehler entdeckt, so erwartet man, dass alle anderen Testfälle in einer Äquivalenzklasse den gleichen Fehler finden. Anders herum ausgedrückt, wenn ein Testfall den Fehler nicht entdeckte, so erwarten wir, dass kein anderer Testfall in der Äquivalenzklassen diesen Fehler findet.
Diese beiden Überlegungen führen zu einer Blackbox-Strategie, die als Äquivalenzklassenbildung bekannt ist. Die zweite Bedingung dient zur Entwicklung einer Menge von Bedingungen, die getestet werden sollen. Mit der ersten Überlegung wird dann ein Minimalsatz von Testfällen definiert, der diese Bedingungen vollständig erfasst.

Bsp.:

Spezifikationen:

Eingabe des Alters für einen Vertragsabschluss.
Erlaubt sind Eingaben >=0.
Bei Eingabe von

0 – 17 muss ein Erziehungsberechtigter herangezogen werden
18 – 99 wird der Vertrag direkt abgeschlossen
> 99 muss das Personal aufgesucht werden

Kl.	***Beschreibung***	***erwarteter Output***
1	Alter zwischen 18 und 99	„Vertragsabschluss“
2	Alter zwischen 0 und 17	„Erziehungsberechtigter“
3	Negatives Alter	„Eingabe fehlerhaft“
4	Text enthält nichtnumerisches Zeichen	„Bitte nur Ziffern eingeben“
5	Alter über 99	„Personal aufsuchen“
6	???	???

Der Entwurf von Testfällen mit Hilfe der Äquivalenzklassenbildung erfolgt in zwei Schritten:

1. Bestimmung der Äquivalenzklassen
2. Definition der Testfälle

- **Definition der Testfälle**

1. Kennzeichnen Sie jede Äquivalenzklasse mit einer eindeutigen Zahl
2. Schreiben Sie jeweils einen neuen Testfall, der möglichst viele der bisher noch nicht behandelten Testfälle behandelt, bis alle gültigen Äquivalenzklassen durch Testfälle geprüft sind.
3. Definieren Sie jeweils einen neuen Testfall, der genau eine der bisher nicht behandelten ungültigen Äquivalenzklassen behandelt, bis alle ungültigen Äquivalenzklassen durch Testfälle geprüft sind.

Obwohl die Bildung von Äquivalenzklassen einer zufälligen Auswahl von Testfällen weit überlegen ist, zeigen sich dabei jedoch Schwächen (d.h. Übergehen von bestimmten Arten von äußerst wirkungsvollen Testfällen).

Bei den nächsten beiden Methoden (Grenzwertanalyse und Ursache-Wirkungsgrad) treten vieler dieser Schwächen nicht auf.

Grenzwertanalyse

Testfälle, die Grenzwerte untersuchen haben einen größeren Erfolg als Testfälle, die das nicht tun – wie Erfahrung zeigt. Damit sind solche Situationen gemeint, wo die Testfälle Werte an den Grenzen oder Ränder der Ein-/Ausgabe-Äquivalenzklassen berücksichtigen. In zwei Punkten unterschiedet sich die Äquivalenzklassenbildung von der Grenzwertanalyse:

1. In der Grenzwertanalyse muss jeder Rand einer Äquivalenzklasse in einem Testfall auftreten; im Gegensatz zur Auswahl irgendeines Elements der Äquivalenzklasse, das für diese repräsentativ angesehen wird.
2. Die Aufmerksamkeit gilt außerdem nicht nur den Eingabebedingungen (Eingaberaum), sondern es werden auch Testfälle entworfen, die den Ergebnisraum berücksichtigen (d.h. Ausgabeäquivalenzklassen).

Es ist schwierig, Rezepte für die Grenzwertanalyse zu geben, einige wenige Richtlinien lassen sich jedoch angeben:

1. Verwendung von Randwerten und ungültigen Werten direkt neben den Rändern
2. Ist die Ein- oder Ausgabe eines Programms eine geordnete Menge (z.B. lineare Liste oder Tabelle), so interessiert uns besonders das erste und letzte Element dieser Menge.
3. Suchen nach nicht offensichtlichen Grenzwerten

Die Grenzwertanalyse ist bei richtiger Anwendung eine der nützlichsten Methoden für den Testfallentwurf. Sie wird jedoch in vielen Fällen nicht effizient genug angewendet, da die Technik – oberflächlich betrachtet – zu einfach erscheint. Oft allerdings ist das Bestimmen von Grenzwerten mit einem beträchtlichen Aufwand an geistiger Arbeit verbunden.

- **Ursache-Wirkungs-Graph**

Grenzwertanalyse und Äquivalenzklassenbildung haben eine gemeinsame Schwäche: sie können keine Kombination von Eingabebedingungen untersuchen. Das Testen von Eingabekombinationen ist keine leichte Aufgabe, da die Anzahl der Kombinationen im Normalfall astronomisch hoch wird, auch wenn man die Eingabekombination in Äquivalenzklassen unterteilt. Geht man bei der Auswahl einer Untermenge von Eingabebedingungen nicht systematisch vor, so ergibt sich durch die Auswahl einer zufälligen Untermenge gewöhnlich ein ineffizienter Test. Bei der systematischen Auswahl einer Menge erfolgreicher Testfälle ist die Technik des Ursache-Wirkungs-Graphen äußerst hilfreich.
Ein Ursache-Wirkungs-Graph ist eine formale Sprache, in die eine Spezifikation aus der natürlichen Sprache übersetzt wird. Der Graph entspricht einer Schaltung der Digitallogik, wobei eine etwas einfachere Form als in der Elektronik verwendet wird. Eine umfassendes Beispiel hierzu findet sich in [Myers, S. 56 – 73].

- **Fehlererwartung (Error guessing)**
  Es wurde oft festgestellt, dass einige Menschen eine natürliche Begabung zum Programmtesten besitzen. Ohne Verwendung einer speziellen Methode wie der Grenzwertanalyse oder des Ursache-Wirkungs-Graphen scheinen sie über eine Spürnase für das Entdecken von Fehlern zu verfügen. Man kann dies folgendermaßen erklären: sie praktizieren – oft unbewusst – eine Technik des Testfallentwurfs, die man als Fehlererwartung (error guessing) bezeichnen könnte. Sie vermuten aufgrund ihrer Erfahrung und Intuition bestimmte Fehlerarten in einem Programm und entwerfen dann Testfälle um diese Fehler aufzuzeigen. Es ist schwer ein Verfahren für diese Technik anzugeben, da dies ein intuitiver und ad-hoc Prozess ist. Prinzipiell legt man eine Liste möglicher Fehler oder fehlerträchtiger Situationen an und definiert dann damit mögliche Testfälle. Das Auftreten einer 0 in einer Ein- oder Ausgabe ist beispielsweise eine fehlerträchtige Situation. Deshalb beschreibt man Testfälle, in denen spezielle Eingabewerte 0 sind und die bestimmte Ausgabewerte auf 0 setzen. Ist eine variable Anzahl von Ein- oder Ausgabewerten möglich (z.B. Anzahl der Einträge in einer Liste), so stellen die Fälle „kein“ oder „ein“ fehlerträchtige Situationen dar (kein Eintrag, gerade ein Eintrag in der Liste).
  Mit anderen Worten, man zählt solche Spezialfälle auf, die beim Entwerfen des Programms übersehen worden sein könnten.

- **Die Strategie**
  Die Testfall-Entwurfsmethoden, die bisher diskutiert wurden, kann man zu einer gemeinsamen Strategie kombinieren: jede Methode trägt einige nützliche Testfälle bei, aber keine kann allein eine Menge vollkommener Testfälle liefern.
  Selbst bei Kombination aller Techniken hat man keinerlei Garantie dafür, dass alle Fehler entdeckt werden, aber es hat sich herausgestellt, dass das ein vernünftiger Kompromiss ist. Es stellt außerdem ein beträchtliches Stück harter Arbeit dar, aber niemand hat behauptet, dass Testen einfach sei.

Testauswertung

Vergleich des Testergebnisses mit Erfahrungswerten aus alten Tests durch:

- Fehler / Codezeile
- Erstellung eines Testprotokolls für alle durchgeführten Tests
- Erstellung einer zusammenfassenden Mängelliste
- Falls nötig: Erfahrungsbericht zur Verbesserung zukünftiger Tests

Sofwaremetriken

Ein Schlüsselwort der Testliteratur und -diskussion ist **Test Coverage** (line coverage, decision coverage, condition coverage). Hierbei werden die Testfälle durch eine Software (z.B. JUnit) getestet und die prozentuale Abdeckung (coverage) angegeben. Kommerzielle Coverage-Tools können bislang nur die Line-Coverage bestimmen.
Wäre unser einziges Ziel, den Wert eines bestimmten Abdeckungsmaßes zu erhöhen, beispielsweise 100% Line Coverage zu erreichen, ist die Gefahr sehr groß, das eigentliche Ziel des Testens, das Finden von Fehlern, aus den Augen zu verlieren.
Dieses Vorgehen ist sinnvoll, um die prozentuale Abdeckung dafür zu nutzen, Codeteile zu finden, die innerhalb der Tests nicht benutzt werden. Es ist jedoch ein fragwürdiges Ziel, unter allen Umständen einen bestimmten Wert erreichen zu wollen um „aufhören zu können“ und sich damit in falscher Sicherheit zu wiegen. Die Ergebnisse der Coverage-Bestimmung können uns jedoch auf Schwachstellen unseres Tests hinweisen.

Folgende Kategorien nicht abgedeckten Codes müssen wir dabei unterscheiden:

1. Code, der nicht getestet wird, aber getestet werden sollte. Diese Entdeckung weist den größten Nutzen für uns auf.
2. Toter Code, der entfernt werden sollte. Auch das ist sehr nützlich.
3. Automatisch generierteter Code, der in unserer Anwendung nicht aufgerufen wird.
4. Code, der nur unter (zu) großem Aufwand in Tests zu erreichen wäre.

Eine interessante Ergänzung zur herkömmlichen Coverage-Analyse stellt **Mutation Testing** dar. Diese Art des Testens basiert auf gezielten Änderungen des Applikationscodes und der nachfolgenden Überprüfung, ob diese Änderungen von der ursprünglichen Testsoftware auch als Fehler erkannt wurden. Im Gegensatz zu herkömmlichen Abdeckungsmetriken können mit dieser Methode Codeteile identifiziert werden, die zwar im Zuge des Programms ausgeführt werden, aber deren Effekte nicht in den Tests überprüft werden.

Testende-Kriterien

Wie viel ist genug?

- Vollständiges Testen mit der erklärten Absicht, die Korrektheit eines Programms zu verifizieren, ist für alle nicht trivialen Programme nicht möglich.
- Für jedes System existiert ein akzeptables Fehlerniveau. Wie hoch dieses Niveau ist, hängt von der Art des Systems ab (z.B. eine Software für med. Geräte muss zuverlässiger sein als ein Computerspiel).
- Die richtige Menge an Unit Tests hat positive Auswirkungen auf die Entwicklungsgeschwindigkeit, sobald die Projektlaufzeit eine bestimmte Länge überschreitet.
- Unit Tests sind nicht die einzigen Tests, die verwendet werden sollten. Daher müssen sie auch nicht vollständig das gewünschte Fehlerniveau garantieren. Dafür sind auch die vom Kunden spezifizierten Akzeptanztests verantwortlich
- Zu wenige Tests bergen auch noch die Gefahr des Gefühls von falscher Sicherheit

Die optimale Testmenge umfasst daher nur all diejenigen Tests, welche die tatsächlichen Fehler finden. Es gilt daher zwei Aspekte gegeneinander abzuwägen: Die ökonomische Seite („Wie viel kostet mich welches Fehlerniveau?“) und die technische Seite („Wie viele Tests bringen mir maximale Geschwindigkeit, flexibles Design und zufriedene Entwickler?“).
In der Praxis ist es so, dass ein Team sein eigenes optimales Testniveau finden muss. Iteratives Rantasten an das richtige Niveau ist gefragt.

Jedoch wendet man tatsächlich oft (fälschlicherweise) folgende Regeln an:
Abbruch des Testens, wenn

1. die geplante Testzeit abgelaufen ist
2. alle Testfälle ohne Fehler durchgeführt wurden

Die erste Bedingung ist sinnlos, da sie durch absolutes Nichtstun erfüllt werden kann.
Die zweite Forderung ist ebenfalls nutzlos, da sie über die Qualität der Testfälle keine Aussage macht.
Es gibt andere, nützlichere Kriterien für die Beendigung des Testens:

Da das Ziel des Testens die Fehlerentdeckung ist, sollte man als Endekriterium die Entdeckung einer festgelegten Anzahl von Fehlern ansehen. Man kann zum Beispiel festlegen, dass ein Systemtest nach 50 gefundenen Fehler abgebrochen wird.
Dieses Vorgehen unterstützt unsere Vorstellung vom Testen. Es tritt dabei ein Problem auf, das aber zu überwinden ist, nämlich die Festlegung der Anzahl der Fehler, die entdeckt werden sollen, wozu man folgende Punkte beachten muss:

1. Eine Abschätzung der Gesamtzahl der Fehler im Programm
2. Eine Abschätzung darüber, welcher Anteil dieser Fehler überhaupt gefunden werden kann
3. Schätzungen darüber, welcher Anteil an den Fehlern in den einzelnen Testphasen entdeckt werden können

Diese Schätzungen kann man am besten aus eigenen Erfahrungswerten gewinnen.

Fehlerabschätzung:

Anwendung von statistischen Methoden:

- Fehlereinpflanzung (error seeding)

  Aber:

  - mehrere Fehler zusammen können ein Fehlverhalten bewirken
  - setzt sehr große Zahl von Fehlern voraus
  - setzt eine gleichmäßige Verteilung der echten Fehler voraus
  - setzt voraus, dass die Verteilung der eingepflanzten Fehler der der echten entspricht
  - setzt voraus, dass echte wie eingepflanzte Fehler mit gleicher Wahrscheinlichkeit gefunden werden
  - setzt auch voraus, dass es keine Wechselwirkung zwischen echten und eingepflanzten Fehlern gib

- Testen mit zwei unabhängigen Gruppen
  - Zwei unabhängige Gruppen G_1 und G_2 entwickeln jeweils Testdatenmengen T_1 resp. T_2 für Programm P
  - Die Gruppe G_i findet F_i Fehler (i=1,2)
  - F sei die (unbekannte) Anzahl der Fehler in Programm P
    es gelte die Annahme, dass beide Gruppen bei allen Fehlern wie auch bei allen Fehlerteilmengen die gleiche Effizienz haben (gleiche Wahrscheinlichkeit der Fehleraufdeckung).
    F wird mit stochastischen Methoden geschätzt.

- **Die unabhängige Testagentur**
  Wie schon früher betont, sollte man es vermeiden, seine eigenen Programme selbst zu testen. Innerhalb der Struktur einer Firma sollte die Testgruppe möglichst weit weg von der Entwicklergruppe angesiedelt werden. Tatsächlich sollte die Testgruppe besser überhaupt nicht zur gleichen Firma gehören, sonst unterliegt sie nämlich dem gleichen Managementdruck wie die Entwicklungsabteilung.

Manuelles Testen

Jahrelang glaubte man, dass Programme nur zur Ausführung am Computer geschrieben wurden, also nicht geeignet sind, von anderen Leuten gelesen zu werden. Mittlerweile hat die Erfahrung gezeigt, dass die sog. „human testing"-Techniken beim Auffinden von Fehlern so effektiv sind, dass eine oder mehrere von ihnen in jedem Programmprojekt eingesetzt werden sollte. Diese Methoden sollten nach Ende der Codierung und vor dem Beginn des Tests am Computer angewendet werden. Obwohl die meisten Leute wegen der informellen Art des manuellen Testens oft skeptisch reagieren, so tragen sie jedoch wesentlich zur Produktivität und Zuverlässigkeit in zwei Richtungen bei.
Erstens wird allgemein anerkannt: je eher Fehler gefunden werden, desto geringer sind die Kosten für die Fehlerkorrektur und desto höher ist die Wahrscheinlichkeit, die Fehler korrekt zu beheben. Zweitens scheinen die Programmierer einen psychologischen Wandel zu erleben, wenn der Test am Computer beginnt. Intern aufgebauter Druck scheint sich schnell zu entwickeln, und es besteht die Tendenz, den Fehler so schnell wie möglich zu beseitigen. Unter diesem Druck neigen die Programmierer dazu, mehr Fehler bei der Korrektur eines im Computertest gefundenen Fehler zu machen als bei der Korrektur eines früher gefundenen Fehlers.

- **Inspektionen und Walkthroughs**
  Codeinspektionen und Walkthroughs sind die beiden ursprünglichen Testmethoden ohne Computereinsatz (human testing), die viele Ähnlichkeiten untereinander aufweisen. Sie erfordern beide das Lesen und die visuelle Überprüfung eines Programms durch ein Testteam. Das Ziel des Testteams ist es Fehler zu finden, aber nicht Fehler zu beheben.
  Die Aktion wird von einer Gruppe durchgeführt (am besten drei/vier Teilnehmer), wobei nur einer der Teilnehmer der Autor des Programms ist. Das Programm wird also im wesentlichen von anderen Leuten und nicht vom Autor selbst getestet. Im Gegensatz zum „Schreibtischtest", an dem der Autor sein eigenes Programm überprüft, sind diese Methoden effektiver, wiederum weil andere Personen als der Autor an diesem Prozess beteiligt sind (analog zu oben: Testprinzipien). Diese Aktionen scheinen auch niedrigere Korrekturkosten zu ergeben, da bei der Fehlerentdeckung die exakte Natur des Fehlers festgestellt wird, sowie eine Menge von Fehlern gefunden wird, die später in einem Zug behoben werden können.
  Diese manuellen Methoden ersetzen allerdings das Testen am Computer nicht. Nur wenn beide Verfahren kombiniert werden, kann man einen effizienten Test erhalten.

- **Code-Inspektionen**
  Hier versucht ein Team beim gemeinsamen Lesen des Codes Fehler zu entdecken. Gewöhnlich besteht dieses Team aus einem kompetenten Programmierer (der Moderator), dem Programmautor, dem Programmdesigner (falls unterschiedlich zum Autor) und einem Testspezialist.
  Die eigentliche Inspektionssitzung läuft dann folgendermaßen ab:

  1. Der Programmierer erklärt die Programmlogik Anweisung für Anweisung. Die Erfahrung hat gezeigt, dass viele Fehler vom Programmierer selbst während des Vortrags entdeckt werden und nicht von den anderen Teilnehmern. Anders gesagt, der einfache Vorgang, ein Programm vor einem Publikum laut vorzutragen, scheint eine bemerkenswert effektive Methode der Fehlerentdeckung zu sein.
  2. Das Programm wird mit Hilfe einer Checkliste bekannter Fehler analysiert

Dabei sollte der Moderator darauf achten, dass sich die Teilnehmer auf die Entdeckung von Fehlern konzentriert und nicht auf ihre Korrektur, die von dem Programmierer selbst nach der Sitzung vorgenommen werden.

Ein wichtiger Bestandteil des Inspektionsprozesses ist die Verwendung einer Prüfliste, um das Programm auf übliche Fehler hin zu untersuchen. Dabei sollten folgende Punkte genau behandelt werden: Datenreferenz, Datenvereinbarung, Berechnungsfehler, Vergleich, Steuerfluss, Schnittstellenfehler, Ein-/Ausgabe-Fehler, sonstige Prüfungen.
Eine ausführliche Prüfliste findet sich in [Myers, S. 21-31].

- **Walkthroughs**
  Der Codewalkthrough ist ein ähnliches Verfahren wie die Codeinspektion, wobei durch die Untersuchung des Codes in einer Gruppe Fehler gefunden werden sollen. Es gibt viele Gemeinsamkeiten mit der Inspektion, nur sind die Prozeduren und Techniken der Fehlerentdeckung etwas verschieden.
  Die Vorbereitungen sind die gleichen wie bei der Codeinspektion: die Teilnehmer erhalten die Unterlagen einige Tage vor der Sitzung, um Gelegenheit zu haben, das Programm „auseinanderzunehmen". Das Vorgehen in der Sitzung jedoch unterschiedet sich von dem in einer Inspektion. Anstatt das Programm zu lesen oder Prüflisten zu verwenden, spielen die Teilnehmer Computer, d.h. ein Teammitglied (der sog. Tester) bereitet Testfälle auf dem Papier vor – eine repräsentative Menge von Eingaben (und erwarteten Ausgaben) für das Programm oder das Modul. Das Programm wird im Laufe der Sitzung mit den Testdaten im Geiste durchgespielt. Der Zustand des Programms wird dabei auf dem Papier oder einer Wandtafel festgehalten.
  Es sollten nicht zu viele und einfache Testfälle sein, da sie von Menschen und nicht vom Computer ausgeführt werden., der um Größenordnungen schneller ist. Die Testfälle selbst spielen daher keine so entscheidende Rolle; sie dienen eher als Anreiz zum Starten und um den Programmierer über seine Logik und Annahmen zu befragen. In den meisten Fällen werden mehr Fehler in der Diskussion mit dem Programmierer als direkt durch die Testfälle selbst entdeckt.
  Ebenso wie bei der Codeinspektion sollten sich Kommentare auf das Programm beziehen und nicht gegen den Programmierer gerichtet sein, um das Vorgehen produktiv zu gestalten. Ähnlich wie bei der Codeinspektion muss auch beim Walkthrough eine Nachbereitung erfolgen.

- **Schreibtischtest**
  Der Schreibtischtest kann als Ein-Mann-Inspektion oder -Walkthrough angesehen werden; dabei liest der Tester das Programm, überprüft es mit Hilfe einer Fehlerliste und/oder simuliert mit Testdaten einen Testfall. Der Schreibtischtest ist für die meisten Leute relativ unproduktiv. Ein zweiter wichtiger Grund ist der Verstoß gegen ein wichtiges Testprinzip (es nicht effektiv ist, sein eigenes Programm zu testen).
  Deshalb sollten zwei Programmierer besser ihre Programm austauschen. Aber auch so ist es weniger effektiv als der Walkthrough oder die Codeinspektion. Der Grund dafür liegt in der Zusammenarbeit des Walkthrough- oder Inspektionsteams, da die Sitzung ein gesundes Konkurrenzverhalten fördert (die Teilnehmer lieben es ihr Können bei der Fehlerentdeckung zu zeigen. Bei einem Schreibtischtest fehlt dieser nützliche Anreiz offensichtlich, da keine Zuschauer da sind. Kurz gesagt, der Schreibtischtest mag mehr bringen als überhaupt nichts zu unternehmen, aber er ist weniger erfolgreich als eine Inspektion oder ein Walkthrough.

- **Peer Ratings**
  Der letzte manuelle Reviewprozess hat nichts mit Programmtesten zu tun (d.h. es wird nicht versucht, Fehler zu entdecken). Dieses Verfahren wird hier jedoch mit aufgeführt, da es mit der Idee des Codelesens in Beziehung steht. Peer Rating ist eine Technik zur Beurteilung von Programmen, deren Autor dem Leser nicht bekannt ist und beschäftigt sich mit Begriffen wie Qualität, Wartungsfreundlichkeit, Erweiterbarkeit, Anwendbarkeit und Klarheit. Diese Technik soll dem Programmierer eine Selbsteinschätzung ermöglichen.
  In einer Sitzung (6 – 20 Teilnehmer) werden mehrere Programm anonym verteilt. Jeder Teilnehmer beschäftigt sich mit einem Programm und füllt ein Beurteilungsformular aus. Nach dem Review erhalten die Teilnehmer die anonyme Beurteilung ihrer Programme und einen statistischen Überblick über die Einstufung ihrer Programme in die Gesamtheit, ebenso eine Analyse ihrer Einstufung anderer Programme im Vergleich zur Einstufung des gleichen Programms durch anderer Teilnehmer. Durch dieses Verfahren sind die Programmierer in der Lage, ihre eigenen Fähigkeiten zu beurteilen.

Fazit:

Murphy sagt: *„Was schief gehen kann, geht schief!“*

Wir sagen: *„Was nicht getestet ist, läuft nicht!“*
„Testen beginnt am ersten Tag!“
[aus J. Siedersleben, S. 317]

Literatur:

G. J. Myers: *Methodisches Testen von Programmen,* Oldenbourg Verlag 1989

J. Link: *Unit Tests mit Java*, dpunkt.verlag, 2002

J. Siedersleben: *Softwaretechnik: Praxiswissen für Sofwareingenieure,*
Carl Hanser Verlag, 2003