Blockseminar Informatik

Themenschwerpunkt Softwaretest

Einführungsvortrag, Psychologie und Ökonomie des Softwaretests

Alfred Roth

FH Giessen-Friedberg
Fachbereich MNI

Abstract
Software testing is an important part of software development. However, it is very difficult to describe how to test software in the right way. The following short introduction gives the reader an impression of this problem. At the beginning there are two examples showing how testing shouldn't be done, then there are mentioned some aspects of testing, especially the psychological and economical issues. Afterwards there is a tiny introduction to the black-box and white-box test methods, and then follows the final conclusion: no program is free of faults or bugs!

Inhalt

1. Einleitung

Warum soll Software getestet werden? Diese Frage stellt sich dem Softwareentwickler und seinem Auftraggeber bei jeder Gelegenheit. Dass das Programm oder das System genau das macht, wofür es spezifiziert wurde, ist doch selbstverständlich, denken jedenfalls beide. Aber so einfach wie es sich anhört, ist die Sachlage nicht, denn bei einem so komplexen Gebilde, wie es Software nun einmal ist, kann nicht nach dem Motto verfahren werden: „Das Programm macht genau das, was ich mir *vorstelle*." An diesem Punkt fangen die Schwierigkeiten an, denn was sich der Programmierer und sein Auftraggeber vorstellen, kann bei einer ungenauen Spezifikation schon grundverschieden sein. Dies liegt z.B. an den unterschiedlichen Auffassungen darüber, wie ein Programm zu arbeiten hat und wie es bedient werden soll. Des Weiteren kann ein Entwickler bei der Erstellung eines Programms nicht alle Effekte auf andere Programme oder Störungen durch andere Programme beachten. Bei der Entwicklung wird zwar die Funktionalität durch den Programmierer getestet, z.B. durch den Kompilierungsvorgang und danach durch die Eingabe von ein paar Testdaten oder Testszenarien, aber das alleine reicht nicht aus, um die gewünschte Produktqualität zu erzielen.

Die geschichtliche Entwicklung auf dem Gebiet der Software-Entwicklung weist einen Übergang von äußerst einfachen Systemen zu Beginn bis zu den heutigen, sehr komplexen Systemen auf. Die frühesten Testmethoden unterscheiden sich daher sehr stark von heutigen: wo früher mit „brute force", d.h. ohne Methodik oder unter sturer Einhaltung von wenigen, erprobten Methoden, gearbeitet werden konnte, führt heute nur noch intelligente Planung und Strategie zum Erfolg. Dieser Unterschied ist allerdings nicht nur geschichtlich verankert: auch heute ist er zwischen kleinen und mittleren bis großen Projekten, zwischen jungen und erfahrenen Entwicklern, Testern und Unternehmen zu sehen.

In dieser Ausarbeitung soll nach zwei einführenden Beispielen, die zeigen, wie eine zu kurze oder mangelhafte Testphase ein Projekt zum Scheitern bringen kann, eine kurze Einführung in die Psychologie und Ökonomie des Testens gegeben werden. Dabei werden auch verschiedene Testverfahren kurz angesprochen.

2. Beispiele

2.1 Ariane 5

Ariane 5 startete ihren Erstflug am 4. Juni 1996. Nach 37 Sekunden Flugzeit wich die Rakete stark von ihrem Kurs ab und musste mitsamt ihrer Nutzlast, den vier Cluster-Satelliten, gesprengt werden. Es stellte sich heraus, dass die in Teilen von der Ariane 4 übernommene Software nicht den nötigen Anforderungen entsprach. Die Ariane 5 konnte schneller beschleunigen als die Ariane 4. 37 Sekunden nach Zünden der Rakete (30 Sekunden nach dem Abheben vom Boden) erreichte Ariane 5 in 3700 m Flughöhe eine Horizontal-Geschwindigkeit von 32768.0 (interne Einheiten). Dieser Wert lag etwa fünfmal höher als bei Ariane 4. Dies führte zu einem Überlauf einer Variablen des Lenksystems (Umwandlung einer 64-Bit-Fließkommazahl (die horizontale Geschwindigkeit) in eine vorzeichenbehaftete 16-Bit-Ganzzahl), der jedoch nicht abgefangen wurde.

Start der Ariane 5

Explosion der Ariane 5

Der Ersatzrechner (Redundanz!) hatte das gleiche Problem schon 72 Millisekunden vorher und schaltete sich sofort ab. Daraus resultierte, dass Diagnose-Daten zum Hauptrechner geschickt wurden, die dieser als Flugbahndaten interpretierte. Daraufhin wurden unsinnige Steuerbefehle an die seitlichen, schwenkbaren Feststoff-Triebwerke, später auch an das Haupttriebwerk gegeben, um die großen Flugabweichungen (über 20 Grad) korrigieren zu können. Die Rakete drohte jedoch auseinander zu brechen und sprengte sich selbst (nach 39 Sekunden).
Ein intensiver Test des Navigations- und Hauptrechners wurde nicht unternommen, da die Software schon bei Ariane 4 erprobt war. Das Unglückliche daran war, dass die Software für den Flug nicht unbedingt notwendig gewesen wäre und nur zu Startvorbereitungen und dem Start selbst diente. Sie sollte nur während einer Übergangszeit von 50 Sekunden aktiv sein (aus Sicherheitsgründen, bis die Bodenstation bei einer Startunterbrechung die Kontrolle übernommen hätte). Trotz des ganz anderen Verhaltens der Ariane 5 wurde dieser Wert nicht neu überlegt.
Hier kam es zu einer Verkettung von unglücklichen Umständen und menschlichem Versagen, da Programmstücke ohne neue Tests einfach übernommen wurden. Bei nur 3 von 7 Variablen wurde ein Überlauf geprüft - für die anderen 4 Variablen existierten Beweise, dass die Werte klein genug bleiben würden (Ariane 4). Diese Beweise galten jedoch nicht für die Ariane 5 und wurden dafür auch gar nicht nachvollzogen. Auch dass der Ersatzrechner die identische Software hatte und die System-Spezifikation festlegte, dass sich im Fehlerfall der Rechner abschalten und der Ersatzrechner einspringen sollte (ein Re-Start eines Rechners war nicht sinnvoll, da die Neubestimmung der Flughöhe zu aufwändig war), führten zu der Katastrophe.

Glücklicherweise kamen keine Menschen ums Leben, doch der materielle Schaden belief sich auf etwa 500 Millionen US-Dollar.

Ada-Programm des Trägheits-Navigationssystems (Ausschnitt):

```
...
declare
  vertical_veloc_sensor: float;
  horizontal_veloc_sensor: float;
  vertical_veloc_bias: integer;
  horizontal_veloc_bias: integer;
  ...
begin
  declare
    pragma suppress(numeric_error, horizontal_veloc_bias);
  begin
    sensor_get(vertical_veloc_sensor);
    sensor_get(horizontal_veloc_sensor);
    vertical_veloc_bias := integer(vertical_veloc_sensor);
    horizontal_veloc_bias := integer(horizontal_veloc_sensor);
    ...
  exception
    when numeric_error => calculate_vertical_veloc();
    when others => use_irs1();
  end;
end irs2;
```

Der hervorgehobene Text zeigt die Typumwandlung, die zum Fehler führte.

2.2 Toll Collect

Toll Collect wurde im März 2002 als Joint Venture der Deutschen Telekom, Daimler Chrysler und der französischen Cofiroute (Compagnie Financière et Industrielle des Autoroutes - Cofiroute war einbezogen worden, weil von den Bewerbern Erfahrung mit vergleichbaren Projekten verlangt worden war und Telekom bzw. Daimler Chrysler dies nicht nachweisen konnten) gegründet. Die Gesellschaftsanteile verteilen sich wie folgt:

- Telekom 45 Prozent
- Daimler Chrysler 45 Prozent
- Cofiroute hält die restlichen zehn Prozent

Das Unternehmen beschäftigt nach eigenen Angaben rund 750 Mitarbeiter an sieben Standorten, darunter Berlin (~200), Bonn (~150) und Potsdam (~400).
Mit dem System von Toll Collect sollte die Gebührenerfassung von LKWs von einer pauschalen Gebühr auf eine wegstreckenerfasste Autobahngebühr umgestellt werden. Nach einigen politischen Querelen hatte Toll Collect im Juli 2002 den Zuschlag erhalten, im September 2002 wurde der Vertrag mit dem Bundesverkehrsministerium unterzeichnet. Für den Betrieb des Mautsystems sollte Toll Collect zwölf Jahre lang jährlich ca. 650 Millionen Euro aus den Mauteinnahmen erhalten.
Die EU Wettbewerbsbehörde meldete jedoch Bedenken wegen einem möglichen Monopol von Daimler Chrysler bei der OBU (On Board Unit – das Gerät im Fahr-

zeug, welches die Wegstrecke per Satellitenortungssystem GPS erfasst, mit der Anzahl der Fahrzeugachsen, der Schadstoffklasse, dem Fahrzeugkennzeichen und mit den gebührenpflichtigen Autobahnen zu einem Paket zusammenfasst und mit einem Mobilfunksender per SMS an den Zentralrechner von Toll Collect sendet) an. Aus diesem Grund mussten verschiedene Auflagen erfüllt werden, z.B. dass die OBU auch mit Geräten anderer Hersteller kompatibel sein sollten. Dies alles führte zu einer Verzögerung bei der Einführung des Mautsystems, die dem Staat einen erheblichen Einnahmeverlust zufügte, denn mit dem geplanten Beginn der neuen Maut wurde das alte System (Zahlung einer Pauschalgebühr) abgeschafft.
Der Start für das neue System sollte am 31. August 2003 sein. Aber es gab erhebliche Probleme, vor allen Dingen mit den OBUs:

- OBUs reagierten nicht auf Eingaben,
- OBUs ließen sich nicht ausschalten,
- OBUs schalteten sich grundlos aus,
- OBUs zeigten unterschiedliche Mauthöhen auf identischen Strecken an,
- OBUs wiesen Autobahnstrecken als mautfrei aus, obwohl diese mautpflichtig waren,
- OBUs wiesen Strecken außerhalb des Autobahnnetzes als mautpflichtig aus;
- Siemens-Geräte passten nicht in den genormten Einbauschacht.

Die Fehler wurden z. T. von der fehlerhaften Software verursacht, die die Geräte auch automatisch per GSM Mobilfunk aktualisieren sollte. Da die Fehler so erheblich waren, konnte der geplante Start des Systems nicht eingehalten werden und wurde mehrfach verschoben, bis schließlich der 1. Januar 2005 als Einführungstermin festgeschrieben wurde. Das Ganze war mit einigen politischen Querelen verbunden, es gab einen zweifachen Führungswechsel der Toll Collect Geschäftsführung und über die entstandenen Kosten, vor allen Dingen durch den Mautausfall, wurde lange diskutiert: die Bundesregierung hatte mit Mehreinnahmen über 2,6 Milliarden Euro gerechnet (im Vergleich zu der alten Maut) und diese schon fest in den Bundeshaushalt eingeplant.
Der aktuelle Stand (März 2005) sieht folgendermaßen aus: Das System läuft fehlerfrei mit einer Zuverlässigkeit von 99%, aber die Software in den OBUs ist in ihrer Funktionalität eingeschränkt, da die Fehler nicht so schnell beseitigt werden konnten. Vor allen Dingen die automatischen Updates der Software über GSM sind deaktiviert und sollen erst zur nächsten Stufe (1. Januar 2006) möglich sein. Über diese Updatefunktion kann auf Streckenneubauten, Änderungen der Kilometerpauschale und Umwidmungen von Strassen (falls eine Bundestrasse wegen „Mautflucht“ zu einer gebührenpflichtigen Strasse wird) eingegangen werden.
Das als Vorzeigeprojekt geplante System von Toll Collect hat sich damit zu einem Lehrstück in Sachen Projektfehler entwickelt. Aber es wird vom Bundesministerium für Verkehr, Bau- und Wohnungswesen als technische Innovation gepriesen:
„Förderung innovativer Technologien: Für die Maut wird in Deutschland eines der anspruchvollsten und innovativsten Erhebungssysteme errichtet. Deutschland wird damit zum Schrittmacher auf dem Feld der elektronischen Mauterhebung.“
[http://www.bmvbw.de/Lkw-Maut-.720.htm]

3. Unterschiedliche Aspekte des Softwaretestens

Das Testen von Software kann von unterschiedlichen Warten aus betrachtet werden. Es gibt zum einen die technischen Aspekte, wie zum Beispiel die Testauswahl, Art und Umfang des Testszenarios, und die psychologischen und ökonomischen Aspekte. Während die technischen Aspekte sich eventuell schon aus der Art des Programms ergeben, sind die anderen Aspekte bei allen Tests gleich. Man könnte sogar sagen, dass die menschliche Psychologie der wichtigste Schlüssel zu einem erfolgreichen Softwaretest ist. Aus den ökonomischen Aspekten ergibt sich erst die Notwendigkeit von geplanten Softwaretests, da zum einen die Korrektheit des Programms gewährleistet sein soll, andererseits aber die Kosten so minimal wie möglich zu halten sind – ein scheinbarer Widerspruch, denn je länger und aufwändiger die Tests sind, desto höher werden auch die Kosten.
Die Auswahl der Testkriterien kann sich schon aus der Art der Software ergeben. So ist es z.B. nicht sinnvoll eine Web-Anwendung nur auf ihre Funktionalität hin zu testen. Es muss auch das Verhalten unter Last, d.h. wenn viele Benutzer gleichzeitig darauf zugreifen und die Anwendung benutzen, getestet werden. Auch die Einflüsse von parallel auf dem Webserver laufenden Programmen sollten unbedingt erprobt werden. Wenn z.B. eine Anwendung im Hintergrund den Server stark belastet und es in der Folge zu einer Zeitüberschreitung bei der zu testenden Anwendung kommt, weil sie ihre Daten nicht rechtzeitig bekommt, so kann dies beim Testen übersehen werden, falls nicht unter realistischer Last getestet wird (wobei realistische Last ein relativer Begriff ist, der erst einmal durch Seitenzugriffe oder ähnliches ermittelt werden muss und selbst dann kann es bei einem plötzlichen Anstieg der Besucherzahlen zu unverhofften Effekten kommen).

3.1 Psychologie

Der Begriff „Testen“ ist auf den ersten Blick trivial, doch bei näherer Betrachtung stellt sich heraus, dass sich dahinter eine etwas komplexere Materie verbirgt.
Jeder Mensch zeigt eine zielorientierte Tendenz, die sich in seinem Handeln und Denken widerspiegelt. Aus diesem Grund ist es für die Psychologie beim Testen wichtig, sich eines immer wieder zu verdeutlichen:

- „Testen ist der Prozess, ein Programm mit der Absicht auszuführen, Fehler zu finden.“[Myers 2004 S.6]

Vielfach sind aber folgende Gedanken in den Köpfen der Tester:

- „Testen ist der Prozess, der zeigen soll, dass keine Fehler vorhanden sind.“
- „Testen ist der Prozess, um zu zeigen, dass das Programm das tut, was es soll.“
- „Der Zweck des Testens ist es zu zeigen, dass ein Programm die geforderten Funktionen korrekt ausführt.“
- usw. [Myers 2004 S. 5]

Diese wichtige Unterscheidung in dem Grundgedanken des Testens mag sich vielleicht unwichtig anhören und erweckt den Anschein, dass es sich hierbei um kleinliche Wortspielereien handelt, aber es ist sehr wichtig, sich dies immer wieder zu verdeutlichen. Durch die Annahme das Programm sei richtig, stellt sich beim Tester eine bestimmte Erwartungshaltung ein, die eventuell dazu führt, dass Testdaten ausgewählt werden, die mit geringerer Wahrscheinlichkeit einen Fehler finden. Dieses Phänomen des zielgerichteten Denkens kennt wahrscheinlich jeder, der schon einmal ein

Programm geschrieben hat. Es werden Fehler im Programmcode gemacht, die auch nach einer Durchsicht nicht gefunden werden. Ein hinzugezogener, neutraler Beobachter sieht diesen Fehler aber auf den (beinahe) ersten Blick. Ein ähnliches Phänomen ist auch beim Schreiben eines Textes zu beobachten. Tippfehler (sehr beliebt sind Buchstabendreher die beim schnellen Tippen auftreten können, wie z.B. „dre“ statt „der“) oder unklare Formulierungen, aber auch Satzstellungen die sich eigenartig anhören, werden vom Autor beim Korrekturlesen leicht überlesen, da er schon vor dem „inneren Auge“ die korrekte Form sieht. Der so genannte „Querleser“ kennt diese innere Form des Autors aber nicht und kann so die Fehler in der Syntax, die Tipp- und Grammatikfehler korrigieren. Spätestens jetzt sieht auch der Schreiber seine Fehler und fragt sich, wie er das übersehen konnte: indem er einfach davon ausging alles richtig korrigiert zu haben und die Fehler einfach übersah. Dies kann auch beim Testen von Software passieren, indem die eigene Erwartungshaltung durch das Unterbewusstsein eine Fehlerfreiheit vorspielt, die eigentlich nicht da ist.
Dadurch ergeben sich weitere psychologische Aspekte des Testens:
Der Tester sollte nicht der Programmierer sein. Es sollten im Idealfall zwei verschiedene Personen sein. Durch diese Trennung wird das Problem der Zielorientiertheit zum effizienteren Testen benutzt. Der Tester bekommt die Vorgabe Fehler zu finden und wird gezielt danach suchen, ohne Rücksicht auf den Programmierer zu nehmen. Der sportliche Ehrgeiz ist geweckt und das Finden von Fehlern wird zu einer Aufgabe, die einen „destruktiven, ja geradezu sadistischen“ [Myers 2004 S. 6] Charakter annimmt. Diese Sichtweise ist vielleicht nicht für jeden geeignet, denn im Grunde möchte jeder Mensch etwas erschaffen oder konstruktiv tätig sein, die Phase des Zerstörens wurde in der Kindheit abgelegt. Hinzu kommt noch der gesellschaftliche Aspekt: jeder zeigt lieber etwas, dass er geschaffen hat, nicht etwas dass durch seine Hilfe zerstört wurde. Diese Aspekte des Testens strahlen bis in die Anfänge der Planung und Definition von Testszenarien und Testfällen.
Der Test ist erfolgreich absolviert worden. Hinter dieser unscheinbaren Behauptung versteckt sich wieder eine Menge an Psychologie. Die meisten Projektmanager nennen einen Test erfolgreich, wenn keine Fehler gefunden wurden. Dies ist natürlich für den Tester kein allzu großer Ansporn, wenn seine Arbeit erfolgreich war und dies als nicht erfolgreicher Test aufgezeichnet wird (denn ein erfolgreicher Test – nach dieser Definition – bedeutet ja, der Tester hat seine Arbeit umsonst gemacht). Das widerspricht wieder dem zielgerichteten Mensch, der seine Arbeit als etwas Positives sieht, das mit einem Erfolgserlebnis beendet wird. Eine Analogie dafür, dass Tests nicht in allen Bereichen mit der gleichen Erfolgsaussicht behandelt werden, ist der Arztbesuch. Hier werden auch Tests gemacht und diese Ergebnisse sind auch positiv oder negativ (in einfacher Form dargestellt). Hier haben diese Worte aber eine andere Bedeutung, denn ein erfolgreicher Test bringt einen Befund zustande, der dann behandelt werden kann. Wohingegen ein erfolgloser Test nur bedeutet, das die Krankheit oder die Beschwerden nicht lokalisiert werden konnten und weitere Tests gemacht werden müssen. Falls diese dann keine Ursache für die Beschwerden finden, d.h. die Tests immer noch negativ sind, muss immer weiter gesucht werden, bis alle technischen und medizinischen Möglichkeiten erschöpft sind.
Testen soll zeigen, dass keine Fehler vorhanden sind. Mit dieser Aussage wird der Tester dazu verdammt, seine Aufgabe bis ins Unendliche auszudehnen, was natürlich seiner Motivation wiederum keinen Auftrieb gibt. Das Resultat aus dieser unmöglichen Aufgabe ist eine innere Resignation, denn eine unlösbare Aufgabe kann ja nicht gelöst werden, also warum sich überhaupt den Kopf zerbrechen. Aus diesem Grund kann es dazu kommen, dass selbst sichtbare Fehler übersehen werden und der ganze Test nicht mit der nötigen Sorgfalt durchgeführt wird. Das Ganze kann

man mit dem Mythos von Sisyphos vergleichen: „Sisyphos Strafe in der Unterwelt bestand darin, einen Felsblock einen steilen Hang hinaufzurollen. Immer kurz bevor er das Ende des Hangs erreichte, entglitt ihm der Stein und er musste wieder von vorne anfangen. Heute spricht man deshalb bei unmöglichen Aufgaben von Sisyphos-Arbeit.“[http://de.wikipedia.org/wiki/Sisyphos]

3.2 Ökonomie

Nach den vorangegangenen psychologischen Betrachtungen stellt sich nun die Frage nach der Ökonomie. Wann ist ein Test abgeschlossen, wie viel Zeit soll mit dem Testen verbracht werden? Auf diese Fragen gibt es keine allgemein gültigen Antworten, da dies von dem jeweiligen Programm oder System und dessen Komplexität abhängt. Es ist einleuchtend, dass ein Programm mit wenigen Zeilen ein anderes Testverfahren benötigt als z. B. ein Betriebssystem, das aus mehreren Millionen Zeilen Programmcode (bei Windows 2000 waren es ca. 30 Millionen Zeilen!) besteht. Wenn man sich diese Dimension betrachtet wird schnell deutlich, dass hier nicht alles getestet werden kann, falls der Aufwand für die Tests nicht ins Unermessliche steigen soll und dass es unmöglich ist, die Fehlerfreiheit eines Programms zu garantieren. So kann auch die Laufzeit eines Programms schon zu Fehlern führen, die niemand bedacht hat, wie z. B. bei der Problematik mit der Umstellung zum Jahr 2000. Welcher Programmierer konnte zu den Anfangszeiten der Informatik (als Speicherplatz teuer war und damit sehr vorsichtig umgegangen werden musste) daran denken, dass dies Jahrzehnte später zu Problemen führen würde? Um dies einigermaßen zufrieden stellend zu meistern, muss ein genauer Testplan spezifiziert werden, bei dem die zu erwartenden Ergebnisse bekannt sind und auf die Einhaltung dieser getestet wird. Es muss also schon *vor* dem Testen eine genaue Vorstellung erarbeitet werden, wie der Test auszusehen hat und der Test endet genau dann, wenn die *vorher* definierten Testziele erreicht sind. Dann kann von einem wirtschaftlichen Testen gesprochen werden. Es gibt verschiedene Tests die eingesetzt werden können (dynamische, statische, aufgabenorientierte, produktorientierte, schwachstellenorientierte Tests, usw.), hier sollen nur zwei prinzipielle Grundgedanken aufgegriffen und kurz erläutert werden:

3.2.1 Black Box-Test

Das *Black-Box-Testen* wird auch als „funktionaler Test“ oder „exterior test“ bezeichnet. Wesentliches Merkmal des Black-Box-Testens ist, dass hier das Ein-/Ausgabeverhalten ohne Berücksichtigung der inneren Struktur des Testobjektes (Schnittstellentest) geprüft wird. Das heißt 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 der Spezifikation verhält, er betrachtet das Programm eben als Blackbox. Die Testdaten werden nur aus der Spezifikation abgeleitet (d.h. ohne Kenntnisse von der internen Struktur des Programms zu verwenden). Wichtig ist, das Verhalten des Testobjekts bei fehlerhaften Eingabedaten zu überprüfen. Black-Box-Testen ist im Wesentlichen für das dynamische Testen, d.h. die Ausführung mit Testdaten, von Bedeutung. Insofern ist Black-Box-Testen im Wesentlichen auf das Testen von Programmen ausgerichtet; darüber hinaus ist diese Testart bei Verfügbarkeit entsprechender Hilfsmittel z.B. zum Prototyping auch für Dokumente von Bedeutung.

Durch Black-Box-Testen kann das Fehlen von Aufgaben/Funktionen bzw. Eigenschaften des Testobjekts erkannt werden. Dagegen können durch Black-Box-Testen keine Informationen darüber geliefert werden, ob alle Funktionen des Testobjekts auch tatsächlich benötigt werden oder ob Datenobjekte manipuliert werden, die keinen Einfluss auf das Ein-/Ausgabeverhalten des Testobjekts haben. Hinweise auf Design- und Implementierungsfehler liefert nur der *White-Box-Test*.

3.2.2 White-Box-Test

Das White-Box-Testen wird auch als „Strukturtest“ oder „interior test“ bezeichnet. Wesentliches Merkmal des White-Box-Testen ist, dass hier das Ein-/Ausgabeverhalten unter Betrachtung der inneren Strukturen (Schnittstellen- und Strukturtest) geprüft wird. Das heißt der Tester definiert die Testdaten mit Kenntnis der Programmlogik. Das Ziel dabei ist, für jeden möglichen Pfad durch das Testobjekt das Verhalten des Testobjekts in Abhängigkeit von den Eingabedaten festzustellen. Bei der Wahl der Eingabedaten ist laut [Pomberger / Blaschek 1996 S. 152] zu berücksichtigen, dass:

- jedes Modul und jede Funktion des Testobjektes mindestens einmal aufgerufen wird,
- jeder Zweig mindestens einmal durchlaufen wird,
- möglichst viele Pfade durchlaufen werden.

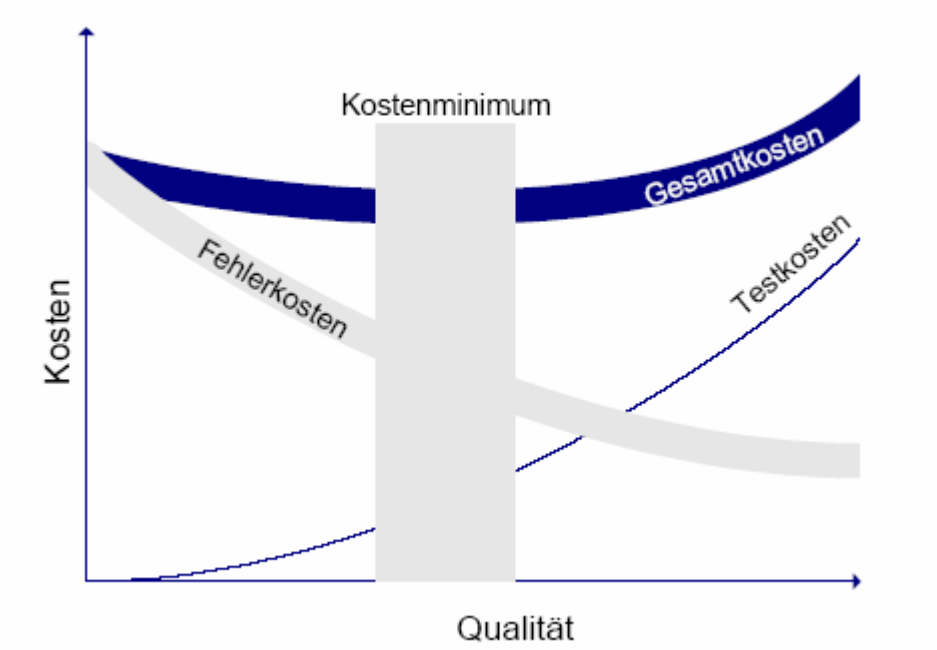
Für gut strukturierte Programme ist es praktisch unmöglich alle Pfade (alle möglichen Anweisungsfolgen) auszutesten. Auch bei einer automatischen Testdatengenerierung ist es unmöglich, alle Pfade zu testen, weil die Anzahl der dazu benötigten Testläufe viel zu hoch ist. Trotzdem ist der White-Box-Test eine der wichtigsten Testmethoden, weil man damit für eine begrenzte Anzahl von Programmpfaden die richtige Manipulation der Datenstrukturen und das Ein-/Ausgabeverhalten von Testobjekten überprüfen kann. White-Box-Testen ist sowohl für das statische als auch dynamische Testen von Bedeutung. Vorteil des White-Box-Testens ist, dass das Vorhandensein nicht geforderter/ gewünschter Aufgaben/Funktionen bzw. Eigenschaften des Testobjekts erkannt werden kann. Demgegenüber wird als Nachteil angeführt, dass das Fehlen erforderlicher Aufgaben/Funktionen nicht festgestellt wird; diese Aussage ist nicht generell gültig, da durch Bereitstellung der Soll-Ergebnisse und entsprechende Ergebnisprüfung auch derartige Fehler gefunden werden können.

Insgesamt ist festzustellen, dass weder allein durch Black-Box-Testen noch durch ausschließliches White-Box-Testen eine vollständige Abdeckung der Testaufgaben gegeben ist; vielmehr sind beide Testarten in Kombination miteinander anzuwenden.
Ein weiterer wichtiger Bestandteil von Test ist das erneute Testen, nachdem Modifikationen, z. B. im Laufe der Fehlerbereinigung, am Programmcode vorgenommen wurden. Dies treibt die Kosten abermals in die Höhe, ist aber unerlässlich, da sich durch die Fehlerbeseitigung wieder neue Fehler in das Programm eingeschlichen haben können.

Durch die Aufteilung in verschiedene Testgegenstände kann auch die Komplexität der Tests und damit auch die Wirtschaftlichkeit gesteigert werden. Es besteht die Möglichkeit erst einmal die einzelnen *Komponenten* zu testen, dann die *Integration*, anschließend das Ganze einem *Systemtest* zu unterziehen. Zum Schluss muss dann noch ein *Abnahmetest* bei der Auslieferung durchgeführt werden. Das Ganze kann mit einem Puzzle verglichen werden, bei dem zuerst die einzelnen Bauteile, dann die Bauteile im Zusammenbau und anschließend das ganze Puzzle geprüft wird. Zum

Schluss, bevor das Ganze auf einer Trägerplatte fixiert wird, wird dann noch einmal ein Abnahmetest durchgeführt – natürlich ist ein Softwaretest viel komplexer, aber das Beispiel verdeutlicht anschaulich die Modularität, durch die Tests einerseits übersichtlicher werden und andererseits auch kostengünstiger.

Wichtig ist auch die möglichst frühe Fehlerbehebung, da Fehler, die erst spät erkannt werden, deutlich höhere Kosten verursachen als Fehler, die in der Anfangsphase des Projekts festgestellt werden. Bis zu 50% aller Fehler sind schon durch eine mangelhafte oder nicht ausreichende Anforderungsanalyse entstanden [Kit, 1995]. Hier ist der Fehler nicht in der Software, sondern schon *vor* der eigentlichen Entwicklung entstanden. Gründe hierfür können Missverständnisse zwischen dem Auftraggeber und dem Entwickler sein, oder mangelhafte Kommunikation während der Entwicklung. Auch auf diese Fehler muss beim Testen von Software eingegangen werden, am einfachsten durch die genaue Festschreibung der Anforderungen und der Entwicklung von Testszenarien auf genau diese Anforderungen hin. Ein Zusammenhang zwischen Fehlerkosten und Testkosten zeigt die folgende Graphik:



Es ist gut ersichtlich, dass der optimale Bereich eines Kostenminimums nicht genau definiert werden kann. Es ist also immer abzuwägen, wann der Test als beendet angesehen werden kann und wann die Kosten für das Testen ein sinnvolles Maß überschreiten. Es wird davon ausgegangen, dass für die Tests 25% bis 50% des Gesamtbudgets erforderlich sind.

4. Weitere Testprinzipien

Mit einigen Maßnahmen im Vorfeld der Testplanung kann das Testen leichter und kostengünstiger gemacht werden, ohne dass die Qualität leidet:

Mit einer sehr genauen *Bedarfsanalyse* kann schon im Vorfeld einiges an Fehlern ausgeräumt werden. *Prototyping* kann auch hilfreich sein, indem die Prototypen mit dem Kunden zusammen erprobt werden. Hier können sich Verständigungsfehler sehr schnell in der Anfangsphase des Projekts zeigen. Durch *testbares Programmdesign* kann das Testen auch erleichtert werden, so sind z. B. sehr lange Funktionen und ein undokumentierter Quellcode für den Tester schwer nachvollziehbar. Auch

der effiziente Einsatz von *Compilern* und *Debuggern* schon bei der Programmerstellung kann im Vorfeld einiges an Fehlern aufdecken. Die *genaue Überprüfung der Testergebnisse* ist sehr wichtig, denn auch hier kann sich bei einer zu schnellen Durchsicht ein Fehler einschleichen, der später wieder zu einer Menge Arbeit führt. Es sollten Tests für alle (un)gültigen und (un)erwarteten Eingabedaten definiert werden, wobei es natürlich schwierig ist, alle ungültigen Daten im Vorfeld zu erahnen.

5. Fazit

Das Thema „Testen“ ist in der Softwaretechnik mit einigen Vorbehalten behaftetet. Die Tester machen sich mit ihren Testdurchläufen (ein wenig übertrieben) unbeliebt, da sie ja immer von der Fehlerhaftigkeit eines Programms ausgehen sollten. Dies behagt dem Entwickler natürlich nicht besonders. Andererseits entsteht das Gefühl, der Tester verzögert die Implementierung. Dies ist auf den ersten Blick zwar richtig, aber eine nachträgliche Korrektur an einem schon fertigen, womöglich ausgelieferten Programm ist sehr kostenintensiv, von den Folgekosten und eventuellen Regressansprüchen einmal abgesehen. Ein Imageverlust durch solche grobe Fehler kann gar nicht beziffert werden (siehe Beispiel Toll Collect). Ein weiter menschlicher Aspekt ist, dass der Entwickler sich nicht sonderlich auf die Fehlerfreiheit konzentriert, da sowieso noch mal alles getestet wird und demnach auch der Tester schuld daran ist, wenn etwas Fehlerhaftes in Produktion geht. Das macht es dem Tester nicht leichter, er hat mehr Arbeit und es kommt zu Unzufriedenheiten. Es ist wichtig sich eines immer wieder zu verdeutlichen: Entwickler und Tester ziehen an einem Strang und sollten sich zuarbeiten und nicht gegeneinander.

Testen ist ein höchst kreativer Akt, der ein Höchstmass an Phantasie und Ideenreichtum erfordert, um auch den abwegigsten Testfall zu entwickeln. Aus diesem Grund braucht ein Tester eine gute Ausbildung, nicht nur im Sinne von Programmiertechniken, sondern auch im Bezug auf das Entwickeln von Tests und der Suche von Fehlern. Auch die Dokumentation dieser Tests ist wichtig und kann durch eine gute Schulung verbessert werden. Ein Test der nicht ordentlich dokumentiert ist bringt nicht viel. So kann z. B. nach einer Korrektur nicht 100% genau der gleiche Ablauf eingehalten werden und somit war der ganze Test sinnlos, da nicht garantiert werden kann, dass der Fehler jetzt nicht mehr auftreten kann. Zum Schluss kann noch eines gesagt werden: **Es gibt kein Programm ohne Fehler.**

6. Literatur

[Myers 1979] Glenford J Myers: The Art of Software Testing <dt.>
Methodisches Testen von Programmen, 1. Aufl., Oldenbourg, 1979

[Myers 2004] Glenford J Myers: The Art of Software Testing
Second Edition, Wiley, 2004

[Beizer 1995] Boris Beizer: Black-Box Testing:
Techniques for FunctionalTesting of Software and Systems, Wiley, 1995

[Pomberger / Blaschek 1996] Gustav Pomberger, Günther Blaschek:
Software Engineering, Prototyping und objektorientierte Software-Entwicklung,
2.Aufl., CarlHanser, 1996, S. 146-160

7. Quellen aus dem Internet

[http://www-aix.gsi.de/~giese/swr/ariane5.html]
[http://de.wikipedia.org/wiki/Ariane_5]

[http://www.bmvbw.de/LKW-Maut-.720.22466/.htm]
[http://de.wikipedia.org/wiki/Toll_Collect]
[http://de.wikipedia.org/wiki/OBU]
[http://www.bmvbw.de/LKW-Maut-.720.22466/.htm]
[http://www.heise.de/mobil/artikel/50923/0]

Quellen der Bilder:

[Start der Ariane 5]
[Explosion der Ariane 5]
[Toll Collect]