Referat

im Rahmen der Veranstaltung

Blockseminar TWA

an der Fachhochschule Giessen-Friedberg

Wintersemester 2004 / 2005

Themen:

Extreme Programming,
OO-Testmuster und
Automatisierung und Tools in der Praxis

Themensteller:	Prof. Dr. Wüst
Ausgabetermin:	09.02.2005
Abgabetermin:	02.03.2005
Vorgelegt von:	Boris Budweg
	Matr.: 678458

Abstract

This document gives a short overview on the three topics listed above. The section "Extreme Programming" has its focus on the testing part called "Extreme Testing", with a small example included. The second one, "OO - Testmuster", describes a few easy techniques on how to test classes and what to take into consideration when designing polymorph structures. "Automation und Tools in der Praxis" introduces some utilities (commercial as well as non-commercial) you can use for automation and testing, which crossed my way while I was searching through literature and google. After reading, you'll have a rough idea of current testing methods and I hope you'll feel like doing some structured tests on your own software creations.

Inhalt

0 Vorwort 3

1 Grundlagen des Testens 4
1.1 Wozu Testen? 4
1.2 Bug-Gruppen 7
1.3 Test-Gruppen 7

2 Extreme Programming 9
2.1 Basics 9
2.2 V-Modell 10
2.3 Extreme Unit Testing 12
2.4 Acceptance Testing 15
2.5 Error Guessing 16
2.6 Fazit / Ausblick 17

3 Objektorientierte Testmuster 18
3.1 Basics 18
3.2 modale Klassen 22
3.3 modale Hierarchie und Polymorphie 25
3.4 Fazit 26

4 Automation und Tools in der Praxis 27
4.1 Capture / Playback 27
4.2 Hardware- & Benutzersimulation 29
4.3 Test Monkey 29
4.4 Unit-Tests 30
4.5 Model Checker 33
4.6 GlowCode 35
4.6.1 Flaschenhals 35
4.6.2 Speicherleck 35
3.6.3 Coverage 36
4.7 Fazit / Ausblick 37

5 Zusammenfassung 38

6 Quellenverzeichnis 39

0 Vorwort

In der vorliegenden Ausarbeitung habe ich versucht, einerseits einen allgemeinen Überblick über die drei Themen zu geben, andererseits aber auch die große Lücke zwischen Theorie und Praxis zu schließen, die in der Fachliteratur häufig entsteht. Es war nicht immer einfach, zu einem theoretischen Muster ein praktisches Beispiel zu finden. Das lag einerseits an der Abstraktionsebene der Formulierungen, andererseits aber auch daran, daß für das Testen von Software noch nicht allzu viele Erfahrungsberichte, Foren und Bibliotheken verfügbar sind, weil das strukturierte Testen noch nicht sehr lange und längst noch nicht von allen Softwarehäusern so konsequent durchgezogen wird, wie es sich die Verbraucher manchmal wünschen.

Oftmals haben die Softwarehäuser auch Eigenentwicklungen im Einsatz, die ihnen, wenn sie eine hohe Erfolgsquote bescheren, einen nicht zu unterschätzenden Wettbewerbsvorteil gegenüber anderen verschaffen, und deshalb natürlich auch nicht publiziert werden ([Q 3]). U.a. betrifft das seltene Geschöpfe wie die Test Monkeys, die auch im großen www-Dschungel nicht so einfach aufzuspüren sind. Treten doch einmal Methoden, Erfahrungen und konkrete Tips mit Praxisbezug in der Literatur auf, dann meist, weil ein ausgeschiedener Mitarbeiter ein Buch schreibt, wie im Falle der Quelle 9, in der ein ehemaliger Microsoft-Programmierer aus dem Nähkästchen plaudert.

Eine weitere Schwierigkeit lag in meinem Bestreben, nicht allzuoft Redundanzen im Laufe der Vorträge des Blockseminars zu erzeugen. Trotzdem habe ich mich bemüht, in dieser Ausarbeitung Herleitungen und Voraussetzungen, sofern sie nötig oder interessant sind, vollständig zu erwähnen, auch wenn sie ggf. beim Vortrag ausgespart werden.

Sollten wichtige Grundlagen gänzlich unerwähnt bleiben, dann deshalb, weil ein vorangegangener Vortrag den Themenkomplex (hoffentlich) schon zur Genüge durchleuchtet hat, wie z.B. bei den einzelnen Black- und Whitebox-Verfahren (Vortrag über Modultests und Testplanung am Vortag).

1 Grundlagen des Testens

1.1 Wozu Testen?

Jedes Tool, welches zum Programmieren eingesetzt werden kann, hat seine Stärken und Schwächen, so natürlich auch unser wesentliches Werkzeug, die Programmiersprache selbst. Typstrenge Compiler verleihen uns oftmals zu unrecht Sicherheit über Korrektheit
Im Jahre 1962 wurde die Trägerrakete einer Venussonde ca. 6 Minuten nach dem Start kontrolliert zerstört, weil sie von der Flugbahn abwich. Der

```
DO 20 N0 = 1, 8
EPS = 5.0*10.0**(N0-7)
CaALL BESJ(X, 0, B0, EPS, IER)
IF (IER .EQ. 0) GOTO 10
20 CONTINUE
DO 5 K = 1. 3
T(K) = W0
Z = 1.0 / (X**2)*B1**2+3.0977E-4*B0**2
D(K) = 3.076E-2*2.0*(1.0/X*B0*b1+3.0977E-4**(B0**2-
X*B0*B1))/Z
E(K) = H**2*93.2943*W0/SIN(W0)*Z
H = D(K)-E(K)
5 CONTINUE
```

Dies ist der Codeausschnitt, der 18,5 Mio. $ kostete.
Obwohl davon ausgegangen werden kann, daß bei so teuren und bedeutenden Projekten Programmierer mit viel Erfahrung am Werk sind, hat sich hier ein (kleiner) Fehler eingeschlichen.
Die Zeile

```
DO 5 K = 1. 3
```

wird zwar von einem ForTran-Compiler übersetzt. Statt aber einer Schleife (5), die mit der Zählvariablen K von 1 bis 3 zählt, weist er der Variablen DO5K die Fließkommazahl 1.3 zu. Richtig wäre die Zeile so:

```
DO 5 K = 1, 3
```

Das US-Verteidigungsministerium reagierte, indem es die Programmiersprache ADA entwickelte, speziell auf sicherheitsrelevante Software spezialisiert. Heute ist bekannt, daß auch das nichts genutzt hat - wer erinnert sich noch an Ariane5 im Jahre 1996? Folgender Code kostete 5,5 Milliarden US-$:

```
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;
```

Was passierte hier? Die Software wurde von der Ariane4 übernommen, die eine Horizontalgeschwindigkeit von nur einem Fünftel der Ariane5 erreichte. Für die Ariane4 existierten Beweise, daß

```
honrizontal_veloc_sensor
```

nicht über den Integerbereich hinauswachsen kann, weshalb mit

```
declare pragma suppress(numeric_error,
                        horizontal_veloc_bias);
```

die Laufzeitüberprüfung zugunsten der Geschwindigkeitssteigerung für `horizontal_veloc_bias` abgeschaltet wurde. Aus Kostengründen wurde auf eine Simulation verzichtet, die eine weitere Millionen gekostet hätte. Die Horizontalgeschwindigkeit lag in einer Flughöhe von 3700 Metern bereits bei 32768.0 internen Zählern des Sensors, was zu einem Überlauf führte. Die Rechner auf der Bodenstation erhielten negative Werte, was zu unsinnigen Steuerungsbefehlen an die Rakete führte. Als diese zu zerbrechen droht, sprengt sie sich selbst.

[Q 8]

Offensichtlich ist niemand vor Bugs sicher.
Im Folgenden wird daher ein Blick auf die verschiedenen Arten von Bugs und Tests geworfen, um einen Überblick zu geben, wo sinnvolle Ansatzpunkte für unsere Testes liegen.

1.2 Bug-Gruppen

Es wird unterschieden zwischen drei Arten von Bugs (benannt nach dem Zeitraum während der Entwicklung, in dem sie meistens auftauchen):

Unit/Component Bugs:
Einfachste Fehler; leicht zu finden und zu beheben. Jede einzelne Komponente (z.B. Klasse / Modul) wird getestet.
Auf diese Buggruppe sollte möglichst früh getestet werden, damit Component Bugs nicht erst in einer späteren Entwicklungsphase bemerkt werden.

Integration Bugs:
Diese Art von Bugs treten zutage, wenn Komponenten zusammenarbeiten. Schwierig zu finden, weil die Interaktion zwischen einzelnen Komponenten sehr vielseitig ausfallen kann, und die Anzahl der Kombinationsmöglichkeiten quadratisch (schlimmstenfalls fakultativ) wächst.

System Bugs:
Wird sich die (deterministische) Software in der (nichtdeterministischen) realen Welt (mit Multitasking, verschiedenen Hardware-Plattformen, DAUs, ...) erwartungsgemäß verhalten?
Solche Bugs werden meistens erst in den ersten Releases entdeckt.

[Q 3]

1.3 Test-Gruppen

Außerdem werden zwei Arten von Tests unterschieden:

Whitebox (strukturierter Test):
Mit vollem Einblick in den Code. Jede Funktion/Methode/... einzeln testen (mit allen Fallunterscheidungen); jede Codezeile muß mindestens einmal durchlaufen werden.

Wird tendenziell eher auf niedrigerem Entwicklungslevel eingesetzt (-> Unit & Integration).

<u>Blackbox (Verhaltenstest / funktionaler Test):</u>
Ohne Einblick in den Code. Alle Features der Spezifikation und Anwendungsfälle (Use Cases) testen. Wird tendenziell eher auf höherem Entwicklungslevel eingesetzt (-> Integration & System).

Sinnvollerweise werden beide Techniken in einem Projekt verwendet, meist keiner zu einem kleineren Anteil als 1/3.
Hat eine Anwendung mehr "hart" codierte ad-hoc-Logik, darfs eher etwas mehr Whitebox sein. Besteht sie dagegen zu einem größeren Anteil aus generischen Algorithmen, deren Laufzeitverhalten stark von Eingabewerten / Datensätzen abhängen, eher etwas mehr Blackbox.

[Q 3]

Beim Testen ist ebenso zu beachten, *wer* testet. Unit Tests machen am besten Programmierer (meist sogar die, die auch schon den Code geschrieben haben) - sie kennen sich sowohl in der Programmiersprache als auch in der Aufgabenstellung aus, kennen die Datentypen, und wissen, wo die Fehler entstehen können (-> Boundary Values etc.).
Systemtests sind sicherer in der Hand der Benutzer aufgehoben (zumindest sollten die Testfälle ins Zusammenarbeit mit ihnen entworfen werden), weil sie die Software unter realistischen Bedingungen einsetzen werden, und die Standardabläufe des Betriebes kennen. Das Programm soll schließlich später von "normalen" Benutzern ohne Programmiererfahrung nutzbar sein, und diese haben meist eine völlig andere Sichtweise als die Programmierer.

[Q 4]

2 Extreme Programming

In den letzten Jahrzehnten hat sich der Konkurrenzdruck der Softwareindustrie vergrößert, was dazu führte, daß Software immer schneller auf den Markt gebracht werden muß. Die Entwicklungszeit schrumpft, gleichzeitig veraltet die Software schnell - sei es durch neue Betriebssysteme, neue Hardware oder neue Anforderungen, die sich im ebenfalls immer schneller entwickelnden und immer mehr automatisierenden Produktionsprozessen ergeben. Die der alten Modelle der Softwareentwicklung sind inzwischen zu schwerfällig geworden, um sie sinnvoll einzusetzen (falls sie je benutzt wurden).
Softwarepleiten (zu lange Entwicklungszeiten, nicht eingehaltene Budgets, Veraltung am Tag der Einführung, zu viele Bugs oder nicht eingehaltene Anforderungsprofile - und damit Unbrauchbarkeit) führten zu einer neuen Methode, die in die heutige Zeit paßt - dem "Extreme Programming".

Es handelt sich beim Extreme Programming nicht um eine gänzlich neue Technik, oder völlig neues Konzept, sondern vielmehr um eine Verknüpfung der bewährten Techniken mit den Erfahrungswerten aus Softwarefabriken.

[Q 4]

2.1 Basics

Die Hauptkonzepte des Extreme Programming lauten (zusammengefaßt):

1. Frühes Kunden-Feedback
2. Zusammenarbeit mit dem Kunden für Entwicklung der Spezifikationen und Testfälle
3. Programmierung mit Partner (an einem PC)
4. Code-Basis testen

Deren Ziele sind:

1. nicht an den Anforderungen "vorbei" programmieren - der Kunde soll möglichst schnell etwas "greifbares" (und sei es ohne ausprogrammierte Funktionen) in der Hand halten und bewerten (vgl. Rapid Prototyping / Evolutionäres Modell)
2. schnelle Erkennung von Fehlplanungen
3. "Echtzeit-Fehlerüberwachung" des Programmierers (durch Partner)
4. Testfälle der Module ("Units") nicht verwerfen, sondern kontinuierlich weiterverwenden: vermeiden, daß durch eine Änderung im Code eines Moduls sich ein anderes nicht mehr erwartungsgemäß verhält

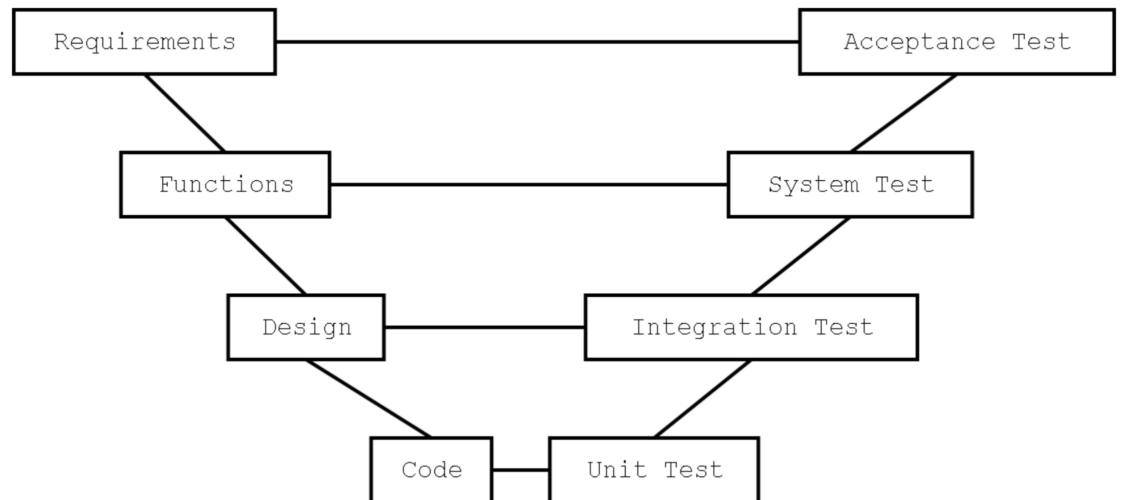
Außerdem folgende Ergänzungen zum Ablauf und Informations-Management:

1. Allen Programmierern gehört der Code. Keiner ist auf bestimmte Abschnitte begrenzt
2. Programmierer erhalten (mindestens) einen Ansprechpartner des Kunden, um unnötig lange Verzögerungen bei Rückfragen zu vermeiden
3. Keine Überstunden (abgesehen von der letzten Woche vor Release)

[Q 1]

2.2 Das V-Modell

Beim Extreme Programming wird nach dem V-Modell gearbeitet:



Die Arbeit beginnt in "herkömlicher" Manie links oben, läuft nach unten in die Mitte, dann hoch in die rechte Ecke. Jeder (Coding-) Punkt auf der linken Seite hat einen äquivalenten Test auf der rechten Seite. Beim Extreme Programming müssen die gegenüberliegenden Tests parallel oder teilweise sogar vorgezogen ablaufen.

Am Anfang werden die Anforderungen Analysiert - dem gegenüber steht der Akzeptanztest: Sind die Anforderungen praxisnah?
Wenn der Funktionsumfang bestimmt ist - funktioniert das ganze als System?
Wenn das Design entworfen wurde - lassen sich die Module integrieren?
Während der Code ausgearbeitet wird - funktionieren die einzelnen Module?

Beim Extreme Programming liegt der Fokus auf dem frühen Testen des Codes (Unit Testing), und dem frühen Feedback des Kunden (Acceptance Testing).

2.3 Extreme Unit Testing

Unit Testing ist das effizienteste Testlevel. Die Vergangenheit hat gezeigt, daß ein Fehler, der im Unit Test gefunden wurde nur ein Zehntel von demjenigen Fehler kostet, der erst nach dem Releasen gefunden wird. Deshalb ist es sinnvoll, diese Bugs so früh wie möglich zu finden.

[Q 2]

Genau aus diesem Grund liegt beim Extreme Testing der Schwerpunkt auf den Unit Tests. Diese werden allerdings ***vor*** der eigentlichen Implementierung geschrieben. Diese Vorgehensweise nennt sich "**Test Driven Development**" (Test-getriebene Entwicklung) und birgt folgende Vorteile:

1. es ist offensichtlich, ob der Code die Spezifikation erfüllt (wenn alle Tests positiv ausfallen)
2. das Endergebnis wird formuliert bevor programmiert wird - dadurch besseres Verständnis der Anforderungen
3. es kann erst eine schnelle, "schmutzige" Lösung implementiert werden, und diese dann zu einem späteren Zeitpunkt optimieren, ohne Sorge, durch Änderungen die Spezifikation zu verletzen (vorausgesetzt, daß die Blackbox-Testfälle erhalten bleiben!)

Der Programmierer wird praktisch gezwungen, erst die Spezifikation zu verstehen, bevor er programmiert.
Diese Vorgehensweise ist etwas gewöhnungsbedürftig - in den ersten Versuchen fällt es schwer, vorrangig an die Tests zu denken. Immer wieder fallen dabei Einzelheiten zum Code auf, die allerdings erst in den Hintergrund treten müssen, bis die Tests fertig verfaßt sind. Aber genau diese Details sind es, mit deren Erkenntnis später von Beginn an besserer Code verfaßt wird.
Es folgt ein Beispiel.

Die einfachste Möglichkeit, Tests einzusetzen bietet die Standard Library mit dem Befehl `assert()`. Er testet, ob der übergebene Ausdruck wahr ist, und gibt eine Fehlermeldung mit Zeilennummer und dem fehlgeschlagenen Ausdruck zurück. Das ist, gemessen an verfügbaren Tool Kits, ein eher magerer Komfort, reicht aber absolut aus, um die Methode zu demonstrieren.

Angenommen, es soll eine Funktion `int kgv(int a, int b)` für das kleinste gemeinsame Vielfache von zwei ganzen positiven Zahlen geschrieben werden. Gibt es keinen Kgv, oder ist die Eingabe falsch, soll 0 zurückgegeben werden.

Zuallererst muß eine Analyseart ausgewählt werden. Whitebox-Tests stehen außer Frage, da für sie der Code benötigt wird, der noch gar nicht existiert. Mit Whitebox-Tests können allerdings später die **Blackbox**-Tests ergänzt werden.
Die Größe des Projekts impliziert die **Boundary Value Analysis**, um Testfälle für die Randbereiche zu erhalten:

einen Kgv von 0 und x gibt es nicht: `assert(kgv(0,5)==0);`
der Kgv von 1 und x ist immer x: `assert(kgv(1,6)==6);`
der Kgv zweier gleicher Zahlen: `assert(kgv(2,2)==2);`

nach oben ist die Begrenzung nur durch die Hardware gegeben (32 Bit):
`assert(kgv(2000000000,3)==0);`

negative Zahlen sind nicht erlaubt: `assert(kgv(-1,2)==0);`
kgv zweier Primzahlen ist deren Produkt: `assert(kgv(3,7)==21);`
ein exemplarischer "Normalfall": `assert(kgv(4,6)==12);`

Jetzt wird "schnell und schmutzig" eine Funktion entworfen, die gerade eben unsere Spezifikation erfüllt:

```
int kgv(int a, int b){
     if (a<1 || b<1) return 0;     // nur positive zahlen
```

```
    for (int i=(a>b?a:b); i>0; i++)
        if (i%a==0 && i%b==0) return i;
    return 0;           // kein kgv im int-zahlenraum
}
```

Der Algorithmus - so ineffizient er auch ist - hält sich mit Sicherheit an die Spezifikationen. Und der Programmierer hat die Gewißheit, daß er nichts grobes vergessen hat, denn die Testergebnisse stimmen mit den vorher bereits geklärten Erwartungen überein.
Optimierungen können später gefahrlos durchgeführt werden, wenn die Testfälle beibehalten, und nach Code-Änderungen wiederholt werden (-> "Regressionstest", kann auch automatisiert werden, s. Kap. Automatisierte Unit Tests).

Jetzt wird es Zeit für einen **Whitebox**-Test. Im aktuellen Fall bietet sich **Decision Coverage** an:

Es gibt vier Stellen, an denen eine Entscheidung getroffen wird, und aus denen Testfälle generiert werden:

	Entscheidung	true	false
1.	`(a<1 || b<1):`	`kgv(0,1)`	`kgv(1,1)`
2.	`(a>b):`	`kgv(2,1)`	`kgv(1,1)`

die Bedingung, durch welche die Schleife verlassen wird, wenn ein Ergebnis gefunden wurde, ist hier unerheblich, da dies auf jeden Fall von einem vorherigen Testfall abgedeckt wird (die Funktion hat schließlich bereits einmal ein Ergebnis zurückgeliefert).

Interessant wird es noch einmal bei der Abbruchbedingung im Schleifenkopf:

	Entscheidung	true	false
3.	`i>0:`	`kgv(2000000000,5)`	`kgv(1,1)`

Also bleiben unter dem Strich die folgenden Test Cases für die Whitebox:

```
assert(kgv(0,1)==0);
assert(kgv(2,1)==2);
assert(kgv(1,1)==1);
assert(kgv(2000000000,5)==0);
```

In diesem Beispiel überlappen sich die Test Cases von Black- und Whitebox. Das muß aber nicht immer der Fall sein, und schließlich handelt es sich hier um ein sehr simples Beispiel, und nur um zwei von vielen möglichen Methoden.
Wer sich für eine komplexere Form des Whitebox-Testens entscheidet, ist gut beraten, mit einem Coverage Tool zu ermitteln, ob er wirklich alle Alternativen einer Kondition mindestens einmal im Test durchlaufen hat (s. Kap. Tools).

Wichtig ist in diesem Zusammenhang auch, ob die Prozeduren (Module) so bleiben, wie sie aktuell sind, oder ob sie (vielleicht sogar im Algorithmus) noch verändert werden. Der große Aufwand den die komplexeren Whitebox-Tests mit sich bringen, lohnt sich nur, wenn gewährleistet werden kann, daß die Tests weiterverwendet werden können - sonst muß, falls der Algorithmus eines schönen Tages in

```
m * n / ggT( m, n);
```

geändert wird, das gesamte Whitebox Test Case Design wiederholt werden. Die Blackbox-Tests bleiben hingegen in jedem Fall erhalten.

2.4 Acceptance Testing

Ein Programm kann alle Unit Tests bestehen und trotzdem ohne weiteres durch den Akzeptanz-Test fallen. Das liegt daran, daß die Unit Tests alleine auf der Spezifikation aufbauen. Sie testen also nur, ob der Code die Spezifikation erfüllt, nicht, ob das Programm einer bestimmten Funktionalität oder einem bestimmten ästhetischen Anspruch genügt.

Die Tests, die durchzuführen sind, können aus den Use Cases abgeleitet werden, wobei ein Use Case mehr als einen Test Case erfordern kann. Werden bei den ausgeführten Tests Fehler gefunden, so müssen die Fehler priorisiert werden, bevor die Liste in die Entwicklung zurück gereicht wird. Der Akzeptanz-Test wird an allen bedeutenden Releases durchgeführt, und wiederholt, wenn die gefundenen Bugs behoben worden sind.

[Q 5,7]

2.5 Error Guessing

Error Guessing ist keine Methode, um Fehler aufzuspüren, es ist mehr eine Art Glücksspiel, gepaart mit Erfahrung. Manche Menschen haben eine natürliche Begabung, Fehler aufzuspüren.

Da Fehler meistens Gruppentiere sind, und keine Einzelgänger, kann zum Beispiel die Statistik helfen.

Mögliche Anhaltspunkte:

1. wo wurden bisher die meisten Fehler gefunden?
1. welche Quellen haben die höchsten Versionsnummern (wurden am häufigsten verändert)?
2. welche Klassen haben überdurchschnittlich viele Methoden?
3. welche Methoden sind überdurchschnittlich lang?
4. wo finden sich überdurchschnittlich viele Kommentare?

So etwas läßt sich leider nicht automatisieren, trotzdem sollte nicht völlig darauf verzichtet werden, denn automatische Tests können nur testen, wofür sie geschrieben wurden, der "Guesser" hingegen ist sehr flexibel.

[Q 8]

2.6 Fazit / Ausblick

Das Konzept ist sicherlich eine gute Entwicklung, aber es ist nicht für jedes Projekt geeignet. Es Funktioniert jedoch in kleinen bis mittelgroßen Entwicklungen, in Umgebungen, die regelmäßige Änderungen der Spezifikation erfährt, wenn direkte Kommunikation möglich ist.

[Q 1, Q 4]

In größeren Projekten steigt die Anzahl der Kommunikationspfade derart, daß deren Koordination in der Praxis einen zu großen organisatorischen Aufwand bedeutet. Der kollektive Besitz der Software ist nicht mehr durchzusetzen, und die Aufteilung der Arbeit auf verschiedene Teams widerspricht ebenfalls dem Konzept von Extreme Programming. Team-Aufteilung und Koordination sind die Knackpunkte an denen Extreme Programming in der heutigen Praxis in großen Projekten (>16 Programmierer) scheitert.

[Q 5]

XP ist trotz allem ein interessanter und moderner Ansatz, weil er immer die Validität des Systems sicherstellt, die Konzentration des Teams auf das Wesentliche lenkt, und die soziale Interaktion der Beteiligten soweit fördert, daß deren volles Potential ausgeschöpft werden kann. Aufgrund der genannten Vorteile und seiner Agilität und Praxisnähe besitzt XP gute Chancen, zukünftig besonders in klein- und mittelständischen Unternehmen verstärkt eingesetzt zu werden.

[Q 7]

3 OO - Testmuster

Die Objektorientierung bietet unbestreitbare Vorteile - hat allerdings auch ihre Tücken, die besonders beim Testen zu beachten sind. Die Methoden, die im folgenden dargestellt werden, lassen sich nicht fest den Blackbox- oder den Whitebox-Tests zuordnen. Es wird ein wenig Einblick in den Code benötigt, zielt aber doch eher auf die Konsistenz bei Vererbung und die Schnittstellen der Klassen, so daß sie in mancher Literatur auch "Greybox Tests" genannt werden.

3.1 Basics

Zuerst zwei Prinzipien, die es bei der Objektorientierung (strikte Vererbung) einzuhalten gilt:

Substitutionsprinzip (B. Liskov):
"Exemplare einer Unterklasse müssen unter allen Umständen an Stelle von Exemplaren jeder Oberklasse einsetzbar sein."

[Q 9]

Verantwortlichkeitsprinzip (B. Oestereich):
"Eine Klasse kann keine Zusicherungen auf Eigenschaften ihrer Oberklasse erzwingen."

[Q 10]

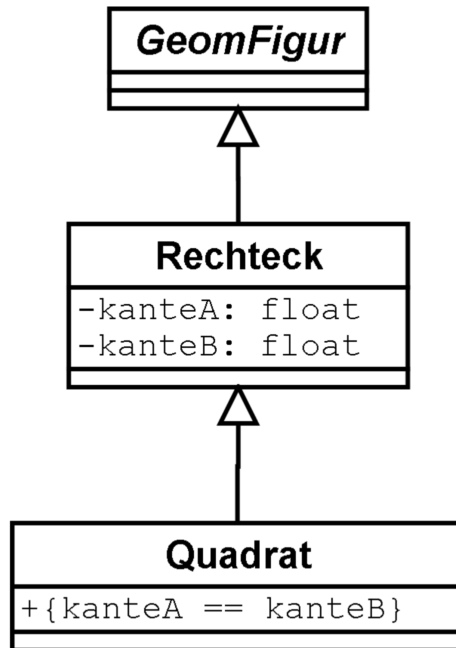
Beispiel zur Erläuterung der Vererbungsproblematik:
Jeder kennt das Standard-Beispiel für die Vererbung:

Wie werden

Geometrische Figur, Rechteck, Quadrat

implementiert?

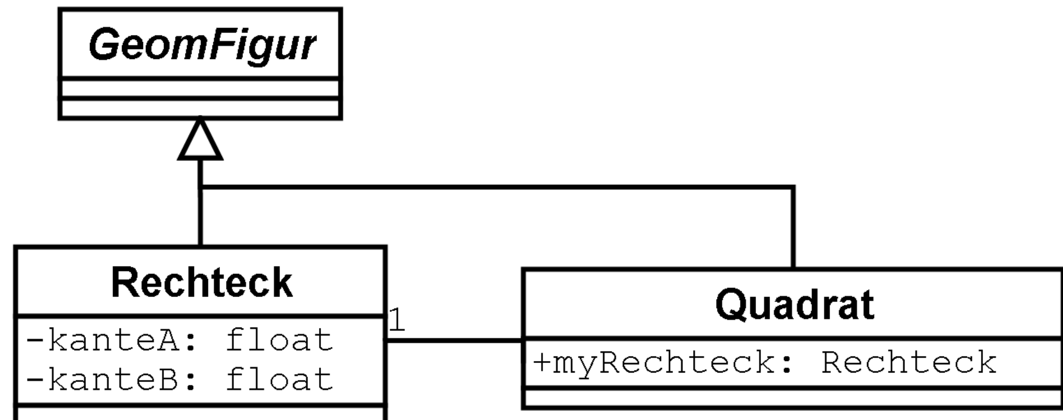
Der erste Ansatz ist sicherlich die Klassenhierarchie so aufzubauen:



Ein Rechteck ist eine Geometrische Figur, ein Quadrat ist ein spezielles Rechteck, mit der Zusatzbedingung, daß beide Kanten gleich lang sind.

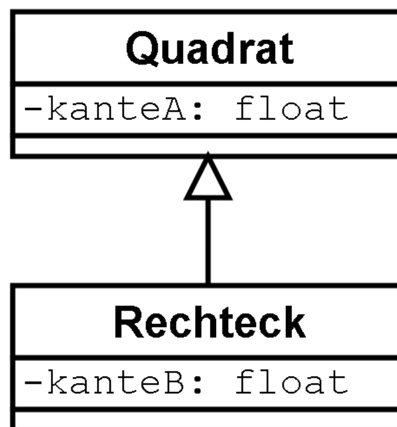
Das Problem hierbei: Verletzung des Verantwortlichkeitsprinzips - In Quadrat dürfen keine Zusicherungen über die Attribute des Rechtecks gemacht werden.

Ein weiterer Ansatz ist die Delegation - beide geometrischen Objekte sind auf gleicher Höhe von GeomFigur abgeleitet, das Quadrat hat eine Instanz von Rechteck:



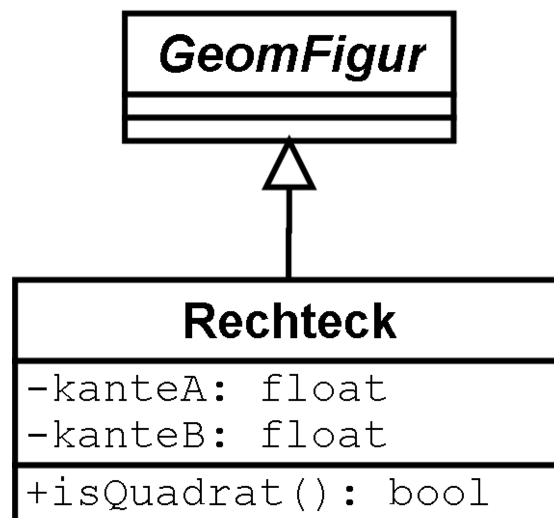
Das hat den Nachteil, daß alle Methoden neu geschrieben werden müssen für das Quadrat.

Findige Menschen könnten noch auf diese Alternative kommen:



Das Problem der Kanten ist damit elegant gelöst, doch jetzt stimmt die Hierarchie nicht mehr - ein Rechteck ist eben keine spezielle Ausprägung eines Quadrates. Es handelt sich hierbei um eine technisch motivierte Vererbung
Außerdem verstößt diese Lösung gegen das Substitutionsprinzip.

Eine weitere Möglichkeit wäre noch, das Rechteck um Quadrat-Funktionen zu erweitern:



Jetzt kann eine Instanz von Rechteck in Eigenregie herausfinden, ob sie ein Quadrat ist.

Doch "schön" ist auch diese Variante nicht gerade.

Es ist also gar nicht so einfach, eine logische, technisch einwandfreie, und optisch ansprechende Hierarchie zu implementieren, ohne dabei eines der beiden Prinzipien zu verletzen.

[Q 8]

Typische Fehler, die nur im Zusammenhang mit der Vererbung auftreten sind z.B.:

Fehlende Überschreibungen: Operatoren werden nicht oder nur unvollständig überschrieben (beliebter Fehler in C++)

Direkter Zugriff auf Public- oder Protected-Attribute: Seiteneffekte führen zu Fehlern in der Oberklasse, oder unbeabsichtigtes Verdecken eines Attributes der Basisklasse

"unartige Kinder": z.B. eine Unterklasse, die nicht alle Nachrichten der Oberklasse akzeptiert oder eine Unterklasse, die einen Status hinterläßt, der für die Oberklasse nicht erlaubt ist. Verletzung des Substitutionsprinzips, meist aus technisch motivierten Vererbungen

"Wurmlöcher": Unterklasse berechnet Ergebnisse, die nicht konform zu den Zusicherungen und Zustandsinvarianten der Oberklasse sind (Verletzung des Verantwortlichkeitsprinzips)

Verantwortlichkeitsverschiebungen: z.B. Unterklasse akzeptiert in einer überschriebenen Methode nur gerade Zahlen, die Oberklasse jedoch alle ganzen Zahlen

"Fat Interface": Unterklasse erbt Methoden, die unpassend oder irrelevant sind

[Q 8]

Zu dem bereits bekannten Modultest gesellen sich im Vererbungskontext zwei weitere Formen der Tests, mit welchen versucht wird, die o.a. Fehler zu finden:

1. Klassentest:
   Jede Methode wird lokal für sich getestet, und jede Folge abhängiger Methoden innerhalb der Klasse wird getestet. Für einen Test wird ein Objekt der Klasse in einen gültigen Zustand versetzt, eine Methode aufgerufen, und anschließend die Gültigkeit des Zustandes nach dem Aufruf geprüft.

   Beispiel: Test einer Datumrechner-Klasse:

   ```
   void testSchaltjahr(){
   ```

```
        int day, month, year;
        assert(true==aDateInterval->reset(29,2,2000));
        aDateInterval->nextDay(day, month, year);
        assert (1 == day);
        assert (3 == month);
        assert (2000 == year);
    }
```

2. Kettentest:
   Lokaler Klassen-Integrationstest. Test der Auswirkungen der ererbten Attribute und Methoden im neuen Verwendungskontext. Auch hier dürfen nur gültige Zustände hinterlassen werden.

Der Modultest mit mehreren, zusammengefaßten Klassen und deren Interaktion untereinander ist natürlich weiterhin elementarer Bestandteil unserer Tests, und sollte nicht vernachlässigt werden.

3.2 Modale Klasse

Eine "Modale Klasse" ist eine Klasse mit gewissen Einschränkungen bezüglich der zugelassenen Reihenfolgen von Methodenaufrufen.

Beispiele:

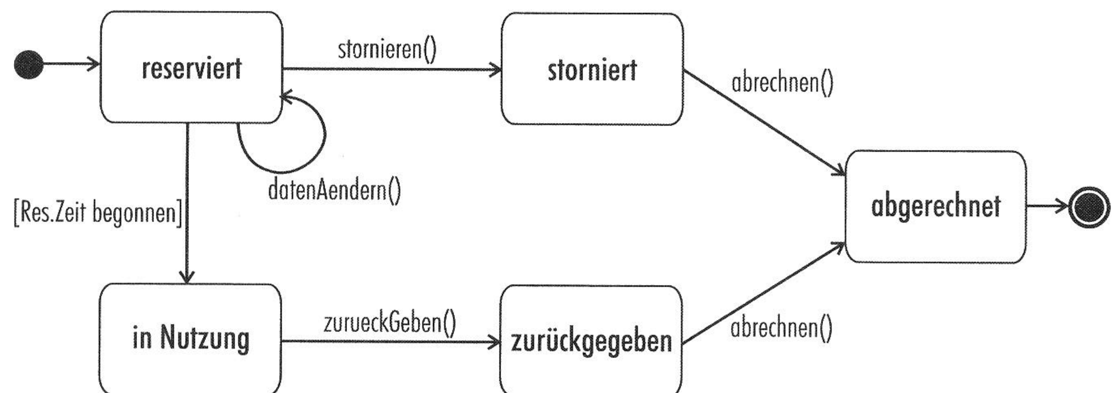
Ampel-Klasse	(**sequenzabhängig**, immer rot-rotgelb-grün-gelb)
Stack-Klasse	(**inhaltsabhängig**, push() nur zugelassen, wenn Stack nicht voll / pop() nur zugelassen, wenn Elemente auf Stack)
Konto-Klasse	(**fachgebundene Abhängigkeit**, buchen() nur, wenn nicht gesperrt)

Das Testmuster für die Modale Klasse sieht folgende Prüfungen vor:

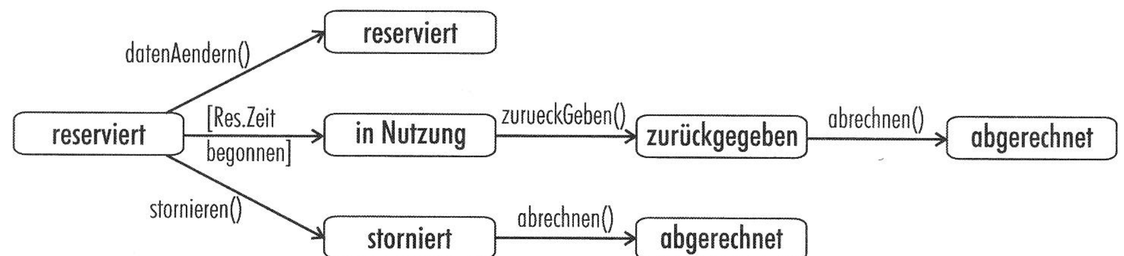
1. alle gültigen Methodenaufrufe müssen akzeptiert werden
2. alle ungültigen Methodenaufrufe müssen abgelehnt werden
3. die Klasse muß sich nach beiden Aufrufen in einem gültigen Zustand befinden

4. die Ergebnisse der Methodenaufrufe müssen korrekt sein

Doch wie ist konkret vorzugehen?
Ein **Zustandsautomat** der Konto-Klasse bildet den Anfang:



Dieser Zustandsautomat ist zwar leicht zu überblicken, aus der Testsicht jedoch ist es schwierig, mit ihm zu arbeiten.
Er wird daher in einen **Zustandsbaum** überführt. Es wird jede Folge von Transaktionen so lange ausmultipliziert, bis ein bereits besuchter Zustand oder der Endzustand erreicht wird:



[Q 8]

Die Tests können nun leicht von der Wurzel bis ins Blatt abgelesen werden:

```
void testGueltigeUebergaenge(){
     Reservierung a, b, c;
     assert ( a.getZustand() == ZUST_RESERVIERT );

     r.datenAendern();
     assert ( a.getZustand() == ZUST_RESERVIERT );
```

```
        b.setDate(date());
        b.setTime(time()-1);
        assert ( b.getZustand() == ZUST_INNUTZUNG );

        b.zurueckgeben();
        assert ( b.getZustand() == ZUST_ZURUECKGEGEBEN );

        b.abrechnen();
        assert ( b.getZustand() == ZUST_ABGERECHNET );

        c.stornieren();
        assert ( c.getZustand() == ZUST_STORNIERT );

        c.abrechnen();
        assert ( c.getZustand() == ZUST_ABGERECHNET );
}
```

Dies reicht bereits um alle gültigen Zustände und Zustandsübergänge abzudecken. Doch wie sieht es mit den ungültigen aus? Die ungültigen könne am leichtesten von einer Wahrheitstabelle abgelesen werden, also muß das Zustandsmodell in eine Tabelle überführt werden.

	reserviert	in Nutzung	zurück-gegeben	storniert	abgerechnet
reserviert	•	•	×	•	×
in Nutzung	×	×	•	×	×
zurück-gegeben	×	×	×	×	•
storniert	×	×	×	×	•
abgerechnet	×	×	×	×	×

[Q 8]

Alle mit X gekennzeichneten Übergänge sind nicht erlaubt und müssen getestet werden:

```
void testUngueltigeUebergaenge(){
     Reservierung r;
     assert (r.getZust() == ZUST_RESERVIERT);

     assert (false == r.zurueckGeben());
     assert (r.getZust() == ZUST_RESERVIERT);

     assert (false == r.abrechnen());
     assert (r.getZust() == ZUST_RESERVIERT);

     r.setZust(ZUST_STORNIERT);
     assert (r.getZust() == ZUST_STORNIERT);

     assert (false == r.stornieren());
     assert (r.getZust() == ZUST_STORNIERT);

     assert (false == r.zurueckGeben());
     assert (r.getZust() == ZUST_STORNIERT);

     (...)
}
```

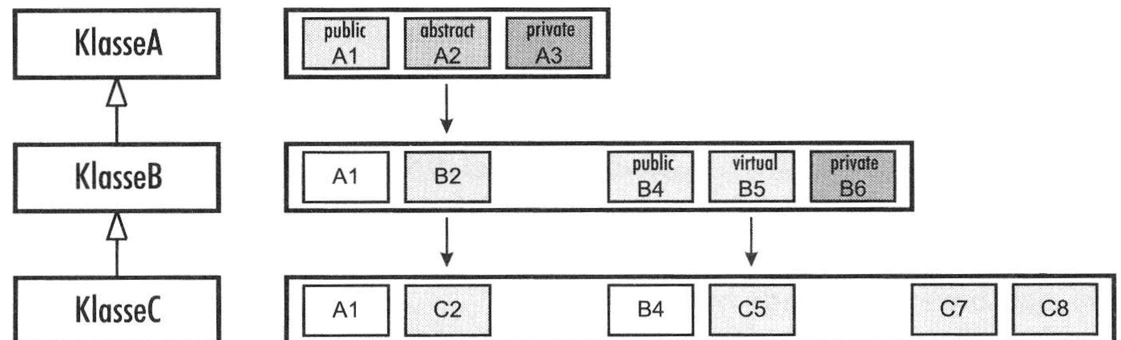
3.3 Modale Hierarchie und Polymorphie

Bei der Modalen Hierarchie wird modales Verhalten über eine Hierarchie von abstrakten und konkreten Klassen implementiert. Dadurch ergeben sich ein paar besondere Situationen, die separater Tests bedürfen.

Konkretes Vorgehen im Test:

1. "Modale Klassen"-Muster auf Oberklasse anwenden.
2. die so gewonnenen Tests werden auch auf die Unterklassen angewendet (zur Überprüfung des Substitutionsprinzips)
3. Status- und Ereignistests für Unterklassen

Für Punkt 3, der auch für die Polymorphie gilt, ist das "Flattening" gut geeignet, um sich eine Übersicht der noch zu Testenden Methoden zu verschaffen:



Die Klassen werden mit ihren Vererbern und allen Methoden Schematisch aufgezeichnet, um zu sehen, welche Methoden in welcher Klasse überschrieben werden.
Abstrakte Methoden müssen in jeder Ausprägung erneut getestet werden, virtuelle dann, wenn sie überschrieben werden (Hier: B2, C2, C5) Dabei wird nach den bereits besprochenen Techniken vorgegangen.

[Q 8]

3.4 Fazit

Objektorientierung ist spätestens mit Polymorphie eine Fehlerträchtige Angelegenheit. Hier gilt es, mit besonderer Sorgfalt schon beim Entwurf der Hierarchien vorzugehen. Die hier dargelegten Techniken können zwar bei der Fehlersuche helfen, sie ersparen aber nicht die sinnvolle Planung der Klassen.

4 Automation/Tools in der Praxis

Da einmal geschriebene Tests wiederverwendet werden müssen, bietet es sich an, die Testläufe soweit es geht zu automatisieren. Diese Kapitel befaßt sich mit der Automatisierung von Tests, sowie der Unterstützung des Testens durch Tools.

4.1 Capture / Playback

Die einfachste Form der Test-Automatisierung (z.B. für GUI-Tests) sind Capture/Playback-Tools. Sie zeichnen für eine definierte Zeitspanne die Aktionen des Benutzers (Maus, Tastatur) auf, und sind in der Lage, die Aktionen in exakt gleicher Reihenfolge zu wiederholen. Die so entstandenen Scripts müssen allerdings in aller Regel später überarbeitet/debugged werden. Ändert sich etwas im Fenster der Zielapplikation, muß oft der gesamte Prozeß des Aufzeichnens und der Überarbeitung von vorne beginnen.

[Q 6]

Zu Demonstrationszwecken wird hier "AutoIt" verwendet - ein relativ ausgereiftes Tool, welches sogar in der Lage ist sog. "Bots" für Spiele wie Diabolo o.ä. zu generieren, mit dem die Spielfigur automatisch Gegenstände aus einem Level suchen und sammeln kann.
Mit Hilfe eines "Revealers" ("Enthüller") können die Koordinaten für die Programmierung der Scripts relativ und absolut abgelesen werden:

Beispielausschnitt aus einem sehr einfachen Script, welches Notepad öffnet und einen kurzen Text ausgibt:

```
Run, notepad.exe
WinWait, %NotepadTitle%
WinActivate, %NotepadTitle%
WinMove, %NotepadTitle%,, 0, 0, default, default
Send, Hello,{SPACE}
Sleep, 500
Send, and welcome to
SetKeyDelay, 100
Send, ...........
SetKeyDelay, 20
Send, A u t o I t
SetKeyDelay, 100
Send, ...........
Sleep, 500
Send, {ENTER 2}
```

Aus den Scripts können dann lauffähige Programme erstellt werden.

4.2 Hardware- & Benutzer-Simulation

Es gibt Projekte wie "FAUmachine" und "Expect", mit denen sich nicht nur einzelne Eingaben, sondern gleich ganze Rechner (ähnlich wie z.B. VMware), allerdings mit speziellen Hardwareausprägungen, Benutzern und Administratoren simulieren. Expect nutzt die Skriptsprache VHDL.
Mit FAUmachine ist es sogar möglich, Hardware-Ausfälle bzw. -Fehler bei Netzwerkkarte, Festplatten oder sogar Bitfehler in Arbeitsspeicher oder Registern zu simulieren.
Bisher es nur unter Linux, und simuliert einen PC, auf dem Linux läuft. Die Simulation der Hardware ist derart genau, daß die eingebauten Linux-Treiber verwendet werden, um die Hardware anzusteuern.
Die simulierten Benutzer können hier, im Gegensatz zu den einfacheren Playback-Skripts, wie die echten Benutzer auf dem Bildschirm nach Symbolen, Knöpfen o.ä. suchen, denn es ist eine Mustererkennung integriert.

[Q 11]

4.3 Test Monkey

Eine weitere Methode zum Testen von GUIs ist die Verwendung eines Test Monkeys - eine Software, die zufällig Knöpfe drückt und Eingaben macht, ohne die geringste Ahnung, was dabei herauskommt. Das ist auch schon das entscheidende Defizit bei Test Monkeys: Selbst wenn sie Fehler produzieren, werden sie es nicht bemerken, sondern blind und taub weiterklicken. Um dann die Fehlersituationen rekonstruieren zu können, lassen manche Entwickler eine Kamera mitlaufen, dann kann später zurückgespult werden, und die Eingaben wiederholt werden, die zu einem Fehler führten.
Test Monkeys haben aber auch Vorteile: Sie kosten nichts, denn sie laufen völlig autonom. Speicherlecks z.B. lassen sich mit ihnen gut finden. Und sie setzen die Applikation unter Dauer-Streß. Wenn sie den Test die ganze Nacht ohne Absturz durchhält, weiß der Programmierer zumindest, daß sein Werk robust ist.

[Q 6]

Tips zur Entwicklung von Monkeys finden sich z.B. in [Q 6].

4.4 Unit Tests

Eine völlig andere Form der Automatisierung bieten z.B. die xUnit-Systeme. Obwohl der Ursprung in JUnit für Java liegt, gibt es inzwischen Implementierungen für die verschiedensten Sprachen:

CppUnit und Unit++ für C++

NUnit für .NET

SQLUnit für SQL

SUnit für Smalltalk

RubyUnit für Ruby

Es unterstützt den Programmierer mit mehreren Klassen, die Testfälle zu verwalten und komfortabeler auszuwerten. Aber xUnit ist mehr als eine Erweiterung des assert()-Befehls der Standard-Bibliothek.
Es kann z.B. auch Exceptions testen, oder die Ausführung einzelner Tests über einer bestimmten Zeitspanne begrenzen.

Die Ausgaben können auf verschiedenste Weise erfolgen; in CppUnit beispielsweise:

Text

Graphisch über MFC

Graphisch über Qt

compilerfreundliches Format (um Sprünge per Mausklick in IDE zu ermöglichen)

XML

Die xUnit-Systeme sind inzwischen so populär geworden, daß sie in einige Entwicklungsumgebungen bereits integriert wurden, wie z.B. JBuilder oder Eclipse.
Zudem ist xUnit Open Source, und wird darum bald für alle relevanten Sprachen und Plattformen verfügbar sein.

Die Popularität hat natürlich ihrerseits wieder für Add-Ons gesorgt, so daß es auch schon eine bemerkenswerte Sammlung an Erweiterungen für z.B. JUnit gibt:

GUI Tools

J2EE - Extensions

Mock Handler

Code-Generatoren

HTML-Exporter

usw.

Jede Testgruppe wird in einer separaten Klasse untergebracht. Mit Hilfe von einigen Makros ist schnell eine eigene Testklasse fertig gestellt:

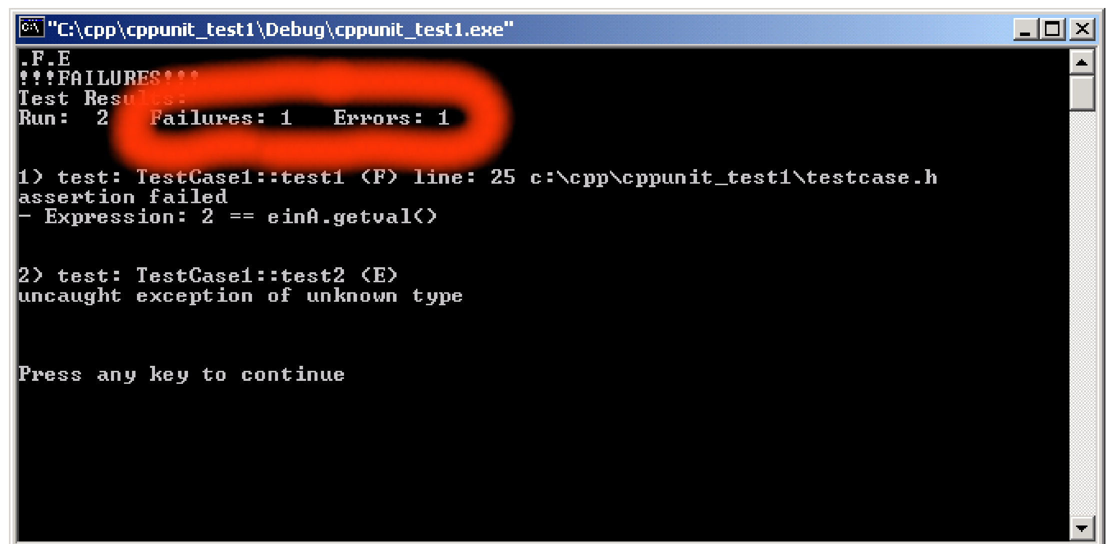
```
class A {
public:
     int getval(){return 1;}
     int doexcept(){throw(54);return 0;}
};
class TestCase1 : public CppUnit::TestFixture{
     CPPUNIT_TEST_SUITE(TestCase1);
     CPPUNIT_TEST(test1);
     CPPUNIT_TEST(test2);
     CPPUNIT_TEST_SUITE_END();
     A einA;
public:
     void setUp(){}
     void tearDown(){}

     void test1(){
          CPPUNIT_ASSERT(2 == einA.getval());
     }

     void test2(){
          CPPUNIT_ASSERT(1 == einA.doexcept());
```

```
        }
};
```

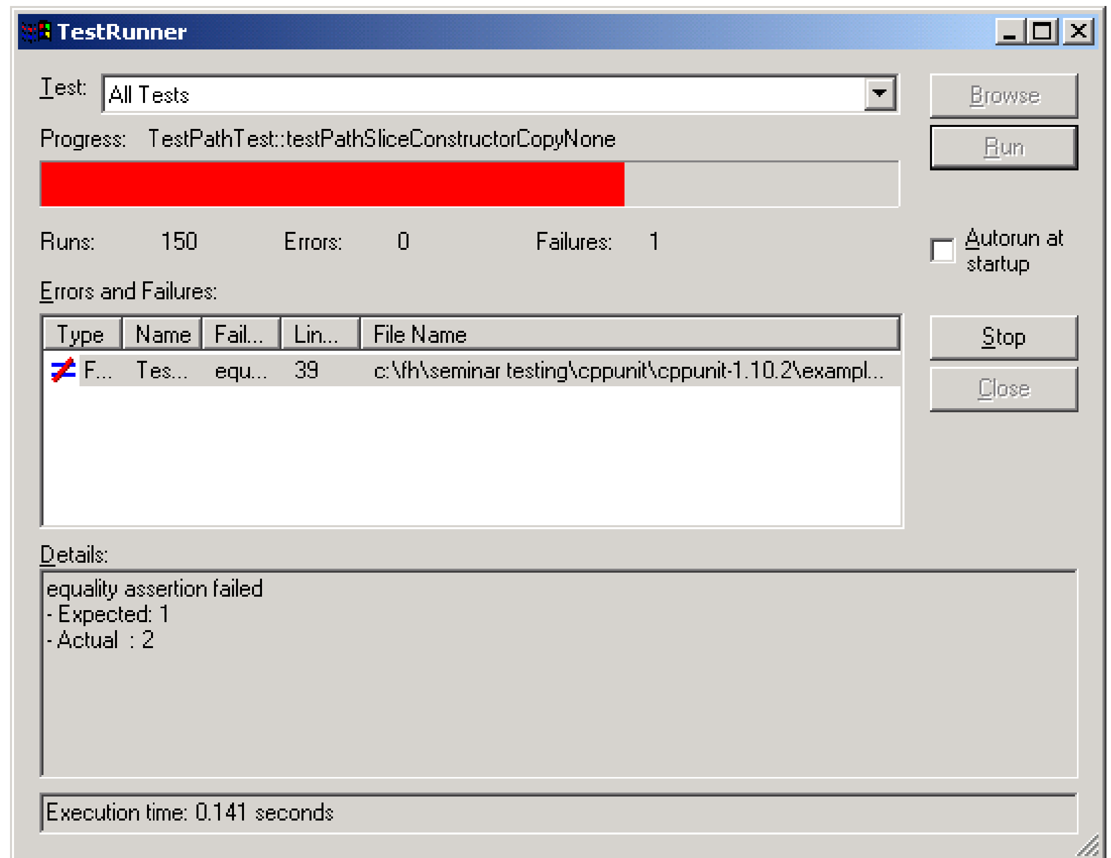
Die Klasse A hat zwei Methoden: Eine, die einfach eine Zahl zurück gibt, und eine, die eine Exception wirft. Die Testklasse heißt TestCase1, und unterteilt die Tests in wiederum zwei verschiedene Methoden. Beide Tests sollen fehl schlagen. Die nicht-erfüllte Zusage 2==einA.getval() führt zu einem Fehler, die unerwartet geworfene Exception führt zu einem Error.
Der Körper einer main()-Funktion könnte z.B. so aussehen:

```
CppUnit::TestSuite suite;
suite.addTest( TestCase1::suite() );
CppUnit::TextTestResult res;
suite.run( &res );
std::cout << res << std::endl;
```

Ausgabe:



Unter MFC sehen Ausgaben ansprechender aus:



(Beispiel wurde dem Source-Paket von CppUnit entnommen)

4.5 Model Checker

Das Problem der "Explodierenden Zustände" ist schon vom Decision-Condition Coverage bekannt. Wird versucht, jeden möglichen Pfad in einer Unit zu beschreiten, explodieren die möglichen Zustände nicht selten bis ins Unendliche. Es gibt jedoch Tools (Bebop von Microsoft oder Moped von der Universität Stuttgart), die trotzdem versuchen, den systematischen Beweis für Fehlerfreiheit zu erbringen, indem sie vorliegende Programme abstrahieren.

Beispiel:

Original-Programm:	Abstraktion mit Prädikat P = i<10
`int i=5;`	`bool P=true;`
`while (i<10) {`	`while (P) {`
`    i++;`	`    P = random_boolean();`
`}`	`}`

Durch die Abstraktion schrumpft der zu prüfende Zustandsraum enorm. Weitere Optimierungen (Summarization,...) sind möglich.
Schwieriger hat es ein Model Checker, wenn Prozesse mit möglichen Race Conditions vorliegen. Hier müssen alle möglichen Befehlsabfolgen geprüft werden, um sicher zu gehen, daß sich die Prozesse nicht gegenseitig behindern.
Da die Model-Überprüfung aber nur an einem abstrahierten Programm vorgenommen werden, kann ein Scheitern auch auf unzureichende Abstraktion oder auf Mängel im Modell zurückgeführt werden.

[Q 13]

4.6 GlowCode

4.6.1 GlowCode ist ein einzigartig funktionales und übersichtliches Tool zur Überwachung laufender Programme in Echtzeit. Es hilft z.B. beim Aufspüren von **Flaschenhälsen**:

4.6.2 und **Speicherlecks**:

Dazu muß GlowCode an einen laufenden Prozeß "angehängt" werden. Es nutzt vorhandene Debugging-Informationen, um verschiedene statische und dynamische Informationen zu gewinnen. Das Speicherleck in einer einfachen Demonstrationsanwendung wurde sofort mit Hinweis auf die verursachende Zeile ausgegeben.

4.6.3 GlowCode eignet sich auch für sog. "**Coverage** Tests", d.h. für eine Bewertung der Whitebox-Tests. Ein Coverage Tool überwacht, welche Zeilen des Codes tatsächlich aufgeführt wurden. Sind unsere Tests (Decision / Condition) sinnvoll gewählt, so wird jede Zeile des Moduls durchlaufen. Beispiel:

```
int dummy(int a){
    int ret;
    if (a>0) {
        if (a>100) {
            ret=1;
        } else {
            ret=2;
        }
    } else {
        if (a<-100) {
            ret=-1;
        } else {
            ret=-2;
        }
    }
    return ret;
}

void testDummy(){
assert( 1==dummy( 200));
assert( 2==dummy(  50));
assert(-1==dummy(-200));
//assert(-2==dummy( -50));
}
```

Wird in einem Decision-/Condition-Coverage Whitebox-Test eine einzige Zeile des Dummys nicht durchlaufen (fett), so findet es GlowCode für uns heraus:

GlowCode--C:\cpp\glowcode_test2\Debug\glowcode_test2.exe

Hooks Units-Millisec View More Help

Report | Profile | Trace | Memory

Run | Clear | Details | Export | Options

Name	Total visit time	Average visit time	Total function visits	Total line visits	Lines visited	% line hooks visited	Lines hooked
dummy	0.006	0.000	21	22	12	92.3%	13
0010: if (a>0) {	0.000	0.000		3	1		1
0011: if (a>100) {	0.000	0.000		2	1		1
0013: } else {	0.000	0.000		1	1		1
0014: ret=2;	0.000	0.000		1	1		1
0016: } else {	0.000	0.000		2	1		1
0017: if (a<-100) {	0.000	0.000		1	1		1
0019: } else {	0.000	0.000		1	1		1
0020: ret=-2;	0.000	0.000		0	0		1
0024: return ret;	0.000	0.000		3	1		1
0025:}	0.000	0.000		3	1		1
0018: ret=-1;	0.000	0.000		1	1		1
0007:int dummy(int a){	0.000	0.000		3	1		1

Der Test ist erst dann vollständig, wenn er 100% der Zeilen durchläuft.

4.7 Fazit / Ausblick

Automatisierung ist zu einem wichtigen Bestandteil aktueller Software-Entwicklungen geworden. Sie macht Programmierern, die gewillt sind, möglichst fehlerfreie Software zu produzieren, die Arbeit entschieden leichter. Doch Vorsicht, die Automatisierung darf auch nicht überbewertet werden. Automatisierung ist kein Garant für fehlerfreie Software (diese gibt es übrigens u.a. nach Murphy nicht).

Die Ansätze, die es bisher in der Welt der Automatisierungen gibt, sind gut, aber sie sind unter dem Strich nur Werkzeuge, die ein wenig Übung zur Beherrschung erfordern, und selbst natürlich weder fehlerlos noch idiotensicher sind.

Blackbox-Regressionstests an Units sind die einzige Form, die sich vollständig automatisieren läßt, wenn sich nicht Grundlegendes an den Schnittstellen ändert. Alle anderen Formen der "Automation" müssen von erfahrenen Programmierern gepflegt und überwacht werden.

Vielleicht bringt uns die Zukunft umfassendere Automatisierungen, vielleicht erübrigen sich große Teile des Testens aber auch dann, wenn der Prozeß der automatisierten Programmierung vorangetrieben wird (vgl. [Q 12]).

5 Zusammenfassung

Verschiedene Methoden zum Testen wurden vorgestellt. Keine von ihnen ist perfekt, und keine einzige kann uns mit Sicherheit sagen, ob ein Programm fehlerfrei ist. Gute Programmierer, die wissen, wo Fehlerquellen liegen, und Code mit wenig Fehlern schreiben, sind unersetzbar.
Ob auch weiterhin Raketen und Sonden gesprengt werden müssen, liegt nicht allein in der Hand der Modelle, Techniken und Tools, sondern nach wie vor auch in der Hand der verantwortlichen Programmierer.
Mit Ausdauer und viel Engagement in der Open Source-Bewegung, und Kooperation mit und unter den Softwarefabriken, werden in Zukunft die Testmöglichkeiten konstruktiv ausgeweitet und für jedermann zugänglich.
Ob es wirklich jemals fehlerfreie Software geben wird? Gibt es überhaupt etwas "perfektes" auf dieser Welt? Vielleicht sind Computer auch nur Menschen, und sie müssen mit ihren Fehlern akzeptiert werden.
Aber auf dem Sektor des systematischen Testens kann noch viel passieren. Die Zukunft darf mit Spannung erwartet werden, doch auch schon in der Gegenwart liegen trickreiche Ideen bereit, die auf experimentierfreudige Programmierer warten.

7 Quellenverzeichnis

[1] Meyers, G.J.: The Art of Software Testing, 7. Auflage 2004

[2] Morton, S.D.: Model-Based Software Development, 2001

[3] Fewster / Graham: Software Test Automation, 2002

[4] Beck / Andres: Extreme Programming Explained, 4. Auflage 2000

[5] Marchesi / Succi / Wells / Williams: Extreme Programming Perspectives, 2002

[6] Li / Wu: Effective GUI Test Automation, 2005

[7] Helfert, B.: Extreme Programming in der Praxis, 2003

[8] Vigenschow, U.: Objektorientiertes Testen und Testautomatisierung in der Praxis, 1. Auflage 2005

[9] McGregor, Sykes: A Practical Guide to Testing Object-Oriented Software, 2001

[10] B. Oestereich: Objektorientierte Softwareentwicklung, 2004

[11] Buchacker, K.: Testautomatisierung mit Linux, in: iX 2 / 2004, S. 104

[12] Care Technologies: Programmiermaschine, in: iX 3 / 2004, S. 38

[13] Weißenbacher, G.: Abstrakte Kunst, in: iX 5 / 2004, S. 116