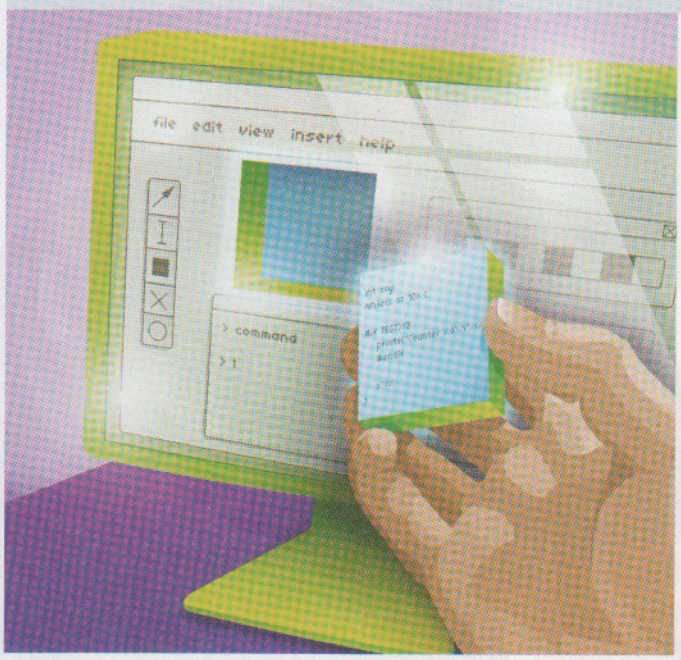
Fette ein Wort!

Wie Entwickler und Maintainer ihre Situation im Handumdrehen verbessern

Software testen

Alle Programmierer auf der Welt machen Fehler. Darum, trotzdem funktionsfähige und sichere Programme zu bekommen, kümmert sich das deswegen so wichtige Sachgebiet „Software-Testing". Zu diesem gehören Code-Reviews genauso wie dynamische Black-Box-Tests. Jan Kleinert, Tim Schürmann



Experten schätzen, dass gängige Software im Mittel 25 Fehler pro 1000 Programmzeilen enthält und gute Programme zwei Fehler pro 1000 Zeilen. Software, die in der Raumfahrt zum Einsatz kommt, soll noch eine Größenordnung weniger Bugs aufweisen. Ob das für die Software der Ariane-5-Rakete (Abbildung 1) zutraf, die am 4. Juni 1996 in den Himmel von Französisch-Guayana aufstieg, darf man bezweifeln.

Das an Bord befindliche Trägheitsnavigationssystem jedenfalls stürzte 36,7 Sekunden nach dem Start ab, als es versuchte, den Wert der horizontalen Geschwindigkeit von einer 64-Bit-Gleitkommadarstellung in einen vorzeichenbehafteten 16-Bit-Integer umzuwandeln. Hier die Ada-Codezeile:

```
P_M_DERIVE(T_ALG.E_BH) := UC_16S_EN_16NS?
 (TDB.T_ENTIER_16S ((1.0/C_M_LSB_BH) *?
 G_M_INFO_DERIVE(T_ALG.E_BH)))
```

Die betreffende Zahl war schlicht größer als 215 und erzeugte einen Overflow der Variablen »E_BH«. Anschließende brach das Lenksystem der Rakete zusammen und gab die Kontrolle an eine zweite identische Einheit ab. Da der Backuprechner das exakt gleiche Programm verwendete, stürzte auch er ab. Der Hauptcomputer interpretierte die Nonsens-Werte als Flugbahnabweichung und versuchte diese zu kompensieren. Nach 40 Sekunden Flugzeit löste eine Automatik die Selbstzerstörung des ganzen Stolzes der europäischen Raumfahrt aus. Materieller Sofortschaden: 370 Millionen Dollar.

Als wäre das nicht ärgerlich genug, förderte die Untersuchung des Vorfalls pikante Details [1] zu Tage:

- Die Typumwandlung war nicht abgesichert, da die Programmierer glaubten, die übergebene Geschwindigkeit könne nie so groß werden.
- Die Annahme war für die Ariane 4 richtig, für die das Programm ursprünglich gedacht war. Die Ariane 5 besitzt aber ein anderes Flugprofil.
- Ariane 4 brauchte das Programm nur während des Countdown, Ariane 5 braucht es eigentlich überhaupt nicht.
- Die Entscheidung, den Ariane-4-Code ungetestet auf der Ariane 5 zu benutzen, hatte das Management des Softwareentwicklerteams gefällt.
- Die Entwicklung der Ariane 5 kostete geschätzt sieben Milliarden Dollar.

Aus dieser wahren Anekdote lassen sich eine Menge Erkenntnisse über falsche Annahmen, die Wirksamkeit von Redundanzen, unerwartete Folgekosten und die Wichtigkeit von Softwaretests ziehen.

Notweniges Übel

Obwohl die Notwendigkeit des Testens selbst geschriebener Software unstreitig ist, gehört es wohl zu den meist gehassten Aufgaben eines Entwicklers. Es ist umständlich, zeitaufwändig, und am Ende weiß er nie, ob er wirklich alle Fehler erwischt hat. Stehen die Programmierer auch noch unter Zeitdruck, schieben diese die Tests immer weiter auf.

Das Ergebnis ist die berühmt-berüchtigte Bananensoftware, die beim Anwender reift – Patches kann man schließlich immer noch nachschieben. Diese Strategie rächt sich jedoch in unzufriedenen Benutzern und hohen Supportkosten. Letzteres gilt auch für Open-Source-Software und für Hobbyentwickler: Bis sich der Programmautor durch die zahlreichen Fehlermeldungen gefräst hat, ist sein schöner Feierabend hinüber.

Inhalt

Abbildung 1: Eine Ariane-5-Rakete wenige Sekunden nach dem Start. Beim Jungfernflug ging die Sache 36,6 Sekunden lang gut.

Folglich kommt niemand um Tests herum, schlauerweise plant man sie schon während der Entwicklung fest ein und führt sie nicht erst ganz am Ende durch. Zu den gesicherten Erkenntnissen gehört, dass je später ein Fehler im Entwicklungszyklus auffällt, desto teurer seine Behebung kommt. In diesem Licht offenbart das sowieso recht antiquierte Wasserfallmodell (Abbildung 2), das Tests nur in den letzten Phasen vorsieht, seinen uneffizienten Charakter. Das ganze Gegenteil praktiziert das Test Driven Development (TDD), ein Gray-Box-Testverfahren, das vor dem Schreiben des Programmcodes ansetzt (siehe den eigenen Schwerpunkt-Artikel zu diesem Thema).

Egal mit welcher Methodik, wichtig ist es, planvoll vorzugehen. Es nützt nichts, das Programm zu starten und auf drei Knöpfe zu klicken. Das bringt nur die Gewissheit, dass die drei Knöpfe gelegentlich funktionieren. Hinzu kommt der psychologische Effekt, dass der Programmierer einer Software davon ausgeht, dass sein Code genau das macht, was er sich erhofft hatte. Ein Zeichen für das Erwartungstäuschung genannte Phänomen sind Gedanken wie: „Wieso gibt er denn jetzt diesen merkwürdigen Wert aus?!"

Weil Blauäugigkeit beim Thema Software-Testing unangebracht ist, sollten sich Projekte formale Strategien und Vorgehensweisen zu Eigen machen. Abbildung 3 zeigt aus einiger Entfernung ein paar Platzierungsmöglichkeiten für Tests.

Reviews und statische Codeanalyse

Der Oberbegriff für eine wichtige Disziplin lautet „Statische Analysen". Das sind White-Box-Testverfahren, deren Kennzeichen es ist, dass sie das Programm nicht ausführen. Code-Reviews und statische Codeanalysen zählen dazu. Die einfachste Variante des Quellcode-Studiums besteht darin, selbst oder noch besser im Team den Code mit eigenen Augen zu begutachten und zu untersuchen. Es geht darum, funktionale und formelle Fehler zu finden. Der Arbeitsablauf ist dreistufig: Identifizieren, konsolidieren, korrigieren. Die Bedeutung von Code-Reviews hat in letzter Zeit in dem Maße wieder zugenommen, wie die agilen Entwicklungsmethoden begannen, sich durchzusetzen (Stichwort: Scrum).

Ein Review kann nicht nur die Befehle betrachten, sondern auch den Fluss der (Eingabe-)Daten durch das Programm verfolgen (siehe auch den Artikel zum Thema Sicherheit von Webapplikationen). Bei dieser Datenflussanomalie-Analyse pickt sich der Tester eine Variable heraus und untersucht, wann und wie die folgenden Anweisungen sie erzeugen, verändern und zerstören. Stellt er etwa fest, dass der Algorithmus die Variable liest, bevor sie erzeugt wurde, liegt zwingend ein Fehler vor – die so genannte UR-Anomalie (Undefined Referencing).

Wer den Quellcode in den Fingern hält, kann gleich noch eine Stilanalyse durchführen, also feststellen, ob der Quellcode entsprechend den Vorgaben formatiert ist oder unbotmäßige Sprachkonstrukte verwendet – etwa Gotos. Obendrein lassen sich ein paar Maße und Statistiken ermitteln, wie etwa die Anzahl der selbst geschriebenen Klassen. Alle diese Untersuchungen lassen sich von Menschen auf Papier durchführen.

Solche manuellen Inspektionen oder Reviews sind allerdings recht zeitaufwändig. Zumindest bei einigen Prüfungen, wie der Stilanalyse, bei den Metriken oder der Datenflussanalyse, helfen zum Glück Werkzeuge. Hier ist auch die Grenze zur statischen Codeanalyse (formale Verifikation) erreicht. Ein Artikel im Schwerpunkt stellt ein paar solcher Anwendungen vor. Ein wichtiger Spieler in diesem Markt ist zudem die Firma Coverity [2].

Dynamische Tests

Um festzustellen, ob die selbst geschriebene Anwendung funktioniert, kommt man jedoch nicht umhin, sie auch auszuführen und mit Testdaten zu füttern, also so genannte dynamische Testtechniken einzusetzen. Dabei stellt sich die Frage, welche und wie viele Testdaten der Entwickler an sein Programm verfüttern muss, um sicherzugehen, dass es reibungslos funktioniert.

Die Testdaten sollten so gewählt sein, dass der Programmablauf jede einzelne Anweisung zumindest ein Mal ausführt.

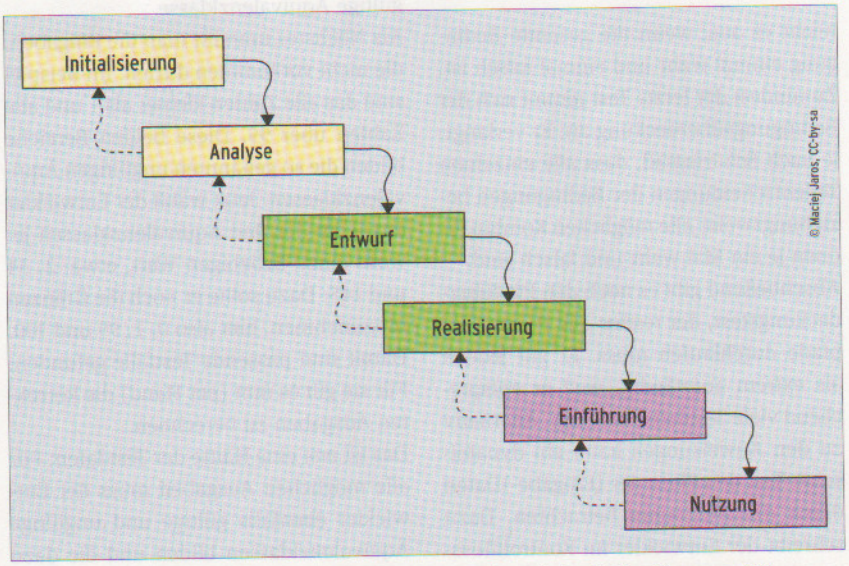


Abbildung 2: Erweitertes klassisches Wasserfallmodell, das Tests nur in den letzten Phasen vorsieht.

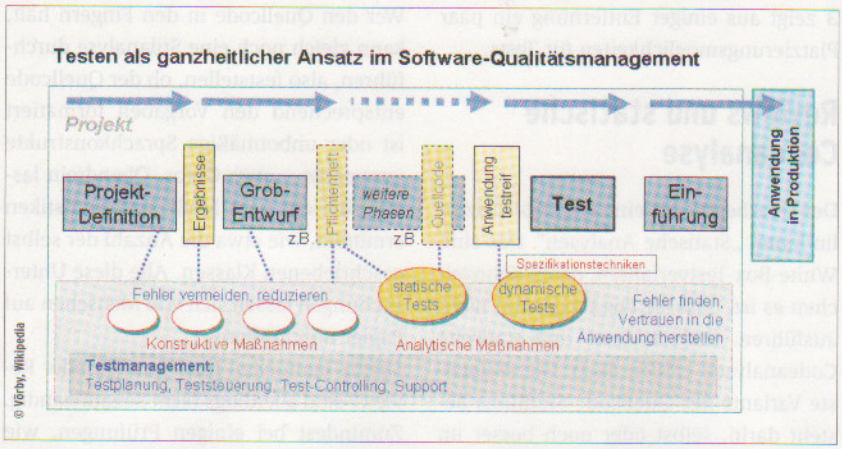


Abbildung 3: Ganzheitlicher Ansatz: Testen nach einer weitergehenden Definition umfasst alle Prüfmaßnahmen in der Softwareentwicklung.

Einen solchen Test bezeichnet man als Anweisungsüberdeckungstest, kurz C0-Test. Ein Problem wird erkennbar, wenn der Entwickler sich den Programmablauf aufmalt. Dabei zeichnet er jede Anweisung als Kreis (Knoten) und zieht dann von einer Anweisung Pfeile zu allen prinzipiell direkt auf sie nachfolgenden Anweisungen. Im Ergebnis entsteht ein so genannter Kontrollflussgraph. Bei einem C0-Test würde im dem Beispiel

```
if(i!=10) then a=a+1;
```

nicht der Fall getestet, in dem »i« gleich 10 ist. Es wäre besser, wenn mindestens ein Mal alle Zweige des Programms ausgeführt würden. Aber auch die Bedingung ist zu lasch: Der Zweigüberdeckungstest (C1-Test) durchliefe Schleifen nur einmal, und bei If-Abfragen der Art:

```
if((A || B) && (C || D)) ...
```

reicht es aus, wenn die gesamte Bedingung einmal wahr und einmal falsch ist. Zumindest der letzte Test nimmt sich der Bedingungsüberdeckung an. Er verlangt, je nach Schärfegrad, dass alle einzelnen Teilentscheidungen der Bedingungen beziehungsweise alle möglichen Kombinationen je ein Mal wahr und falsch sind.

Abschließend gibt es noch den Pfadüberdeckungstest, der restlos alle Programmpfade durchlaufen muss. Er gilt jedoch als extrem aufwändig, weil er entsprechend viele Testdaten benötigt. Alternativ zu den Anweisungen kann ein dynamischer Test den Fluss der (Eingabe-)Daten durch das Programm betrachten. Dazu schreibt der Entwickler im Kontrollflussgraph an die Knoten alle Variablen, die dieser Knoten benutzt oder verändert. Geht die Variable an einem Knoten in eine Berechnung ein oder schreibt die entsprechende Anweisung einen neuen Wert in die Variable, heißt der Fall C-Use (von Computational Use). Kommt ihr Wert in einer Entscheidung zum Einsatz, etwa in einer If-Abfrage, spricht man von P-Use (Predicate-Use).

Geeignete Testdaten finden

Um nun konkrete Testdaten zu finden, bildet man auf den möglichen Eingaben und Ausgaben Äquivalenzklassen. Das klingt kompliziert, die meisten Programmierer machen das bereits intuitiv. Zunächst stellen sie die Eingabebereiche des Programm fest. Beispielsweise könnte es Werte zwischen 1 und 99 entgegennehmen. Der Bereich bildet eine so genannte gültige Äquivalenzklasse.

Als Nächstes umreißt man die Eingaben, die nicht vorkommen dürfen. Im Beispiel sind das alle Zahlen kleiner als 1 und alle Zahlen über 99. Diese beiden Bereiche bilden die so genannten ungültigen Äquivalenzklassen. Jetzt wählt der Entwickler aus jeder der drei Äquivalenzklassen jeweils einen beliebigen Wert, etwa -2, 34 und 145. Dazu sollte er noch die Grenzen hinzunehmen, hier also 0, 1, 99 und 100. Damit sind passende Testfälle gefunden. Für sie gilt es nun (per Hand) die korrekten Ausgaben zu berechnen.

Das ist erst eine Hälfte der Testdaten: Für alle möglichen Ausgaben muss der Entwickler ebenfalls gültige und ungültige Äquivalenzklassen bilden und die dazu passenden Eingaben finden. Abschließend prüft er, ob die gefundenen Testdaten auch die erwähnten Kriterien abdecken, also beispielsweise einmal alle Anweisungen ausführen. Wenn nicht, muss er weitere Testdaten hinzunehmen.

Das Gegenteil veranstalten Smoketests: Hier bestimmt der Programmierer die Testdaten nicht mit systematischen Methoden, sondern wählt sie bewusst zufällig. Smoketests kommen meist in frühen Entwicklungsphasen zum Einsatz. Sie klären, ob das Programm funktioniert ohne abzurauchen – daher der Name. Die Praxis zeigt, dass diesem Stochern im Nebel witzigerweise oft eine ganz passable Testabdeckung gelingt.

Überhaupt sind die Ausprägungen, Spielarten und deren Kategorisierungsmöglichkeiten beim Software-Testing offenbar unendlich. Wer sich der Wikipedia-Seite [3] und den dort verzeichneten Verlinkungen widmet, verliert schnell den Überblick. Es gibt objektorientiertes Testen, zustandsbasiertes, Klassentests, Modul- und Unittests, Regressionstests, Integrationstests, Datenkonsistenztests, Systemtests, Akzeptanztests … Letztlich geht es aber immer darum, Fehler zu finden.

Fehl ohne Tadel

Auch wenn die wenigsten Programmierer Ada-Code für Trägerraketen entwickeln, sollten sie sich selbstbewusst ihrer eigenen Fehlbarkeit stellen – wo Menschen arbeiten, machen sie auch Fehler. Das ist kein Beinbruch, solange man die rchtigen Schlüsse daraus zieht: Mit Köpfchen und Verantwortung geplante und durchgeführte Softwaretests machen den Bugs wirksam den Garaus. Das Sachgebiet ist viel zu umfangreich, als dass ein Magazin-Schwerpunkt es komplett zu würdigen vermag. Die folgenden Artikel setzen aber Spots auf einige Teildisziplinen, die manchem Software-Manager der Ariane 5 zur Erleuchtung hätten verhelfen können – und nicht nur dem. ■

Infos

[1] Ken Garlington, „Critique of ‚Put it in the contract: The lessons of Ariane'": [http://lore.ua.ac.be/Teaching/SE3BAC/ariane.html]
[2] Coverity: [http://www.coverity.com/de/]
[3] „Softwaretest" bei Wikipedia: [http://de.wikipedia.org/wiki/Softwaretest]