SWE1
Testmanagement

Inhalt

Artikel

Referenzen

Artikellizenzen

Einleitung

Softwaretest

Ein **Softwaretest** ist *ein Test*, der im Rahmen der Softwareentwicklung durchgeführt wird. Er prüft und bewertet Software gegen die für ihren Einsatz definierten Anforderungen und misst ihre Qualität. Die aus dem Softwaretest gewonnenen Erkenntnisse werden zur Erkennung und Behebung von Softwarefehlern genutzt und dienen dazu, die Software möglichst fehlerfrei in "Produktion" zu nehmen.

Von diesem, *einzelne* Testdurchführungen bezeichnenden Begriff ist die gleich lautende Bezeichnung 'Test' (auch 'Testen') zu unterscheiden, unter der die *Gesamtheit* der Maßnahmen zur Überprüfung der Softwarequalität (inkl. Planung, Vorbereitung, Steuerung, Durchführung, Dokumentation usw.; siehe auch Definitionen) verstanden wird.

Den Nachweis, dass keine Fehler (mehr) vorhanden sind, kann das Softwaretesten nicht erbringen, es kann lediglich feststellen, dass bestimmte Testfälle erfolgreich waren. E. W. Dijkstra hierzu: *Program testing can be used to show the presence of bugs, but never show their absence!* (Programmtesten kann angewandt werden, um die Existenz von Fehlern zu zeigen, aber niemals deren Abwesenheit.) Der Grund ist, dass alle Programmfunktionen und auch alle möglichen Werte in den Eingabedaten in allen ihren Kombination getestet werden müssten – was (außer bei sehr einfachen Testobjekten) praktisch nicht möglich ist. Aus diesem Grund beschäftigen sich verschiedene Teststrategien und -konzepte mit der Frage, wie mit einer möglichst geringen Anzahl von Testfällen eine große Testabdeckung zu erreichen ist.

Pol, Koomen, Spillner [1] zu Testen: *"Tests sind nicht die einzige Maßnahme im Qualitätsmanagement der Softwareentwicklung, aber oft die letztmögliche. Je später Fehler entdeckt werden, desto aufwändiger ist ihre Behebung, woraus sich der Umkehrschluss ableitet: Qualität muss (im ganzen Projektverlauf) implementiert und kann nicht 'eingetestet' werden."*

Beim Testen in der Softwareentwicklung wird i. d. R. eine mehr oder minder große Fehleranzahl als 'normal' unterstellt oder akzeptiert. Hier herrscht ein erheblicher Unterschied zur Industrie: Dort werden im Prozessabschnitt 'Qualitätskontrolle' oft nur noch in Extremsituationen Fehler erwartet.

Definition

Es gibt unterschiedliche Definitionen für den Softwaretest:

Nach ANSI/IEEE Std. 610.12-1990 ist Test *„the process of operating a system or component under specified conditions, observing or recording the results and making an evaluation of some aspects of the system or component.“*

Eine andere Definition liefert Denert[2] , wonach der *„Test […] der überprüfbare und jederzeit wiederholbare Nachweis der Korrektheit eines Softwarebausteines relativ zu vorher festgelegten Anforderungen“* ist.

Eine weitergehende Definition verwenden Pol, Koomen und Spillner in [1] : *Unter Testen versteht man den Prozess des Planens, der Vorbereitung und der Messung, mit dem Ziel, die Eigenschaften eines IT-Systems festzustellen und den Unterschied zwischen dem tatsächlichen und dem erforderlichen Zustand aufzuzeigen.* Bemerkenswert hierbei: Als Messgröße gilt 'der erforderliche Zustand', nicht nur die (möglicherweise fehlerhafte) Spezifikation.

'Testen' ist ein wesentlicher Teil im Qualitätsmanagement von Projekten der Softwareentwicklung.

Ziele

Globales Ziel des Softwaretestens ist das *Messen der Qualität des Softwaresystems*. Dabei werden definierte Anforderungen überprüft und dabei ggf. vorhandene Fehler aufgedeckt. ISTQB: *Der Wirkung von Fehlern (im produktiven Betrieb) wird damit vorgebeugt*.

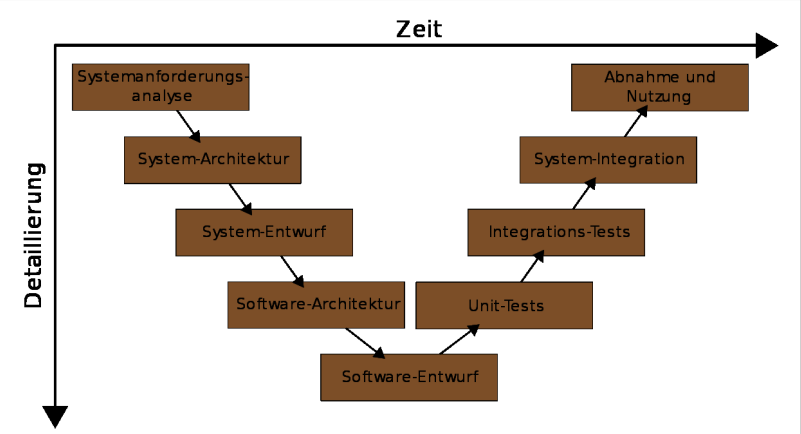
Ein Rahmen für diese Anforderungen können die *Qualitätsparameter gem. ISO/IEC 9126* sein, denen jeweils konkrete Detailanforderungen z. B. zur Funktionalität, Bedienbarkeit, Sicherheit usw. zugeordnet werden können. Im Besonderen ist auch die Erfüllung gesetzlicher und / oder vertraglicher Vorgaben nachzuweisen.

Die Testergebnisse (die über verschiedene Testverfahren gewonnen werden) tragen zur Beurteilung der realen Qualität der Software bei – als Voraussetzung für deren Freigabe zum operativen Betrieb. *Das Testen soll Vertrauen in die Qualität der Software schaffen* [1] .

Individuelle Testziele: Da das Softwaretesten aus zahlreichen Einzelmaßnahmen besteht, die i. d. R. über mehrere Teststufen hinweg und an vielen Testobjekten ausgeführt werden, ergeben sich individuelle Testziele für jeden einzelnen Testfall und für jede Teststufe – wie z. B. Rechenfunktion X in Programm Y getestet, Schnittstellentest erfolgreich, Wiederinbetriebnahme getestet, Lasttest erfolgreich, Programm XYZ getestet usw.

Teststufen

Die Einordnung der Teststufen (auch Testzyklen genannt) folgt dem Entwicklungsstand des Systems. Sie orientieren sich in der Regel an den Entwicklungsstufen von Projekten nach dem V-Modell. Dabei wird in jeder Teststufe (rechte Seite im 'V') gegen die Systementwürfe und Spezifikationen der zugehörigen Entwicklungsstufe (linke Seite) getestet.



Stufen des V-Modells

In der Realität werden diese Ausprägungen, abhängig von der Größe und Komplexität des Software-Produkts, weiter untergliedert. So könnten beispielsweise die Tests für die Entwicklung von sicherheitsrelevanten Systemen in der Transportsicherungstechnik folgendermaßen untergliedert sein: Komponententest auf dem Entwicklungsrechner, Komponententest auf der Ziel-Hardware, Produkt-Integrationstests, Produkttest, Produkt-Validierungstests, System-Integrationstest, Systemtest, System-Validierungstests, Feldtests und Akzeptanztest.

Komponententest

Der Modultest, auch *Komponententest* oder *Unittest* genannt, ist ein Test auf der Ebene der einzelnen Module der Software. Testgegenstand ist die Funktionalität innerhalb einzelner abgrenzbarer Teile der Software (Module, Programme oder Unterprogramme, Units oder Klassen). Testziel dieser häufig durch den Softwareentwickler selbst durchgeführten Tests ist der Nachweis der technischen Lauffähigkeit und korrekter fachlicher (Teil-) Ergebnisse.

Integrationstest

Der Integrationstest bzw. Interaktionstests testet die Zusammenarbeit voneinander abhängiger Komponenten. Der Testschwerpunkt liegt auf den Schnittstellen der beteiligten Komponenten und soll korrekte Ergebnisse über komplette Abläufe hinweg nachweisen.

Systemtest

Der **Systemtest** ist die Teststufe, bei der das gesamte System gegen die gesamten Anforderungen (funktionale und nicht funktionale Anforderungen) getestet wird. Gewöhnlich findet der Test auf einer Testumgebung statt und wird mit Testdaten durchgeführt. Die Testumgebung soll die Produktivumgebung des Kunden simulieren. In der Regel wird der Systemtest durch die realisierende Organisation durchgeführt.

Abnahmetest

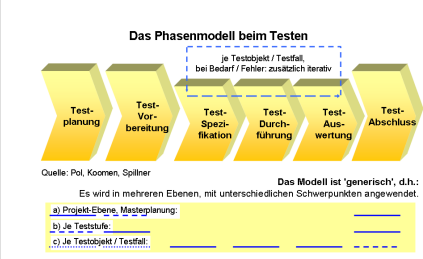
Ein **Abnahmetest**, **Verfahrenstest**, **Akzeptanztest** oder auch **User Acceptance Test** (UAT) ist ein Test der gelieferten Software durch den Kunden bzw. Auftraggeber. Oft sind Akzeptanztests Voraussetzung für die Rechnungsstellung. Dieser Test kann bereits auf der Produktivumgebung mit Kopien aus Echtdaten durchgeführt werden.

Die vorgenannten Teststufen sind in der Praxis oft nicht scharf voneinander abgegrenzt, sondern können, abhängig von der Projektsituation, fließend oder über zusätzliche Zwischenstufen verlaufen. So könnte zum Beispiel die Abnahme des Systems auf der Grundlage von Testergebnissen (Reviews, Testprotokolle) von Systemtests erfolgen.

Besonders für System- und Abnahmetests wird das Blackbox-Verfahren angewendet, d. h. der Test orientiert sich nicht am Code der Software, sondern nur am Verhalten der Software bei spezifizierten Handlungen (Eingaben des Benutzers, Grenzwerte bei der Datenerfassung, etc.).

Testprozess / Testphasen

Pol, Koomen und Spillner [1] empfehlen ein Vorgehen nach dem in der Grafik dargestellten *Phasenmodell*. Sie nennen dieses Vorgehen *Testprozess*, die Einzelschritte *Testphasen* und unterscheiden dabei: Testplanung, Testvorbereitung, Testspezifikation, Testdurchführung, Testauswertung, Testabschluss



Der generische Testprozess und seine Testphasen

Dieses Vorgehen ist *generisch*, d.h. es wird - jeweils nach Erfordernis - für unterschiedliche Ebenen angewendet, nämlich für das Gesamtprojekt, für jede Teststufe und letztlich auch je Testobjekt und Testfall. Die in diesen Ebenen i. d. R. anfallende Arbeitsintensität ist in der Grafik durch Punkte (= gering), Striche (= mittel) und durchgehende Linien (= Schwerpunkt) dargestellt.

Testaktivitäten werden (nach Pol, Koomen und Spillner [1]) rollenspezifisch zu sog. *Testfunktionen* zusammengefasst: Testen, Testmanagement, Methodische Unterstützung, Technische Unterstützung, Funktionale Unterstützung, Verwaltung, Koordination und Beratung, Anwendungsintegrator, TAKT-Architekt und TAKT-Ingenieur (bei Einsatz von Testautomatisierung; TAKT = Testen, Automatisierung, Kenntnisse, Tools). Diese

Funktionen (Rollen) haben Schwerpunkte in bestimmten Testphasen; sie können im Projekt selbst eingerichtet sein oder über spezialisierte Organisationseinheiten einbezogen werden.

Bei *anderen Autoren oder Instituten* finden sich zum Teil andere Gruppierungen und andere Bezeichnungen, die aber inhaltlich nahezu identisch sind. Z. B. wird bei ISTQB der "fundamentale Testprozess" mit folgenden Hauptaktivitäten definiert:
Testplanung und Steuerung, Testanalyse und Testdesign, Testrealisierung und Testdurchführung, Testauswertung und Bericht, Abschluss der Testaktivitäten

Testplanung

Ergebnis dieser i. d. R. parallel zur Softwareentwicklung stattfindenden Phase ist i. W. der *Testplan*. Er wird für jedes Projekt erarbeitet und soll den gesamten Testprozess definieren. In TMap wird dazu ausgeführt: *Sowohl die zunehmende Bedeutung von IT-Systemen für Betriebsprozesse als auch die hohen Kosten des Testens rechtfertigen einen optimal verwaltbaren und strukturierten Testprozess.* Der Plan kann und soll je Teststufe aktualisiert und konkretisiert werden, sodass die Einzeltests im Umfang zweckmäßig und effizient ausgeführt werden können.

Inhalte im Testplan sollten z. B. folgende Aspekte sein: Teststrategie (Testumfang, Testabdeckung, Risikoabschätzung); Testziele und Kriterien für Testbeginn, Testende und Testabbruch - für alle Teststufen; Vorgehensweise (Testarten); Hilfsmittel und Werkzeuge; Dokumentation (Festlegen der Art, Struktur, Detaillierungsgrad); Testumgebung (Beschreibung); Testdaten (allgemeine Festlegungen); Testorganisation (Termine, Rollen), alle Ressourcen, Ausbildungsbedarf; Testmetriken; Problemmanagement

Testvorbereitung

Aufbauend auf der Testplanung werden die dort festgelegten Sachverhalte zur operativen Nutzung vorbereitet und zur Verfügung gestellt.

Beispiele für einzelne Aufgaben (global und je Teststufe): Bereitstellen der Dokumente der Testbasis; Verfügbar machen (z.B. Customizing) von Werkzeugen für das Testfall- und Fehlermanagement; Aufbauen der Testumgebung(en) (Systeme, Daten); Übernehmen der Testobjekte als Grundlage für Testfälle aus der Entwicklungsumgebung in die Testumgebung; Benutzer und Benutzerrechte anlegen; ...
Beispiele für Vorbereitungen (für Einzeltests): Transfer / Bereitstellung von Testdaten bzw. Eingabedaten in die Testumgebung(en).

Testspezifikation

Hier werden alle Festlegungen und Vorbereitungen getroffen, die erforderlich sind, um einen bestimmten Testfall ausführen zu können.

Beispiele für einzelne Aktivitäten: Testfallfindung und Testfalloptimierung; Beschreiben je Testfall (was genau ist zu testen); Vorbedingungen (incl. Festlegen von Abhängigkeiten zu anderen Testfällen); Festlegen und Erstellen der Eingabedaten; Festlegungen zum Testablauf und zur Testreihenfolge; Festlegen Soll-Ergebnis; Bedingung(en) für 'Test erfüllt'; ...

Testdurchführung

Bei dynamischen Tests wird dazu das zu testende Programm ausgeführt, in statischen Tests ist es Gegenstand analytischer Prüfungen.

Beispiele für einzelne Aktivitäten: Auswählen der zu testenden Testfälle; Starten des Prüflings - manuell oder automatisch; Bereitstellen der Testdaten und des Ist-Ergebnisses zur Auswertung; Umgebungsinformationen für den Testlauf archivieren, ...

Weitere Anmerkung: Ein Testobjekt sollte nicht vom Entwickler selbst, sondern von anderen, wenn möglich unabhängigen, Personen getestet werden.

Testauswertung

Die Ergebnisse aus durchgeführten Tests (je Testfall) werden überprüft. Dabei wird das Ist-Ergebnis mit dem Soll-Ergebnis verglichen und anschließend eine Entscheidung über das Testergebnis (ok oder Fehler) herbeigeführt.

- Bei Fehler: Klassifizierung (z. B. nach Fehlerursache, Fehlerschwere etc.), angemessene Fehlerbeschreibung und -Erläuterung, Überleitung ins Fehlermanagement; Testfall bleibt offen
- Bei OK: Testfall gilt als erledigt
- Für alle Tests: Dokumentation, Historisieren / Archivieren von Unterlagen

Testabschluss

Abschluss-Aktivitäten finden auf allen Testebenen statt: Testfall, Testobjekt, Teststufe, Projekt. Der Status zum Abschluss von Teststufen wird (z. B. mit Hilfe von Teststatistiken) dokumentiert und kommuniziert, Entscheidungen sind herbeizuführen und Unterlagen zu archivieren. Grundsätzlich ist dabei zu unterscheiden nach:

- Regel-Abschluss = Ziele erreicht, nächste Schritte einleiten
- Alternativ möglich: Teststufe ggf. vorzeitig beenden oder unterbrechen (aus diversen, zu dokumentierenden Gründen); in Zusammenarbeit mit dem Projektmanagement

Klassifikation für Testarten

In kaum einer Disziplin der Softwareentwicklung hat sich, der Komplexität der Aufgabe 'Testen' entsprechend, eine derart große Vielfalt an Begriffen für Verfahrensansätze gebildet wie beim Softwaretest. Dies beginnt bereits bei den *Typ-Ausprägungen* für Testvarianten, die mit Begriffen wie Teststufe, Testzyklus, Testphase, Testart, Testtyp, Testmethode, Testverfahren ... bezeichnet werden.

Die Bezeichnung *konkreter Testarten* leitet sich meistens aus ihren individuellen Zielen und Charaktermerkmalen ab - wodurch sich eine Vielzahl an Bezeichnungen ergibt. Dieser Vieldimensionalität entsprechend können für einen konkreten Test die Bezeichnungen mehrerer Testarten zutreffen. Beispiel: *Entwicklertest, dynamischer Test, Blackbox-Test, Fehlertest, Integrationstest, Äquivalenzklassentest, Batchtest, Regressionstest*. Durchaus im Sinn effizienter Testprozesse ist es, mehrere Testfälle mit nur einem konkreten Test abzudecken, z. B. eine technische Datenschnittstelle, die Prüfung korrekter Wertebereiche und eine Rechenformel.

Die nachfolgend beschriebenen Testart-Bezeichnungen sind Beispiele aus der Literatur. Im praktischen Einsatz werden aber viele dieser Bezeichnungen nicht verwendet, sondern (zum Beispiel) einfach als 'Funktionstest' bezeichnet und nicht als Fehlertest, Batchtest, High-Level-Test etc. Die Testeffizienz wird hierdurch nicht beeinträchtigt - wenn die Tests ansonsten angemessen geplant und ausgeführt werden. Die nachfolgenden Aufzählungen können auch eine Vorstellung davon vermitteln, was beim Testen, vor allem in der Testplanung berücksichtigt werden sollte oder könnte.

Ein Mittel zum Verständnis dieser Begriffsvielfalt ist die nachfolgend angewendete *Klassifikation* - bei der Testarten nach unterschiedlichen Kriterien gegliedert werden.

Klassifikation nach der Prüftechnik

Analytische Maßnahmen

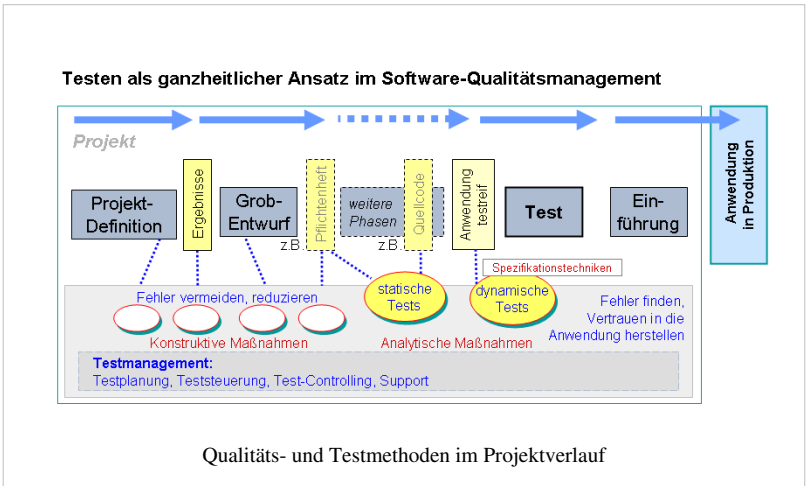
Softwaretests werden oft als analytische Maßnahmen, die erst nach Erstellung des Prüfgegenstandes durchgeführt werden können, definiert. Liggesmeyer [3] klassifiziert diese Testmethoden folgendermaßen (verkürzt und z. T. kommentiert):

Statischer Test (Test ohne Programmausführung)

- Review
- Statische Code-Analyse, auch Formale Verifikation

Dynamischer Test (Test während Programmausführung)

- Strukturorientierter Test
  - Kontrollflussorientiert (Maß für die Überdeckung des Kontrollflusses)
    - Anweisungs-, Zweig-, Bedingungs- und Pfadüberdeckungstests
  - Datenflussorientiert (Maß für die Überdeckung des Datenflusses)
    - Defs-/Uses Kriterien, Required k-Tupels-Test, Datenkontext-Überdeckung
- Funktionsorientierter Test (Test gegen eine Spezifikation)
  - Funktionale Äquivalenzklassenbildung, Zustandsbasierter Test, Ursache-Wirkung-Analyse z. B. mittels Ursache-Wirkungs-Diagramm, Syntaxtest, Transaktionsflussbasierter Test, Test auf Basis von Entscheidungstabellen
  - Positivtest (versucht die Anforderungen zu verifizieren) und Negativtest (prüft die Robustheit einer Anwendung)
- Diversifizierender Test (Vergleich der Testergebnisse mehrerer Versionen)
  - Regressionstest, Back-To-Back-Test, Mutationen-Test
- *Sonstige* (nicht eindeutig zuzuordnen, bzw. Mischformen)
  - Bereichstest bzw. Domain Testing (Verallgemeinerung der Äquivalenzklassenbildung), Error guessing, Grenzwertanalyse, Zusicherungstechniken



Qualitäts- und Testmethoden im Projektverlauf

Konstruktive Maßnahmen

Diesen analytischen Maßnahmen, bei denen Testobjekte "geprüft" werden, gehen die sog. konstruktiven Maßnahmen voraus, die bereits im Verlauf der Software-Erstellung zur Qualitätssicherung betrieben werden. Beispiele: Anforderungsmanagement, Prototyping, Review von Pflichtenheften.

Spezifikationstechniken

Weiterhin sind von den Prüftechniken die Spezifikationstechniken zu unterscheiden: Sie bezeichnen keine Testarten, mit denen Testobjekte aktiv geprüft werden, sondern nur die Verfahren, nach denen die Tests vorbereitet und spezifiziert werden.

Beispiele Äquivalenzklassentest und Überdeckungstest: Testfälle werden nach diesen Verfahren identifiziert und spezifiziert, konkret überprüft jedoch (z. B.) im Integrationstest, Batchtest, Sicherheitstest etc.

Klassifikation nach dem Testkriterium

Die Einordnung erfolgt hier je nachdem welche inhaltlichen Aspekte getestet werden sollen.

Funktionale Tests bzw. Funktionstests

: überprüfen ein System in Bezug auf funktionale Anforderungsmerkmale wie Korrektheit und Vollständigkeit.

Nicht-funktionale Tests

: überprüfen die nicht funktionalen Anforderungen, wie z. B. die Sicherheit, die Gebrauchstauglichkeit oder die Zuverlässigkeit eines Systems. Dabei steht nicht die Funktion der Software (Was tut die Software?) im Vordergrund, sondern ihre Funktionsweise (Wie arbeitet die Software?).

Schnittstellentests

: testen die Funktionalität bei der Zusammenarbeit voneinander unabhängiger Komponenten (für jede der beteiligten Komponente, anhand der Spezifikation, beispielsweise mit Hilfe von Mock-Objekten)

Fehlertests

: testen, ob die Verarbeitung von Fehlersituationen korrekt, d.h. wie definiert erfolgt.

Datenkonsistenztests

: testen die Auswirkung der getesteten Funktion auf die Korrektheit von Datenbeständen (Testbezeichnungen: Datenzyklustest, Wertebereichstest, Semantiktest, CRUD-Test)

Wiederinbetriebnahmetests

: testen ob ein System nach einem Abschalten oder Zusammenbruch (z. B. ausgelöst durch Stresstest) wieder in Betrieb genommen werden kann.

Interoperabilitätstests

: testen die Funktionalität bei der Zusammenarbeit voneinander unabhängiger Komponenten unter Einsatz mehrerer Komponenten.

Installationstests

: testen die Softwareinstallationsroutinen, ggfs. in verschiedenen Systemumgebungen (z. B. mit verschiedener Hardware oder unterschiedlichen Betriebssystemversionen)

Oberflächentests

: testen die Benutzerschnittstelle des Systems (z. B. Verständlichkeit, Anordnung von Informationen, Hilfefunktionen); für Dialogprogramme auch *GUI-Test oder UI-Test* genannt.

Stresstests

: sind Tests, die das Verhalten eines Systems unter Ausnahmesituationen prüfen und analysieren.

Crashtests

: sind Stresstests, die versuchen, das System zum Absturz zu bringen.

Lasttests

: sind Tests, die das Systemverhalten unter besonders hohen Speicher-, CPU-, o. ä. -Anforderungen analysieren. Besondere Arten von Last-Tests können Multi-User-Tests (viele Anwender greifen auf ein System zu, simuliert oder real) und Stresstests (dabei wird das System an die Grenzen der Leistungsfähigkeit geführt) sein.

Performance Tests

: sind Tests, die ein korrektes Systemverhalten bei bestimmten Speicher- und CPU-Anforderungen sicherstellen sollen.

Rechnernetz-Tests

testen das Systemverhalten in Rechnernetzen (z. B. Verzögerungen der Datenübertragung, Verhalten bei Problemen in der Datenübertragung).

Sicherheitstests

testen ein System gegen potentielle Sicherheitslücken.

Weitere Klassifikationen für Testarten

Aus den Qualitätsmerkmalen gem. ISO/IEC 9126 (die für die meisten Testanforderungen den Rahmen bilden können) leitet sich eine große Anzahl von Testarten ab. Auf Grund ihrer Vielfalt werden nachfolgend nur wenige Beispiele genannt: *Sicherheitstest, Funktionstest, Wiederanlauftest, GUI-Test, Fehlertest, Installationstest, Lasttest.*

Zum Testen ausgewählte methodischen Ansätze spiegeln sich ebenfalls in Testart-Bezeichnungen wieder. Das sind z. B.:

Spezielle Teststrategien: *SMART-Testing, Risk based testing, Data driven Testing, exploratives Testen, top-down / bottom-up, hardest first, big-bang.*

Besondere Methoden sind für *Entscheidungstabellentests, Use-Case- oder anwendungsfallbasierte Tests, Zustandsübergangs- / zustandsbezogene Tests, Äquivalenzklassentests und Pair-wise-Tests* die Grundlage für Testartbezeichnungen.

Testart-Bezeichnungen leiten sich u. a. auch vom **Zeitpunkt der Testdurchführung** ab:

Die bedeutendsten und meist auch im allgemeinen Sprachgebrauch benutzten Testartbezeichnungen sind die Teststufen, die mit *Modultest, Integrationstest, Systemtest, Abnahmetest* bezeichnet werden.

Als Testarten für frühe Tests sind auch bekannt: *Alpha-Test* (erste Entwicklertests), *Beta-Test* (oder Feldtest; spätere Benutzer testen eine Pre-Version der Software)

Produktionstests werden auf den Systemen der Produktionsumgebung, evtl. sogar erst im produktiven Betrieb der Software (nur für unkritische Funktionen geeignet) durchgeführt. Möglicher Grund: Nur die Produktionsumgebung verfügt über bestimmte, zum Testen erforderliche Komponenten.

Auch Test-Wiederholungen gehören zum Aspekt 'Testzeitpunkt': Diese Tests werden *Regressionstest oder Retest* etc. genannt.

Indirekt mit Zeitbezug sind zu nennen: *Entwicklertest (vor Anwendertest ...), statisches Testen (vor dynamischem Testen).*

Auch **nach der Testintensität** werden einige Testarten speziell benannt:

Unterschiedliche Grade der Testabdeckung (Test-Coverage- bzw. Code-Coverage) werden mit *Überdeckungstests* erreicht, die (auf Grund der geringen Größe der Testobjekte) besonders für *Komponententests* geeignet sind. Testartenbezeichnungen hierzu: Anweisungs- (C0-Test, C1-Test), Zweig-, Bedingungs- und Pfadüberdeckungstest.

Bewusst nach dem Zufallsprinzip werden *Smoketests* ausgeführt, frühe, oberflächliche Tests, bei denen lediglich ausprobiert werden soll, ob das Testobjekt 'überhaupt irgend etwas tut - ohne abzurauchen'. Beim *Error Guessing* provozieren (erfahrene) Tester bewusst Fehler.

Mit *Vorbereitungstests* werden vorerst nur wichtige Hauptfunktionen getestet.

Testarten werden auch danach bezeichnet, welcher **Informationsstand über die zu testenden Komponenten** vorhanden ist (auf dessen Basis Testfälle spezifiziert werden könnten):

Black-Box-Tests werden ohne Kenntnisse über den inneren Aufbau des zu testenden Systems, sondern auf der Basis von Entwicklungsdokumenten entwickelt. In der Praxis werden Black-Box-Tests meist nicht von den Software-Entwicklern, sondern von fachlich orientierten Testern oder von speziellen Test-Abteilungen oder Test-Teams entwickelt. In diese Kategorie fallen auch Anforderungstests (Requirements Tests; auf der Grundlage spezieller Anforderungen); stochastisches Testen (statistische Informationen als Testgrundlage)

White-Box-Tests, auch strukturorientierte Tests genannt, werden auf Grund von Wissen über den inneren Aufbau der zu testenden Komponente entwickelt. Entwicklertests sind i. d. R. White-Box-Tests - wenn Entwickler und Tester dieselbe Person sind. Hierbei besteht die Gefahr, dass Missverständnisse beim Entwickler auch zu 'falschen' Testfällen führen, der Fehler also nicht erkannt wird.

Grey-Box-Tests werden von den gleichen Entwicklern entwickelt wie das zu testende System selbst, gemäß den Regeln der testgetriebenen Entwicklung, also vor der Implementierung des Systems, und damit (noch) ohne Kenntnisse über das zu testende System. Auch hier gilt: Missverständnisse zur Aufgabenstellung können zu falschen Testfällen führen.

Aus der Art und dem Umfang des Testobjekts ergeben sich die folgenden Testart-Bezeichnungen:

Systemtest, Geschäftsprozesstest (Integration von Programmteilen im Geschäftsprozess), *Schnittstellentest* (zwischen 2 Programmen), *Modultest*, für einzelne Codeteile *Debugging* (Testen des Programmcodes unter schrittweiser oder abschnittsweiser Kontrolle und ggf. Modifikation des Entwicklers) und *Mutationstest*.

Batchtest werden Tests von Stapelprogrammen, *Dialogtest* Tests für Dialogprogramme genannt.

Internet- oder Intranet-Funktionen werden mit *Web-Tests* (auch Browsertest genannt) durchgeführt.

Hardwaretests sind Tests, die auf konkrete, Hardwarekomponenten betreffende Last- und andere Kriterien ausgerichtet sind - wie Netzlast, Zugriffszeit, Parallelspeichertechniken etc. Auch *Produktionstests* gehören hierzu.

Testarten können auch danach benannt sein, **wer die Tests ausführt oder spezifiziert**:

Entwicklertests (Programmierertests), Benutzertests, Anwendertests, Benutzerakzeptanztests (User Accaptence Tests - UAT) *werden von der jeweiligen Testergruppe durchgeführt.*

Abnahmetests führen die fachlich für die Software verantwortlichen Stellen aus.

Betriebstest, Installationstests, Wiederanlauftests, Notfalltests nehmen u. a. auch Vertreter des Rechenzentrums vor - zur Sicherstellung der Einsatzfähigkeit nach definierten Vorgaben.

Von eher untergeordneter Bedeutung sind Testbegriffe, die sich an der **Art der Softwaremaßnahme** orientieren, aus der der Testbedarf resultiert:

In Wartungsprojekten werden *Wartungstests und Regressionstests* ausgeführt; dabei werden i. d. R. bereits vorhandenen Testfälle und Testdaten benutzt.

In Migrationsprojekten werden *Migrationstests* durchgeführt; die Datenüberführung und spezielle Migrationsfunktionen sind hierbei z. B. Testinhalte.

Weitere Teilaspekte beim Testen

Teststrategie

Pol, Koomen und Spillner beschreiben in [1] die Teststrategie als **umfassenden Ansatz:** *Eine Teststrategie ist notwendig, da ein vollständiger Test, d. h. ein Test, der alle Teile des Systems mit allen möglichen Eingabewerten unter allen Vorbedingungen überprüft, in der Praxis nicht durchführbar ist. Deswegen muss in der Test-Planung anhand einer Risikoabschätzung festgelegt werden, wie kritisch das Auftreten eines Fehlers in einem Systemteil einzuschätzen ist (z.B. nur finanzieller Verlust oder Gefahr für Menschenleben) und wie intensiv (unter Berücksichtigung der verfügbaren Ressourcen und des Budgets) ein Systemteil getestet werden muss oder kann.*

Demnach ist in der Teststrategie festzulegen, welche Teile des Systems mit welcher Intensität unter Anwendung welcher Testmethoden und -Techniken unter Nutzung welcher Test-Infrastruktur und in welcher Reihenfolge (siehe auch Teststufen) zu testen sind.

Sie wird vom Testmanagement im Rahmen der Testplanung erarbeitet, im Testplan dokumentiert und festgelegt und als Handlungsrahmen für das Testen (durch die Testteams) zu Grunde gelegt.

Nach einer anderen Interpretation wird "Teststrategie" als **methodischer Ansatz** verstanden, nach dem das Testen angelegt wird.

So benennt z. B. ISTQB Ausprägungen für Teststrategien wie folgt:

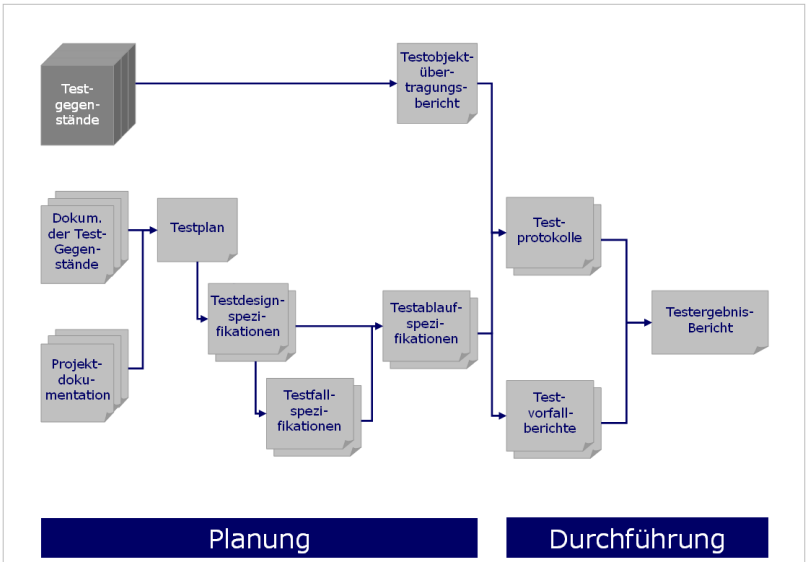
- top-down: Haupt- vor Detailfunktionen testen; untergeordnete Routinen werden beim Test zunächst ignoriert oder (mittels "Stubs") simuliert
- bottom-up: Detailfunktionen zuerst testen; übergeordnete Funktionen oder Aufrufe werden mittels "Testdriver" simuliert
- hardest first: Situationsbedingt das Wichtigste zuerst
- big-bang: Alles auf einmal

Weitere Prinzipien und Techniken für Teststrategien sind:

- Risk based testing: Testprinzip, nach dem die Testabdeckung an den Risiken ausgerichtet wird, die in den Testobjekten (für den Fall des Nichtfindens von Fehlern) eintreten können.
- Data driven Testing: Testtechnik, mit der über Einstellungen in den Testscripts die Datenkonstellationen gezielt geändert werden können, um damit mehrere Testfälle hintereinander effizient testen zu können
- Testgetriebene Entwicklung
- SMART: Testprinzip "Specific, Measurable, Achievable, Realistic, time-bound"
- Keyword driven testing
- framework based: Test-Automatisierung mittels Testwerkzeugen für bestimmte Entwicklungsumgebungen / Programmiersprachen

Dokumentation

Zur Testplanung gehört auch die Vorbereitung der Dokumentation. Eine normierte Vorgehensweise dazu empfiehlt der Standard IEEE 829.[4] [5] . Laut diesem Standard gehören zu einer vollständigen Testdokumentation folgende Unterlagen:



Zusammenhang der Dokumente und Verfahrensschritte laut IEEE 829

Testplan

> Beschreibt Umfang, Vorgehensweise, Terminplan, Testgegenstände.

Testdesignspezifikation

> Beschreibt die im Testplan genannten Vorgehensweisen im Detail.

Testfallspezifikationen

> Beschreibt die Umgebungsbedingungen, Eingaben und Ausgaben eines jeden Testfalls.

Testablaufspezifikationen

> Beschreibt in Einzelschritten, wie jeder Testfall durchzuführen ist.

Testobjektübertragungsbericht

> Protokolliert, wann welche Testgegenstände an welche Tester übergeben wurden.

Testprotokoll

Listet chronologisch alle relevanten Vorgänge bei der Testdurchführung.

Testvorfallbericht

Listet alle Ereignisse, die eine weitere Untersuchung erforderlich machen.

Testergebnisbericht

Beschreibt und bewertet die Ergebnisse aller Tests.

Testautomatisierung

Insbesondere bei Tests, die häufig wiederholt werden, ist deren Automatisierung angeraten. Dies ist vor allem bei Regressionstests und bei testgetriebener Entwicklung der Fall. Darüber hinaus kommt Testautomatisierung bei manuell nicht oder nur schwer durchführbaren Tests zum Einsatz (z.B. Lasttests).

- Durch Regressionstests wird nach Softwareänderungen meist im Zuge des System- oder Abnahmetests der fehlerfreie Erhalt der bisherigen Funktionalität überprüft.
- Bei der testgetriebenen Entwicklung werden die Tests im Zuge der Softwareentwicklung im Idealfall vor jeder Änderung ergänzt und nach jeder Änderung ausgeführt.

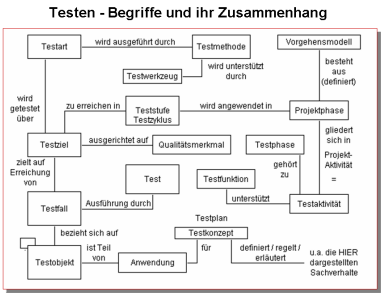
Bei nicht automatisierten Tests ist in beiden Fällen der Aufwand so groß, dass häufig auf die Tests verzichtet wird.

Übersichten / Zusammenhänge

Begriffe beim Testen

Die nebenstehende Grafik zeigt Begriffe, die im Kontext 'Testen' auftreten - und wie sie mit anderen Begriffen in Verbindung stehen.

Zur Erläuterung: Die Grafik ist optisch ein ER-Diagramm, sie zeigt aber keine Datenobjekte, sondern nur *Begriffe*. Der den Zusammenhang beschreibende Text ist aus Sicht des 'ersten' Begriffs formuliert und steht links von der Linie, die (aus seiner Sicht) zum zweiten Begriff führt.



Begriffe und ihr Zusammenhang

Schnittstellen beim Testen

Die Grafik zeigt die wichtigsten Schnittstellen, die beim Testen auftreten. Zu den von Thaller[6] genannten 'Partnern' beim Testen wird nachfolgend beispielhaft angeführt, WAS jeweils kommuniziert / ausgetauscht wird.

Wichtige Schnittstellen beim Testen

- Projektmanagement: Termin- und Aufwandsrahmen, Status je Testobjekt ('testready'), Dokumentationssysteme
- Linienmanagement (und Linienabteilung): Fachlicher Support, Testabnahme, fachliche Tester stellen
- Rechenzentrum: Testumgebung(en) und Testwerkzeuge bereitstellen und betreiben
- Datenbankadministrator: Testdatenbestände installieren, laden und verwalten
- Konfigurations Management: Testumgebung einrichten, Integrieren der neuen Software

- Entwicklung: Test-Basisdokumente, Prüflinge, Support zum Testen, Testergebnisse erörtern
- Problem- und CR-Management: Fehlermeldungen, Rückmeldung zum Retest, Fehlerstatistik
- Lenkungsausschuss: Entscheidungen zur Test(stufen)abnahme oder zum Testabbruch

Einzelnachweise

[1] M. Pol, T. Koomen, A. Spillner: *Management und Optimierung des Testprozesses.* dpunkt.Verlag, Heidelberg 2002, ISBN 3-89864-156-2.
[2] Ernst Denert: *Software-Engineering.* Springer, Berlin 1991, 1992. ISBN 3-540-53404-0
[3] Peter Liggesmeyer: *Software-Qualität - Testen, Analysieren und Verifizieren von Software.* Spektrum Akademischer Verlag, Heidelberg/Berlin 2002, ISBN 3-8274-1118-1, S. 34.
[4] IEEE Standard for Software Test Documentation (IEEE Std. 829-1998)
[5] IEEE Standard for Software and System Test Documentation (IEEE Std. 829-2008)
[6] Georg-Edwin Thaller: *Softwaretest*

Literatur

- Andreas Spillner, Tilo Linz: *Basiswissen Softwaretest. Aus- und Weiterbildung zum Certified Tester - Foundation Level nach ISTQB-Standard* 4. Aufl. dpunkt.verlag, Heidelberg 2010, ISBN 978-3-89864-642-0.
- Harry Sneed, Manfred Baumgartner, Richard Seidl: *Der Systemtest - Anforderungsbasiertes Testen von Software-Systemen.* Carl Hanser, München/Wien 2006, ISBN 3-446-40793-6.
- Georg Erwin Thaller: *Software-Test. Verifikation und Validation.* Heise, Hannover 2002, ISBN 3-88229-198-2.

Hauptteil

Testfall

Ein **Testfall** (engl. **Test case**) beschreibt einen elementaren, funktionalen Softwaretest, der der Überprüfung einer z. B. in einer Spezifikation zugesicherten Eigenschaft eines Testobjektes dient. Ein Testfall wird mittels Testmethoden erstellt.

Wichtige Bestandteile der Beschreibung eines Testfalls sind:

1. die Vorbedingungen, die vor der Testausführung hergestellt werden müssen,
2. die Eingaben/Handlungen, die zur Durchführung des Testfalls notwendig sind,
3. die erwarteten Ausgaben/Reaktionen des Testobjektes auf die Eingaben,
4. die erwarteten Nachbedingungen, die als Ergebnis der Durchführung des Testfalls erzielt werden.
5. die Prüfanweisungen, d. h. wie Eingaben an das Testobjekt zu übergeben sind und wie Sollwerte abzulesen sind

Weichen trotz Einhaltung der Vorbedingungen und trotz korrekter Eingaben/Handlungen in der Testdurchführung die Ausgaben oder Nachbedingungen während der Testdurchführung von den erwarteten Werten ab, so liegt eine Anomalie vor (das kann z. B. ein zu behebender Mangel oder Fehler sein.)

Testfälle lassen sich in *Positiv-Testfälle* und *Negativ-Testfälle* unterteilen:

1. In Positivtests wird das Verhalten des Testobjekts in Folge gültiger Vorbedingungen und Eingaben überprüft.
2. In Negativtests (auch Robustheitstest genannt) wird das Verhalten des Testobjekts in Folge ungültiger Vorbedingungen und/oder Eingaben überprüft.

Durch Variation der Eingabewerte und von Parametern der Vorbedingungen lassen sich verschiedene Varianten eines Testfalls überprüfen. Hier sind insbesondere die Grenzwerte der Eingabewerte und Parameter von Interesse (siehe Grenzwertanalyse), deren Erreichen oder Überschreiten ein anderes Verhalten des Testobjektes erwarten lässt.

Testarten

Bananenprinzip

Das **Bananenprinzip** ist ein sarkastischer Ausdruck, der die Hoffnung ausdrückt, das noch unreife (sprich mangelhafte) Produkt könne beim Verbraucher reifen. Grundlage ist die Tatsache, dass Bananen unreif geerntet, grün ausgeliefert und erst nach einer Reifezeit beim Zwischenhändler oder gar beim Endverbraucher genießbar werden.

Positiv wird der Begriff in der Qualitätskontrolle bezeichnet; im Allgemeinen hat er aber eine negative Bedeutung.

Der Ausdruck wird vorrangig in Branchen verwendet, die mit Kraftfahrzeugen, Computerbauteilen oder Software handeln.

Eine solche Software wird von den Benutzern auch als *Bananenware*, *Bananen-Software* oder (engl.) *Bananaware* bezeichnet. Dabei werden die Programme gleich an die Endverbraucher verkauft, ohne ausreichende Fehlersuche in Betatests – oder gar ohne jemals einen Alphatest durchgeführt zu haben. Die betreffenden Hersteller weisen oft darauf hin, dass es unmöglich sei, alle Endanwender-Konfigurationen auf dadurch möglicherweise auftretende Fehler zu testen. Die Behebung durch Aktualisierungen nach Problembeschreibungen sei daher nicht nur kostengünstiger, sondern auch die einzig realistische Möglichkeit. Der Vorgang wird von den Herstellern daher auch als *„kundenseitige Anpassung“* bezeichnet. Letztlich finanziert der Kunde aber durch seinen Kauf die Entwicklung des eingeschränkt nutzbaren Produktes mit.

Die Anwender können die bereits bezahlte Software oft nicht oder nur eingeschränkt nutzen. Sie müssen nach der Beschwerde beim Hersteller auf die Aktualisierung warten, dafür manchmal extra zahlen und bei einem verbesserten Programm Daten erneut einpflegen. Zudem müssen bei durch den Kunden erweiterbaren Produkten möglicherweise Eigenentwicklungen an die aktualisierte Version angepasst werden.

Weblinks

- Glossar auf projektmagazin.de [1]

Referenzen

[1] http://www.projektmagazin.de/glossar/gl-0874.html

Positivtest

Der **Positivtest** (auch **Verifizierender Test** oder **Gut-Test** genannt) will beweisen, dass eine Anforderung an die Anwendung fehlerfrei ausführbar ist. Der Testfall prüft also die korrekte Verarbeitung bei korrekter Handhabung ab.

Zweck des Positivtests ist es nachzuweisen, dass die Anwendung bei richtiger Bedienung das tut, was sie tun soll.

Mit einem Positivtestfall wird das korrekte Verhalten der Anwendung und des Anwenders geprüft. Dabei werden ausschließlich gültige Werte eingegeben, Masken werden prinzipiell korrekt und vollständig ausgefüllt, Schnittstellen werden korrekt beliefert.

Eine Erweiterung des Positivtests ist der Negativtest.

Literatur

- Manuel Arnold: *Soft(ware) Skills des Testens: Irren ist menschlich.* Javamagazin ISSN 1619-795X [1] 13. Jg., H. 5, 2010, S. 84

Referenzen

[1] http://dispatch.opac.d-nb.de/DB=1.1/CMD?ACT=SRCHA&IKT=8&TRM=1619-795X

Negativtest

Der **Negativtest** (auch **Provokationstest, Robustheitstest, Falsifizierender Test** oder **Schlecht-Test** genannt) ist eine Erweiterung des Positivtests. Der Negativtest prüft, ob die Anwendung auf eine (falsche) Eingabe oder Bedienung, die nicht den Anforderungen an die Anwendung entspricht, erwartungsgemäß (also ohne Programmabbruch) reagiert, z. B. durch eine Fehlermeldung. Beim Negativtest werden absichtlich ungültige Werte eingegeben, Masken werden nicht oder nur unvollständig ausgefüllt, Schnittstellen werden mit falschen Werten beliefert oder die Datenbank wird abgeklemmt. Der Testfall prüft also auf "korrekte" Verarbeitung bei fehlerhafter Handhabung ab.

Zweck des Negativtests ist es nachzuweisen, dass die Anwendung robust auf Bedienfehler reagiert. Damit wird sichergestellt, dass die Sicherheit der Anwendung gegen falsche Bedienung und technische Störungen gewährleistet ist.

Der Negativtestfall ist bereits erfüllt, wenn eine Fehlermeldung erscheint (z. B. "Eingabe ist falsch"). Dieser sagt aber nichts über die Benutzerfreundlichkeit der Fehlermeldung aus. Benutzerfreundlich wäre eine Fehlermeldung wie "Das Datum muss in der Zukunft liegen" oder "Das Datum muss größer sein als das Beginn-Datum" oder "Der Wert muss größer als 0,00 und kleiner als 500,01 sein".

Literatur

- Manuel Arnold: *Soft(ware) Skills des Testens: Irren ist menschlich.* Javamagazin ISSN 1619-795X [1] 13. Jg., H. 5, 2010, S. 84

Black-Box-Test

Black-Box-Test bezeichnet eine Methode des Softwaretests, bei der die Tests ohne Kenntnisse über die innere Funktionsweise des zu testenden Systems entwickelt werden. Er beschränkt sich auf funktionsorientiertes Testen, d. h. für die Ermittlung der Testfälle werden nur die Anforderungen, aber nicht die Implementierung des Testobjekts herangezogen. Die genaue Beschaffenheit des Programms wird nicht betrachtet, sondern vielmehr als Black Box behandelt. Nur nach außen sichtbares Verhalten fließt in den Test ein.

Zielsetzung

Ziel ist es, die Übereinstimmung eines Softwaresystems mit seiner Spezifikation zu überprüfen. Ausgehend von formalen oder informalen Spezifikationen werden Testfälle erarbeitet, die sicherstellen, dass der geforderte Funktionsumfang eingehalten wird. Das zu testende System wird dabei als Ganzes betrachtet, nur sein Außenverhalten wird bei der Bewertung der Testergebnisse herangezogen.

Testfälle aus einer informalen Spezifikation abzuleiten, ist vergleichsweise aufwändig und je nach Präzisionsgrad der Spezifikation u. U. nicht möglich. Oft ist daher ein vollständiger Black-Box-Test ebenso wenig wirtschaftlich wie ein vollständiger White-Box-Test.

Auch ist ein erfolgreicher Black-Box-Test keine Garantie für die Fehlerfreiheit der Software, da in frühen Phasen des Softwareentwurfs erstellte Spezifikationen spätere Detail- und Implementationsentscheidungen nicht abdecken.

Die außerdem existierenden Grey-Box-Tests sind ein Ansatz aus dem Extreme Programming, mit Hilfe testgetriebener Entwicklung die gewünschten Vorteile von Black-Box-Tests und White-Box-Tests weitgehend miteinander zu verbinden und gleichzeitig die unerwünschten Nachteile möglichst zu eliminieren.

Black-Box-Tests verhindern, dass Programmierer Tests „um ihre eigenen Fehler herum“ entwickeln und somit Lücken in der Implementierung übersehen. Ein Entwickler, der Kenntnisse über die innere Funktionsweise eines Systems besitzt, könnte unabsichtlich durch gewisse zusätzliche Annahmen, die außerhalb der Spezifikation liegen, einige Dinge in den Tests vergessen oder anders als die Spezifikation sehen. Als weitere nützliche Eigenschaft eignen sich Black-Box-Tests auch als zusätzliche Stütze zum Überprüfen der Spezifikation auf Vollständigkeit, da eine unvollständige Spezifikation häufig Fragen bei der Entwicklung der Tests aufwirft.

Weil die Entwickler der Tests keine Kenntnisse über die innere Funktionsweise des zu testenden Systems haben dürfen, ist bei Black-Box-Tests praktisch ein separates Team zur Entwicklung der Tests nötig. In vielen Unternehmen sind dafür sogar spezielle Testabteilungen zuständig.

Vergleich mit White-Box-Tests

Black-Box-Tests werden eingesetzt um Fehler gegenüber der Spezifikation aufzudecken, sind aber kaum geeignet, Fehler in bestimmten Komponenten oder gar die fehlerauslösende Komponente selbst zu identifizieren. Für letzteres benötigt man White-Box-Tests. Zu bedenken ist auch, dass zwei Fehler in zwei Komponenten sich zu einem vorübergehend scheinbar korrekten Gesamtsystem aufheben könnten. Dies kann durch White-Box-Tests leichter aufgedeckt werden, bei Black-Box-Tests nach der nicht auszuschließenden Korrektur nur einer der beiden Fehler jedoch als vermeintliche Regression zu Tage treten.

Im Vergleich zu White-Box-Tests sind Black-Box-Tests wesentlich aufwendiger in der Durchführung, da sie eine größere organisatorische Infrastruktur (eigenes Team) benötigen.

Die Vorteile von Black-Box-Tests gegenüber White-Box-Tests:

- bessere Verifikation des Gesamtsystems
- Testen von semantischen Eigenschaften bei geeigneter Spezifikation
- Portabilität von systematisch erstellten Testsequenzen auf plattformunabhängige Implementierungen

Die Nachteile von Black-Box-Tests gegenüber White-Box-Tests:

- größerer organisatorischer Aufwand
- zusätzlich eingefügte Funktionen bei der Implementierung werden nur durch Zufall getestet
- Testsequenzen einer unzureichenden Spezifikation sind unbrauchbar

Zudem sei genannt, dass die Unterscheidung Black-Box-Test vs. White-Box-Test teilweise von der Perspektive abhängt. Das Testen einer Teilkomponente ist aus Sicht des Gesamtsystems ein White-Box-Test, da für das Gesamtsystem aus der Außenperspektive keine Kenntnisse über den Systemaufbau und damit die vorhandenen Teilkomponenten vorliegen. Aus Sicht der Teilkomponente wiederum kann derselbe Test unter Umständen als Black-Box-Test betrachtet werden, wenn er ohne Kenntnisse über die Interna der Teilkomponente entwickelt und durchgeführt wird.

Auswahl der Testfälle

Die Anzahl der Testfälle einer systematisch erstellten Testsequenz, auf Basis einer geeigneten Spezifikation, ist in fast allen Anwendungen für die Praxis zu hoch. Es gibt z.B. folgende Möglichkeiten, diese systematisch zu verringern:

- Grenzwerte und spezielle Werte,
- Äquivalenzklassenmethode, Klassifikationsbaum-Methode,
- (simplifizierte) Entscheidungstabellen
- zustandsbezogene Tests,
- use case Tests,
- Ursache- und Wirkungsgrad
- Auffinden von Robustheits- und Sicherheitsproblemen Fuzzing
- Risikoanalyse bzw. Priorisierung der Anwendung der gewünschten Ergebnisse (wichtige bzw. unwichtige Funktionen).

Im Gegensatz dazu kann die Reduktion auch auf intuitive Weise (Error Guessing) durchgeführt werden. Von dieser Methode sollte allerdings Abstand genommen werden, da hier immer unbewusst Annahmen berücksichtigt werden, die sich bei der späteren Anwendung der Applikation als negativ herausstellen können. Es gibt aber auch andere erfolgreiche Testrichtungen, die sehr erfolgreich damit sind. Vertreter sind z.B. James Bach [1] mit Rapid Testing oder Cem Kaner [2] mit Exploratory Testing (Ad-hoc-Test). Diese Testarten sind also den erfahrungsbasierten oder auch unsystematischen Techniken zuzuordnen. Dazu gehört auch schwachstellenorientiertes Testen.

Repräsentatives Testen

Alle Funktionen werden entsprechend der Häufigkeit, mit der sie später im Einsatz sein werden, getestet.

Schwachstellen-orientiertes Testen

Man beschränkt sich häufig nur auf intensives Testen jener Funktionen, bei denen die Wahrscheinlichkeit eines Auftretens von Fehlern hoch ist (komplexe Algorithmen, Teile mit ungenügender Spezifikation, Teile von unerfahrenen Programmierern, ...). Intensivere Tests können mit Fuzzing-Werkzeugen durchgeführt werden, da diese eine weitestgehende Automatisierung der Robustheits- und Schwachstellen-Tests erlauben. Die Ergebnisse dieser Tests sind dann Informationen über Datenpakete, die das SUT (System under Test) kompromittieren können. Schwachstellen-Tests können z. B. durch Vulnerability Scanner oder Fuzzer durchgeführt werden.

Schadensausmaß-orientiertes Testen (Risikoanalyse)

Man beschränkt sich auf intensives Testen von Funktionen, bei denen Fehler besonders gravierende Folgen haben können (z. B. Verfälschung oder Zerstörung einer umfangreichen Datei / Lebensgefahr für Personen (KFZ, Maschinensteuerungen) etc..).

Diese werden priorisiert oder klassifiziert (1,2,3,...) und dann entsprechend dieser Ordnung getestet.

Literatur

- BCS SIGIST (British Computer Society Specialist Interest Group in Software Testing): *Standard for Software Component Testing* [3], Working Draft 3.4, 27. April 2001.

Referenzen

[1] http://satisfice.com/

[2] http://kaner.com/

[3] http://www.testingstandards.co.uk/BS7925_3_4.zip

Grey-Box-Test

Grey-Box-Test ist eine Methode des Softwaretests, welche mit Hilfe testgetriebener Entwicklung (siehe auch Extreme Programming) die Vorteile von Black-Box-Tests und White-Box-Tests miteinander verbinden soll.

Der Grey-Box-Test hat mit dem White-Box-Test gemeinsam, dass er ebenfalls von den gleichen Entwicklern wie das zu testende System geschrieben wird. Mit dem Black-Box-Test teilt er sich anfänglich die Unkenntnis über die Interna des zu testenden Systems, weil der Grey-Box-Test vor dem zu testenden System geschrieben wird (Test-First-Programmierung).

Somit können Teilkomponenten und Gesamtsysteme mit dem geringen organisatorischen Aufwand der White-Box-Tests geprüft werden, ohne eventuell „um Fehler herum“ zu testen. Grey-Box-Tests erfordern als Bestandteil agiler Prozesse hohe Disziplin oder weitere Prozessmethoden des Software-Engineerings wie zum Beispiel Paarprogrammierung oder Akzeptanztests, um praktikabel und erfolgreich einsetzbar zu sein. Andernfalls könnten sich Grey-Box-Tests als fatal erweisen. Grey-Box-Tests sollten nicht unbedacht als vollwertiger Ersatz für Black-Box-Tests gesehen werden.

Ohne die stützenden Säulen agiler Prozesse sollte beim Einsatz von Grey-Box-Tests keinesfalls auf die üblichen Black-Box-Tests verzichtet werden. Grey-Box-Tests sollten als qualitative Verbesserung von White-Box-Tests betrachtet werden.

White-Box-Test

Der Begriff **White-Box-Test** (auch **Glass-Box-Test**) bezeichnet eine Methode des Software-Tests, bei der die Tests mit Kenntnissen über die innere Funktionsweise des zu testenden Systems entwickelt werden. Im Gegensatz zum Black-Box-Test ist für diesen Test also ein Blick in den Quellcode gestattet, d. h. es wird am Code geprüft.

Ein Beispiel für einen White-Box-Test ist ablaufbezogenes Testen (Kontrollflussorientierte Testverfahren), bei welchem der Ablaufgraph im Vordergrund steht. Ziel des Tests ist es, sicherzustellen, dass Testfälle in Bezug auf die Überdeckung des Quellcodes gewisse *Hinlänglichkeitskriterien* erfüllen. Gängig sind dabei u. a. folgende Maße (bzw. Qualitätskriterien):

- *Zeilenüberdeckung*: Ausführung aller Quellcode-Zeilen
- *Anweisungsüberdeckung*: Ausführung aller Anweisungen
- *Zweigüberdeckung* bzw. *Kantenüberdeckung*: Durchlaufen aller möglichen Kanten von Verzweigungen des Kontrollflusses
- *Bedingungsüberdeckung* bzw *Termüberdeckung* (mehrere Varianten): Durchlaufen aller möglichen ausschlaggebenden Belegungen bei logischen Ausdrücken in Bedingungen
- *Pfadüberdeckung* (mehrere Varianten): Betrachtung der Pfade durch ein Modul

Die Zahl der benötigten Testfälle für die einzelnen Maße unterscheidet sich z.T. deutlich. Kantenüberdeckung wird im Allgemeinen als minimales Testkriterium angesehen. Je nach Art und Struktur der zu testenden Software können andere Maße für ein System als Ganzes oder für Module sinnvoll sein.

Selbst wenn ein Softwaresystem in Bezug auf ein Hinlänglichkeitskriterium erfolgreich getestet wurde, schließt das nicht aus, dass es Fehler enthält. Dies liegt in der Natur des White-Box-Tests begründet und kann eine der folgenden Ursachen haben:

- Der White-Box-Test leitet Testfälle nicht aus der Spezifikation des Programms her, sondern aus dem Programm selbst. Getestet werden kann nur die Korrektheit eines Systems, nicht, ob es eine geforderte Semantik erfüllt.
- Auch wenn alle Programmpfade getestet worden sind, bedeutet dies nicht, dass ein Programm fehlerfrei arbeitet. Der Fall, dass im Graphen des Kontrollflusses Kanten fehlen, wird nicht erkannt.

Zusammenfassend kann man sagen, dass White-Box-Tests alleine als Testmethodik nicht ausreichen. Eine sinnvolle Testreihe sollte Black-Box-Tests und White-Box-Tests kombinieren. Nach der Überdeckungsmessung der Testfälle des Black-Box-Tests (durch ein geeignetes Werkzeug) werden durch Betrachten der nicht überdeckten Codeteile neue Testfälle aufgestellt, um die Überdeckung zu erhöhen.

Will man ein System auch in seinen Teilsystemen testen, benötigt man dazu Kenntnisse über die innere Funktionsweise des zu testenden Systems. White-Box-Tests eignen sich besonders gut, um in Erscheinung getretene Fehler zu lokalisieren, d. h. die fehlerverursachende Komponente zu identifizieren und als Regressionstest ein Wiederauftreten des Fehlers bereits in der Komponente zu vermeiden.

Weil die Entwickler der Tests Kenntnisse über die innere Funktionsweise des zu testenden Systems besitzen müssen, werden White-Box-Tests von demselben Team, häufig sogar von denselben Entwicklern entwickelt wie die zu testenden Komponenten. Spezielle Testabteilungen werden für White-Box-Tests in der Regel nicht eingesetzt, da der Nutzen speziell für diese Aufgabe abgestellter Tester meist durch den Aufwand der Einarbeitung in das System eliminiert wird.

Vergleich mit Black-Box-Tests

White-Box-Tests werden eingesetzt, um Fehler in den Teilkomponenten aufzudecken und zu lokalisieren, sind aber aufgrund ihrer Methodik kein geeignetes Werkzeug, Fehler gegenüber der Spezifikation aufzudecken. Für letzteres benötigt man Black-Box-Tests. Zu bedenken ist auch, dass zwei Komponenten, die für sich genommen korrekt gemäß ihrer jeweiligen Teilspezifikation arbeiten, zusammen nicht zwangsläufig eine korrekte Einheit gemäß der Gesamtspezifikation bilden. Dies kann durch Black-Box-Tests leichter festgestellt werden als durch White-Box-Tests.

Im Vergleich zu Black-Box-Tests sind White-Box-Tests wesentlich einfacher in der Durchführung, da sie keine besondere organisatorische Infrastruktur benötigen.

Vorteile von White-Box-Tests gegenüber Black-Box-Tests

- Testen von Teilkomponenten und der internen Funktionsweise
- Geringerer organisatorischer Aufwand
- Automatisierung durch gute Tool-Unterstützung

Nachteile von White-Box-Tests gegenüber Black-Box-Tests

- Erfüllung der Spezifikation nicht überprüft
- Eventuell Testen „um Fehler herum“

Grey-Box-Tests sind ein Ansatz aus dem Extreme Programming, mit Hilfe testgetriebener Entwicklung die gewünschten Vorteile von Black-Box-Tests und White-Box-Tests weitgehend miteinander zu verbinden und gleichzeitig die unerwünschten Nachteile möglichst zu eliminieren.

Zudem sei genannt, dass die Unterscheidung zwischen Black-Box-Test und White-Box-Test teilweise von der Perspektive abhängt. Das Testen einer Teilkomponente ist aus Sicht des Gesamtsystems ein White-Box-Test, da für das Gesamtsystem aus der Außenperspektive keine Kenntnisse über den Systemaufbau und damit die vorhandenen Teilkomponenten vorliegen. Aus Sicht der Teilkomponente wiederum kann derselbe Test unter Umständen als Black-Box-Test betrachtet werden, wenn er ohne Kenntnisse über die Interna der Teilkomponente entwickelt und durchgeführt wird.

Literatur

- Andreas Spillner, Tilo Linz: *Basiswissen Softwaretest - Basiswissen Softwaretest: Aus- und Weiterbildung zum Certified Tester: Foundation Level nach ISTQB-Standard.* 3., überarbeitete und aktualisierte Auflage, dpunkt.verlag GmbH, Heidelberg 2005, ISBN 3-89864-358-1.
- Lee Copeland: *A Practitioner's Guide to Software Test Design.* first printing, Artech House Publishers, Norwood MA, USA 2003, ISBN 1-58053-791-X.
- BCS SIGIST (British Computer Society Specialist Interest Group in Software Testing): *Standard for Software Component Testing* [3], Working Draft 3.4, 27. April 2001.

Code-Walkthrough

Code-Walkthrough bezeichnet das Durchdenken eines Problems in Bezug auf Computerprogramme in der Entwicklung. Er wird häufig in der Softwareindustrie gebraucht und beschreibt das Verifizieren von Algorithmen und Quellcode. Bei diesem Verifizieren folgt man Pfaden durch den Algorithmus bzw. Quellcode, die durch die Vorbedingungen und die möglichen, vom Nutzer getroffenen Entscheidungen festgelegt sind.

Der Zweck eines Walkthroughs ist es festzustellen, ob der Algorithmus bzw. Quellcode den gestellten Anforderungen genügt.

Modellbasiertes Testen

Modellbasiertes Testen (MBT) ist ein Oberbegriff für die Nutzung von Modellen zur

- Automatisierung von Testaktivitäten
- Generierung von Testartefakten im Testprozess.

Darunter fällt insbesondere die Generierung von Testfällen aus Modellen (z.B. unter Verwendung der UML), die das Sollverhalten des zu testenden Systems beschreiben.

Ziele und Nutzen

Hauptziel ist es, nicht nur die Durchführung von Tests (siehe Testautomatisierung), sondern schon deren Erstellung zu (teil-)automatisieren. Man verspricht sich davon Transparenz und Steuerbarkeit in der Testfallentstehung, wodurch der Testprozess wirtschaftlicher und die Testqualität personenunabhängiger gestaltet werden kann.

Modellkategorien für MBT

Beim MBT wird die zu testende Software oder ihre Umgebung (z.B. in Form von Nutzungsprofilen) oder eben der Test selbst als Verhaltens- und/oder Strukturmodell dargestellt. [Roßner 2010] unterscheidet folglich zwischen

- Systemmodellen
- Umgebungsmodellen
- Testmodellen

im MBT-Einsatz.

Systemmodelle beschreiben Anforderungen an das Softwaresystem und können in Form von Analyse- oder Design-Modellen vorliegen. Sie fokussieren i.A. nicht den Test und beinhalten deshalb insbesondere keine Testdaten (i.S. einer Stichprobe aus allen möglichen Eingabedaten in das System). Folglich kann eine Generierung auf solchen Modellen maximal zu abstrakten Testfällen (d.h. ohne Angabe konkreter Werte für Testdaten) führen.

Testmodelle können aus vorhandenen Systemmodellen entstehen, bieten aber mehr Möglichkeiten für den Test. Sie beschreiben den Test eines Systems und bilden Testentscheidungen, die ein Tester vielleicht nur „im Kopf“ gefällt hat, explizit ab. So können in ihnen nicht nur Abläufe von Testschritten, sondern Testdaten, Prüfschritte und ggf. Testorakel modelliert werden. Als Folge wird es möglich, aus ihnen nicht nur abstrakte, sondern konkrete, ja sogar vollständige und ausführbare Testfälle zu generieren. Verwendet man in Testmodellen eine schlüsselwortbasierte Notation für Testschritte (siehe Keyword-Driven Testing), können sogar automatisiert ausführbare Testfälle entstehen.

Testmodelle setzen aber bei den Testern Kenntnisse in der Erstellung von Modellen voraus. Diese Skillanforderung kann entscheidend für den Erfolg einer MBT-Einführung sein.

MBT-Ausprägungen

Je nachdem, welche Modellkategorien zum Einsatz kommen und welche Rolle die Modelle im modellbasierten Testprozess spielen, kann man von unterschiedlichen Ausprägungen und Reifegraden von MBT sprechen. [Roßner 2010] definiert die folgenden:

- Beim *modellorientierten Testen* dienen Modelle als Leitfaden und Grundlage für das Testdesign, ohne dass zwingend Generatoren zum Einsatz kommen müssen. Da Modelle im Vergleich zu natürlichsprachlichen Anforderungsspezifikationen deutliche Qualitätsvorteile mit sich bringen können, profitiert schon hier die Testqualität.
- *Modellgetriebenes Testen* bezeichnet den Einsatz von Generatoren, um Testartefakte, insb. Testfälle, aus Modellen zu generieren. Häufig handelt es sich hierbei aber um eine unidirektionale Werkzeug-Einbahnstraße, bei der z.B. die Testergebnisse nicht ins Modell zurück übertragen werden.
- Werden alle relevanten Testinformationen in Modellform gepflegt und die Werkzeuge zu einem Ring verbunden, kann von *modellzentrischem Testen* gesprochen werden.

MBT-Werkzeuge

Je nachdem, welche Modelle Grundlage der Testgenerierung sein sollen und wie weit die Generierung reichen soll, ist der Einsatz von kommerziellen Generatoren oder die Eigenentwicklung eines Generators vorzusehen. Eine Übersicht über kommerzielle MBT-Werkzeuge ist zu finden in [Götz 2009]. Darin findet man folgende Taxonomie für MBT-Werkzeuge:

- *Modellbasierte Testdatengeneratoren* sind Werkzeuge, die basierend auf einem Modell der Eingangs- und Ausgangsdaten des Testobjekts und bestimmten Steuerinformationen Testdaten für die Erstellung von abstrakten und/oder konkreten Testfällen erzeugen. (Beispiel: CTE, siehe Klassifikationsbaum-Methode)
- *Modellbasierte Testfalleditoren* sind Werkzeuge, die basierend auf einem abstrakten Modell von Testfällen konkrete Repräsentationen des Testfalls zur manuellen Durchführung bzw. Testskripte zur automatischen Testdurchführung erzeugen.
- *Modellbasierte Testfallgeneratoren* sind Werkzeuge, die basierend auf einem Modell des Systemverhaltens, der Systemumgebung oder des Tests sowie bestimmter Steuerinformationen mehrere (logisch zusammengehörende) Testfälle bzw. Testskripte automatisch nach konfigurierbaren Abdeckungskriterien erzeugen.

Eine andere und komplexere Taxonomie ist zu finden in [Utting 2007].

In der Regel wird es nötig sein, die Testabdeckung beim Generiervorgang steuern zu können. Typischerweise finden hierbei die aus den White-Box-Test-Verfahren bekannten graphenbasierten Abdeckungsmaße Verwendung (siehe Kontrollflussorientierte Testverfahren), aber es sind auch andere Abdeckungsstrategien denkbar (z.B. aufgrund von Risikoinformationen im Modell).

Vor der Generierung ist es i.A. unverzichtbar, nicht nur den Generator zu testen, sondern die zu verarbeitenden Modelle geeigneten QS-Maßnahmen zuzuführen. Dazu zählen z.B. Reviews, werkzeuggestützte Prüfungen gegen Metamodelle oder Model Checking.

Literatur

- [Baker 2008] Baker, P.; Dai, Z. R.; Grabowski, J.; Haugen, Ø.; Schieferdecker, I.; Williams, C.: "Model-Driven Testing – Using the UML Testing Profile". Springer-Verlag, Berlin, 2008, ISBN 3-642-09159-8
- [Eckardt 2009] Eckardt, T.; Spijkerman, M.: "Modellbasiertes Testen auf Basis des fundamentalen Testprozesses". Beitrag zur TAV 28 in Dortmund, 2009
- [Götz 2009] Götz, H.; Nickolaus, M.; Roßner, T.; Salomon, K.: "iX-Studie Modellbasiertes Testen". Heise Zeitschriften Verlag, Hannover, 2009
- [Güldali 2010] Güldali, B.; Jungmayr, S.; Mlynarski, M.; Neumann, S.; Winter, M.: "Starthilfe für modellbasiertes Testen: Entscheidungsunterstützung für Projekt- und Testmanager". OBJEKTspektrum 3/10, S. 63-69, 2010
- [Pretschner 2006] Pretschner, A.: "Zur Kosteneffektivität modellbasierten Testens". Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme II, Braunschweig, 2006
- [Roßner 2010] Roßner, T.; Brandes, C.; Götz, H.; Winter, M.: "Basiswissen Modellbasierter Test". dpunkt-Verlag, 2010, ISBN 3-898-64589-4
- [Schieferdecker 2007] Schieferdecker, I.: "Modellbasiertes Testen". OBJEKTspektrum 3/07, S. 39-45, 2007
- [Utting 2006] Utting, M.; Pretschner, A.; Legeard, B.: "A Taxonomy of Model-Based Testing". Working Paper 4/2006, University of Waikato, 2006
- [Utting 2007] Utting, M.; Legeard, B.: "Practical Model-Based Testing – A Tools Approach". Morgan Kaufmann Publ., Amsterdam, 2007, ISBN 0-123-72501-1
- Justyna Zander, Ina Schieferdecker, Pieter J. Mosterman: *Model-Based Testing for Embedded Systems (Computational Analysis, Synthesis, and Design of Dynamic Systems)*, CRC Press 2011, ISBN 1439818452.

Weblinks

- Starthilfe für modellbasiertes Testen: Entscheidungsunterstützung für Projekt- und Testmanager [1] in OBJEKTspektrum 03/2010 (PDF-Datei; 856 kB)
- Modellgetriebene Testentwicklung [2] Ausführlichere Darstellung Modellgetriebener Testentwicklung (Model Driven Test Development, MDTD). (PDF-Datei; 7,39 MB)
- Modellbasiertes Testen automobiler Steuergeräte [3] In: ICST, pp.485-493, 2008 International Conference on Software Testing, Verification, and Validation, 2008. (PDF-Datei; 228 kB)
- Download von oAW-Test auf der Basis von MDTD [4]
- Flexibilität beim modellbasierten Testen [5] mit Enterprise Architect von www.sparxsystems.de (PDF-Datei; 921 kB)

Referenzen

[1] http://www.sigs-datacom.de/fileadmin/user_upload/zeitschriften/os/2010/03/gueldali_OS_03_10.pdf
[2] http://www.sigs.de/publications/os/2006/03/kunz_sensler_OS_03_06.pdf
[3] http://www.piketec.com/downloads/papers/Kraemer2008-Model_based_testing_of_automotive_systems.pdf
[4] http://www.mdtd.de
[5] http://www.sigs.de/publications/os/2009/Testing/kargl_OS_testing_09.pdf

Dynamisches Software-Testverfahren

Dynamische Software-Testverfahren sind bestimmte Prüfmethoden um beim Softwaretest Fehler in Software aufzudecken.

Während bei statischen Verfahren die zu testende Software nicht ausgeführt wird, setzen dynamische Verfahren die Ausführbarkeit der Software voraus. Grundprinzip der dynamischen Verfahren ist die Ausführung der zu testenden Software mit systematisch festgelegten Eingabedaten (Testfälle). Für jeden Testfall werden zu den Eingabedaten auch die erwarteten Ausgabedaten angegeben. Die vom Testlauf erzeugten Ausgabedaten werden mit den jeweils erwarteten Daten verglichen. Bei Abweichungen liegt ein Fehler vor.

Wesentliche Aufgabe der einzelnen Verfahren ist die Bestimmung geeigneter Testfälle für den Test der Software.

Die dynamischen Verfahren lassen sich wie folgt kategorisieren:

Funktionsorientierte Testmethoden

Funktionsorientierte Testmethoden werden zur Bestimmung von Testfällen benutzt, mit denen geprüft werden soll, inwieweit der Prüfling (auch Testling, Testobjekt oder Testgegenstand genannt) die vorgegebenen Spezifikationen erfüllt. Man spricht auch davon, dass „gegen die Spezifikationen“ getestet wird. Je nach Testling und Testart sind die Spezifikationen dabei von unterschiedlicher Art. Beim Modultest wird z. B. gegen die Modulspezifikation getestet, beim Schnittstellentest gegen die Schnittstellenspezifikation und beim Abnahmetest gegen die fachlichen Anforderungen, wie sie etwa in einem Pflichtenheft niedergelegt sind.

Äquivalenzklassenbildung

Bei der Äquivalenzklassenbildung werden die möglichen Werte der Eingaben (oder auch der Ausgaben) in Klassen eingeteilt, von denen vermutet werden kann, dass Fehler, die bei der Verarbeitung eines Wertes aus dieser Klasse auftreten, auch bei allen anderen Vertretern der Klasse auftreten. Wenn andererseits ein Vertreter der Klasse korrekt verarbeitet wird, dann wird vermutet, dass auch die Eingabe aller anderen Elemente der Klasse nicht zu Fehlern führt. Insofern können die Werte einer Klasse als (in dieser Hinsicht) äquivalent zueinander angesehen werden.

Auf Basis der Äquivalenzklassen werden die Testfälle gebildet. Zum Test der gültigen Äquivalenzklassen werden die Testdaten aus möglichst vielen gültigen Äquivalenzklassen gebildet. Zum Test der ungültigen Äquivalenzklassen wird jeweils ein Testdatum aus einer ungültigen Äquivalenzklasse mit ausschließlich gültigen Testdaten aus den übrigen Äquivalenzklassen kombiniert.

Ein Beispiel, um diese recht abstrakte Beschreibung zu verdeutlichen: In der Spezifikation eines Online-Banking-Systems wird gefordert, dass nur Beträge von 0,01 bis 500 € eingegeben werden dürfen. Man kann dann vermuten, dass eine Überweisung in Höhe von 123,45 € akzeptiert und korrekt ausgeführt wird, wenn ein Test ergeben hat, dass eine Überweisung von 123,44 € korrekt durchgeführt wird. Verallgemeinert kann angenommen werden, dass alle Beträge von 0,01€ bis 500,00 € korrekt verarbeitet werden, wenn dies für einen beliebigen Betrag aus diesem Bereich der Fall ist. Es reicht also aus, einen beliebigen Vertreter aus dem Bereich zu testen, um einem möglichen Fehler auf die Spur zu kommen.

In gleicher Weise kann man für negative Werte und für Werte größer als 500 € argumentieren. Für Tests sollte es daher ausreichen, drei Äquivalenzklassen zu bilden (eine gültige und zwei ungültige Äquivalenzklassen):

- Werte von 0,01 bis und mit 500,00 € (gültig)
- Werte kleiner gleich null (ungültig)
- Werte größer als 500,00 € (ungültig)

Jede Überweisung im Online-Banking-System muss durch die Eingabe einer TAN autorisiert werden. Analog zur ersten Äquivalenzklasse können hier für die Eingabe der TAN zwei Äquivalenzklassen gebildet werden:

- Eingabe einer korrekten TAN
- Eingabe einer falschen TAN

Auf Basis der beiden Äquivalenzklassen werden die nachfolgenden Testfälle definiert:

- Eingabe eines gültigen Werts (z. B. 123,45 €) und Bestätigung mit einer korrekten TAN
- Eingabe eines gültigen Werts (z. B. 123,45 €) und Bestätigung mit einer falschen TAN
- Eingabe eines ungültigen Werts (z. B. 600,00 €) und Bestätigung mit einer korrekten TAN

Mit der Äquivalenzklassenbildung ist es nicht möglich, Abhängigkeiten zwischen verschiedenen Eingabewerten zu berücksichtigen. So ist es etwa bei der Prüfung einer eingegebenen Adresse nicht ausreichend, für Ortsnamen, Straßennamen und Postleitzahlen jeweils (z.B. anhand einer Datenbank) zu prüfen, ob diese zur Klasse der gültigen Werte gehören. Sie müssen auch zusammenpassen.

Grenzwertanalyse

Die Grenzwertanalyse ist ein Spezialfall der Äquivalenzklassenanalyse. Sie ist aus der Beobachtung entstanden, dass Fehler besonders häufig an den „Rändern“ der Äquivalenzklassen auftreten. Daher werden hier nicht beliebige Werte getestet, sondern sog. Randwerte oder Grenzwerte. Im Beispiel wären dies die Werte

- 0,00 € (ungültige Eingabe)
- 0,01 € (gültige Eingabe)
- 500,00 € (gültige Eingabe)
- 500,01 € (ungültige Eingabe)

Pairwise-Methode

Die Pairwise-Methode ist ein Verfahren, die Anzahl von Tests von Kombinationen mehrerer Eingabewerte zu reduzieren, indem nicht alle möglichen Kombinationen getestet werden. Stattdessen wird lediglich jeder Eingabewert eines Feldes paarweise mit jedem Eingabewert der anderen Felder zusammen getestet. Dies reduziert die Anzahl der nötigen Tests radikal, bedeutet aber natürlich auch, dass unter Umständen Fehler nicht entdeckt werden, die nur bei ganz bestimmten Kombinationen von mehr als zwei Feldern auftreten.

Zustandsbasierte Testmethoden

Zustandsbasierte Testmethoden basieren auf Zustandsautomaten, die heute oft als UML-Diagramm dargestellt werden.

In der Beschreibung des Zustandsautomaten sind üblicherweise keine Fehlerfälle vorgesehen. Diese sind zusätzlich zu spezifizieren, indem zu jeder Kombination „{Ausgangszustand; Ereignis}“ (auch den im Automaten nicht spezifizierten Kombinationen) der Folgezustand und die ausgelösten Aktionen angegeben werden. Diese Kombinationen können dann alle getestet werden. Einsetzbar sind zustandsbasierte Methoden außer in technischen Anwendungen auch beim Test grafischer Benutzeroberflächen und von Klassen, die durch Zustandsautomaten definiert sind.

Um die Vollständigkeit der so ermittelten Testfälle zu prüfen, gibt es verschiedene Kriterien:

- Werden bei den Tests alle Zustände durchlaufen?
- Werden alle Zustandsübergänge durchlaufen?
- Werden alle Ereignisse, die Zustandsübergänge hervorrufen, getestet?

Ursache-Wirkungs-Analyse

Bei dieser Methode werden Ursachen und Wirkungen jeder Teilspezifikation identifiziert. Eine Ursache ist dabei eine einzelne Eingabebedingung oder eine Äquivalenzklasse von Eingabebedingungen; eine Wirkung ist eine Ausgabebedingung oder eine Systemtransformation.

Die Teilspezifikation wird in einen Boole'schen Graphen überführt, der Ursachen und Wirkungen durch logische Verknüpfungen verbindet. Anschließend wird der Graph um Abhängigkeiten auf Grund von syntaktischen Zwängen oder Umgebungsbedingungen ergänzt. Die folgenden Arten von Abhängigkeiten zwischen Ursachen werden dabei unterschieden:

1. Exklusive Abhängigkeit: Die Anwesenheit einer Ursache schließt andere Ursachen aus.
2. Inklusive Beziehung: Mindestens eine von mehreren Ursachen liegt vor (z. B. ist eine Ampel immer grün, gelb oder rot – oder rot und gelb zusammen. Wenn sie funktioniert und eingeschaltet ist).
3. One-and-only-one-Beziehung: Es liegt genau eine von mehreren Ursachen vor (z. B. männlich oder weiblich).
4. Requires-Abhängigkeit: Die Anwesenheit einer Ursache ist Voraussetzung für das Vorhandensein einer anderen (z. B. Alter > 21 und Alter > 18).
5. Maskierungsabhängigkeit: Eine Ursache verhindert, dass eine andere Ursache eintreten kann.

Der so entstandene Graph wird zu einer Entscheidungstabelle umgeformt. Dabei werden bestimmte Regeln angewandt, die aus einer Verknüpfung von n Ursachen n+1 Testfälle generieren (statt 2^n, wie es bei einer vollständigen Entscheidungstabelle der Fall wäre). Dabei entspricht jeder Testfall einer Spalte der Entscheidungstabelle.

Strukturorientierte Testmethoden

Strukturorientierte Testmethoden bestimmen Testfälle auf Basis des Softwarequellcodes (Whiteboxtest).

Software-Module enthalten Daten, die verarbeitet werden, und Kontrollstrukturen, die die Verarbeitung der Daten steuern. Entsprechend unterscheidet man Tests, die auf dem Kontrollfluss basieren, und Tests, die Datenzugriffe als Grundlage haben.

Kontrollflussorientierte Tests beziehen sich auf logische Ausdrücke der Implementierung. Datenflussorientierte Kriterien konzentrieren sich auf den Datenfluss der Implementierung. Genau genommen konzentrieren sie sich auf die Art und Weise in welcher Hinsicht die Werte mit ihren Variablen verbunden sind und wie diese Anweisungen die Durchführung der Implementierung beeinflussen.

Kontrollflussorientierte Tests

Die kontrollflussorientierten Testverfahren orientieren sich am Kontrollflussgraphen des Programms.

Es werden folgende Arten unterschieden:

- Anweisungsüberdeckung (C0)
- Kantenüberdeckung (C1)
- Bedingungsüberdeckung (C2, C3)
- Pfadüberdeckung (C4)

Anweisungsüberdeckungstest (C_0-Test)

Beim Anweisungsüberdeckungstest werden alle Knoten des Kontrollflussgraphen von mindestens einem Testfall abgedeckt, d.h., jede Anweisung im Quelltext wird mindestens einmal ausgeführt.

Zweigüberdeckungstest (Kantenüberdeckungstest) (C_1-Test)

Hier werden die Testfälle so spezifiziert, dass alle Zweige (Kanten) des Kontrollflussgraphen mindestens einmal durchlaufen werden.

Diese Methode hat Nachteile da es oft schwierig ist, alle Programmzweige zu durchlaufen, da bestimmte Betriebssystemzustände oder schwierig zu erzeugende Datenkonstellationen erforderlich sind. Zudem werden Kombinationen von Zweigen und komplexe Bedingungen nicht angemessen berücksichtigt. Schließlich werden Programmschleifen nicht ausreichend getestet, da ein einzelner Durchlauf durch den Schleifenkörper von abweisenden Schleifen und eine einzelne Wiederholung von nicht abweisenden Schleifen für die Zweigüberdeckung ausreichend ist.

Einfacher Bedingungsüberdeckungstest (C_2-Test)

Hier wird die Struktur der Entscheidungen innerhalb des Testlings für die Testfallermittlung herangezogen. Dabei wird gefordert, die Testfälle so zu bestimmen, dass die Auswertung jeder atomaren Teilentscheidung einmal den Wert wahr und einmal den Wert falsch ergibt.

Dies berücksichtigt jedoch nicht, dass Bedingungen oft aus geschachtelten logischen Verknüpfungen von Teilentscheidungen bestehen. Dadurch werden Fehlersituationen maskiert: Bei „oder“ - Verknüpfungen reicht es, wenn die erste Teilbedingung wahr ist, bei „und“ – Verknüpfungen, wenn die erste Bedingung falsch ist, um den weiteren Kontrollfluss festzulegen. Fehler in den anderen Bedingungsteilen werden nicht erkannt. Daher garantiert der einfache Bedingungsüberdeckungstest nicht einmal eine Zweigüberdeckung.

Mehrfach-Bedingungsüberdeckungstest (C_3-Test)

Der maximale Mehrfach-Überdeckungstest verlangt, dass neben den atomaren Teilentscheidungen und der Gesamtentscheidung auch alle zusammengesetzten Teilentscheidungen gegen wahr und falsch geprüft werden.

Der wesentliche Nachteil ist der extrem hohe Testaufwand: Bei n Teilentscheidungen ergeben sich 2^n Testfälle. Der modifizierte Bedingungsüberdeckungstest verlangt Testfälle, die demonstrieren, dass jede atomare Teilentscheidung den Wahrheitswert der Gesamtentscheidung unabhängig von den anderen Teilentscheidungen beeinflussen kann (Das wird etwa vom Standard RTCA DO-178B für flugkritische Software gefordert).

Bei n Teilentscheidungen ergeben sich minimal (n+1), maximal 2n Testfälle.

Pfad-Überdeckungstests (C_4-Test)

Beim Pfadüberdeckungstest werden sämtliche unterschiedlichen Pfade des Testlings in mindestens einem Testfall durchlaufen. In der Regel ist dies allerdings nicht durchführbar, da es sehr häufig unendlich viele Pfade gibt – bedingt durch Schleifen ohne feste Wiederholungszahl. Der Pfadüberdeckungstest hat alleinstehend daher wenig praktische Bedeutung.

Datenflussorientierte Tests

Datenflussorientierte Testmethoden basieren auf dem Datenfluss, also dem Zugriff auf Variablen. Sie sind besonders geeignet für objektorientiert entwickelte Systeme.

Für die datenflussorientierten Tests gibt es unterschiedliche Kriterien, welche im Folgenden beschrieben werden.

All defs-Kriterium

Für jede Definition (all defs) einer Variablen wird eine Berechnung oder Bedingung getestet. Für jeden Knoten und jede Variable muss ein definitionsfreier Pfad zu einem Element getestet werden. Die Fehlererkennungsrate liegt bei diesem Kriterium bei ca. 24%.

All p-uses-Kriterium Die „p-uses“ dienen zur Bildung von Wahrheitswerten innerhalb eines Prädikates (predicate-uses). Die Fehlererkennungsrate liegt bei diesem Kriterium bei ca. 34%. Es werden insbesondere Kontrollflussfehler erkannt.

All c-uses Kriterium

Unter dem „c-uses-Kriterium“ wird die Berechnung (computation-uses) von Werten innerhalb eines Ausdrucks verstanden. Dieses Kriterium deckt ca. 48% aller c-uses-Fehler auf. Es identifiziert insbesondere Berechnungsfehler.

Wegen ihrer Kompliziertheit sind die Techniken zur datenflussorientierten Testfallermittlung nur werkzeuggestützt einsetzbar. Da es aber kaum Werkzeuge gibt, haben sie zur Zeit noch keine praktische Relevanz.

Diversifizierende Testmethoden

Die Gruppe der **Diversifizierenden Tests** umfasst eine Menge von Testtechniken, die verschiedene Versionen einer Software gegeneinander testen. Es findet kein Vergleich zwischen den Testergebnissen und der Spezifikation, sondern ein Vergleich der Testergebnisse der verschiedenen Versionen statt. Ein Testfall gilt als erfolgreich absolviert, wenn die Ausgaben der unterschiedlichen Versionen identisch sind. Auch wird im Gegensatz zu den Funktionsorientierten und Strukturorientierten Testmethoden kein *Vollständigkeitskriterium* spezifiziert. Die notwendigen Testdaten werden mittels einer der anderen Techniken, per Zufall oder Aufzeichnung einer Benutzer-Session erstellt.

Die diversifizierenden Tests umgehen die oft aufwändige Beurteilung der Testergebnisse anhand der Spezifikation. Dies birgt natürlich die Gefahr, dass ein Fehler, der in den zu vergleichenden Versionen das gleiche Ergebnis produziert, nicht erkannt wird. Aufgrund des einfachen Vergleichens lassen sich solche Tests sehr gut automatisieren.

Back-to-Back-Test

Beim **Back-to-Back-Test** entstehen verschiedene gegeneinander zu testende Versionen aus der **n-Versionen-Programmierung**, d. h. die Programmierung verschiedener Versionen einer Software nach der gleichen Spezifikation. Die Unabhängigkeit der Programmierteams ist dabei eine Grundvoraussetzung.

Dieses Verfahren ist sehr teuer und nur bei entsprechend hohen Sicherheits-Anforderungen gerechtfertigt.

Mutationen-Test

Der **Mutationen-Test** ist keine Testtechnik im engeren Sinne, sondern ein Test der Leistungsfähigkeit anderer Testmethoden, sowie der verwendeten Testfälle. Die verschiedenen Versionen entstehen durch künstliches Einfügen von typischen Fehlern. Es wird dann geprüft, ob die benutzte Testmethode mit den vorhandenen Testdaten diese künstlichen Fehler findet. Wird ein Fehler nicht gefunden, so werden die Testdaten um entsprechende Testfälle erweitert. Diese Technik beruht auf der Annahme, dass ein erfahrener Programmierer meist nur noch "typische" Fehler macht.

Regressionstest

Aufgrund der mehrdeutigen Verwendung des Begriffes ist dem Regressionstest ein eigener Artikel gewidmet.

Weblinks

- http://www.pst.informatik.uni-muenchen.de/lehre/WS0405/mse/folien/D2-BlackWhite6p.pdf (PDF-Datei; 275 kB)

Hardware in the Loop

Hardware in the Loop (HiL, auch HitL, HITL) bezeichnet ein Verfahren, bei dem ein eingebettetes System (z. B. reales elektronisches Steuergerät oder reale mechatronische Komponente) über seine Ein- und Ausgänge an ein angepasstes Gegenstück, das im Allgemeinen HiL-Simulator genannt wird und als Nachbildung der realen Umgebung des Systems dient, angeschlossen wird.

Hardware in the Loop ist eine Methode zum Testen und Absichern von eingebetteten Systemen, zur Unterstützung während der Entwicklung sowie zur vorzeitigen Inbetriebnahme von Maschinen und Anlagen.

HiL für eingebettete Systeme

Dabei wird das zu steuernde System (z. B. Auto) über Modelle simuliert, um die korrekte Funktion des zu entwickelnden Steuergerätes (z. B. Motorsteuergerät) zu testen.

Die Eingänge des Steuergeräts werden mit Sensordaten aus dem Modell stimuliert. Um die Reglerschleife (Loop) zu schließen, wird die Reaktion der Ausgänge des Steuergeräts, z. B. das Ansteuern eines Elektromotors, in das Modell zurückgelesen.

Die HIL-Simulation muss meist in Echtzeit ablaufen und wird in der Entwicklung benutzt, um Entwicklungszeiten zu verkürzen und Kosten zu sparen. Insbesondere lassen sich wiederkehrende Abläufe simulieren. Dies hat den Vorteil, dass eine neue Entwicklungsversion unter den gleichen Kriterien getestet werden kann wie die Vorgängerversion. Somit kann detailliert nachgewiesen werden, ob ein Fehler beseitigt wurde oder nicht (siehe auch Fehlernachtest (engl. *re-testing)*).

Die Tests an realen Systemen lassen sich dadurch stark verringern und zusätzlich lassen sich Systemgrenzen ermitteln, ohne das Zielsystem (z. B. Auto und Fahrer) zu gefährden.

Die HIL-Simulation ist immer nur eine Vereinfachung der Realität und kann den Test am realen System deshalb nicht ersetzen. Falls zu große Diskrepanzen zwischen der HIL-Simulation und der Realität auftreten, sind die zugrundeliegenden Modelle in der Simulation zu stark vereinfacht. Dann müssen die Simulations-Modelle weiterentwickelt werden.

HiL im Automobilbereich

Mit der rapiden Zunahme von elektronischen Steuergeräten und steigendem Funktionsumfang, insbesondere in der Antriebselektronik, mit einer Fülle neuer regelbasierter Funktionen, wurde Anfang der 1990er Jahre Hardware in the Loop als Maßnahme zur Verbesserung der Testmöglichkeiten im Automobilbereich eingeführt. Hierbei wird HiL in zwei wesentlichen Ausprägungen für den Test angewandt.

1. Adaption von einem elektronischen System (z. B. Motor-, Getriebe- oder Bremsenelektronik) an einen HiL-Simulator als sogenannter Komponenten- oder Modulprüfstand.

2. Adaption von mehreren elektronischen Systemen an einen bzw. mehreren gekoppelten HiL-Simulatoren als sogenannter Integrationsprüfstand. Hierbei gehören die elektronischen Systeme im Allgemeinen zum gleichen Teilbereich des Automobils (Antriebselektronik, Komfort- bzw. Karosserieelektronik, Infotainmentelektronik). Die Verwendung der Bezeichnung HiL im Zusammenhang mit Komfort- bzw. Karosserieelektronik oder Infotainmentelektronik ist umgangssprachlich zwar üblich, aufgrund des Fehlens echter Regelkreise bei diesen Systemen jedoch nur bedingt korrekt.

Bei der Durchführung von Tests mit HiL werden die in der Anfangsphase manuell durchgeführten Tests durch automatische Testabläufe ersetzt. Dieses Verfahren nennt man Testautomatisierung. Dadurch lassen sich Tests nahezu beliebig parametrieren und präzise wiederholen. Eine Kontrolle der Fehlerabstellung ist somit wesentlich besser möglich. Die Testautomatisierung hat dem HiL Testverfahren zu einem Durchbruch verholfen und aus dem entwicklungsbegleitenden Testverfahren zu einem festen Bestandteil des Erprobungsprozesses gemacht.

Durch die mittlerweile hohe Güte der verwendeten Modelle im Fahrdynamik oder auch Motorbereich findet das HiL Verfahren seit Anfang der 2000er Jahre immer mehr Anwendung in der Entwicklung neuer Regelalgorithmen. Dies führt mittlerweile zu erheblichen Verkürzungen der Entwicklungszeiten.

Neben der reinen Anbindung des elektronischen Steuergeräts an einen HiL-Simulator gibt es auch die Variante des mechatronischen Verfahrens. Hierbei wird auch ein Teil der Mechanik in die Regelschleife integriert. Dieses Verfahren wird oft bei elektronischen Lenksystemen verwendet, wobei ein Teil des Lenkgestänges als reale Mechanik an den HiL-Simulator gekoppelt ist.

HiL im Maschinen- und Anlagenbau

Im Maschinen- und Anlagenbau wird für Hardware in the Loop in der Regel eine Speicherprogrammierbare Steuerung über einen Feldbus an ein Physikmodell einer Maschine bzw. Anlage angeschlossen. Man verwendet hierfür auch die Bezeichnung Anlagensimulation Die Anlagensimulation enthält in der Regel eine Abbildung des Verhaltens sowie des Materialflusses. Über eine optionale 3D-Visualisierung sowie Ausgaben der Physiksimulation kann dann ein Beobachter die Maschinenfunktion überwachen.

Zweck ist die Erstellung und Erprobung von Steuerungsprogrammen, bevor die Bauteile einer Maschine gefertigt und montiert sind. Dadurch lässt sich die Inbetriebnahmephase verkürzen. Ein weiterer Vorteil liegt in der Möglichkeit, ohne Gefahr für den Bediener Grenzsituationen zu testen, wie z. B. das Fahren auf Hardware-Endschalter.

Zukünftige Anwendungsfelder können die Ferndiagnose und Fernwartung von Maschinen und Anlagen mit einschließen. Über eine Telekommunikationsleitung (z. B. über Internet) wird der aktuelle Zustand einer Steuerung vom Maschinenbetreiber in ein Service-Center beim Maschinenhersteller übertragen. Dort können dann anhand des physikalischen Modells erste Diagnosen gestellt und Empfehlungen abgeleitet werden.

HiL in der Luft- und Raumfahrt

In der Luft- und Raumfahrt werden in HiL Systemen Zustände getestet, die am Boden nicht immer nachzubilden sind. Für die Zulassung der Flugsteuerung wird bereits für den Superjet 100 der Iron Bird durch den virtuellen oder Electronic Bird ersetzt.

HiL und die reale Welt

Durch den technologischen Fortschritt und die Entwicklung von leistungsstarken Mikroprozessoren ist es mittlerweile Stand der Technik, dass HiL-Systeme zunehmend die reale Umwelt ersetzen. Gerade für die Entwicklung elektronischer Steuergeräte wird mit Hilfe des HiL-Simulators so die Erstellung einer idealen Testumgebung im Labor ermöglicht. Je nach Systemanforderungen bewegt sich die Berechnungszeit eines kompletten Simulationszyklus' im Bereich von 1ms, bei Spezialanwendungen reicht es bis in den Mikrosekundenbereich.

Unabhängig von den technischen Möglichkeiten stellt der HiL-Simulator jedoch immer nur einen begrenzten und großteils reduzierten Ausschnitt der realen Umgebungswelt dar. Insbesondere bei Funktionen, bei denen der Kunde in Interaktion mit der Technik steht, stößt man schnell an die Grenzen der Simulationsfähigkeit.

Eine Umweltsimulation basiert immer nur auf den vorliegenden Erkenntnissen, erhobenen Mess- und Erfahrungswerten, welche in vereinfachte mathematische Formeln überführt werden und sich dann als Modelle im HiL-Simulator wiederfinden. Auf Grund dessen und trotz der Fortschritte in der Umweltsimulation kann der HiL-Simulator nur in einem begrenzten Rahmen den Test in der realen Welt ersetzen.

Software in the Loop

Bei der Methode Software in the Loop (SiL) wird im Gegensatz zum HiL keine besondere Hardware eingesetzt. Das erstellte Modell der Software wird lediglich in den für die Zielhardware verständlichen Code umgewandelt (beispielsweise von einem MATLAB/Simulink-Modell nach C-Code). Dieser Code wird auf dem Entwicklungsrechner zusammen mit dem simulierten Modell ausgeführt, anstatt wie bei Hardware in the Loop auf der Zielhardware zu laufen. Es handelt sich dabei also um eine Methode, die vor dem HiL anzuwenden ist.

Vorteile von SiL sind unter anderem, dass die Zielhardware noch nicht feststehen muss, und dass die Kosten aufgrund der fehlenden Simulationsumgebung weitaus geringer ausfallen. Das hier benutzte Modell der Strecke kann auch beim HiL weiter verwendet werden, und somit die einzelnen Testläufe miteinander verglichen werden.

Weblinks

- Thermodynamische Simulation von Verbrennungsmotoren in Echtzeit: Simulation von Dieselmotoren bis hin zum 12-Zylinder-Motor [1]
- Maschinensimulation zur Steuerungsinbetriebnahme: Echtzeit-HIL-Maschinensimulation [2]
- Quelle zu HiL von Maschinen: Virtuelle Inbetriebnahme [3]
- Electronic Bird - Superjet 100 [4]
- Gegenüberstellung von HiL und Realität in der Steuergeräteentwicklung im Automobilbereich: Fahrversuch versus HiL-Test - neutrale Gegenüberstellung [5].
- Zusammenhang Echtzeitsimulation und HIL: Echtzeitsimulation [6]

Referenzen

[1] http://www.micronova.de/de/hil/referenzen/222-man-nutzfahrzeuge-ag.html
[2] http://www.isg-stuttgart.de/virtuos
[3] http://www.machineering.de/lexikon/virtuelle-inbetriebnahme
[4] http://www.prcenter.de/Electronic-Bird-fuer-Liebherr-Aerospace-Lindenberg-GmbH.9475.html
[5] http://www.berner-mattner.com/cms/upload/pdf/whitepaper/BernerMattner_WhitePaper_modularHiL.pdf
[6] http://www.joerg.frochte.de/echtzeit.html

Model in the Loop

Model in the Loop (MIL) ist die Simulation eines eingebetteten Systems in einer frühen Entwicklungsphase, der Modellierung, bei der modellbasierten Softwareentwicklung. Eingebettete Systeme kommunizieren mit ihrer Umwelt und erwarten häufig plausible Sensorsignale als Eingang und stimulieren dann das physikalische System. Um richtig zu funktionieren, muss die Umgebung des eingebetteten Systems simuliert werden. Wird nun das eingebettete System (Modell) in einer Schleife zusammen mit dem Umgebungsmodell simuliert, spricht man von Model in the Loop Simulation.

MIL ist eine kostengünstige Möglichkeit um eingebettete Systeme zu testen. Entwicklungs- und Simulationsumgebungen für die modellbasierte Entwicklung sind beispielsweise MATLAB/Simulink oder ASCET.

In nachfolgenden Entwicklungsstufen wird dann von Software in the Loop (SIL), Processor in the Loop (PIL) und Hardware in the Loop (HIL) gesprochen. Das Prinzip ist gleich. Das eingebettete System wird zusammen mit einem Modell simuliert, das die Umgebung des Systems abbildet.

Referenzen

- Menno Mennenga, Christian Dziobek, Iyad Bahous: Modell- und Software-Verifikation vereinfacht. In: Elektronik automotive, Heft 4.2009, 2009 [1]
- Plummer: Model-in-the-Loop Testing; In: Proceedings of the Institution of Mechanical Engineers, Part I: Journal of Systems and Control Engineering, 2006 [2]
- Isermann: Fahrdynamik-Regelung: Modellbildung, Fahrerassistenzsysteme, Mechatronik, Vieweg, 2006 [3]

Referenzen

[1] http://www.piketec.com/downloads/papers/Effizient-Testen-Elektronik-Automotive-04-09.pdf
[2] http://journals.pepublishing.com/content/uh512182515v47q7/fulltext.pdf
[3] http://books.google.de/books?id=G4E08FM3GK8C&pg=PA104&lpg=PA104&dq=mil+simulation&source=bl&ots=sOWnhamr55&sig=uYRfVaw8RyOySsgh6LAAVCXESmA&hl=de&ei=AENHS9yPNJTG_gaDr92PAg&sa=X&oi=book_result&ct=result&resnum=9&ved=0CDAQ6AEwCA#v=onepage&q=mil%20simulation&f=false

Testautomatisierung

Unter **Testautomatisierung** (auch **Testautomation**) ist die Automatisierung von Aktivitäten im Test zu verstehen, sowohl beim Softwaretest als auch beim automatisierten Test von Hardware, dem Hardwaretest.

Motivation

In der Softwareentwicklung ist es besonders wichtig, einen festen, definierten Status der Software zu kennen, so z. B.:

- Ist die jetzige, neue Softwareversion besser als die alte Version?

Automatische Tests, die jeden Tag zu definierten Zeiten die Software testen, machen Software bezüglich ihrer Qualität erst messbar und zeigen mögliche Nebeneffekte von vorgenommenen Änderungen direkt und erkennbar an. Sie dienen als direkte Rückkopplung für Entwickler und für Tester, die unter Umständen nicht in der Lage sind, das Gesamtsoftwaresystem auf einmal zu überschauen, sowie zur Erkennung von Nebeneffekten und Folgefehlern.

Die Testautomatisierung liefert demnach eine Metrik der Anzahl der erfolgreichen Testfälle pro Testlauf. Ist die Software so messbar, können jederzeit folgende Fragen beantwortet werden:

- Wann ist eine neue Anforderung durch eine Software vollständig erfüllt?
- Wann ist ein Programmfehler behoben?
- Wann ist die Arbeit des Entwicklers beendet?
- Wer ist zu welchem Zeitpunkt wofür verantwortlich?
- Welche Qualität hat eine neue Software-Version (siehe Entwicklungsstadium (Software))?
- Ist die neue Software-Version qualitativ besser als die vorherige Version?
- Hat ein behobener Fehler oder eine neue Anforderung eine Auswirkung auf bestehende Software (Änderung des Verhaltens der Software)?
- Ist sichergestellt, dass der Echtbetrieb mit der neuen Software erfolgreich und sicher ist?
- Was ist nun wirklich an neuer Funktionalität und Fehlerkorrekturen in der Software; kann man das nachvollziehen?
- Lässt sich der Liefertermin der Software noch einhalten, wenn eine Einschätzung der momentanen Qualität der Software nicht möglich ist?

Zur Beispielfrage: „Wann ist ein Programmfehler behoben?“ lautet die Antwort in diesem Fall:

> „Genau dann, wenn alle schon existierenden Testfälle und auch die für den Programmfehler selbst geschriebenen Testfälle erfolgreich beendet wurden.“

Eine Rückmeldung liefert nur der ständige Test, und dieser ist durch Automatisierung erst möglich und realisierbar.

Ein weiterer Vorteil der Testautomatisierung ist die Beschleunigung des Entwicklungsprozesses. Wo bei Software-Projekten ohne Automatisierung die Produktion, die Installation und der Test nacheinander manuell durchgeführt werden, können bei vollautomatisierten Projekten (also wenn außer dem Test auch Produktion und Installation automatisierbar sind) diese drei Schritte automatisch nacheinander gestartet werden, z. B. in einem Nachlauf. Je nach Umfang des Projektes kann man gegebenenfalls diesen Ablauf abends starten und am nächsten Morgen das Testergebnis verfügbar haben.

Automatisierbare Aktivitäten

Prinzipiell lassen sich folgende Aktivitäten automatisieren:

- Testfallerstellung
  - Testdatenerstellung
  - Testskripterstellung
- Testdurchführung
- Testauswertung
- Testdokumentation
- Testadministration

Testfallerstellung

Abhängig vom verwendeten Format zur Beschreibung eines Testfalles lässt sich die Testfallerstellung automatisieren, indem höhersprachliche Beschreibungen *(Testspezifikationen)* in dieses Format transformiert werden. Zur Testspezifikation werden Sprachen unterschiedlicher Abstraktionsstufe verwendet: einfache tabellenartige Notationen für Testdaten und Funktionsaufrufe, Skriptsprachen (z. B. Tcl, Perl, Python), imperative Sprachen (z. B. C, TTCN-3), objektorientierte Ansätze (JUnit) und deklarative und logische Formalismen sowie modellbasierte Ansätze (z. B. TPT). Dabei wird eine weitgehende und möglichst vollautomatische Übersetzung von Artefakten in einer maschinenfernen fachlichen Sprachebene in Artefakte in einer maschinennahen technischen Sprachebene angestrebt. Ein anderer Ansatz ist es, die Testfallerstellung an Hand von zu deklarierenden Geschäftsobjekten dynamisch zu generieren. Liegt eine Testspezifikation nicht schon in ablauffähiger Form vor, sondern in einer nicht ausführbaren Sprache (z. B. UML, Excel-Tabelle, oder ähnliches), kann diese unter Umständen mit geeigneten Werkzeugen automatisch in ablauffähige Testfälle übersetzt werden.

Testdatenerstellung und Testskripterstellung

Da die Anzahl möglicher Eingabewerte und Abläufe eines Programms oft sehr groß ist, müssen bei der Generierung von Testfällen aus Testspezifikationen Eingabedaten und Abläufe gemäß der zu erzielenden Testabdeckung ausgewählt werden. Zur Testdatenerstellung kann dabei oft das Datenmodell der Software genutzt werden, zur Testskripterstellung werden beim modellbasierten Testen Verhaltensmodelle der Software verwendet. Lösungen, die ohne Skripten auskommen, sind auf dem kommerziellen Markt auch verfügbar.

Testdurchführung

Die Testdurchführung erfolgt heute weitgehend durch vollautomatische Testwerkzeuge. Abhängig vom Zielsystem kommen hier Unit-Test-Tools, Testsysteme für Grafische Benutzeroberflächen, Lasttestsysteme, Hardware-in-the-loop-Prüfstände oder andere Werkzeuge zum Einsatz.

Testauswertung

Zur Testauswertung muss das erhaltene Testergebnis mit dem Erwartungswert verglichen werden. Im einfachsten Fall ist hier nur ein Tabellenvergleich vorzunehmen; falls das Sollverhalten allerdings durch logische Constraints definiert ist oder extrem komplexe Berechnungen enthält, kann das so genannte Orakelproblem auftreten. Werden zwei Software-Versionen oder zwei Testzyklen und damit zwei Testergebnisse gegen das Soll-Ergebnis verglichen, so lassen sich Tendenzaussagen und Qualitätsstatistiken berechnen.

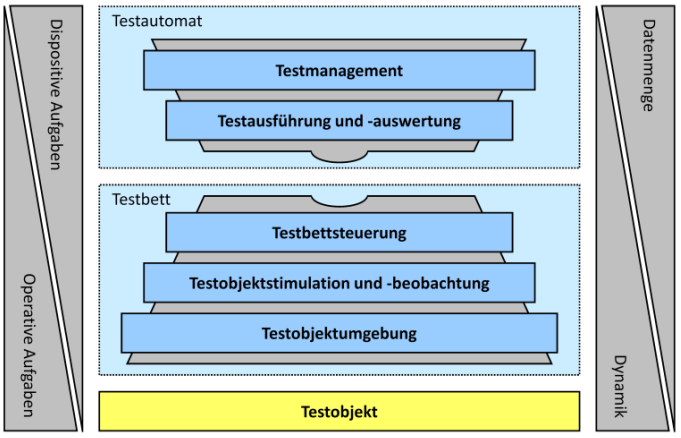
Testdokumentation

Bei der Testdokumentation wird aus den erhaltenen Testergebnissen ein nachvollziehbarer und verständlicher Testbericht erzeugt. Hierfür können Dokumentgeneratoren und Schablonenwerkzeuge eingesetzt werden.

Testadministration

Aufgabe der Testadministration ist die Verwaltung und Versionierung von Testsuiten sowie die Bereitstellung einer adäquaten Benutzungsumgebung. Neben Standardwerkzeugen (z. B. CVS, Eclipse) gibt es eine Reihe von Spezialwerkzeugen, die speziell auf die Belange des Softwaretests zugeschnitten sind.

Universelle Architektur zur Testautomatisierung



Universelle Testsystem-Architektur

Für die Automatisierung der oben genannten Aktivitäten existieren verschiedene Tools. Diese fokussieren stets die Lösung spezieller Aufgaben und unterscheiden sich in Bedienphilosophie, Syntax und Semantik. Daher ist es oft schwierig die richtigen Tools für eine bestimmte Menge von Aktivitäten auszuwählen bzw. die Tools richtig einzusetzen. Eine Strukturierung und Einordnung der automatisierbaren Aktivitäten zur abstrahierten lösungsneutralen Toolfunktionalität bietet die universelle Testsystem-Architektur. Dazu definiert sie fünf Funktionsebenen: Testmanagement, Testausführung und -auswertung, Testbettsteuerung, Testobjektstimulation und -beobachtung sowie Testobjektumgebung. Die Testsystem-Architektur unterstützt die Integration vorhandener Test-Tools und -Komponenten in Testsysteme und stellt somit eine universelle Grundlage zur Testautomatisierung dar.

Werkzeuge für automatisierte Software-Tests

Werkzeuge für automatisiertes Testen von Software sind z. B.

- ABAP Unit
- CppUnit für C++
- csvdiff – ein Tool zur Testauswertung um Datenbankentladedateien/Tabellen zu vergleichen
- eCATT von SAP
- EXAM - Methodik zur grafischen Entwicklung von Testfällen für Automobilanwendungen
- Fit - Framework for Integrated Test
- Hyades
- JUnit und Cactus für Java
- JMeter oder Selenium für Webprojekte
- NUnit für alle Sprachen (C#, C++.Net, VisualBasic usw.), die eine Implementierung der CLI verwenden, z. B. das Microsoft .Net Framework oder das Mono-Projekt
- QF-Test – ein Testwerkzeug für die Automatisierung von Java-und Web-Anwendungen mit grafischer Benutzeroberfläche (GUI)
- Tessy – ein Werkzeug zum automatisierten Modul- bzw. Unit-Test von in C/C++ geschriebener Software

- TOSCA Testsuite - beinhaltet ein integriertes Testmanagement, eine grafische Benutzeroberfläche (GUI) und eine Anwendungsprogrammierschnittstelle (API).
- TPT – eine Methode und Werkzeug für den automatischen systematischen modellbasierten Test von Steuerungs- und Regelsystemen
- WinRunner – ein automatisches GUI-Testwerkzeug für Windows (per Add-Ins auch für andere Anwendungen z.B. in Java)

Weblinks

- www.testingfaqs.org [1] Eine Liste mit Testautomatisierungswerkzeugen (en)

Literatur

- Dmitry Korotkiy: *Universelle Testsystem-Architektur in der Mechatronik.* Sierke Verlag, Göttingen 2010. ISBN 978-3-86844-238-0

Referenzen

[1] http://www.testingfaqs.org/

Modultest

In der Softwareentwicklung wird ein Computerprogramm üblicherweise in einzelne Teile mit klar definierten Schnittstellen, sogenannte Module, unterteilt. Der **Modultest** (auch **Komponententest** oder oft vom engl. **unit test** als **Unittest** bezeichnet) ist der Softwaretest dieser Programmteile, die zu einem späteren Zeitpunkt zusammengefügt (integriert) werden (vgl. Integrationstest). Ziel des Modultests ist es, frühzeitig Programmfehler in den Modulen einer Software (z. B. von einzelnen Klassen) zu finden. Die Funktionalität der Module kann so meist einfacher getestet werden, als wenn die Module bereits integriert sind, da in diesem Fall die Abhängigkeit der Einzelmodule mit in Betracht gezogen werden muss.

Automatisierte Modultests

Mit der Verbreitung von agilen Softwareentwicklungsmethoden und insbesondere testgetriebener Entwicklung hat sich die Ansicht verbreitet, Modultests ausschließlich automatisiert durchzuführen. Dazu werden üblicherweise mit Hilfe sogenannter Test Frameworks wie beispielsweise JUnit Testprogramme geschrieben, welche die zu testenden Module, in der Regel etwa eine Methode oder eine Funktion, aufrufen und auf korrekte Funktionsweise testen.

Die automatisierten Modultests haben den Vorteil, dass sie rasch und von jedermann ausgeführt werden können. Somit besteht die Möglichkeit, nach jeder Programmänderung durch Ablauf aller Modultests nach Programmfehlern zu suchen. Dies wird üblicherweise empfohlen, da damit etwaige neu entstandene Fehler schnell entdeckt und somit kostengünstig behoben werden können. Dies setzt allerdings voraus, dass die Modultests vor Programmänderungen 100% durchlaufen, also Entwickler nur dann ihren Sourcecode einchecken, wenn alle Modultests erfolgreich durchlaufen.

Eigenschaften von Modultests

Isoliert

Modultests testen die Module isoliert, d. h. ohne die Interaktion der Module mit anderen. Ist das nicht der Fall spricht man nicht von Modultests, sondern von Integrationstests. Dazu müssen andere Module beziehungsweise externe Komponenten wie etwa eine Datenbank, Dateien oder Backendsysteme, die von dem Modul verwendet werden, simuliert (engl. to mock) werden. Diese Module nennt man Mock-Objekte. Auch dafür verwendet man üblicherweise Frameworks wie beispielsweise Easymock.

Test von Fehlverhalten

Modultests testen ausdrücklich nicht nur das Verhalten des Moduls im Gutfall, beispielsweise bei korrekten Eingabewerten, sondern auch im Fehlerfall, beispielsweise bei unkorrekten Eingabewerten oder von anderen Modulen gelieferten Fehlermeldungen. Auch dafür ist das Mocken anderer Module hilfreich, da sich ein Fehlzustand anderer Module, wie beispielsweise der Verbindungsabbruch bei Netzwerkverbindungen, durch Mock-Objekte leichter simulieren als in der Realität nachstellen lässt.

Laufende Ausführung

Modultests sollten im Laufe der Entwicklung regelmäßig durchgeführt werden, um zu verifizieren, dass Änderungen keine unerwünschten Nebeneffekte haben. Modultests sollten daher von jedem Entwickler vor dem Einchecken durchgeführt werden, automatisierte Modultests auch bei Kontinuierlicher Integration auf sogenannten Continuous Integration Servern wie beispielsweise Hudson.

Test des Vertrages und nicht der Algorithmen

Modultests sollen gemäß dem Design-by-contract-Prinzip möglichst nicht die Interna einer Methode testen, sondern nur ihre externen Auswirkungen (Rückgabewerte, Ausgaben, Zustandsänderungen, Zusicherungen). Werden auch interne Details der Methode geprüft (dies wird als White-Box-Testing bezeichnet), droht der Test fragil zu werden, er könnte auch fehlschlagen, obwohl sich die externen Auswirkungen nicht geändert haben. Daher wird in der Regel das sogenannte Black-Box-Testing empfohlen, bei dem man sich auf das Prüfen der externen Auswirkungen beschränkt.

Testgetriebene Entwicklung

Bei der Testgetriebenen Entwicklung (TDD von *Test driven development*) wird jeweils ein Modultest vor dem Erstellen bzw. Ändern des eigentlichen Programmcodes erstellt und gepflegt. Dies hat verschiedene Vorteile: Es wird verifiziert, dass der Test ohne die Änderung wirklich fehlschlägt, es reduziert die Gefahr des übermäßigen White-Box-Testings, es verbessert das Design, da sich der Entwickler so besser über das benötigte Verhalten der Methode klar werden kann, bevor er mit der Entwicklung beginnt. Darum ist es in der Praxis meist weniger aufwendig einen Test vor der Implementierung zu schreiben als umgekehrt.

Pro Bug ein Test

Die Erstellung eines Modultests vor Behebung eines Fehlers bietet einige Möglichkeiten. Einerseits stellt der Test sicher, dass der Bug wirklich in dem Programmcode vorhanden ist, wo man ihn auszubessern gedenkt, andererseits dass der Bug auch durch die Programmänderung tatsächlich behoben wurde und auch in Zukunft nicht mehr auftreten kann. Einige Open-Source-Projekte fordern daher, dass bei Fehlern des Programms nicht nur ein Fix, sondern auch ein Test mitgeliefert wird, der ohne den Fix fehlschlägt, aber mit dem Fix korrekt abläuft.

Modultests sind die Vorstufe zu Integrationstests, die wiederum zum Testen mehrerer voneinander abhängiger Komponenten im Zusammenspiel geeignet sind. Im Gegensatz zu Modultests werden Integrationstests meist manuell ausgeführt.

Anwendungsbeispiele

Modultests werden auch im Automotive-Bereich an programmierbaren Steuereinheiten verwendet. Damit wird die Steuereinheit verifiziert (ihre Übereinstimmung mit der Absicht des Entwicklers geprüft). Hier haben die Modultests auch rechtliche Bedeutung innerhalb des Vertragsdokumentes. Falls eine programmierbare Steuerung versagt, kann es zu Personenschäden kommen. Bei einem solchen Test wird die Durchführung einschließlich aufgetretener Fehler protokollartig festgehalten. Dabei wird dann zwischen Unit-Test (kann der Test einer einzelnen C-Funktion sein) und Modultest (Test des gesamten Moduls, dazu gehören Tests der Units und Tests der Funktionsschnittstellen zwischen den Units) unterschieden. Im Automotive-Bereich stehen bei diesen Tests weniger textuelle Daten als vielmehr Variablen physikalischer Werte und damit Grenzwerte im Vordergrund. So muss z. B. geprüft werden, ob das Ergebnis einer Addition von Ganzzahlen sich in jedem Fall innerhalb des Wertebereiches des Ganzzahl-Datentyps befindet. Man erhält dabei über die Code-Abdeckung hinaus große Mengen an Zahlenlisten, die zu testen sind.

Bei Fluxtests werden die Datenflüsse der Schnittstellen integrierter Systeme abgehört und dem Regressionset für die Unittests beigefügt, da ja sowohl Input als auch Output bekannt ist. Die eigentliche Fluxtestphase erfolgt dabei erst beim nächsten Zyklus der Softwareentwicklung, in der den Units dann die bekannten Aufrufparameter übergeben werden beziehungsweise anhand der bekannten gewünschten Ausgabedaten die Units auf ihre Richtigkeit überprüft werden. Fluxtests können nur aus funktionierenden (teil-)integrierten Systemen gewonnen werden. Damit wird im nächsten Zyklus der Integration vorgegriffen. Bereits fertiggestellte Units werden früher getestet, der Release- oder Entwicklungszyklus verkürzt sich. Besonders bei "hardest-first" Integrationsstrategien zahlen sich Fluxtests somit aus.

Beispieltest

Das folgende Beispiel ist eine Testsuite für eine Model-Klasse einer Ruby on Rails-Anwendung mit dem Shoulda-Test-Framework. Zunächst wird ein einfacher Test für Verknüpfungen zu anderen Objekten (hier Übersetzungen) geprüft. Mit einer setup-Methode wird ein Ort in die Datenbank eingetragen, anhand dessen dann zwei Methoden der Klasse mit unterschiedlichen Werten geprüft werden. Anhand der verschachtelten context- und should-Blöcke generiert Shoulda Methoden mit Namen wie Given I have a place name should return "Schweiz" for Switzerland in German oder Given I have a place name_with_local should return "Switzerland (Schweiz)" or "Switzerland (Suisse)" for Switzerland in English. I18n.locale setzt dabei die Spracheinstellung, während assert_equal und assert_contains die Methode aufrufen und den Rückgabewert mit den erwarteten Werten vergleichen.

```
class PlaceTest < ActiveRecord::TestCase
  should_have_many :translations

  context "Given I have a place" do
    setup do
      @Switzerland = Place.create!
      @Switzerland.translations.create! :name => "Schweiz", :locale =>
"de", :is_local => true
      @Switzerland.translations.create! :name => "Switzerland", :locale
=> "en"
      @Switzerland.translations.create! :name => "Suisse", :locale =>
"fr", :is_local => true
    end

    context "name" do
      should "return \"Schweiz\" for Switzerland in German" do
```

```
        I18n.locale = :de
        assert_equal "Schweiz", @Switzerland.name
      end

      should "return \"Switzerland\" for Switzerland in English" do
        I18n.locale = :en
        assert_equal "Switzerland", @Switzerland.name
      end

      should "return \"Switzerland\" for Switzerland in Korean" do
        I18n.locale = :ko
        assert_equal "Switzerland", @Switzerland.name
      end
    end

    context "name_with_local" do
      should "return \"Schweiz\" for Switzerland in German" do
        I18n.locale = :de
        assert_equal "Schweiz", @Switzerland.name_with_local
      end

      should "return \"Switzerland (Schweiz)\" or \"Switzerland
(Suisse)\" for Switzerland in English" do
        I18n.locale = :en
        assert_contains ["Switzerland (Schweiz)", "Switzerland
(Suisse)"], @Switzerland.name_with_local
      end
    end
  end
end
```

Literatur

- Gerard Meszaros: *xUnit Test Patterns: Refactoring Test Code.* Addison-Wesley, Amsterdam 31. Mai 2007, ISBN 0-1314-9505-0, S. 833. (Englisch)
- Paul Hamill: *Unit Test Frameworks.* O'Reilly Media, 19. November 2004, ISBN 0-5960-0689-6, S. 304. (Englisch)

Weblinks

- Links zum Thema Unit-Testing (engl.) [1] im Open Directory Project
- Einführung in die Prinzipien von Unit-Tests [2]
- Artikel zum Thema Komponententests unter Java (engl.) [3]
- White Paper „Unit Test of Embedded Software“ [4] (PDF)

Referenzen

[1] http://www.dmoz.org/Computers/Programming/Languages/Java/Development_Tools/Performance_and_Testing/Unit_Testing/
[2] http://www.EvoComp.de/softwareentwicklung/unit-tests/unittests.html
[3] http://junit.sourceforge.net/doc/testinfected/testing.htm
[4] http://www.hitex.com/index.php?id=923&L=2

Liste von Modultest-Software

Modultestsoftware (meist aber "Test-Frameworks" von engl. "Unit test frameworks" genannt) bezeichnet Software-Frameworks zur Durchführung von Modultests (oft auch Komponententests genannt). Sie dienen dem Nachweis von Fehlern in einzelnen Komponenten (Modulen) einer Software, beispielsweise einzelnen Klassen. Als Voraussetzung für Refactoring kommt ihm besondere Bedeutung zu. Nach jeder Änderung sollte durch Ablauf aller *Testfälle* nach Programmfehlern gesucht werden.

Modultestsoftware bzw Test-Frameworks gibt es mittlerweile für fast jede Programmiersprache. Oft haben die Namen dieser Frameworks die Form xyzUnit (z. B. JUnit für Java, siehe unten). Testrahmen dienen dazu, den Quelltext besser und automatisch testen zu können. Damit kann nach Modifikationen am Quelltext relativ schnell festgestellt werden, ob die vorher erstellten Tests immer noch dieselben Ergebnisse haben.

ABAP

ABAP Unit ist die Implementierung des Frameworks für ABAP und steht ab SAP NetWeaver Release 2004 zur Verfügung.

C

Für die Programmiersprache C gibt es verschiedene Implementationen, die sich hauptsächlich in ihrem Funktionsumfang, Lizenzen und Einsatzgebieten unterscheiden:

- CUnit:[1] Umfangreiche Implementation. Lizenziert unter der GNU General Public License (GPL). Das Projekt wird unter SourceForge gehostet.
- cfix:[2] Spezialisiert für Win32 und Windows NT kernel mode-Entwicklung. Lizenziert unter der GNU Lesser General Public License (LGPL). Basierend auf cfix existiert mit Visual Assert[3] ein Add-In, welches Visual Studio um Unit Testing-Funktionalität erweitert.
- Embedded Unit: Spezielles Framework für Embedded Systems. Lizenziert unter der MIT-Lizenz. Das Projekt wird ebenfalls unter SourceForge gehostet.
- uCUnit: Spezielles Framework für kleine Mikrocontrollersysteme. Lizenziert unter der Common Public License (CPL) v1.0
- MinUnit: Demonstration eines minimalistischen Frameworks, lizenziert als Public Domain.
- Tessy (Software): Professionelles Werkzeug für Modul-/Unit-Tests, speziell von Embedded Software.
- Testwell ctc++ Test Coverage Analyser für Modul-/Unit-Tests für alle embedded Systeme
- Time Partition Testing: Modellbasiertes Testen eingebetteter Steuerungs- und Regelungssysteme.

C++

- CppUnit ist die Portierung von JUnit auf C++. Ursprünglich wurde es von Michael Feathers geschrieben, ist jetzt aber ein offenes Projekt unter SourceForge. Da er allerdings den Eindruck hatte, dass CppUnit mittlerweile zu kompliziert zu installieren war, entschied sich Feathers, die abgespeckte Variante CppUnitLite zu schreiben.
- cfix:[2] Spezialisiert für Win32 und Windows NT kernel mode-Entwicklung. Lizenziert unter der GNU Lesser General Public License (LGPL). Basierend auf cfix existiert mit Visual Assert[3] ein Add-In, welches Visual Studio um Unit Testing-Funktionalität erweitert.
- CxxTest, CppTest (beides ebenfalls offene Projekte unter SourceForge)
- Boost (C++-Bibliothek) Test Library
- Tessy (Software): Professionelles Werkzeug für Modul-/Unit-Tests, speziell von Embedded Software.
- Testwell CTA++ C++ Test Aider / Testwell ctc++ Test Coverage Analyser
- Time Partition Testing: Modellbasiertes Testen eingebetteter Steuerungs- und Regelungssysteme.
- Qt unterstützt Modultests (QTestLib).
- Google Mock von Google entwickeltes C++ Test Framework.

Cobol

- CobolUnit ist ein zu XUnit kompatibles Framework.
- *savvytest*[4] ist ein auf Eclipse basierendes (kommerzielles) Testtool zur Erfassung und Durchführung von Komponententests, das vorwiegend zum Testen von Mainframe-Komponenten (insbesondere COBOL unter z/OS) konzipiert wurde. Zudem wird die rein technische Schnittstelle durch zusätzliche Spezifikationen in einer fachlichen Sicht dargestellt und mit Testdaten versorgt. Die Tests an sich werden sprach- und plattformunabhängig gespeichert.

Delphi

DUnit ist eine Portierung von JUnit für Borland Delphi. In der Version „Delphi 2005“ wurde es von Borland (heute CodeGear) als fester Bestandteil in die Entwicklungsumgebung aufgenommen.

Java

JUnit wurde für die Programmiersprache Java von Erich Gamma und Kent Beck geschrieben. (weitere Werkzeuge für Modultests: TestNG, HttpUnit, Cactus, …)

JavaScript

JSUnit wurde nach dem Vorbild JUnit gebaut und bietet manuelle wie auch komplett integrierte Tests für JavaScript mit Browser- und Betriebssystemübergreifenden Testservern.

Lingo (Macromedia Director)

Für die interpretierte Skriptsprache Lingo in Macromedias Autorensystem Director gibt es das Framework LingoUnit, welches ebenfalls unter SourceForge geführt wird.

.NET

NUnit ursprünglich eine 1:1 Portierung von JUnit auf die Plattform .NET, insbesondere für C# und Visual Basic .NET. Wurde mittlerweile mit spezifischen .NET Features erweitert neu geschrieben und unterstützt alle .NET Sprachen.

Microsoft bietet das in Visual Studio integrierte Unit-Test-Framework MSTest an.

Perl

Perl hat eine weit zurückreichende Geschichte automatischer Tests. Perl selbst wird automatisch getestet und zur Perldistribution gehören eine Reihe von Testmodulen. Hier beginnt man am besten mit Test::Simple, arbeitet sich zu Test::More vor und taucht dann in die Tiefen von Test::Class und den weiteren Modulen ab. Diese verwenden alle das Test Anything Protocol (TAP)[5] . Des weiteren gibt es Test::Unit und Test::Unit::Lite, welche Derivate von JUnit sind.

PHP

PHPUnit ist die Portierung von JUnit auf PHP und wurde von Sebastian Bergmann geschrieben.

Simpletest ist ebenfalls eine Portierung von JUnit, die um weitere Funktionen wie Mock Objects und Funktionen zum Testen von Web-Seiten erweitert wurde.

PL/SQL

utPLSQL[6] ist ein UnitTest-Framework für PL/SQL welches ebenfalls unter SourceForge geführt wird. Quest Code Tester for Oracle ist ein kommerzielles Produkt zur Defininion und Durchführung Unit Tests für PL/SQL. Es handelt sich um eine Weiterentwicklung von utPLSQL und wird vom Unternehmen Quest vertrieben.

Python

Unittest ist fester Bestandteil der Python-Standard-Bibliothek

Ruby

Unittest ist bei Ruby in der Standard-Bibliothek als Test::Unit oder RUnit verfügbar.

RSpec ist ein verhaltensgetriebenes Entwicklungs- und Testframework für Ruby.

MATLAB/Simulink

- *Simulink Design Verifier* von The MathWorks generiert Testfälle zur vollständigen Überdeckung und einen Testrahmen. Zum Einsatz kommen dabei Formale Methoden.
- *SystemTest* von The MathWorks kann Modelle testen und dabei auch Parameter verändern. Inputstimuli können manuell oder durch statistische Verteilungen generiert werden.
- *TPT* von PikeTec. TPT unterstützt die automatische Testrahmengenerierung inklusive einer automatischen Schnittstellenanalyse sowie die automatische Testdurchführung, Auswertung und Protokollierung.

Smalltalk

Das vermutlich erste Framework zum Erstellen von Komponententests (SUnit) wurde von Kent Beck für die Programmiersprache Smalltalk geschrieben. Die Idee wurde schnell auf andere Programmiersprachen übertragen.

Tcl

Tcl enthält das Modul tcltest für Modultests. Auch andere Test-Frameworks wurden in Tcl geschrieben, bekannt ist etwa DejaGnu, mit dem der gcc getestet wird. Zudem kann man in wenigen Zeilen ein „Framework“ selbst erstellen:

```
proc test {command expected} {
   catch {uplevel 1 $command} res
   if {$res ne $expected} {
      puts "$command->$res, not $expected"
   }
}
```

Tests (in eigenen Quelldateien, oder direkt beim Code) sehen dann so aus: man gibt eine Anweisung und das erwartete Ergebnis an. Wenn ein unerwartetes Ergebnis ausgewertet wird, so wird dies angezeigt:

```
test {expr 3 + 4} 7
```

Transact-SQL

TSQLUnit[7] ist ein Framework für Unittests in Transact-SQL. Es hält sich an die Tradition des *xUnit* Frameworks, das es für viele Programmiersprachen gibt.

Visual Basic 6

Das deutsche Unternehmen Maaß Computertechnik aus Bochum hat das xUnit Framework für Visual Basic 6 implementiert.

vbUnit ist nicht vollkommen Open Source. In der kostenpflichtigen Version 3 Professional erhält man das Unit Testing Framework für Visual Basic 6 inklusive einem Microsoft Visual Studio 6 Add-on.

Die Basic Version von vbUnit wird unter der GNU Lesser General Public License (LGPL) angeboten, das heißt dass man zu dieser Version sowohl die kompilierten Komponenten als auch den Quellcode erhält. Der TestRunner und das Add-on für Microsofts Visual Studio 6 in der Professional Version sind kostenpflichtig und nicht Open Source.

Weblinks

- opensourcetesting.org [8] (englisch) – Eine große Sammlung von Open Source Software-Testtools sowie ein recht aktives Forum mit Neuigkeiten und Diskussionen.
- xprogramming.com [9] – Liste von Unittest Frameworks für viele Programmiersprachen.

Einzelnachweise

[1] cunit.sourceforge.net (http://cunit.sourceforge.net/)
[2] cfix-testing.org (http://www.cfix-testing.org/)
[3] visualassert.com (http://www.visualassert.com/)
[4] savvytest (http://www.savignano.net/savvytest) beim Hersteller savignano software solutions (http://www.savignano.net/)
[5] testanything.org (http://testanything.org/wiki/index.php/Main_Page)
[6] utplsql (http://sourceforge.net/projects/utplsql/) bei SourceForge
[7] TSQLUnit (http://sourceforge.net/projects/tsqlunit/) bei SourceForge
[8] http://opensourcetesting.org/
[9] http://www.xprogramming.com/software.htm

Integrationstest

Der Begriff **Integrationstest** bezeichnet in der Softwareentwicklung eine aufeinander abgestimmte Reihe von Einzeltests, die dazu dienen, verschiedene voneinander abhängige Komponenten eines komplexen Systems im Zusammenspiel miteinander zu testen. Die erstmals im gemeinsamen Kontext zu testenden Komponenten haben jeweils einen Modultest erfolgreich bestanden und sind für sich isoliert fehlerfrei funktionsfähig.

Systematik

Für jede Abhängigkeit zwischen zwei Komponenten eines Systems wird ein Testszenario definiert, welches in der Lage ist nachzuweisen, dass nach der Zusammenführung sowohl beide Komponenten für sich wie auch der Datenaustausch über die gemeinsame(n) Schnittstelle(n) spezifikationsgemäß funktionieren. Als Methoden werden sowohl Funktionstests als auch Schnittstellentests angewendet. Da die Funktionstests meistens im Rahmen der Komponententests bereits durchgeführt wurden, dienen sie an dieser Stelle dazu festzustellen, ob die *richtige* Komponente verwendet wird. Die Schnittstellentests dienen zur Überprüfung der Daten, die zwischen den Komponenten ausgetauscht werden.

Umfang

Der Umfang von Integrationstests ist nicht auf ein Gesamtsystem festgelegt. Da der zeitliche Aufwand für Integrationstests mit wachsender Komponentenanzahl überproportional ansteigt, ist es üblich, Integrationstests für einzelne, abgegrenzte Subsysteme durchzuführen und diese dann im weiteren Verlauf als eine Komponente zu betrachten (Bottom-Up-Methode). Bei dieser Methode enden die Integrationstests erst mit den erfolgreichen Testläufen in einer mit dem späteren Produktivsystem identischen Testumgebung.

In kleineren Softwareprojekten finden Integrationstests häufig während der Programmierung durch den oder die Programmierer statt. Unmittelbar im Anschluss an die Programmierung eines Moduls wird das Modul selbst und das Zusammenspiel mit dem bisher erstellten Programmcode getestet. Bei großen, umfangreichen Software-Entwicklungen, die meist im Rahmen eines Projekts durchgeführt werden, erhöht sich der Aufwand für Tests generell so stark, dass diese zur Steigerung der Effizienz automatisiert durchgeführt werden.

Literatur

- Ian Sommerville: Software Engineering; Addison-Wesley, 6. Auflage, 2001, ISBN 3-827-37001-9
- IEEE: IEEE Standard Glossary of Software Engineering Terminology; IEEE, 1991, ISBN 1-559-37067-X
- British Computer Society: Glossary of terms used in software testing, Version 6.3; http://www.testingstandards.co.uk/bs_7925-1_online.htm
- Article Managing Your Way through the Integration and Test Black Hole [1] from Methods & Tools [2]

Referenzen

[1] http://www.methodsandtools.com/archive/archive.php?id=13

[2] http://www.methodsandtools.com/

Programmfehler

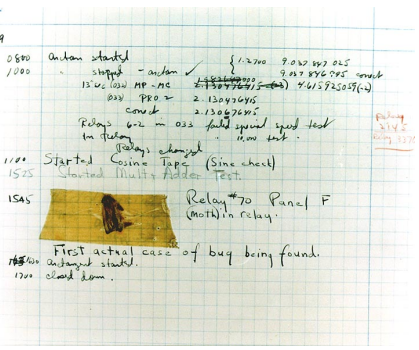
Ein **Programmfehler** oder **Softwarefehler**, häufig auch als **Bug** (bʌg) benannt, bezeichnet im Allgemeinen ein Fehlverhalten von Computerprogrammen. Dies tritt auf, wenn der Programmierer einen bestimmten Zustand in der Programmlogik beim Umsetzen der Vorgaben nicht berücksichtigt hat, oder wenn die Laufzeitumgebung fehlerhaft arbeitet. Weiterhin können auch Unvollständigkeit, Ungenauigkeit oder Mehrdeutigkeiten in der Spezifikation des Programms zu Fehlern führen. In der Praxis treten Computerprogramme ohne Programmfehler selten auf. Statistische Erhebungen in der Softwaretechnik weisen im Mittel etwa zwei bis drei Fehler je 1000 Zeilen Code aus.

Bei der Suche nach den Ursachen für Fehler in Programmen sind sogenannte Debugger hilfreich, mit denen ein Programm Schritt für Schritt ausgeführt werden kann. Die internen Zustände der (Variablen) können hierbei angezeigt werden.

Zur Erfassung und Dokumentation werden sogenannte *Bug-Tracker* (wie Bugzilla oder Mantis) eingesetzt. Diese nehmen sowohl Fehlerberichte, als auch Verbesserungsvorschläge und Wünsche der Nutzer oder allgemeine Vorgänge auf.

Der Vorgang des Beseitigens eines Programmfehlverhaltens wird umgangssprachlich *bugfixing* genannt. Das Ergebnis der Verbesserung wird in der Fachsprache als *Bugfix*, *Patch* oder *Softwarepatch* bezeichnet.

Das engl. Wort „Bug“ als Synonym für Programmfehler



Logbuch-Seite des Mark II Aiken Relay Calculator mit dem ersten *Bug* (1947)

Das Wort „Bug“ wurde schon im 19. Jahrhundert für kleine Fehler in mechanischen und elektrischen Teilen verwendet. Knistern und Rauschen in der Telefonleitung würden daher rühren, dass kleine Tiere („Bugs“: engl: Wanze) an der Leitung knabbern. Edison schrieb an seinen Freund Tivadar Puskás, den Erfinder der Telefonzentrale und Gründer einer Telefonzeitung einen Brief über die Entwicklung seiner Erfindungen, in dem er kleine Störungen und Schwierigkeiten als „Bugs“ bezeichnete. “The first step [in all of my inventions] is an intuition, and comes with a burst, then difficulties arise - this thing gives out and [it is] then that 'Bugs' - as such little faults and difficulties are called - show themselves” (Edison 1878[1] , deutsch: „Immer wenn man denkt, ein neues Maschinchen läuft wie geschmiert, streuen einem so genannte „Bugs“ Sand ins Getriebe.“) Edison ist zwar nicht Erfinder, aber immerhin Kronzeuge für eine schon damals kursierende Neuschöpfung.

Die Verknüpfung des Begriffs mit Computern geht möglicherweise auf die Computerpionierin Grace Hopper zurück.[2] Sie verbreitete die Geschichte, dass am 9. September 1945 eine Motte in einem Relais des Computers Mark II Aiken Relay Calculator zu einer Fehlfunktion führte. Die Motte wurde entfernt und in das Logbuch geklebt mit den Worten “First actual case of bug being found.” (deutsch: „Das erste Mal, dass tatsächlich ein ‚Käfer‘ gefunden wurde.“). Die Legende von der Begriffsfindung hält sich hartnäckig, obwohl die Logbuch-Eintragung gerade auf die frühere Verwendung des Begriffs hinweist. Zudem irrte Grace Hopper sich hinsichtlich des Jahres: Der Vorfall ereignete sich tatsächlich am 9. September 1947. Die entsprechende Seite des Logbuchs wurde bis Anfang der 1990er Jahre am *Naval Surface Warfare Center Computer Museum* der US-Marine in Dahlgren, Virginia, aufbewahrt. Zur Zeit befindet sich diese Logbuchseite mit der Motte am Smithsonian Institute.

Arten von Programmfehlern

In der Softwaretechnik wird zwischen folgenden Typen von Fehlern in Programmen unterschieden:

- **Syntaxfehler** sind Verstöße gegen die grammatischen Regeln der benutzten Programmiersprache. Ein Syntaxfehler verhindert die Kompilierung des fehlerhaften Programms. Bei Programmiersprachen, die sequentiell interpretiert werden, bricht das Programm an der syntaktisch fehlerhaften Stelle undefiniert ab.
- **Laufzeitfehler** sind alle Arten von Fehlern, die auftreten, während das Programm abgearbeitet wird. Da diese Fehler die Programmlogik und damit die Bedeutung des Programmcodes betreffen, spricht man hier auch von semantischen Fehlern. Ihre Ursache liegt in den meisten Fällen in einer inkorrekten Implementierung der gewünschten Funktionalität im Programm. Gelegentlich tritt als Ursache auch eine ungeeignete Laufzeitumgebung auf (z. B. eine falsche Betriebssystem-Version). Laufzeitfehler können sich auf die unterschiedlichsten Arten zeigen. Oftmals zeigt das Programm ungewünschtes Verhalten, im Extremfall wird die Ausführung des Programms abgebrochen („Absturz“), oder das Programm geht in einen Zustand über, in dem es keine Benutzereingaben mehr akzeptiert („Einfrieren“, „Hängen“). Wird in Programmiersprachen ohne automatische Speicherbereinigung (etwa C oder C++) Speicher nach der Verwendung nicht mehr freigegeben, so wird durch das Programm auf Dauer immer mehr Speicher belegt. Diese Situation wird Speicherleck genannt. Aber auch in Programmiersprachen mit automatischer Speicherbereinigung (etwa Java oder C#) können ähnliche Probleme auftreten, wenn zum Beispiel Objekte durch systemnahe Programmierung unkontrolliert angesammelt werden. Noch kritischer sind versehentlich vom Programmierer freigegebene Speicherbereiche, die oft trotzdem noch durch hängende Zeiger referenziert werden, da dies zu völlig unkontrolliertem Verhalten der Software führen kann. Manche Laufzeitumgebungen erlauben daher solche programmierbaren Speicherfreigaben grundsätzlich nicht. Des weiteren gibt es auch Bugs im Zusammenhang mit Multithreading, etwa Race Conditions, welche Konstellationen bezeichnen, in denen das Ergebnis einer Operation vom zeitlichen Verhalten bestimmter Einzeloperationen abhängt, oder Deadlocks.
- Fehler im Compiler, der Laufzeitumgebung oder sonstigen Bibliotheken. Solche Fehler sind meist besonders schwer nachzuvollziehen, da das Verhalten des Programms in solchen Fällen nicht seiner Semantik entspricht. Insbesondere von Compiler und Laufzeitumgebung wird daher besondere Zuverlässigkeit erwartet, welche jedoch gerade bei kleineren Projekten nicht immer gegeben ist.
- **Logische Fehler** und **semantische Fehler** bestehen in einem falschen Problemlösungsansatz, beispielsweise auf Grund eines Fehlschlusses oder eines fehlerhaften oder falsch interpretierten Algorithmus. Die Toleranz gegenüber solchen Fehlern und die diese einschränken sollende Attributgrammatik von Programmiersprachen, wie etwa bei der Zuweisungskompatibilität von Datentypen, sind je nach verwendeter Programmiersprache sehr unterschiedlich ausgeprägt.
- **Designfehler** sind Fehler im Grundkonzept, entweder bei der Definition der Anforderungen an die Software, oder bei der Entwicklung des Softwaredesigns, auf dessen Grundlage das Programm entwickelt wird. Fehler bei der Anforderungsdefinition beruhen oft auf mangelnder Kenntnis des Fachgebietes, für das die Software geschrieben wird oder auf Missverständnissen zwischen Nutzern und Entwicklern. Fehler direkt im Softwaredesign hingegen

sind oft auf mangelnde Erfahrung der Softwareentwickler oder auf Folgefehler durch Fehler in der Anforderungsspezifikation zurückzuführen. In anderen Fällen ist das Design historisch gewachsen und wird mit der Zeit unübersichtlich, was wiederum zu Designfehlern bei Weiterentwicklungen des Programms führen kann. Vielen Programmierern ist das Softwaredesign auch lästig, sodass oftmals ohne richtiges Konzept direkt entwickelt wird, was dann insbesondere bei steigendem Komplexitätsgrad der Software unweigerlich zu Designfehlern führt. Sowohl für Fehler in der Anforderungsdefinition als auch im Softwaredesign kommen darüber hinaus vielfach Kosten- oder Zeitdruck in Frage. Ein typischer Designfehler ist die Codewiederholung, die zwar nicht unmittelbar zu Programmfehlern führt, aber bei der Softwarewartung, der Modifikation oder der Erweiterung von Programmcode sehr leicht übersehen werden kann und dann unweigerlich zu unerwünschten Effekten führt.

- Ein **Regressionsbug** ist ein Fehler, der in einer früheren Programmversion bereits behoben wurde, der aber in einer späteren Programmversion wieder auftaucht.
- Fehler im Bedienkonzept. Das Programm verhält sich anders als es einzelne oder viele Anwender erwarten, obwohl es technisch an sich fehlerfrei arbeitet.
- Fehler infolge der Betriebsumgebung. Verschiedenste Begebenheiten wie elektromagnetische Felder, Strahlen, Temperaturschwankungen, Erschütterungen, usw., können auch bei sonst einwandfrei konfigurierten und innerhalb der Spezifikationen betriebenen Systemen zu Fehlern führen. Fehler dieses Typs sind sehr unwahrscheinlich, können nur sehr schwer festgestellt werden und haben bei Echtzeitanwendungen unter Umständen fatale Folgen. Sie dürfen aber aus statistischen Gründen nicht ausgeschlossen werden. Das berühmte "Umfallen eines Bits" im Speicher oder auf der Festplatte auf Grund der beschriebenen Einflüsse stellt zum Beispiel solch einen Fehler dar. Da die Auswirkungen eines solchen Fehlers (z.B. Absturz des Systems oder Boot-Unfähigkeit, weil eine Systemdatei beschädigt wurde) von denen anderer Programmfehler meist nur sehr schwer unterschieden werden können, vermutet man oft eine andere Ursache, zumal ein solcher Fehler häufig nicht reproduzierbar ist.

Bei manchen Projekten wird nicht der Begriff Bug verwendet, sondern man spricht zum Beispiel von Metabugs, bei denen ein Bug ein Element einer Aufgabenliste darstellt. Bei einigen Projekten spricht man stattdessen auch von „Issues“ (Angelegenheiten), da sich dieser Ausdruck nicht auf Programmfehler beschränkt.

Vermeidung und Behebung von Programmfehlern

Generell gilt der Leitsatz: *Je früher in einem Entwicklungsprozess der Fehler auftritt und je später er entdeckt wird, umso aufwendiger wird es, den Fehler zu beheben.*

Während der Planung

Am wichtigsten ist eine gute und geeignete Planung des Entwicklungsprozesses. Hierfür gibt es bereits etliche Prozessmodelle, aus denen ein geeignetes ausgewählt werden kann.

In der Analysephase

Ein Problem ist, dass die Korrektheit eines Programms nur gegen eine entsprechend formalisierte Spezifikation bewiesen werden kann. Eine solche Spezifikation zu erstellen kann jedoch im Einzelfall ähnlich kompliziert und fehlerträchtig sein, wie die Programmierung des Programms selbst.

Auch die Entwicklung immer abstrakterer Programmierparadigmen und Programmierstile wie die funktionale Programmierung, objektorientierte Programmierung, Design By Contract und die aspektorientierte Programmierung dienen unter anderem der Fehlervermeidung und Vereinfachung der Fehlersuche. Aus den zur Verfügung stehenden Techniken für das Problem ist eine geeignete auszuwählen. Ein wichtiger Punkt hierbei ist aber auch, dass für das jeweilige Paradigma erfahrene Programmierer zur Verfügung stehen müssen, sonst entsteht oft der gegenteilige

Effekt.

Ferner ist es sehr nützlich, von den Entwicklungswerkzeugen möglichst viele Aufgaben der Fehlervermeidung zuverlässig und automatisch erledigen zu lassen. Dies betrifft zum einen bereits Kontrollen wie Sichtbarkeitsregeln und Typsicherheit, sowie die Vermeidung von Zirkelbezügen, die bereits vor der Übersetzung von Programmen vom Compiler übernommen werden können, aber auch Kontrollen, die erst zur Laufzeit durchgeführt werden können, wie zum Beispiel Indexprüfung bei Datenfeldern oder Typprüfung bei Objekten der objektorientierten Programmierung.

In der Entwurfsphase

Softwareexperten sind sich darüber einig, dass praktisch jedes nicht-triviale Programm Fehler enthält. Deshalb wurden Techniken entwickelt, mit Fehlern innerhalb von Programmen tolerant umzugehen. Zu diesen Techniken gehören defensives Programmieren, Ausnahmebehandlung, Redundanz und die Überwachung von Programmen (z. B. durch Watchdog-timer) sowie die Plausibilisierung des Programmes während der Entwicklung und der Daten während des Programmablaufs.

Bei der Programmierung

Darüber hinaus wird eine Reihe fortgeschrittener Anwendungen angeboten, die entweder den Quellcode oder den Binärcode analysieren und versuchen, häufig gemachte Fehler automatisiert zu finden. In diese Kategorie fallen etwa Programme zur Ausführungsüberwachung, die üblicherweise fehlerhafte Speicherzugriffe und Speicherlecks zuverlässig aufspüren. Beispiele sind das frei erhältliche Tool Valgrind und das kommerzielle Purify. Eine weitere Kategorie von Prüfprogrammen umfasst Anwendungen, die Quell- oder Binärcode statisch analysieren und etwa nicht geschlossene Ressourcen und andere Probleme auffinden und melden können. Darunter fallen etwa Findbugs, Lint und Splint.

Beim Testen

Es ist durchaus sinnvoll, dass der Test vor dem eigentlichen Programm entwickelt wird. Damit wird erreicht, dass nicht ein Test geschrieben wird, der zu dem bereits geschriebenen Programm *passt*. Dies kann durch Ermittlung von Testfällen anhand der Spezifikation bereits während der Analyse- bzw. Designphase erfolgen. Die Ermittlung von Testfällen in diesem frühen Stadium der Softwareentwicklung ermöglicht zudem die Prüfung der Anforderungen an das Programm auf Testbarkeit und Vollständigkeit. Die anhand der Spezifikation ermittelten Testfälle stellen die Basis für die Abnahmetests und können kontinuierlich über den gesamten Entwicklungsprozess verfeinert werden.

Manche Softwareanbieter führen Testphasen teilweise öffentlich durch und geben Betaversionen heraus, um die unvorhersehbar vielfältigen Nutzungsbedingungen verschiedener Anwender durch diese selbst testen und kommentieren zu lassen.

Im Betrieb

Tritt ein Fehler während des Betriebs auf, so muss versucht werden, seine Auswirkungen möglichst gering zu halten und seinen Wirkungskreis durch Schaffung von „Schutzwällen“ oder „Sicherungen“ einzudämmen. Dies erfordert zum einen Möglichkeiten der Fehlererkennung und zum anderen, adäquat auf einen Fehler reagieren zu können.

Ein Beispiel zur Fehlererkennung zur Laufzeit eines Computerprogrammes sind Assertions, mit deren Hilfe Bedingungen abgefragt werden, die gemäß Programmdesign immer erfüllt sind. Weitere Mechanismen sind Ausnahmebehandlungen wie Trap und Exception.

Durch die Implementierung von Proof-Carrying Code kann die Software zur Laufzeit ihre Zuverlässigkeit in gewissem Rahmen gewährleisten und sicherstellen.

Fehlerfreiheit

Völlige Fehlerfreiheit für Software, die eine gewisse Komplexitätsgrenze überschreitet, ist weder erreichbar noch nachweisbar. Mit steigender Komplexität sinkt die Überblickbarkeit, insbesondere auch, wenn mehrere Personen an der Programmierung beteiligt sind. Selbst teure oder vielfach getestete Software enthält unweigerlich Programmierfehler. Man spricht dann bei gut brauchbaren Programmen nicht von Fehlerfreiheit, sondern von Stabilität und Robustheit. Eine Software gilt dann als stabil bzw. robust, wenn Fehler nur sehr selten auftreten und diese dann nur kleinere Unannehmlichkeiten mit sich bringen und keine größeren Schäden oder Verluste verursachen.

In Spezialfällen ist ein Beweis der Fehlerfreiheit eines Programms möglich. Insbesondere in Bereichen, in denen der Einsatz von Software mit hohen finanziellen, wirtschaftlichen oder menschlichen Risiken verbunden ist, wie z. B. bei militärisch oder medizinisch genutzter Software oder in der Luft- und Raumfahrt, verwendet man zudem eine (formale) Verifizierung genannte Methode, bei der die Korrektheit einer Software formal-mathematisch nachgewiesen wird. Dieser Methode sind allerdings wegen des enormen Aufwands enge Grenzen gesetzt und sie ist daher bei komplexen Programmen praktisch unmöglich durchzuführen (siehe auch Berechenbarkeit). Allerdings gibt es mittlerweile Werkzeuge, die diesen Nachweis laut eigenen Angaben zumindest für Teilbereiche (Laufzeitfehler) schnell und zuverlässig erbringen können.

Neben der mathematischen Verifizierung gibt es noch eine praxistaugliche Form der Verifizierung, die durch die Qualitätsmanagement-Norm ISO 9000 beschrieben wird. Bei ihr wird nur dann ein Fehler konstatiert, wenn eine Anforderung nicht erfüllt ist. Umgekehrt kann demnach ein Arbeitsergebnis (und damit auch Software) als fehlerfrei bezeichnet werden, wenn es nachweisbar alle Anforderungen erfüllt. Die Erfüllung einer Anforderung wird dabei durch Tests festgestellt. Wenn alle Tests das erwartete Ergebnis bringen, ist eine Anforderung erfüllt.

Folgen von Programmfehlern

Die Folgen eines Programmfehlers können außerordentlich sein und sich in vielfältiger Weise zeigen. Die folgende Liste ist zeitlich geordnet:

- 1982 stürzte ein Prototyp des F117 Kampfjets ab, da bei der Programmierung die Steuerung des Höhenruders mit der des Seitenruders vertauscht worden war.
- Zwischen 1985 und 1987 gab es mehrere Unfälle[3] mit dem medizinischen Bestrahlungsgerät Therac-25. Infolge einer Überdosis, die durch fehlerhafte Programmierung und fehlende Sicherungsmaßnahmen verursacht wurde, mussten Organe entfernt werden, drei Patienten verstarben aufgrund der Überdosis.
- Am 12. März 1995 kam es wegen eines um wenige Byte zu klein bemessenen Stapelspeichers in der Software eines Hamburger Stellwerks, bei dem auch das Ersatzsystem aus Sicherheitsgründen abgeschaltet wurde, zu massiven Verzögerungen im bundesweiten Zugverkehr.[4] Am 16. Dezember 2009 kam es zu einem ähnlichen Fehler im Stellwerk des Hauptbahnhofs Hannover.[5]
- Am 4. Juni 1996 wurde der Prototyp der Ariane-5-Rakete der Europäischen Raumfahrtbehörde eine Minute nach dem Start in vier Kilometern Höhe gesprengt. Der Programmcode für die Steuerung war von der Ariane 4 übernommen worden und funktioniert nur in einem von der Ariane 4 nicht überschreitbaren Bereich (Beschleunigungswert). Die Steuersysteme kamen zum Erliegen, als ebendieser Bereich von der Ariane 5 überschritten wurde, die stärker beschleunigt als die Ariane 4. Bei der Programmierung war es zu einem Fehler bei einer Typumwandlung gekommen[6] , dessen Auftreten durch die verwendete Programmiersprache Ada eigentlich hätte entdeckt und behandelt werden können. Diese Sicherheitsfunktionalität ließen die Verantwortlichen jedoch abschalten. Der Schaden betrug etwa 370 Millionen US-Dollar.
- 1999 verpasste die NASA-Sonde Mars Climate Orbiter den Landeanflug auf den Mars, weil die Programmierer das falsche Maßsystem verwendeten - Pound-force · Sekunde statt Newton · Sekunde. Die NASA verlor dadurch die Sonde.

- Bei Toll Collect kam es 2003[7] unter anderem wegen der fehlenden Kompatibilität von Softwaremodulen zu drastischen Verzögerungen mit Vertragsstrafen und Einnahmeausfällen in Milliardenhöhe.
- Am 8. Oktober 2005 führte im russischen Plessezk ein Programmfehler zum Fehlstart einer Trägerrakete und zum Verlust des Satelliten CryoSat.
- Anfang November 2005 konnte an der Tokioter Börse wegen eines Programmfehlers stundenlang kein Handel betrieben werden. Auch in den nachfolgenden Wochen gab es viele fehlerhafte Wertpapierordern, die in einem Fall sogar einen finanziellen Schaden von über 300 Millionen Dollar ausmachte. Der Präsident der Börse, Takuo Tsurushima, trat daraufhin von seinem Amt zurück.[8]
- 2007 kam es in Österreich bei der Umstellung auf digitales Satellitenfernsehen zu Problemen in rund 30.000 Haushalten, weil Software im Digitalreceiver fehlerhaft war[9] .
- In der Nacht vom 29. zum 30. Oktober 2007 kam es bei der Deutschen Telekom zu bundesweiten Ausfällen und Fehlverbindungen bei Sprachdiensten im Kommunikationssystem, weil eine aktualisierte Server-Software fehlerhaft war. Das System konnte erst nach mehreren Stunden durch die Installation der alten Software wieder zum Laufen gebracht werden[10] [11] .
- Im Oktober 2007 kamen zehn Angehörige der südafrikanischen Armee aufgrund eines Programmfehlers in einem vollautomatisierten 35-mm-Flakgeschütz ums Leben.[12]
- Zu Beginn des Jahres 2010 konnten wegen eines Softwarefehlers im EMV-Sicherheitschip bei der Verarbeitung der Jahreszahl 2010 viele Bankkunden tagelang an Automaten kein Geld abheben.[13]

Reproduzierbarkeit von Programmfehlern

Manche Programmfehler sind nur äußerst schwer oder gar nicht zuverlässig reproduzierbar. Bei der Wiederholung eines zuvor gescheiterten Vorgangs unter scheinbar unveränderten Bedingungen ist die Wahrscheinlichkeit hoch, dass sich diese Fehler nicht erneut äußern. Es gibt zwei mögliche Gründe für dieses Verhalten: Zum einen kann es zu Verzögerungen zwischen der Fehleraktivierung und dem letztlich auftretenden Problem beispielsweise einem Programmabsturz kommen, welche die tatsächliche Ursache verschleiern und deren Identifikation erschweren. Zum anderen können andere Elemente des Softwaresystems (Hardware, Betriebssystem, andere Programme) das Verhalten der Fehler in dem betrachteten Programm beeinflussen. Ein Beispiel hierfür sind Fehler, die in nebenläufigen Umgebungen mit mangelnder Synchronisation (genauer: Sequentialisierung) auftreten. Wegen der hieraus folgenden Wettlaufsituationen (*Race Conditions*) können die Prozesse in einer Reihenfolge abgearbeitet werden, welche zu einem Laufzeitfehler führt. Bei einer Wiederholung der gleichen Aktion ist es möglich, dass die Reihenfolge der Prozesse unterschiedlich ist und kein Problem auftritt.

Weblinks

- Die 25 gefährlichsten Programmierfehler [14] *(englisch)*
- Ansicht von Windowsfehlermeldungen [15] *(deutsch)*

Literatur

- William E. Perry: Software Testen, Mitp-Verlag, November 2002, ISBN 3-8266-0887-9
- Elfriede Dustin, Jeff Rashka, John Paul: Software automatisch testen, Springer, Berlin, Januar 2001, ISBN 3-540-67639-2
- Cem Kaner, Jack Falk, Hung Q. Nguyen: Testing Computer Software, John Wiley & Sons, Juni 1999, ISBN 0-471-35846-0

Einzelnachweise

[1] (https://www.vdi-nachrichten.com/allgemein/denkmal.asp)

[2] Vgl. den Beitrag zu *Rear Admiral Grace Murray Hopper* in *The History of Computing* (engl.) (http://ei.cs.vt.edu/~history/Hopper.Danis.html)

[3] Nancy G. Leveson and Clark S. Tyler, *An Investigation of the Therac-25 Accidents*, IEEE Computer 26(7), p. 18-14, 1993

[4] Software-Katastrophen - Hamburger Stellwerk (Link zur Wayback Machine des Internet Archive) (http://web.archive.org/web/20070928164336/http://www.munopag.de/archiv/SoftwareEngineering/2004Sommersemester/1__Folien/I-Einfuehrung-2-auf-1.pdf)

[5] Software-Panne: Chaos am Hauptbahnhof Hannover (http://www.heise.de/newsticker/meldung/Software-Panne-Chaos-am-Hauptbahnhof-Hannover-887996.html)

[6] Vortrag über Software Reliability (http://www-aix.gsi.de/~giese/swr/ariane5.html) von I. Giese (GSI), u.a. mit Ausschnitt des ursächlichen Quellecodes

[7] Pressemitteilung Toll Collect (http://www.toll-collect.de/frontend/press/PressEntryVP.do;jsessionid=FC27809D4527D427594A75570E8A0B68?pressId=522)

[8] Tokioter Börse - Der Chef tritt ab (http://www.manager-magazin.de/koepfe/artikel/0,2828,391434,00.html), Manager Magazin vom 20. Dezember 2005

[9] ORF - 22. August 2007 - *Fehlerhafte Software verhindert Empfang der ORF-Programme* (http://help.orf.at/?story=6637)

[10] Server-Ausfall legt bundesweit Telekom-Leitungen lahm (http://www.tagesspiegel.de/wirtschaft/Unternehmen-Telekom;art129,2409995) - Tagesspiegel vom 30. Oktober 2007

[11] Online FOCUS - Hunderttausendfach „falsch verbunden“ (http://www.focus.de/digital/handy/telefonausfall_aid_137569.html)

[12] heise online - Defektes Computersystem für den Tod von zehn Soldaten verantwortlich? (http://www.heise.de/newsticker/meldung/97656/from/atom10)

[13] Panne zum Jahreswechsel (http://www.tagesspiegel.de/wirtschaft/EC-Karten-Postbank-Commerzbank;art271,2992727) - Tagesspiegel vom 5. Januar 2010

[14] http://cwe.mitre.org/top25/

[15] http://www.kublikon.de/fehlermeldungen/

Proof-Carrying Code

Proof-Carrying Code (PCC) ist ein 1996 von George Necula und Peter Lee entwickelter, effizienter Algorithmus für Computer, mit dessen Hilfe die Eigenschaften von Anwendungssoftware und insbesondere die Einhaltung von Sicherheitsrichtlinien überprüft und verifiziert werden können.

Der automatische Algorithmus benutzt ein Axiomensystem, um den Programmcode begleitende Metadaten zu analysieren. Dabei kann geschlussfolgert und gewährleistet werden, dass bestimmte sicherheitsrelevante Kriterien eingehalten werden. Zur Laufzeit müssen dann keine entsprechenden zusätzlichen Maßnahmen ergriffen werden, wie zum Beispiel die Ausnahmebehandlung bei kritischem Verhalten der Software. Proof-Carrying Code ist ferner besonders nützlich, um Sicherheitslücken, wie zum Beispiel Pufferüberläufe oder Mehrdeutigkeiten (beispielsweise Typverletzung, Überladen oder Polymorphie), zu verhindern, die häufig durch die Benutzung von unzureichenden Programmiersprachen bedingt sind.

Mit dem Proof-Carrying Code kann auf einem Client bei der Installation und der Ausführung von Computerprogrammen die Zuverlässigkeit und Vertrauenswürdigkeit einer Programmquelle in einem Rechnernetz überprüft werden. Dabei werden Metadaten vom Host, dem sogenannten *Programmcodeproduzenten* abgerufen, mit deren Hilfe die Überprüfung auf dem Client, dem sogenannten *Programmcodeverbraucher*, stattfinden kann.

Weblinks

- George C. Necula und Peter Lee. *Safe, Untrusted Agents Using Proof-Carrying Code* [1]. Mobile Agents and Security, Giovanni Vigna (Herausgeber), Lecture Notes in Computer Science, Volume 1419, Springer-Verlag, Berlin, ISBN 3-540-64792-9, 1998
- George C. Necula. *Compiling with Proofs* [2]. PhD thesis, School of Computer Science, Carnegie Mellon University, September 1998
- Proof-Carrying Code im Projekt MOBIUS [3]

Referenzen

[1] http://www.cs.cmu.edu/~petel/publications/lncs98.pdf
[2] http://www.cs.berkeley.edu/~necula/Papers/thesis.pdf
[3] http://mobius.inria.fr/twiki/bin/view/Mobius/ProofCarryingCode

Assertion (Informatik)

Eine **Zusicherung** oder **Assertion** (lat./engl. für Aussage; Behauptung) ist eine Aussage über den Zustand eines Computer-Programms oder einer elektronischen Schaltung. Mit Hilfe von Zusicherungen können logische Fehler im Programm oder Defekte in der umgebenden Hard- oder Software erkannt und das Programm kontrolliert beendet werden. Bei der Entwicklung elektronischer Schaltungen kann mittels Assertions die Einhaltung der Spezifikation in der Verifikationsphase überprüft werden. Des Weiteren können Assertions Informationen über den Grad der Testabdeckung während der Verifikation liefern.

Anwendung

Durch die Formulierung einer Zusicherung bringt der Entwickler eines Programms seine Überzeugung über bestimmte Bedingungen während der Laufzeit eines Programms zum Ausdruck und lässt sie *Teil des Programms* werden. Er trennt diese Überzeugungen von den normalen Laufzeitumständen ab und nimmt diese Bedingungen als stets wahr an. Abweichungen hiervon werden nicht regulär behandelt, damit die Vielzahl möglicher Fälle nicht die Lösung des Problems vereitelt, denn natürlich kann es während der Laufzeit eines Programms dazu kommen, dass 2+2=4 einmal nicht gilt, z. B. weil Variablen durch Programmfehler im Betriebssystem überschrieben wurden.

Damit unterscheiden sich Zusicherungen von der klassischen Fehlerkontrolle durch Kontrollstrukturen oder Ausnahmen (Exceptions), die einen Fehlerfall als mögliches Ergebnis *einschließen*. In einigen Programmiersprachen wurden Zusicherungen auf Sprachebene eingebaut, häufig werden sie als Sonderform der Ausnahmen verwirklicht.

Geschichte

Eingeführt wurde der Begriff **Assertion** von Robert Floyd 1967 in seinem Artikel *Assigning Meanings to Programs*. Er schlug eine Methode vor, mit der man die Korrektheit von Flussdiagrammen beweisen konnte indem man jedes Element des Flussdiagramms mit einer Zusicherung versieht. Floyd gab Regeln an, nach denen die Zusicherungen bestimmt werden konnten. Tony Hoare entwickelte diese Methode zum Hoare-Kalkül für prozedurale Programmiersprachen weiter. Im Hoare-Kalkül wird eine Zusicherung, die vor einer Anweisung steht, *Vorbedingung* (engl. precondition), eine Zusicherung nach einer Anweisung *Nachbedingung* (engl. postcondition) genannt. Eine Zusicherung, die bei jedem Schleifendurchlauf erfüllt sein muss, heißt *Invariante*.

Niklaus Wirth benutzte Zusicherungen zur Definition der Semantik von Pascal und schlug vor, dass Programmierer in ihre Programme Kommentare mit Zusicherungen schreiben sollten. Aus diesem Grund sind Kommentare in Pascal mit geschweiften Klammern {...} umgeben, eine Syntax, die Hoare in seinem Kalkül für Zusicherungen verwendet

hatte.

In Borland Delphi wurde die Idee übernommen und ist als System-Funktion `assert` eingebaut. In der Programmiersprache Java steht seit Version 1.4 das Schlüsselwort `assert` zur Verfügung.

Die zur Entwicklung elektronischer Schaltungen eingesetzten Hardwarebeschreibungssprachen VHDL und SystemVerilog unterstützen Assertions. PSL ist eine eigenständige Beschreibungssprache für Assertions, die Modelle in VHDL, Verilog und SystemC unterstützt. Während der Verifikation wird vom Simulationswerkzeug erfasst, wie oft die Assertion ausgelöst wurde und wie oft die Zusicherung erfüllt oder verletzt wurde. Wurde die Assertion ausgelöst und die Zusicherung nie verletzt, gilt die Schaltung als erfolgreich verifiziert. Wurde jedoch die Assertion während der Simulation nie ausgelöst, besteht eine mangelnde Testabdeckung und die Verifikationsumgebung muss erweitert werden.

Programmierpraxis

In der Programmiersprache C könnte eine Zusicherung in etwa so eingesetzt werden:

```c
#include <assert.h>
/// diese Funktion liefert die Länge eines nullterminierten Strings
/// falls der übergebene auf die Adresse NULL verweist, wird das
/// Programm kontrolliert abgebrochen. (strlen prüft das nicht selbst)
int strlenChecked(char* s) {
  assert(s != NULL);
  return strlen(s);
}
```

In diesem Beispiel wird über die Einbindung der Header-Datei assert.h das Makro `assert` verwendbar. Dieses Makro sorgt im Falle eines Fehlschlags für die Ausgabe einer Standardmeldung, in der die nicht erfüllte Bedingung zitiert wird und Dateiname und Zeilennummer hinzugefügt werden. Eine solche Meldung sieht dann so aus:

```
Assertion "s!=NULL" failed in file "C:\Projects\Sudoku\utils.c", line 9
```

Java kennt das Konzept der Zusicherungen ab Version 1.4. Hier wird allerdings das Programm nicht notwendigerweise beendet, sondern eine so genannte Ausnahme (englisch *exception*) geworfen, die innerhalb des Programms weiter verarbeitet werden kann.

Ein einfaches Beispiel einer Assertion (hier in Java-Syntax) ist

```java
int n = readInput();
n = n * n; //Quadrieren
assert n >= 0;
```

Mit dieser Assertion sagt der Programmierer "*Ich bin mir sicher, dass nach dieser Stelle n größer gleich null ist*".

Bertrand Meyer hat die Idee von Zusicherungen in dem Programmierparadigma Design by Contract verarbeitet und in der Programmiersprache Eiffel umgesetzt. Vorbedingungen werden durch *require*-Klauseln, Nachbedingungen durch *ensure*-Klauseln beschrieben. Für Klassen können Invarianten spezifiziert werden. Auch in Eiffel werden Ausnahmen geworfen, wenn eine Zusicherung nicht erfüllt ist.

Verwandte Techniken

Assertions entdecken Programmfehler zur Laufzeit beim Anwender, also erst wenn es zu spät ist. Um Meldungen über "Interne Fehler" möglichst zu vermeiden, versucht man durch geeignete Formulierung des Quelltextes logische Fehler bereits zur Kompilierzeit (durch den Compiler) in Form von Fehlern und Warnungen aufdecken zu lassen. Logische Fehler, die man auf diese Weise nicht finden kann, können häufig mittels Modultests aufgedeckt werden.

Umformulieren des Quelltextes

Indem Fallunterscheidungen auf ein Minimum reduziert werden, ist mancher Fehler nicht ausdrückbar, dann ist er als logischer Fehler auch nicht möglich. Nehmen wir an es gäbe nur zwei mögliche Fälle (das ist in der Tat häufig so) dann könnten wir einen einfachen Wahrheitswert verwenden anstelle einer spezielleren Kodierung:

```
/// Funktion liefert den Text zu einem Geschlechtskode
const char* genderStr(bool isFemale) {
  return isFemale ? "weiblich" : "männlich";
}
```

auch wenn es politisch korrekter wäre, enthält das folgende Beispiel eine (scheinbar) absurde Fehlermöglichkeit:

```
/// Geschlechtskennung als Enumeration:
enum GENDER { GENDER_MALE, GENDER_FEMALE };
/// Funktion liefert den Text zu einem Geschlechtskode
const char* genderStr(GENDER gender) {
  switch(gender) {
    case GENDER_FEMALE: return "weiblich";
    case GENDER_MALE:   return "männlich";
    default:            assert(0); return ""; // ???
  }
}
```

Zusicherungen zur Kompilierzeit

Während die oben beschriebenen Assertionen zur Laufzeit des Programms geprüft werden, gibt es in C++ die Möglichkeit, Bedingungen auch schon beim Übersetzen des Programms durch den Compiler zu überprüfen. Es können nur Bedingungen nachgeprüft werden, die zur Übersetzungszeit bekannt sind, z. B. sizeof(int) == 4. Schlägt ein Test fehl, lässt sich das Programm nicht übersetzen:

```
#if sizeof(int) != 4
  #error "unerwartete int-Größe"
#endif
```

Allerdings hängt das stark von der Funktionalität des jeweiligen Compilers ab und manche Formulierungen sind gewöhnungsbedürftig:

```
/// Die Korrektheit unserer Implementierung hängt davon ab,
/// dass ein int 4 Bytes groß ist. Falls dies nicht gilt,
/// bricht der Compiler mit der Meldung ab, dass
/// Arrays mindestens ein Element haben müssen:
void validIntSize(void) {
  int valid[sizeof(int)==4];
```

```
}
```

Zusicherungen in Modultests

Ein Bereich in dem ebenfalls Zusicherungen eine Rolle spielen, sind Modultests (u. A. Kernbestandteil des Extreme Programmings). Wann immer man am Quelltext Änderungen (z. B. Refactorings) vornimmt, z. B. um weitere Funktionen zu integrieren, führt man Tests auf Teilfunktionen (Modulen, also z. B. Funktionen und Prozeduren, Klassen) aus, um die bekannte (gewünschte) Funktionalität zu evaluieren.

Im Gegensatz zu assert wird bei einem Fehlschlag nicht das ganze Programm beendet, sondern nur der Test als gescheitert betrachtet und weitere Tests ausgeführt, um möglichst viele Fehler zu finden. Laufen alle Tests fehlerfrei, dann sollte davon ausgegangen werden können, dass die gewünschte Funktionalität besteht.

Von besonderer Bedeutung ist der Umstand, dass Modultests üblicherweise nicht im Produktivcode ausgeführt werden, sondern zur Entwicklungszeit. Das bedeutet, dass Assertions im fertigen Programm als Gegenpart zu betrachten sind.

Spezielle Techniken für Microsoft-Compiler

Neben Assert wird häufig auch **Verify** verwendet. Verify führt alle Anweisungen aus, die in die Berechnung der Bedingung einfließen, gleichgültig, ob mit oder ohne Debug-Absicht kompiliert wurde. Die Überprüfung findet aber nur in der Debug-Variante statt, d.h. auch hier kann ein Programm in obskurer Weise scheitern, falls die Zusicherung nicht erfüllt ist.

```
// die Variable anzahl wird immer erhöht
Verify(++anzahl>2);
```

Assert_Valid wird eingesetzt, um Objekte auf ihre Gültigkeit zu testen. Dazu gibt es in der Basisklasse der Objekthierarchie die virtuelle Methode AssertValid(). Die Methode sollte für jede konkrete Klasse überschrieben werden, um die internen Felder der Klasse auf Gültigkeit zu testen.

```
// Example for CObject::AssertValid.
void CAge::AssertValid() const
{
  CObject::AssertValid();
  ASSERT( m_years > 0 );
  ASSERT( m_years < 105 );
}
```

Weblinks

- Assertions in Java 5.0 [1]
- Assertions in C++ mit Boost [2]

Literatur

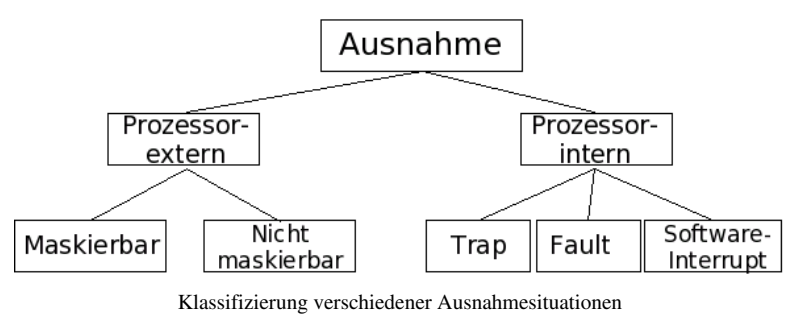
- Robert W. Floyd: *Assigning meanings to programs,* Proceedings of Symposia in Applied Mathematics, Volume 19 (1967), Seite 19–32.

Referenzen

[1] http://java.sun.com/j2se/1.5.0/docs/guide/language/assert.html
[2] http://www.boost.org/doc/html/boost_staticassert.html

Ausnahmebehandlung

Eine **Ausnahme** oder **Ausnahmesituation** (engl. *exception*) bezeichnet in der Computertechnik ein Verfahren, Informationen über bestimmte Programmzustände – meistens Fehlerzustände – an andere Programmebenen zur Weiterbehandlung weiterzureichen.



Klassifizierung verschiedener Ausnahmesituationen

Kann in einem Programm beispielsweise einer Speicheranforderung nicht stattgegeben werden, wird eine *Speicheranforderungs-Ausnahme* ausgelöst. Ein Computerprogramm kann zur Behandlung dieses Problems dafür definierte Algorithmen abarbeiten, die den Fehler ignorieren, beheben oder anzeigen.

Sachgemäße Anwendung von Ausnahmen ermöglicht es Programmen, flexibler auf unvorhergesehene Probleme zu reagieren und sich fehlertoleranter zu verhalten.

Strukturierte Ausnahmebehandlung

Die größte Rolle in der Programmierung spielt dabei die **strukturierte Ausnahmebehandlung** (englisch *structured exception handling*, kurz *SEH*), eine Technik, die den Code zur Ausnahmebehandlung vom normalen Anwendungscode getrennt hält. Anstatt beispielsweise bei jedem Funktionsaufruf einen Rückgabewert, der den Erfolg anzeigt, zu prüfen und darauf zu reagieren, kann man die betreffende Funktion in Ausnahmesituationen eine Ausnahme auslösen lassen, die alle für die Problemerkennung und -behandlung erforderlichen Informationen in sich trägt.

Da die verursachende Funktion (oder die Funktion, die das Problem feststellt) in ihrem Kontext den Fehler möglicherweise nicht angemessen behandeln kann, wird die Exception so lange an aufrufende Funktionen zurückgereicht, bis schließlich eine die Exception „fängt“. Dies baut man sinnvollerweise an einer Stelle im Code ein, wo abzusehen ist, was für Konsequenzen diese Ausnahme haben kann und wie man darauf am besten reagiert (zum Beispiel neue Benutzereingabe, Programmabbruch, Abbruch einer bestimmten Operation). Außerdem ist jeder Code-Abschnitt, der bezüglich der Ausnahmebehandlung nichts ausrichten kann, frei von Fehlerbehandlungsroutinen.

Ein weiterer entscheidender Vorteil gegenüber der Fehlerbehandlung über Rückgabewerte ist, dass eine Exception nicht ignoriert werden kann. Ein Programmierer könnte vergessen, einen Rückgabewert zu prüfen, aber eine Exception wird immer weiter zurückgereicht - im Extremfall so lange, bis sie in der programmstartenden Funktion ankommt, falls der Programmierer es unterlassen hat, sie abzufangen. In diesem Fall führt das oft zu einem Programmabbruch. Dies mag etwas subtil erscheinen, das Resultat ist jedoch meistens, dass Programme, die *trotz* Fehlerbehandlung per Exceptions stabil laufen, auch *wirklich* stabil laufen. Demgegenüber kann das Ignorieren eines Fehlerwertes viele Male keine drastischen Konsequenzen haben und irgendwann könnte das Programm eventuell doch unerwartete Ergebnisse produzieren.

Programmiersprachen, die Ausnahmebehandlung unterstützen, sind zum Beispiel C++, Java, C#, Delphi, Visual Basic.Net, PHP (ab Version 5), Python, Common Lisp, Ruby, Eiffel, Ada und Objective CAML.

Verschiedene Hardware-Architekturen (wie zum Beispiel die IA-32-Architektur von Intel) unterstützen eine Exception-Behandlung auf Hardware-Ebene durch das Betriebssystem. Hierbei werden bei bestimmten ungültigen Operationen Software-Interrupts ausgelöst, die einen Einsprung in den privilegierten Betriebssystemkern verursachen. Dieser kann dann anhand der Exception das Programm mit einer Fehlermeldung beenden oder den Fehler an einen Debugger weiterleiten.

Checked Exceptions

Bei Java gibt es als Weiterentwicklung der Ausnahme die „Checked Exception“ (dt. etwa: *überprüfte Ausnahme*). Das ist eine Ausnahme, bei der der Compiler prüft, ob alle Stellen, wo sie auftreten kann, durch Code zum Abfangen der Ausnahme abgedeckt sind. Der Code zum Abfangen kann dabei innerhalb derselben Methode stehen, in der die Ausnahme auftreten kann, oder auch in aufrufenden Methoden. In letzterem Fall muss der Programmierer die Ausnahmen in der Methodensignatur deklarieren.

Die zugrunde liegende Idee beim Entwurf von Java war, dass Ausnahmen, auf die der Anwendungscode sinnvoll reagieren kann, als Checked Exception ausgeführt werden. Durch den Zwang zur Behandlung der Ausnahme sollte robuster Code erreicht werden und fehlende Fehlerbehandlungen bereits vom Compiler entdeckt werden.[1] Es gibt aber weiterhin Ausnahmen, die keine Checked Exceptions sind. Als Konvention gilt dabei, solche Fehler als Checked Exception zu realisieren, bei denen man vom Aufrufer erwartet, dass er auf ihn reagieren und einen geregelten Programmablauf wiederherstellen kann. Darunter fallen beispielsweise Netzwerk-, Datenbank- oder sonstige E/A-Fehler. So kann das Öffnen einer Datei aus verschiedenen Gründen fehlschlagen (keine Rechte, Datei nicht vorhanden), der Aufbau einer Netzwerkverbindung kann aus vom Programm nicht zu beeinflussenden Gründen fehlschlagen. Nicht-Checked-Exceptions sind zum Melden verschiedener Arten von Programmfehlern vorgesehen (zum Beispiel Indexfehler bei Array-Indizierung). Es wird davon abgeraten, die Anwendung in solchen Fällen versuchen zu lassen, einen geregelten Programmablauf wiederherzustellen.[2] Die Klassen der Java-Plattform selber halten sich weitgehend an diese Konvention.

Kritiker führen gegen die Checked Exceptions an, dass sie die Lesbarkeit des Quellcodes verschlechtern würden und dass sie viele Programmierer, weil sie in dieser Funktionalität keinen dem Aufwand entsprechenden Nutzen erkennen, zu Ausweichkonstrukten verleiten, die dem Compiler genügen, aber kaum Fehler behandeln.[3] Ein anderer Einwand ist, dass aufgrund der Deklaration der Exceptions in den Methodensignaturen allgemein verwendbare Hilfsklassen oder Interfaces, insbesondere als Teil von Entwurfsmustern, oft nicht sinnvoll operabel sind mit Klassen, die Checked Exceptions verwenden.[4] Als Ausweichlösung werden getunnelte Checked Exceptions vorgeschlagen, die aber den Nutzen der Checked Exception aufheben.[5]

Auslösen von Exceptions

Eine Exception kann an jeder Stelle im Programmcode ausgelöst werden. Dabei wird fast immer ein Objekt einer Exception-Klasse erzeugt und mit dem Schlüsselwort throw oder raise abgeschickt. Bei manchen Programmiersprachen (zum Beispiel C++) darf statt der Exception-Klasse auch jeder andere Datentyp verwendet werden.

Abfangen von Exceptions

Wird eine Exception im Programmablauf nicht explizit abgefangen, dann wird sie von der Laufzeitbibliothek aufgefangen. Die Exception wird als Fehlermeldung angezeigt; je nach Art der Exception wird die Anwendung abgebrochen oder fortgesetzt. Anwendungen ohne eigene Benutzeroberfläche werden immer abgebrochen.

Häufige Fehler bei der Ausnahmebehandlung sind

- Exceptions werden ohne weitere Aktionen geschluckt. Somit geht die eigentliche Fehlerursache verloren.
- Exceptions werden durch eine eigene (häufig unzutreffende) Meldung ersetzt.

Es ist sinnvoll, Exceptions abzufangen, um zusätzliche Informationen anzureichern und erneut auszulösen.

Beispiele

Borland Delphi

```
ObjektA := TObjectA.Create;
try
  try
    BerechneEinkommen(name);
  except
    on E:Exception do
    begin
      // Exception wurde abgefangen und wird um einen aussagekräftigen
Hinweis ergänzt
      E.Message := 'Fehler beim Berechnen des Einkommens von
'+name+#13#10+
        E.Message; // ursprüngliche Meldung anhängen
      Raise;  // veränderte Exception erneut auslösen
    end;
  end;
finally
  FreeAndNil(ObjektA); //dies wird auf jeden Fall ausgeführt
end;
```

C++

```
try {
    funktion1();
    funktion2();
   ...
} catch (const std::invalid_argument &e) {
    std::cerr << "Falsches Argument:"  << e.what() << std::endl;
} catch (const std::range_error &e) {
    std::cerr << "Ungültiger Bereich:" << e.what() << std::endl;
```

```
} catch (...) {
    std::cerr << "Sonstige Exception" << std::endl;
}
```

C#

```
try
{
  TryToDoSomething(23, 42);
}
catch (Exception ex)
{
  // Fehler behandeln.
  System.Console.WriteLine("Ausnahme aufgetreten: {0}", ex.Message);
}
finally
{
  // Wird auf jeden Fall ausgeführt.
  System.Console.WriteLine("Versuch durchgeführt.");
}
```

Visual Basic .NET

```
' Versuche ...
Try
  ' ... die Methode oder Prozedur ...
  BerechneEinkommen(name)
' bei Ausnahme
Catch ex AS Exception
  ' gib Ausnahme aus
  MessageBox.Show("Fehler -> " & ex.message)
' Führe auf jeden Fall aus
Finally
  MessageBox.Show("Das wird trotzdem ausgeführt")
End Try
```

Java

```
try {
    //Berechne ...
} catch (OutOfMemoryError e) {
    //Ein Error ist keine Exception und muss separat abgefangen werden
    e.printStackTrace();
} catch (RuntimeException e) {
    //z.B. IndexOutOfBoundsException, NullPointerException usw.
    System.err.println("Offensichtlich ein Programmierfehler!");
    throw e; //Leite nach oben weiter
} catch (Exception e) {
    //Fange alle restlichen Ausnahmefehler ab
    e.printStackTrace();
```

```
} catch (Throwable t) {
    //Das hier fängt wirklich alles ab
    t.printStackTrace();
} finally {
    //Ob Exception oder nicht, führe das hier auf jeden Fall aus.
    System.out.println("Berechnung beendet oder abgebrochen");
}
```

PHP

```
//Exceptionhandling ab PHP Version 5!
try {
    //Berechne ...
    throw new RuntimeException('Fehlermeldung',543); //Fehlermeldung,
Fehlercode
} catch (RuntimeException $e) {
    //z.B. IndexOutOfBoundsException, NullPointerException usw.
    //Wichtig: Um die hier verwendeten Basistypen zu nutzen muss die
"SPL" installiert sein
    echo($e->getMessage());
    throw $e; //Leite nach oben weiter
} catch (Exception $e) {
    //Fange alle restlichen Ausnahmefehler ab und verwende die
__toString() - Methode zur Ausgabe
    echo($e);
}
```

Python

```
try:
    result = doSomething()
    if result < 0 :
        raise StandardError

except StandardError:
    print("catching exception")
    doSomethingElse()

except :
    print("exception in method doSomething")
```

Siehe auch Python, Abschnitt Ausnahmebehandlung.

Perl

```
eval {
    something_fatal(...);
    1;
} or do {
    warn "Exception: $@";
};
```

```
# alternativ/ergänzend kann man auch objektorientierte Exception-Module
 verwenden
```

Stacktrace

Der Stacktrace kann insbesondere für eine manuelle Sichtung der Situation bei Auftreten einer Ausnahme wertvoll sein, um insbesondere bei nicht aufgefangenen Ausnahmen die Ursache schnell erkennen und beheben zu können. Der Stacktrace, das ist die Aufrufreihenfolge aller Methoden bis zur Ausnahme, kann beispielsweise in ein Log-System ausgegeben werden und auf diese Weise einem Entwicklerteam zur Verfügung stehen, auch wenn eine nicht behandelte Ausnahme in einer Software beim Kunden auftritt. In Java ist das Exceptionhandling unmittelbar mit dem Stacktrace verbunden.

Technische Umsetzung

In nativ compilierten Sprachen wie C++ wird meistens Code generiert, der zur Laufzeit Informationen zur Ausnahmebehandlung protokolliert. In C++ werden „abgesicherte“ Bereiche von einem try-Block umfasst. Jeder Eintritt in und Austritt aus einem try-Block wird auf einer dafür vorgesehenen Datenstruktur (meistens ein Stack) notiert. Wird eine Ausnahme ausgelöst, kann diese Datenstruktur nach einer passenden Fehlerbehandlung durchsucht werden und der Kontrollfluss so lange aufgerufene Funktionen verlassen, bis er den ausgewählten Handler erreicht.

Bei vielen Virtuellen Maschinen wie zum Beispiel JVM oder .NET kann auf den Overhead des ständigen Protokollierens verzichtet werden, da der Bytecode bei Auftreten einer Ausnahme nach einem passenden Handler abgesucht werden kann. Somit verursachen nur noch tatsächlich ausgelöste Ausnahmen Overhead und nicht mehr jede mögliche.

Weblinks

- Exceptional practices [6] Artikelserie von Brian Goetz im Onlinejournal JavaWorld
- Programming with Exceptions in C++ [7] *O'Reilly Network*-Artikel von Kyle Loudon
- Understanding and Using Exceptions in .NET [8] Hinter den Kulissen des Win32-SEH (Structured Exception Handling)
- Errors and Exceptions [9] The Python Tutorial (englisch)
- Ausnahmebehandlung unter PHP [10]

Einzelnachweise

[1] Ann Wohlrath: *Re: Toward a more "automatic" RMI = compatible with basic RMI phi loso phy* (http://archives.java.sun.com/cgi-bin/wa?A2=ind9901&L=rmi-users&F=P&P=36083), archives.java.sun.com, abgerufen 9. Oktober 2008

[2] Joshua Bloch: *Effective Java*, Addison-Wesley, 2001, ISBN 0-201-31005-8, S. 172 ff.

[3] Bruce Eckel: *Does Java need Checked Exceptions?* (http://www.mindview.net/Etc/Discussions/CheckedExceptions), abgerufen 9. Oktober 2008

[4] *Java's checked exceptions were a mistake* (http://radio.weblogs.com/0122027/stories/2003/04/01/JavasCheckedExceptionsWereAMistake.html), Rod Waldhoff's Weblog, abgerufen 9. Oktober 2008

[5] c2:ExceptionTunneling

[6] http://www.javaworld.com/javaworld/jw-08-2001/jw-0803-exceptions.html

[7] http://oreillynet.com/pub/a/network/2003/05/05/cpluspocketref.html

[8] http://codebetter.com/blogs/karlseguin/archive/2006/04/05/142355.aspx

[9] http://docs.python.org/tutorial/errors.html

[10] http://coding-bereich.de/2010/03/15/php/ausnahmebehandlung.html

Bugtracker

Bugtracker (dt. „*[Software-]Fehler-Verfolger*“) sind Fallbearbeitungssysteme (engl. *trouble ticket system*) für die Softwareentwicklung, die als Werkzeug eingesetzt werden, um Programmfehler zu erfassen und zu dokumentieren. Mit ihnen werden – oft interaktiv und im Internet – auch Zustands- oder Feature-Berichte geschrieben.

Bugtracker können die Kommunikation zwischen Anwendern und Entwicklern von Computerprogrammen verbessern. Häufig erfordert die Eingrenzung eines Programmfehlers eine Folge von Fragen und Antworten zwischen Anwendern und Programmierern. Eine Erfassung dieser Kommunikation in einem Bugtracking-System bedeutet eine zentrale Archivierung und ermöglicht somit spätere Recherchen, z. B. bei ähnlichen Problemen (im Gegensatz zu E-Mail oder Telefon).

Neben Programmfehlern können Bugtracker auch Verbesserungsvorschläge und Wünsche der Nutzer oder allgemeine Vorgänge aufnehmen. Bei manchen Projekten spricht man dann zum Beispiel von „Metabugs“, wo ein Bug ein Element einer Aufgabenliste darstellt. Bei anderen Projekten spricht man stattdessen von „Issues“ (Angelegenheiten, Vorgänge), da sich dieser Ausdruck nicht nur auf Programmfehler beschränkt (Issue-Tracking-System).

Bekannte freie Bugtracker sind Bugzilla, Mantis, Roundup, Redmine, Trac und Flyspray. Bekannte kommerzielle Bugtracker sind Track+ und Jira.

Der Begriff **Bugtracking** bezeichnet zum einen den Vorgang der Fehlersuche an sich (durch Methoden des Debugging und Versuch und Irrtum), zum anderen den Vorgang, einen Fehler samt seiner Dokumentation über die Zeit zu verfolgen.

Motivation

Entwicklungsabteilungen, die ihre Problemerfassung in Form von einfachen Problemdokumenten oder gar simplen E-Mail-Sammlungen organisieren, haben mit einer Reihe von Problemen zu kämpfen; darunter z. B.:

- Es kann immer nur jeweils eine Person Änderungen am Dokument vornehmen.
- Es sind nur simple Suchanfragen möglich.
- Eine Organisation aller Problemdokumente ist sehr mühselig.
- Erstellung und Pflege von Problemdokumenten ist mit sehr hohem und manuellem Aufwand verbunden.
- Die Nachverfolgung von Problemen gestaltet sich sehr schwierig, wodurch bereits erfasste Probleme in Vergessenheit geraten können.
- Meist wird in proprietären Formaten gespeichert, was die Verwendung auf unterschiedlichen Plattformen erschwert oder unmöglich macht.
- Problemdokumente sind eventuell nicht von überall her zugreifbar. (vgl. Dokumentfreigaben, web-Zugriffe, etc.)

Im Gegensatz dazu stehen ausgewachsene Bugtracker mit angebundener Datenbank, die nicht nur die eben angesprochenen Probleme aufheben, sondern richtig eingesetzt folgende Fragen beantworten:

- Welche Probleme sind aufgetreten?
- Welcher Art sind die Probleme?
- Welcher Entwickler ist für das Problem zuständig?
- Welche Programmversionen sind davon betroffen?
- Was wurde unternommen, um das Problem zu beheben?
- Gibt es Möglichkeiten, das Problem zu umgehen und wenn ja, welche?
- Ab welcher Programmversion wurde das Problem behoben?
- Ist das Problem wirklich behoben, also nachgetestet?
- Ist ein einmal behobenes Problem wieder aufgetaucht?

Berechenbarkeit

In der Berechenbarkeitstheorie nennt man eine Funktion **berechenbar**, wenn es einen Algorithmus gibt, der die Funktion berechnet. Die Funktion, die ein Algorithmus **berechnet**, ist gegeben durch die Ausgabe, mit der der Algorithmus auf eine Eingabe reagiert. Der Definitionsbereich der Funktion ist die Menge der Eingaben, für die der Algorithmus überhaupt eine Ausgabe produziert. Wenn der Algorithmus nicht terminiert, dann liegt die Eingabe nicht im Definitionsbereich.

Dem Algorithmusbegriff liegt ein Berechnungsmodell zugrunde. Verschiedene Berechnungsmodelle sind entwickelt worden, es hat sich aber herausgestellt, dass die stärksten davon zum Modell der Turingmaschine gleich stark (Turing-mächtig) sind. Die Church-Turing-These behauptet daher, dass die Turingmaschinen den intuitiven Begriff der Berechenbarkeit wiedergeben.

Zu den Berechnungsmodellen, die schwächer sind als Turingmaschinen, gehören zum Beispiel die Loop-Programme. Diese können zum Beispiel die Turing-berechenbare Ackermann-Funktion nicht berechnen.

Ein dem Begriff der Berechenbarkeit eng verwandter Begriff ist der der Entscheidbarkeit. Eine Teilmenge einer Menge (zum Beispiel eine Formale Sprache) heißt entscheidbar, wenn ihre charakteristische Funktion (im Wesentlichen das zugehörige Prädikat) berechenbar ist.

Formale Definition

Man sagt, der Algorithmus P *berechnet* die Funktion $f\colon T \to \mathbb{N}$ mit $T \subseteq \mathbb{N}^k$, wenn P bei Eingabe von $(n_1, \ldots, n_k) \in T$ nach einer endlichen Zahl von Schritten den Wert $f(n_1, \ldots, n_k)$ ausgibt und bei Eingabe von $(n_1, \ldots, n_k) \in \mathbb{N}^k \setminus T$ nicht terminiert.
Eine Funktion heißt *berechenbar*, wenn es einen Algorithmus gibt, der sie berechnet.

Den Berechenbarkeitsbegriff kann man gleichwertig auf partielle Funktionen übertragen. Eine partielle Funktion $f\colon \mathbb{N}^k \rightsquigarrow \mathbb{N}$ heißt berechenbar, wenn sie eingeschränkt auf ihren Definitionsbereich $f\colon \mathrm{Def}(f) \to \mathbb{N}$ eine berechenbare Funktion ist.

Zahlenfunktionen

Definition berechenbarer Funktionen mit Registermaschinen

Eine Funktion $f\colon \mathbb{N}^k \to \mathbb{N}$ ist berechenbar genau dann, wenn es eine k-stellige Registermaschine M gibt, deren Maschinenfunktion mit f übereinstimmt, also $f = f_M$ gilt.
Z. B. ist die Funktion

$$f(x) = \mathrm{div}$$

(die für kein Argument terminiert) berechenbar, da es eine entsprechende Registermaschine gibt.

Definition mit WHILE-Programmen

Eine Funktion f (wie oben) ist berechenbar genau dann, wenn es ein WHILE-Programm P gibt mit

$$f = AC \circ \tau(P) \circ EC\,.$$

Dabei ist EC die Eingabecodierung, AC die Ausgabecodierung und $\tau(P)$ die von P über die Semantik realisierte Maschinenfunktion.

Definition durch Rekursion

Seien μ, Sub und Prk die Operationen der μ-Rekursion, der Substitution und primitiven Rekursion. Funktionen, die sich aus der Menge der primitiv-rekursiven Grundfunktionen durch wiederholtes Anwenden dieser Operatoren erzeugen lassen, heißen μ-rekursiv. Die Menge der μ-rekursiven Funktionen ist genau die Menge der berechenbaren Funktionen.

Übergang von einstelligen zu mehrstelligen Funktionen

Über die cantorsche Paarungsfunktion führt man den Begriff der Berechenbarkeit einer *k*-stelligen Funktion auf den der Berechenbarkeit von einstelligen Funktionen zurück. Insbesondere kann man damit in natürlicher Weise definieren, welche Funktionen auf den rationalen Zahlen berechenbar sind.

Wortfunktionen

Die Berechenbarkeit von Wortfunktionen lässt sich entweder mit Hilfe von Turingmaschinen zeigen. Alternativ führt man eine Standardnummerierung über die Wörter über Σ^* ein und zeigt, dass die so erzeugten Zahlenfunktionen berechenbar sind.

Uniforme Berechenbarkeit

Eine zweistellige Funktion $f(x,y)$ mit der Eigenschaft, dass für jeden festen Wert a die durch $f_a(y) = f(a,y)$ definierte einstellige Funktion f_a berechenbar ist, muss selbst nicht unbedingt berechenbar sein; für jeden Wert a gibt es zwar einen Algorithmus (also etwa ein Programm für eine Turingmaschine) T_a, der f_a berechnet, aber die Abbildung $a \to T_a$ ist im Allgemeinen nicht berechenbar.

Eine Familie $(f_a\colon a=0,1,2,...)$ von berechenbaren Funktionen heißt **uniform berechenbar**, wenn es einen Algorithmus gibt, der zu jedem a einen Algorithmus T_a liefert, welcher f_a berechnet. Man kann leicht zeigen, dass so eine Familie genau dann uniform berechenbar ist, wenn die zweistellige Funktion $(x, y) \to f_x(y)$ berechenbar ist.

Eigenschaften

- Die Komposition von berechenbaren Funktionen ist berechenbar.
- Der Definitionsbereich einer berechenbaren Funktion ist rekursiv aufzählbar.
- Der Wertebereich einer berechenbaren Funktion ist rekursiv aufzählbar.
- Die universelle Funktion nimmt ihren ersten Parameter als Gödelnummer eines Algorithmus und wendet diesen Algorithmus an auf ihren zweiten Parameter. Die universelle Funktion ist berechenbar zum Beispiel durch eine universelle Turingmaschine.

Code-Freeze

Der **Code-Freeze** bezeichnet innerhalb eines Software-Projekts den Zeitpunkt, ab dem sich der Quellcode der Software bis zur endgültigen Veröffentlichung der aktuellen Version nicht mehr ändern soll. Erlaubt sind allerdings noch Änderungen zur Behebung von im Test der Software entdeckten Fehlern von größerer Relevanz.

In der Praxis der Software-Entwicklung wird der Code-Freeze in der Regel mehrere Wochen, u. U. auch Monate vor der geplanten Veröffentlichung einer Software-Version festgelegt, damit noch ausreichend Zeit für das Testen der endgültigen Version der Software ist. Das Ziel ist, die Zahl der Fehler in der veröffentlichten Software zu minimieren.

Allgemeines

In einem Software-Projekt werden während der Implementierungsphase, das heißt, während der Erstellung des Quellcodes, regelmäßig Änderungen am bestehenden Code vorgenommen, zum Hinzufügen neuer Features und zum Beheben von aufgetretenen Fehlern. Nach dem Code-Freeze-Zeitpunkt dürfen Änderungen zum Hinzufügen neuer Features nicht mehr vorgenommen werden; die Software wurde praktisch auf ihrem aktuellen Stand *eingefroren*. Änderungen zur Behebung von im Test entdeckten Fehlern dürfen im Allgemeinen noch vorgenommen werden, solange der Nutzen, der durch das Beheben des Fehlers entsteht, in einem vernünftigen Verhältnis steht zum Risiko, das durch die erneute Änderung des Quellcodes und der damit verbundenen Möglichkeit des Einfügens neuer Fehler in die Software unvermeidbar entsteht. In der Regel werden hier spezielle Anforderungen an jede Änderung gestellt. So ist es üblich, dass ein Software-Entwickler die Entscheidung zur Änderung des Codes nach dem Code-Freeze nicht selbst treffen kann. Die Entscheidung wird stattdessen meistens durch ein mehrköpfiges Gremium gefällt.

Praktische Umsetzung

Da sich der Quellcode in Software-Projekten in der Regel in einem System zur Versionsverwaltung befindet, ist ab dem Code-Freeze ein „einchecken“ (*check in*), das heißt, ein Einbringen einer neuen Version einer Quellcode-Datei, für den einzelnen Entwickler nicht oder nur noch nach Erfüllen bestimmter Bedingungen möglich.

Entwicklungsstadium (Software)

Im Prozess der Softwareentwicklung durchläuft die zu erstellende Software verschiedene **Entwicklungsstadien**, die auch als Meilensteine betrachtet werden.

Die Stadien der Entwicklung sind: *pre-Alpha* → *Alpha* → *Beta* → *Release Candidate* → *Release*

Nach dem Erreichen des Endzustands wird der Zyklus, durch Wiederaufnahme der Arbeit an einer neuen Version der Software, wieder von vorne begonnen. Je nach Größe des Softwareprojektes und des Vorgehensmodells fallen einige Stadien weg oder werden zusammengelegt.

pre-Alpha-Version

Im Allgemeinen kann jeder beliebige Stand vor der ersten *Alpha-Version* als eine *pre-Alpha-Version* bezeichnet werden. Oft wird eine solche Version verwendet, wenn ein halbwegs fertiges Modul der Software vorgestellt werden soll.

Alpha-Version

Die erste zum Test durch Fremde (also nicht die eigentlichen Entwickler) bestimmte Version eines Computerprogramms wird oft *Alpha-Version* genannt. Obwohl der Begriff nicht exakt definiert ist, enthält in der Regel eine Alpha-Version bereits die grundlegenden Bestandteile des Softwareprodukts – es ist aber fast unerlässlich, dass in späteren Versionen der Funktionsumfang noch erweitert wird.

Insbesondere enthalten Alpha-Versionen zumeist Programmfehler in Ausmaß oder Menge, die sie für den produktiven Einsatz ungeeignet machen.

Beta-Version

Beta-Versionen sind meist durch einfache Logos gekennzeichnet.

Eine *Beta-Version* ist eine unfertige Version eines Computerprogramms.

Häufig sind Beta-Versionen die ersten Versionen eines Programms, die vom Hersteller zu Testzwecken veröffentlicht werden. Als Betatester bezeichnet man im Allgemeinen den oder die ersten unabhängigen beziehungsweise anonymen Fremdtester und Anwender.

Der Begriff ist nicht exakt definiert, als Faustregel zur Abgrenzung einer Beta-Version von anderen Versionen gilt in der Regel, dass zwar alle wesentlichen Funktionen des Programms implementiert, aber noch nicht vollständig getestet sind und das Programm daher vermutlich noch viele, auch schwerwiegende Fehler enthält, die einen produktiven Einsatz nicht empfehlenswert machen.

Beta-Versionen von Programmen sind in der Regel an der 0 als Hauptversionsnummer – diese Variante gilt natürlich nur für die Beta-Versionen vor der ersten fertigen Version (1.0) – oder dem Namenszusatz *Beta* zu erkennen.

Der Nutzen eines Betatests besteht darin, dass Fehler, die typischerweise erst in der Praxis auftreten, wie zum Beispiel Konflikte mit anderen Programmen oder Probleme mit bestimmten Hardwarekomponenten, schon vor dem Release des Programms erkannt und behoben oder zumindest dokumentiert werden können.

Beta-Versionen werden normalerweise nicht auf dem gleichen Weg wie Release Candidates oder fertige Versionen vertrieben. Folgende Möglichkeiten finden Verwendung:

- In (un)regelmäßigen Abständen werden definierte Snapshots (aktuelle Entwicklungszustände) aus dem Quellcodeverwaltungssystem generiert und en bloc entweder im Quellcode oder als vorkompiliertes Paket angeboten. Dies kann täglich (Nightly Build), wöchentlich oder zu beliebigen anderen Terminen, die die Entwickler für angemessen halten (z. B. nach Fertigstellung eines Subsystems), erfolgen. Eine solche Version kann auch ein automatisches Bugtracking-Modul enthalten (siehe Amarok), um den Betatestern die Fehlerberichte an die Entwickler zu erleichtern. Dies ist bei großen Projekten mit definierten Entwicklungszielen und einem festen Release-Zeitplan üblicherweise der Normalfall (GNOME).
- Die Betaversion wird im Quellcodeverwaltungssystem zu einer definierten Revision mit einem *Tag* (einer Markierung) versehen, aber sonst nicht gesondert behandelt. Unabhängige Anbieter können dann diesen Entwicklungsstand als Basis für ihre vorkompilierten Pakete verwenden. Dies kommt bei sich sehr schnell ändernden Projekten, die unter Umständen ganz ohne oder nur mit seltenen festen Releases arbeiten, bei denen aber trotzdem allgemeines Interesse an aktuellen Versionen besteht, zum Einsatz (Dirac, Xine).
- Es gibt keine feste Betaversion, Beta ist das aktuelle *HEAD*, also der sich ständig ändernde, tatsächliche Entwicklungsstand. Betatester müssen den derzeitigen Stand selbst aus dem Quellcodeverwaltungssystem herunterladen, konfigurieren und kompilieren, diese Tätigkeit wird normalerweise durch vom Projekt bereitgestellte Skripte automatisiert erledigt. Dies ist der häufigste Fall, kann aber auch mit einer der beiden vorherigen Methoden kombiniert werden (das ist die Regel).

Perpetual Beta

Ein Begriff, der beschreibt, dass sich in Bezug auf die ständige Entwicklung des Internets auch Websites und Software kontinuierlich weiterentwickeln und somit nie wirklich fertig sind. Somit ist ein immerwährender Entwicklungszustand eingetreten, die „Perpetual Beta“. Entstanden als Schlagwort innerhalb des Web-2.0-Konzeptes, das dem Extreme-Programming-Konzept der „Continuous Integration“ Rechnung trägt.

Release Candidate/Prerelease

Ein *Release Candidate* (RC) (auf Deutsch: *Freigabekandidat*), gelegentlich auch als *Prerelease* (auf Deutsch etwa: *Vorabveröffentlichung*) bezeichnet, ist eine abschließende Testversion einer Software. Darin sind alle Funktionen, die die endgültige Version der Software enthalten soll, schon verfügbar (sogenannter *feature complete*), alle bis dahin *bekannten* Fehler sind behoben. Aus dem Release Candidate wird vor der Veröffentlichung die endgültige Version erstellt, um einen abschließenden Produkttest oder Systemtest durchzuführen. Dabei wird die Qualität der Software überprüft und nach verbliebenen Programmfehlern gesucht.

Wird auch nur eine Kleinigkeit geändert, muss ein weiterer Release Candidate erstellt werden und die Tests werden wiederholt. Die Release Candidates werden daher auch oft nummeriert (RC1, RC2, usw.). Erfolgen keine weiteren Änderungen und hält ein Release Candidate schließlich die geforderten Qualitätsstandards ein, so wird das Suffix *RCx* entfernt und damit die Version als Release erklärt und veröffentlicht.

Versionen, die deutlich stabiler sind als Beta-Versionen, aber noch nicht den Teststand eines Release Candidate besitzen, werden in manchen Entwicklungsprojekten als *Gamma*-Version bezeichnet.

Bei Gerätetreibern für Windows (z. B. bei Nvidia) gibt es manchmal den Status *WHQL Candidate*. Hierbei handelt es sich um eine dem RC entsprechende Treiberversion, die der Hersteller zur WHQL-Prüfung eingereicht hat, die entsprechende Zertifizierung ist allerdings noch nicht erfolgt.

Release

Die fertige und veröffentlichte Version einer Software wird als *Release* bezeichnet. Damit geht eine Veränderung der Versionsbezeichnung, meist ein Hochzählen der Versionsnummer einher. Bei einer mediengebundenen Verteilung wird diese Version zur Produktion an die Presswerke ausgeliefert, wo sie auf Datenträger wie CD-ROMs oder DVDs kopiert, also als tatsächlich greifbares Produkt hergestellt wird.

Für diesen Status haben sich außerdem verschiedene Bezeichnungen etabliert:

Release to Manufacturing/Web (RTM/RTW)

: Bereit für die Veröffentlichung

Stable

: für eine stabile Version, die nicht mehr verändert wird

Final

: für die endgültige Version

General Availability (GA)

: verdeutlicht, dass die Version für den Praxiseinsatz freigegeben ist und über verschiedene Medien verteilt wurde bzw. erhältlich ist[1] [2] GA steht für eine allgemeine Verfügbarkeit. Im Gegensatz dazu stehen kundenspezifische Versionen. Es kann sein, dass eine zunächst kundenspezifische Variante später in eine allgemeine Verfügbarkeit überführt wird.

Gold

: Die Herkunft der Bezeichnung ist umstritten. Sie geht auf die Zeit vor den CDs zurück, hat also nichts mit der Farbe der CDs oder des Trägermaterials zu tun. Die wahrscheinlichste Erklärung ist die Aufnahmetechnik für Schallplatten, bei der manche Master-Formen mit Gold beschichtet wurden. Besonders im Bereich der Computerspiele wird dieser Begriff verwendet, wohl wegen der plakativen Wirkung des Edelmetalls. Der Begriff soll – ähnlich wie die *Final* und *General Availability* – symbolisieren, dass das Produkt ausgereift und (möglichst) frei von Fehlern ist.

Fehlerbehebung nach Veröffentlichung

Um Fehler in bereits veröffentlichter Software zu beheben, geben Softwarehersteller sogenannte Hotfixes, Patches und Service Packs heraus. Bei vielen modernen Anwendungen und Betriebssystemen können diese dann manuell oder automatisch direkt als Software über das Internet bezogen werden.

Literatur

- Manfred Precht, Nikolaus Meier, Dieter Tremel: *EDV-Grundwissen.* Pearson Education, 2004, ISBN 3827321298.
- Mike Gunderloy: *Coder to Developer.* Wiley_Default, 2004, ISBN 078214327X.

Einzelnachweise

[1] http://www.microsoft.com/presspass/press/2008/nov08/11-12WESSPR.mspx

[2] http://www.vmware.com/company/news/releases/vmware_view_3.html

FMEA

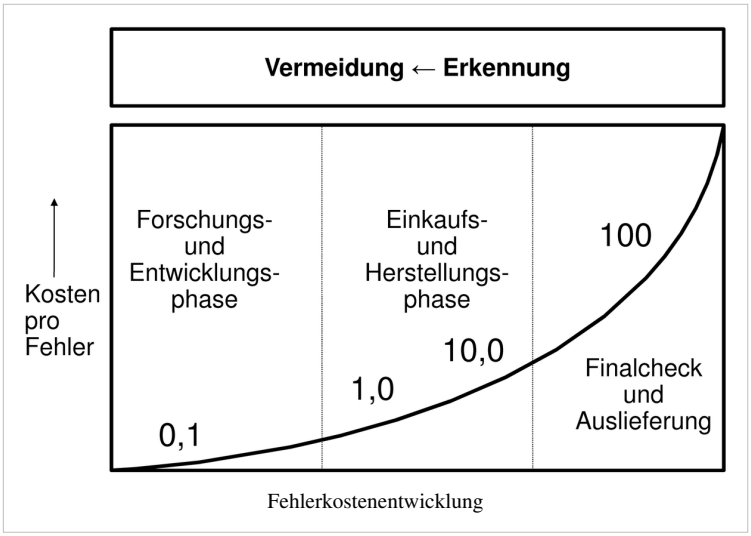
FMEA (*Failure Mode and Effects Analysis* oder auch deutsch: *Fehler-Möglichkeits- und Einflussanalyse* oder kurz *Auswirkungsanalyse*) sowie **FMECA** (*Failure Mode and Effects and Criticality Analysis*) sind analytische Methoden der Zuverlässigkeitstechnik, um potenzielle Schwachstellen zu finden. Im Rahmen des Qualitätsmanagements bzw. Sicherheitsmanagements wird die FMEA zur Fehlervermeidung und Erhöhung der technischen Zuverlässigkeit vorbeugend eingesetzt. Die FMEA wird insbesondere in der Design- bzw. Entwicklungsphase neuer Produkte oder Prozesse angewandt und von Lieferanten von Serienteilen für die Automobilhersteller aber auch anderen Industrien gefordert.

FMEA folgt dem Grundgedanken einer vorsorgenden Fehlerverhütung anstelle einer nachsorgenden Fehlererkennung und -korrektur (Fehlerbewältigung) durch frühzeitige Identifikation und Bewertung potenzieller Fehlerursachen bereits in der Entwurfsphase. Damit werden ansonsten anfallende Kontroll- und Fehlerfolgekosten in der Produktionsphase oder gar im Feld (beim Kunden) vermieden und die Kosten insgesamt gesenkt. Durch eine systematische Vorgehensweise und die dabei gewonnenen Erkenntnisse wird zudem die Wiederholung von Designmängeln bei neuen Produkten und Prozessen vermieden.

Die Methodik der FMEA soll schon in der frühen Phase der Produktentwicklung (Planung und Entwicklung) innerhalb des Produktlebenszyklus angewandt werden, da eine Kosten-/Nutzenoptimierung in der Entwicklungsphase am wirtschaftlichsten ist. Denn je später ein Fehler entdeckt wird, desto schwieriger und kostenintensiver wird seine Korrektur sein.

Arten

Die FMEA kann in mehrere Arten unterteilt werden:



Fehlerkostenentwicklung

Design-FMEA

: Die Design-FMEA dient der Entwicklung und Konstruktion dazu, die Fertigungs- und Montagegerechtigkeit eines Produkts möglichst frühzeitig zu evaluieren.

System-FMEA

: Die System-FMEA (auch S-FMEA) untersucht das Zusammenwirken von Teilsystemen in einem übergeordneten Systemverbund bzw. das Zusammenwirken mehrerer Komponenten in einem komplexen System. Sie zielt dabei auf die Identifikation potenzieller Schwachstellen, insbesondere auch an den Schnittstellen, die durch das Zusammenwirken der einzelnen Komponenten oder die Interaktion des eigenen Systemes mit der Umwelt entstehen könnten.

Konstruktions-FMEA

: Die Konstruktions-FMEA (auch K-FMEA) zielt auf die Konstruktion einzelner Produkte oder Bauteile und untersucht diese auf potenzielle Schwachstellen oder Ausfallmöglichkeiten.

Hardware-FMEA

: Eine Hardware-FMEA hat zum Ziel, Risiken aus dem Bereich Hardware & Elektronik zu analysieren, zu bewerten und mit Maßnahmen abzustellen.

Software-FMEA

Eine Software-FMEA leistet dieselbe Aufgabe für erzeugten Programmcode.

Prozess-FMEA

Die Prozess-FMEA (auch P-FMEA) stützt sich auf die Ergebnisse der Konstruktions-FMEA und befasst sich mit möglichen Schwachstellen im *Produktions*- oder Leistungsprozess.

Dabei können die *System-FMEA* und die *HW- / SW- / Konstruktions-FMEA* zur sogenannten *Produkt-FMEA* zusammengefasst werden, da das zu betrachtende System meist nicht eindeutig aufgelöst werden kann.

Die System-FMEA Produkt wird innerhalb des Entwicklungsprozesses angewandt. Ihre Aufgabe ist es einerseits, das Produkt auf Erfüllung der im Pflichtenheft festgelegten Funktionen hin zu untersuchen, andererseits aber vor allem, Fehlermöglichkeiten, die zur Nichterfüllung der Anforderungen führen, zu sammeln und zu bewerten. Dabei sind für alle risikobehafteten Teile eines Produktes geeignete Maßnahmen zur Vermeidung oder Entdeckung der potenziellen Fehler zu planen. Die System-FMEA auf Bauteilebene entspricht der bisherigen Definition der Konstruktions-FMEA. Sie dient zur Analyse aller Bauteilmerkmale, die zur Erfüllung der geforderten Bauteilfunktion notwendig sind.

Der System-FMEA Prozess wird noch innerhalb des Produktionsplanungsprozesses angewandt. Er baut logisch auf den Ergebnissen der untergeordneten FMEA auf. Ein Fehler der System-FMEA Produkt, dessen Ursache im Herstellungsprozess liegt, wird folgerichtig als Fehler in die Prozess-FMEA übernommen. Aufgabe der System-FMEA Prozess ist es, den gesamten Herstellungsprozess eines Produktes auf die Eignung zur Herstellung des Produktes hin zu untersuchen. Dabei sind für alle Fehler, die bei der Herstellung des Produktes auftreten können, geeignete Maßnahmen zu deren Vermeidung oder Entdeckung zu planen.

Anwendung

Bei der Anwendung wird zunächst ein Team aus Mitarbeitern verschiedener Unternehmensfunktionen (Interdisziplinäres Team) gebildet. Einzubeziehen sind insbesondere Konstruktion, Entwicklung, Versuch, Fertigungsplanung, Fertigungsausführung, Qualitätsmanagement etc. Der Analyseprozess selbst wird dann mit Hilfe von Formblättern (QS-9000) oder entsprechender Software in formalisierter Weise (VDA 4.2) durchgeführt.

Die FMEA enthält

- eine Eingrenzung des betrachteten Systems,
- eine Strukturierung des betrachteten Systems,
- Definitionen von Funktionen der Strukturelemente,
- eine Analyse auf potenzielle Fehlerursachen, Fehlerarten und Fehlerfolgen, die sich direkt (z.B. unter Anwendung der W-Fragen[1]) aus den Funktionen der Strukturelemente ableiten,
- eine Risikobeurteilung,
- Maßnahmen- bzw. Lösungsvorschläge zu priorisierten Risiken
- eine Verfolgung vereinbarter Vermeidungs- und Entdeckungsmaßnahmen und
- eine Restrisikobeurteilung bzw. -bewertung.

Potenzielle Fehler werden analysiert, indem der Fehlerort lokalisiert wird, die Fehlerart bestimmt, die Fehlerfolge beschrieben und anschließend die Fehlerursache ermittelt wird. Zur Ermittlung denkbarer Fehlerursachen wird häufig ein sogenanntes Ursache-Wirkungs-Diagramm erstellt. Es ist möglich, dass schon aufgrund einer erkannten Fehlerursache unmittelbar Hinweise auf mögliche Maßnahmen zur Fehlervermeidung abgeleitet werden können.

Kennzahlen zur **B**edeutung (der Fehlerfolge, engl. *Severity S*), zur **A**uftretenswahrscheinlichkeit (der Fehlerursache, engl. *Occurrence O*) und zur **E**ntdeckungswahrscheinlichkeit (des Fehlers oder seiner Ursache; ggf. auch der Folge; engl. *Detection D*) sind eine Grundlage zur Risikobeurteilung. Die Kennzahlen sind ganzzahlige Zahlenwerte zwischen 1 und 10 und werden unter Zuhilfenahme von Bewertungskatalogen vergeben.

Mit der Berechnung der Risiko-Prioritätszahl (**RPZ**) wird der Versuch gemacht eine Rangfolge der Risiken zu erstellen. Die RPZ entsteht durch Multiplikation der B-, A- und E-Bewertungszahlen ($RPZ = B * A * E$) und

kann dementsprechend Werte zwischen 1 und 1000 annehmen. Es besteht der Anspruch, dass die RPZ, mindestens im Vergleich mit anderen RPZ der gleichen FMEA, eine Aussage im Sinne *besser/schlechter* erlaubt.

Das Ziel der RPZ, die Bedeutung und den Rang eines Fehlers abzuschätzen, um hieraus Prioritäten für die zu ergreifenden Maßnahmen abzuleiten, wird immer wieder in Frage gestellt. Es gibt Versuche mit der Kenngröße ($B * A$) zusätzlich oder alternativ zu arbeiten. Bei DRBFM, der bei Toyota eingesetzten FMEA-Systematik, unterbleibt die Festlegung von Kennzahlen in Gänze. Maßnahmen werden dort ausschließlich nach dem *gesunden Menschenverstand* bzw. als Ergebnis der Teamdiskussion festgelegt.

Maßnahmen

Maßnahmen sind darauf gerichtet,

- die **A**uftretenswahrscheinlichkeit einer Fehlerursache zu reduzieren (z. B. durch den Einbau verbesserter Bauteile).
- die **E**ntdeckenswahrscheinlichkeit für eine potenzielle Fehlerursache zu erhöhen, indem bspw. zusätzlich Prüfungen vorgesehen werden.

Die Risikobewertung findet in der aktuellen FMEA nicht mehr alleine durch die bereits genannte RPZ statt, sondern vielmehr nach folgendem Ablauf:

Höchste Prioritäten haben hohe Bedeutungen (10), danach wird das Produkt aus **B**edeutung und **A**uftretenswahrscheinlichkeit betrachtet (B*A), dieses wird auch als **Kritikalität** oder **Technisches Risiko** bezeichnet (Zu berücksichtigen sind die den Bewertungszahlen hinterlegten Kataloge, A=x gibt einen Bereich und keine feste ppm-Zahl für die Auftretenswahrscheinlichkeit des Fehlers an).

Erst dann greift zur Priorisierung der restlichen Punkte die RPZ.

Bewertung

Die Bewertung erfolgt durch interdisziplinäre Teams, die jeweils Punkte von „10“ bis „1“ vergeben. Es wird immer von der höheren Bewertung zur niedrigeren Bewertung abgestuft.

- **A**uftretenswahrscheinlichkeit der Ursache (hoch = „10“ bis gering = „1“)
- **E**ntdeckenswahrscheinlichkeit der Ursache oder des Fehlers im Prozess, vor Übergabe an den Kunden (gering = „10“ bis hoch = „1“)
- **B**edeutung oder **S**chwere der Fehlerfolge wird aus der Sicht des Kunden bewertet (hoch = „10“ bis gering = „1“).

Der Kunde kann hierbei sowohl der Endkunde als auch ein (z. B. firmeninterner) Zwischenkunde sein, der die FMEA fordert. Risikoprioritätszahlen (RPZ) können zur Rangfolge für die Vereinbarung von Gegenmaßnahmen im Entwicklungsprozess genutzt werden. Die RPZ allein ist zur Beurteilung von Risikopotentialen nicht geeignet. Eine RPZ bspw. von 120 kann auf verschiedene Art und Weisen entstanden sein, wie z.B. aus B x A x E = 10 x 3 x 4 oder aber aus 5 x 8 x 3. Eine Bedeutung von B=10 und eine eher mäßige Entdeckung von E=4 ist weniger akzeptabel als eine Fehlerfolge, bewertet mit B=5, die sehr häufig auftritt (A=8)aber gut entdeckt wird (E=3).

Verschiedene Firmen haben eigene, an die Normenwerke angelehnte, Kataloge zur Bewertung von Risiken und Kriterien für die Ergreifung von Maßnahmen zur Risikoreduzierung definiert.

Nach der Erstbewertung und abgearbeiteten Maßnahmen erfolgt eine erneute Risikobewertung: Es wird durch nochmalige Ermittlung einer **R**isiko-**P**rioritäts-**Z**ahl RPZ geprüft, ob die geplanten Maßnahmen ein befriedigendes Ergebnis versprechen (Die Bedeutung der Fehlerfolge bleibt unverändert.). Entspricht das Ergebnis noch nicht den geforderten Qualitätsansprüchen des Kunden, so müssen weitere Vermeidungs- oder Entdeckungsmaßnahmen ergriffen und/oder Lösungsansätze entwickelt werden.

Die VDA Bände 4, Teil 2 und 3 empfehlen detailliert eine systematische Vorgehensweise.

Historisches

Erstmals wurde eine Beschreibung zur FMEA-Methode als United States Military Procedure veröffentlicht: MIL-P-1629 – Procedures for Performing a Failure Mode, Effects and Criticality Analysis; November 9, 1949. Der flächendeckende Einsatz der FMEA im Bereich der Automobilindustrie wurde von Ford initiiert, nachdem es in den 70er Jahren beim Modell Ford Pinto aufsehenerregende Probleme gab[2] .

In den 1970er konfrontierten die drei größten US-amerikanischen Automobilunternehmen GM, Ford und Chrysler ihre Zulieferer mit jeweils individuellen FMEA-Richtlinien. Eine davon hatte z.B. nur fünf Bewertungszahlen für B, A und E. Auf Initiative der Zulieferer wurde eine Vereinheitlichung in Form der QS-9000 FMEA Anfang der 1980er erreicht; die Big Three (Ford, GM und Chrysler) nahmen dabei die FMEA-Methodenbeschreibung von Ford als Grundlage und fügten nur wenige unverzichtbare Ergänzungen in Form von Anlagen hinzu, z.B. jeweils eigene Symbole für die Klassifikation. In den Folgejahren erfolgte die flächendeckende Einführung von FMEA in der Zulieferindustrie. 1996 wurde vom Verband der Automobilindustrie (VDA) eine verbesserte FMEA-Systematik veröffentlicht. In der seit 2002 verfügbaren dritten Auflage der QS-9000 FMEA Methodenbeschreibung wurden einige Elemente des VDA-Ansatzes übernommen. 1997 beschrieb Toyota erstmals eine änderungsfokussierte FMEA, die heute als DRBFM-Methodik bekannt ist. Im März 2007 ging die VDA FMEA 2. Auflage in Druck. Nachdem die QS-9000 Standards nicht mehr aktuell sind wurde die AIAG als Herausgeber der jetzt verfügbaren AIAG-FMEA 4th edition (Juni 2008) gewählt.

Die Einsatzfelder der FMEA haben sich im Laufe der Zeit ausgeweitet. Ursprünglich im militärischen Bereich angesiedelt hat die FMEA über eine Zwischenstufe „Luft- und Raumfahrt“ die Anerkennung im Automotivebereich gefunden. Da der FMEA ein universelles Methoden-Modell zugrunde liegt, findet sie auch in anderen Bereichen, in denen systematisch gearbeitet wird, ihre Einsatzfelder, z. B. Medizintechnik, Lebensmittelindustrie (als *HACCP-System*), Anlagenbau, Software-Entwicklung.

Normen und Standards

Für die FMEA gibt es vielfältige Normen und Standards je nach Anwendungskontext. Eine allgemeine kontextunspezifische Normierung erfolgte 1980 durch die DIN 25448 unter dem Stichwort „Ausfalleffektanalyse“. Diese Norm wurde 2006 aktualisiert durch die DIN EN 60812 unter dem Stichwort „Fehlzustandsart- und -auswirkungsanalyse“. Daneben gibt es zahlreiche kontextspezifische Standardisierungen, nachfolgend eine kleine Auswahl:

Design Review Based on Failure Mode (DRBFM)

> Von Toyota wurde die DRBFM auf Änderungen fokussierte FMEA-Methode entwickelt. Die DRBFM soll die Trennung zwischen Entwicklungs- und Qualitätsprozess aufheben und den Entwicklungs-Ingenieur direkter in den Qualitätsprozess mit einbinden.

Hazard Analysis and Critical Control Points (HACCP)

> Auf Lebensmittel ist das HACCP-Konzept (deutsch: Gefährdungsanalyse und kritische Kontrollpunkte) ausgerichtet. Ursprünglich von der NASA zusammen mit einem Lieferanten entwickelt, um die Sicherheit der Astronautennahrung zu gewährleisten, wird es heute von der US-amerikanischen National Academy of Sciences sowie von der Food and Agriculture Organization der UNO empfohlen. In der Europäischen Union ist HACCP seit 2006 für den Handel/Produktion mit/von Lebensmitteln verpflichtend.

Failure Mode, Effects, and Criticality Analysis (FMECA)

> Die FMECA ist eine erweiterte FMEA für die Analyse und Bewertung der Ausfallwahrscheinlichkeit und des zu erwarteten Schadens

Failure Mode, Effects and Diagnostic Analysis (FMEDA)

Die FMEDA bestimmt zusätzlich die Safe Failure Fraction (SFF) als Bewertungsgröße für das Functional Safety Management nach IEC 61508.

Literatur

- DIN EN 60812: *Analysetechniken für die Funktionsfähigkeit von Systemen – Verfahren für die Fehlzustandsart- und -auswirkungsanalyse (FMEA)*, November 2006
- QS-9000: *FMEA – Fehler-Möglichkeits- und -Einfluss-Analyse*, 3. Aufl. 10.2001, Carwin Ltd. (ersetzt: siehe AIAG)
- AIAG: *Potential Failure Mode and Effects Analysis (FMEA)*, 4th Edition, June 2008, AIAG
- VDA: *Sicherung der Qualität vor Serieneinsatz – System-FMEA*, 1. Aufl. 1996, ISSN 0943-9412 (ersetzt durch 2. Auflage 2006)
- VDA: *Sicherung der Qualität vor Serieneinsatz - Produkt- und Prozess-FMEA*, 2. Auflage, 2006 (Loseblattsammlung)
- DGQ: Band 13–11 *FMEA – Fehlermöglichkeits- und Einflussanalyse*, 3. Aufl. 2004, ISBN 3-410-32962-5
- Dieter H. Müller, Thorsten Tietjen: *FMEA-Praxis*, 2. Aufl. 2003, ISBN 3-446-22322-3
- Otto Eberhard, *Gefährdungsanalyse mit FMEA*, 2003, ISBN 3-8169-2061-6
- Martin Werdich, *FMEA - Einführung und Moderation*, 2011, ISBN 978-3834814333

Einzelnachweise

[1] Thorsten Tietjen,Dieter H. Müller:FMEA-Praxis, Ausgabe 2, Hanser Verlag, 2003 (http://books.google.de/books?id=lAEosVeVXIwC&pg=PA53&lpg=PA53&dq=fmea+w-fragen&source=bl&ots=zGx5ecNidt&sig=A_1p5ZWKAiE3X7QnFJxrqPeIjIg&hl=de&ei=tP6AS_PIK4mYnQPgtrHvBg&sa=X&oi=book_result&ct=result&resnum=3&ved=0CA8Q6AEwAg#v=onepage&q=fmea w-fragen&f=false)

[2] Ford Fuel System Recalls (http://www.autosafety.org/article.php?scid=96&did=479)

Weblinks

- FMEA Info Centre – Portal zum Thema FMEA (http://www.fmeainfocentre.com)
- Gefahren-und Risikoanalyse – Durchführung der FMEA-Gefahrenanalyse (http://www.riedel-doku.de/images/download/FMEA_Methode.pdf) (PDF-Datei; 1,64 MB)

Gödelscher Unvollständigkeitssatz

Der **Gödelsche Unvollständigkeitssatz** ist einer der wichtigsten Sätze der modernen Logik. Er beschäftigt sich mit der Ableitbarkeit von Aussagen in Formalen Sprachen. Der Satz zeigt die Grenzen der formalen Systeme ab einer bestimmten Mächtigkeit auf und weist nach, dass es in hinreichend mächtigen Systemen (wie der Arithmetik) Aussagen gibt – und geben muss – die man weder formal beweisen, noch widerlegen kann. Der Satz beweist damit die Unmöglichkeit des Hilbertprogramms, welches von David Hilbert unter Anderem gegründet wurde, um die Widerspruchsfreiheit der Mathematik zu beweisen.

In der Wissenschaftstheorie und in anderen Gebieten der Philosophie zählt der Satz zu den meistrezipierten der Mathematik. Das Buch Gödel Escher Bach und die Werke von John Randolph Lucas, der versuchte eine Theorie der Menschenrechte mit der Aussage zu zeigen, werden in dem Zusammenhang, zusammen mit ihren ebenso zahlreichen Kritikern, gern exemplarisch herausgehoben. Der Satz wurde zuerst in der Arbeit von Kurt Gödel formuliert und bewiesen: *Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I.* in: *Monatshefte für Mathematik und Physik* 38 (1931), S. 173 ff.

Grundbegriffe

Aussagen sind dabei Folgen von Zeichen, die ähnlich wie ein Programm einer Programmiersprache einer gewissen Syntax genügen müssen. Für solche Aussagen definiert man auf naheliegende Weise das Konzept der Gültigkeit oder Wahrheit in Strukturen (siehe Modelltheorie). Dabei kann die Wahrheit einer Aussage durchaus von der betrachteten Struktur abhängen: Eine Aussage mit der intendierten Bedeutung „Es gibt ein Element, das echt größer als 0 und echt kleiner als 1 ist“ gilt zum Beispiel in der Struktur $\mathbb{R}$ der reellen Zahlen, nicht jedoch in der Struktur $\mathbb{N}$ der natürlichen Zahlen.

Der Begriff der Gültigkeit einer Aussage in einer Struktur führt auf natürliche Weise auf einen logischen Folgerungsbegriff, den sogenannten semantischen Folgerungsbegriff: Aussage A folgt aus einer Menge von Aussagen T (einer sogenannten Theorie) genau dann, wenn in jeder Struktur, in der alle Aussagen aus T gelten, auch A gilt.

Dem semantischen Folgerungsbegriff wird in der mathematischen Logik ein zweiter, der syntaktische Folgerungsbegriff, gegenübergestellt. Man gibt einen, aus gewissen logischen Axiomen und Regeln bestehenden, Kalkül an, der spezifiziert, wie man eine Aussage mechanisch und rein syntaktisch aus anderen folgern kann; auf diese Weise erhält man einen formalisierten Beweisbegriff.

Von einem geeignet gewählten Kalkül lässt sich dann zeigen, dass er korrekt und vollständig ist; das heißt, dass der von ihm festgelegte syntaktische Folgerungsbegriff mit dem oben beschriebenen semantischen zusammenfällt (sogenannter gödelscher Vollständigkeitssatz). Dieses Resultat sichert, dass man die natürliche semantische Folgerungsrelation durch Überlegungen zum durch den Kalkül vorgegebenen syntaktischen Beweisbegriff adäquat untersuchen kann. Solche Überlegungen spielen bei der Herleitung der Unvollständigkeitssätze eine zentrale Rolle.

Im Umfeld der gödelschen Unvollständigkeitssätze sind die Begriffe der Widerspruchsfreiheit und der Vollständigkeit von Bedeutung. Eine Theorie T heißt genau dann widerspruchsfrei, wenn es keine Aussage A gibt, sodass aus T sowohl A als auch die Verneinung oder Negation von A folgt. Diese Bedingung ist, wie man leicht zeigen kann, äquivalent dazu, dass nicht jede Aussage aus T folgt. Eine widerspruchsfreie Theorie T heißt genau dann vollständig, wenn für alle Aussagen A aus T die Aussage A oder deren Negation folgt.

Gödels Satz

Satz

Der Mathematiker Kurt Gödel wies mit seinem im Jahre 1931 veröffentlichten Unvollständigkeitssatz nach, dass man in (hier stets als rekursiv aufzählbar vorausgesetzten) Systemen wie der Arithmetik nicht alle Aussagen formal beweisen oder widerlegen kann. Sein Satz besagt:

> *Jedes hinreichend mächtige formale System ist entweder widersprüchlich oder unvollständig.*

Eine einfache Formulierung des ersten Unvollständigkeitssatzes sowie des daraus unmittelbar folgenden zweiten Gödelschen Unvollständigkeitssatzes lautet:

> *In jedem formalen System der Zahlen, das zumindest eine Theorie der Arithmetik der natürlichen Zahlen ($\mathbb{N}$) enthält, gibt es einen unentscheidbaren Satz, also einen Satz, der nicht beweisbar und dessen Negierung ebenso wenig beweisbar ist.* (1. Gödelscher Unvollständigkeitssatz).

Daraus folgt unmittelbar, dass kein formales System der Zahlen, das zumindest eine Theorie der natürlichen Zahlen ($\mathbb{N}$) samt Addition und Multiplikation enthält, sich innerhalb seiner selbst als widerspruchsfrei beweisen lässt (2. Gödelscher Unvollständigkeitssatz).

Beweis des 1. Unvollständigkeitssatzes

Gödels Argumentation läuft auf eine Abzählung aller Sätze innerhalb des formalen Systems hinaus, jeder Satz erhält eine eigene Nummer. Er konstruiert dann mit Hilfe einer Diagonalisierung eine Aussage der Form: „*Der Satz mit der Nummer x ist nicht ableitbar*“ und zeigt, dass es eine Einsetzung für *x* gibt, so dass *x* die Nummer dieser Aussage ist. Insgesamt erhält er einen Satz der Form „*Ich bin nicht ableitbar*“. Es gibt nun zwei Möglichkeiten: Entweder dieser „*Satz x*“ ist wahr, dann ist er nicht ableitbar (genau das ist sein Inhalt: Ich bin nicht ableitbar!). Oder „*Satz x*“ ist falsch, dann muss der Satz ableitbar sein. Ein formales System, aus dem ein falscher Satz abgeleitet werden kann, ist aber widersprüchlich. Demnach kann dieser Satz nur wahr sein, wenn das formale System unvollständig ist, oder falsch, wenn das formale System widersprüchlich ist (siehe hierfür auch das klassische Problem des Lügner-Paradox).

Man beachte: Falls das formale System nicht widersprüchlich ist, ist der Satz mit Nummer *x* und der Bedeutung „*Satz x ist nicht ableitbar*“ damit gezeigt. Wir vertrauen auf diesen „Beweis“, obwohl es innerhalb des Systems keine Ableitung gibt, die zu diesem Satz führt.

Damit obiger Ansatz funktioniert, muss das zugrundegelegte formale System also mindestens Zählungen und eine Multiplikation mit einer Konstanten größer als 1 (für Kodierungszwecke) erlauben. Für zu einfache Systeme gilt der Unvollständigkeitssatz daher nicht. Die Möglichkeit von Addition und Multiplikation sind ganz wesentliche Eigenschaften in vielen Theorien, so dass hier dieser Satz gilt. Insbesondere muss aber eine Substitution wie im Beweis Gödels möglich sein. Es gibt sehr einfache Systeme, für die diese Bedingungen erfüllt sind.

Nun könnte man sich dadurch behelfen, dass man für alle Sätze, die weder bewiesen noch widerlegt werden können, einfach festlegt, ob sie als wahr oder falsch gelten. Das formale System würde dann durch diese zusätzlichen Axiome erweitert. Lesen wir jedoch erneut den Unvollständigkeitssatz, so sehen wir, dass auch hier die Voraussetzungen erfüllt sind und somit auch das erweiterte System unvollständig bleibt, da stets unbeweisbare Sätze übrigbleiben.

Bedeutung des Unvollständigkeitssatzes

Gödel versetzte mit seinem Unvollständigkeitssatz einem Ansatz von David Hilbert zur vollständigen Begründung und Formalisierung der Mathematik einen schweren Schlag. Dieser Ansatz ist als Hilbertprogramm bekannt geworden und wurde von ihm im Jahre 1921 veröffentlicht. Hilbert hatte vorgeschlagen, die Widerspruchsfreiheit von komplexeren Systemen durch diejenige einfacherer Systeme nachzuweisen. Hintergrund ist der, dass einem Beweis zur Widerspruchsfreiheit eines Systems, der in diesem System selbst gegeben ist, nicht getraut werden kann. Der Grund ist, dass sich aus einem Widerspruch heraus alles beweisen lässt (Ex falso quodlibet), also ließe sich aus einem Widerspruch im System auch die Widerspruchsfreiheit des Systems beweisen. Daher sollte die Widerspruchsfreiheit in einem einfacheren System bewiesen werden.

Eine streng formalisierte Prädikatenlogik erster Stufe war eines von Hilberts Konzepten. Am Ende seines Programms sollte die gesamte Mathematik auf die einfache Arithmetik zurückgeführt und auf ein axiomatisches System gestellt werden, aus dem alle mathematischen Sätze streng ableitbar sind.

Gödels Arbeit war durch Hilberts Programm motiviert. Er verwendete die von Hilbert vorgeschlagenen Methoden, um seinen Unvollständigkeitssatz zu zeigen. Gödel bewies auch den folgenden Satz

> *Ein System kann nicht zum Beweis seiner eigenen Widerspruchsfreiheit verwendet werden.*

Gödel hatte damit gewissermaßen Hilbert mit dessen Methoden gezeigt, dass der Vorschlag nicht funktioniert.

Die Folge daraus ist, dass man die Korrektheit von (gewissen) formalen Systemen *als gegeben annehmen* muss, sie lassen sich nicht beweisen.

Ein anderer Ansatz, der unüberbrückbare Lücken in Hilberts Programm nachweist, stammt von dem Mathematiker Alan Turing. Er erfand die *Turingmaschine* und formulierte deren Halteproblem.

Philosophische Interpretationen

Obwohl Gödel sich im Laufe seines Lebens wiederholt als Platoniker zu erkennen gab, wurde sein Unvollständigkeitssatz wiederholt in einem subjektivistischen Sinn interpretiert. Auch schien Gödels Teilnahme am Wiener Kreis eine Nähe des Unvollständigkeitssatzes mit dem logischen Positivismus nahezulegen, der dem Platonismus in vielerlei Hinsicht entgegengesetzt ist. Gödels zurückhaltende, konfliktscheue Art trug dazu bei, die Fehlinterpretationen am Leben zu erhalten.

Gödel selbst verstand seinen Satz jedoch insbesondere als einen Schlag gegen den von Hilbert propagierten Formalismus in der Mathematik, der in letzter Konsequenz die gesamte Mathematik zu einem rein formalen Gebilde ohne Bezug zur „realen Welt“ machen sollte. Für Gödel als Platoniker waren jedoch die mathematischen Objekte durchaus „real“. Sie waren zwar nicht durch Sinneswahrnehmungen zu bestätigen (wie es die Positivisten einforderten), doch waren sie der Erkenntnis zugänglich. Der Unvollständigkeitssatz zeigte für Gödel, dass man dieser Realität nicht mit rein formalen Mitteln beikommen konnte.

Obwohl Gödel sich in seiner Grundhaltung gegenüber dem damals bedeutsamen logischen Positivismus nicht sehr von Ludwig Wittgenstein unterschied, der eine Realität jenseits der möglichen Bedeutung von Sätzen anerkannte (und sie sogar für wichtiger hielt als das Sagbare), hielten Wittgenstein und Gödel Zeit ihres Lebens nicht viel voneinander. In Wittgensteins Werk wird der Unvollständigkeitssatz eher abschätzig behandelt. Für Wittgenstein taugte der Satz lediglich für „logische Kunststücke“. Gödel hingegen wies in späteren Interviews jeglichen Einfluss Wittgensteins auf sein eigenes Denken weit von sich.

Genauere Formulierung

Der gödelsche Satz besagt genauer, dass jedes Beweissystem für die Menge der wahren arithmetischen Formeln unvollständig ist (sofern man voraussetzt, dass die Arithmetik widerspruchsfrei ist – was, wie Gödel auch zeigt, nicht mit Mitteln der untersuchten Theorie allein bewiesen werden kann). Das heißt:

In jeder formalen Theorie, welche mindestens so mächtig wie die Theorie der natürlichen Zahlen (Peano-Arithmetik) ist, bleiben wahre (und falsche) arithmetische Formeln übrig, die nicht innerhalb der Theorie beweisbar (widerlegbar) sind. Paul Cohen bewies 1963, dass sowohl das Auswahlaxiom als auch die Kontinuumshypothese auf Grundlage der Zermelo-Fraenkel-Mengenlehre formal unentscheidbar sind. Er fand damit die ersten Beispiele mathematisch bedeutsamer unentscheidbarer Sätze, deren Existenz Gödel bewiesen hatte.

Damit eine Theorie (in der Prädikatenlogik erster Stufe, PL1) die Voraussetzungen für die Unvollständigkeit erfüllt, muss gelten:

- Zu jeder durch einen Ausdruck G(x) beschriebenen Menge ist das Komplement beschreibbar.
- Zu jeder durch einen Ausdruck G(x) beschriebenen Menge M ist die Menge M*={x|d(x)∈M} beschreibbar; Dabei ist d(x) die Diagonalisierung von x.
- Die Menge der beweisbaren Ausdrücke der Theorie ist durch einen Ausdruck der Form G(x) beschreibbar.

Nach dem Satz von Löwenheim-Skolem findet man zu jeder Theorie in PL1 ein Modell mit der Mächtigkeit der Signatur. Für *normale* Theorien existiert also ein abzählbares Modell, beispielsweise die natürlichen Zahlen (das heißt, es lässt sich für jede Theorie in PL1 auch ein Modell finden, in dem die Objekte natürliche Zahlen sind). Die Idee von Gödel war, Formeln der Theorie selbst zum Objekt derselben zu machen. Dazu wurden die Formeln gödelisiert, das heißt eine (injektive) Abbildung von Formeln auf natürliche Zahlen gebildet. Das kann man zum Beispiel dadurch erreichen, dass man jedem Symbol der Signatur eine Zahl zuordnet und einer Symbolkette entsprechend eine Kette von Zahlen. Ordnet man der 0 die 1 und = die 2 zu, so ist die Gödelnummer der Formel (in dem Spezialfall) 0=0 die 121. Die Zahlenkette wird dann durch Exponentieren in eine einzelne Zahl übersetzt. Es lassen sich auch die syntaktisch wohlgeformten, und schließlich die beweisbaren Formeln durch arithmetische Ausdrücke (Addition, Multiplikation, Exponentiation) beschreiben.

Die Diagonalisierung in Gödels Beweis ist nun eine Anwendung eines Ausdrucks P(x) auf die eigene Gödelnummer. Ist die Gödelnummer des Ausdrucks (und damit der Zeichenreihe) P(x) zum Beispiel 12345, so ist die Diagonalisierung *d* der Zahl 12345 die Gödelnummer von *P*(12345) (selbstverständlich hat eine Zahl, hier 12345, auch eine Gödelnummer, die entsteht, indem man alle vorkommenden Ziffern gödelisiert).

Besagt der Ausdruck *Bew(x)* also, dass *x* ableitbar ist, so sagt *B(x)*=¬Bew(d(x)), dass die Formel mit der Gödelnummer d(x) nicht ableitbar ist. Ist nun zum Beispiel 12345 die Gödelnummer von *B(x)*, so ist B(12345) eine nicht ableitbare Aussage. Diese Aussage besagt dann nämlich: Die Formel mit der Gödelnummer *d(12345)* ist nicht ableitbar. 12345 ist aber die Gödelnummer von B(x), also *d(12345)* die Gödelnummer von B(12345). Also sagt B(12345): Ich bin nicht ableitbar. Wenn PA korrekt ist, so ist dieser Satz wahr (in PA), aber nicht ableitbar.

Gödels ursprünglicher Beweis ging noch weiter. Er wollte Rückgriffe auf die Semantik, insbesondere die Korrektheit, vermeiden. Deswegen bewies er seinen Unvollständigkeitssatz unter der Voraussetzung der ω-Konsistenz: Eine Theorie ist ω-inkonsistent, wenn ein Ausdruck mit einer einzigen freien Variable x existiert, für den $\exists x P(x)$ ableitbar ist, zugleich aber für alle $n < \omega$ $\neg P(n)$ ableitbar ist.

Rosser erweiterte das gödelsche Resultat, indem er einen Unvollständigkeitsbeweis lieferte, für den nicht die Menge der Ausdrücke, deren Diagonalisierung beweisbar ist, beschrieben wird, sondern eine zu dieser Menge disjunkte Obermenge der Ausdrücke, deren Diagonalisierung widerlegbar ist. Dadurch ist auch der Bezug auf die ω-Konsistenz überflüssig.

Gödels zweiter Unvollständigkeitssatz ist eine leicht zu sehende Konsequenz aus dem ersten. Da Gödel beweisbare Aussagen innerhalb der Prädikatenlogik formalisierte (beispielsweise durch das Prädikat $Bew(x)$), lässt sich auch folgende Aussage bilden: $\neg Bew(\bot)$, wobei $\bot$ die Gödelnummer von einer beliebigen Kontradiktion, zum

Beispiel $\neg 0 = 0$, ist. Die Aussage $W = \neg Bew(\bot)$ *behauptet* die Nichtbeweisbarkeit einer Kontradiktion, und damit die Widerspruchsfreiheit der gesamten Theorie (der Peano-Arithmetik). W ist in PA nicht beweisbar. Um die Nicht-Beweisbarkeit zu zeigen, wird eine Fixpunktkonstruktion verwendet. Sei $g(x)$ die Gödelisierungsfunktion, A eine Aussage, $B(x)$ ein Prädikat. A heißt Fixpunkt für $B(x)$, wenn $A \iff B(g(A))$ beweisbar ist. Über einfache aussagenlogisch sich beweisen, dass $W \rightarrow A$ beweisbar ist, wenn A Fixpunkt von $\neg Bew(x)$ ist. Außerdem kann leicht gezeigt werden, Fixpunkt von $\neg Bew(x)$ ist, A nicht beweisbar ist, falls PA konsistent ist. Daraus folgt dann, dass W nicht beweisbar ist.

Durch diese erstaunlichen Sätze ist der Mathematik eine prinzipielle Grenze gesetzt: Nicht jeder wahre mathematische Satz kann aus den wie auch immer gewählten Axiomen eines mathematischen Teilgebietes (zum Beispiel Arithmetik, Geometrie, Algebra etcetera) formal abgeleitet werden.

Viel Verwirrung entsteht aus dem Zusammenhang der gödelschen Unvollständigkeitssätze mit dem Gödelschen Vollständigkeitssatz. Hier ist zu beachten, dass der Begriff der Vollständigkeit in zwei verschiedenen Bedeutungen gebraucht wird. Der Vollständigkeitssatz beweist die *semantische* Vollständigkeit der Prädikatenlogik der ersten Stufe, behandelt also eine Eigenschaft von formalen Systemen. Der Unvollständigkeitssatz hingegen beweist, dass gewisse Mengen von Ausdrücken nicht vollständig im klassischen Sinne sind.

Ein häufiger Fehlschluss aus dem Unvollständigkeitssatz ist, dass gewisse mathematische Objekte, wie z. B. die natürlichen Zahlen nicht mit einer vollständigen Menge von Ausdrücken formalisierbar seien. Man erkennt aber sofort, dass die Menge aller Ausdrücke (in einer geeigneten formalen Sprache), die in der Menge der natürlichen Zahlen gelten, sowohl vollständig ist als auch die natürlichen Zahlen formalisiert. Eine Konsequenz aus dem Unvollständigkeitssatz ist aber, dass eine solche Menge nicht entscheidbar sein kann, womit sie eine in vielen Bereichen der Logik notwendige Eigenschaft nicht erfüllt.

Ein weiterer Fehlschluss ist, dass die meisten in der Mathematik benutzten Theorien unvollständig seien. Es gibt aber einige wichtige vollständige Theorien, wie z. B. die Theorie der algebraisch abgeschlossenen Körper von Charakteristik p, die Theorie der dichten linearen Ordnungen ohne größtes und kleinstes Element oder Tarskis Axiomatisierung der euklidischen Geometrie.

Sonstiges

Gödel nannte seinen Aufsatz *Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I*, weil er plante, einen zweiten Aufsatz zu verfassen, in dem er den Beweis genauer erläutern wollte. Der erste Aufsatz fand jedoch bereits so große Anerkennung, dass der Bedarf für einen zweiten entfiel, der daher auch nie geschrieben wurde.

Konkret bezog sich Gödels Aufsatz auf die Principia Mathematica, ein großes formales System, das Bertrand Russell und Alfred North Whitehead zwischen 1910 und 1913 veröffentlichten. Gödel zeigte jedoch auf, dass jedes System mit der gleichen Mächtigkeit wie die Principia Mathematica ebenso anfällig ist.

Weiterhin konnte Gerhard Gentzen zeigen, dass eine konstruktive Mathematik und Logik durchaus widerspruchsfrei ist. Hier zeigt sich ein Grundlagenstreit der Mathematik. Der Philosoph Paul Lorenzen hat eine widerspruchsfreie Logik und Mathematik erarbeitet (Methodischer Konstruktivismus), und sein Buch Metamathematik (1962) eigens geschrieben, um zu zeigen, dass der gödelsche Unvollständigkeitssatz keinen Einwand gegen einen widerspruchsfreien Aufbau der Mathematik darstellt.

Literatur

- Kurt Gödel: *Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I*. Monatshefte für Mathematik und Physik 38, 1931, S. 173–198, doi:10.1007/BF01700692 [1]. Zentralblatt MATH [2].
- Kurt Gödel: *Diskussion zur Grundlegung der Mathematik: Erkenntnis 2*. Monatshefte für Math. und Physik, 1931–32, S. 147–148.
- Ernest Nagel u. James R. Newman: *Der Gödelsche Beweis*. 8. Auflage 2007. ISBN 978-3-486-45218-1.
- Douglas R. Hofstadter: *Gödel, Escher, Bach, ein Endloses Geflochtenes Band*. München 1991 ISBN 3-423-30017-5 (Eine interessante und relativ leicht verständliche Erklärung von Gödels Satz und seinen Implikationen).
- Paul Lorenzen: *Metamathematik*. Mannheim 1962. ISBN 3-86025-870-2.
- Wolfgang Franz: *Über mathematische Aussagen, die samt ihrer Negation nachweislich unbeweisbar sind. Der Unvollständigkeitssatz von Gödel.* Franz Steiner Verlag, Wiesbaden, 1977, 27 S., ISBN 3-515-02612-6. (verständlicher Vortrag mit wissenschaftsgeschichtlichen Bezügen).
- Torkel Franzen: *Gödel's Theorem. An incomplete guide to its use and abuse*. Wellesley, Massachusetts 2005. ISBN: 1-56881-238-8 (Eine leicht verständliche Erläuterung von Gödels Unvollständigkeitssätzen, ihren Implikationen und ihren Fehlinterpretationen).
- Max Woitschach: *Gödel, Götzen und Computer: eine Kritik der unreinen Vernunft*. Stuttgart 1986. ISBN 3-87959-294-2.
- Paul Lorenzen: *Lehrbuch der konstruktiven Wissenschaftstheorie*. Stuttgart 2000. ISBN 3-476-01784-2.
- Wolfgang Stegmüller: *Unvollständigkeit und Unentscheidbarkeit. Die metamathematischen Resultate von Gödel, Church, Kleene, Rosser und ihre erkenntnistheoretische Bedeutung. 3. Auflage, Springer-Verlag, Wien, New York, 1973. ISBN 3-211-81208-3. 116 S.*
- Sybille Krämer: *Symbolische Maschinen: die Idee der Formalisierung in geschichtlichem Abriß*. Darmstadt 1988. ISBN 3-534-03207-1.
- Ludwig Fischer: *Die Grundlagen der Philosophie und der Mathematik*. Leipzig 1933.
- Peter Smith: *An Introduction to Gödel's Theorems*. Cambridge Introductions to Philosophy. Cambridge 2007. ISBN 978-0-521-85784-0.
- S. G. Shanker (Hg.): *Gödel's Theorem in focus*. London/New York 1988. ISBN 0-415-04575-4.
- Raymond Smullyan: *Dame oder Tiger – Logische Denkspiele und eine mathematische Novelle über Gödels große Entdeckung*. Fischer-Verlag Frankfurt am Main 1983. Das amerikanische Original erschien bei Alfred A. Knopf, New York 1982.
- Raymond Smullyan: *To Mock a Mockingbird*. 1985.
- Raymond Smullyan: *Forever undecided: a puzzle guide to Gödel*. 1987.
- Raymond Smullyan: *Gödel's Incompleteness Theorems*. Oxford Logic Guides. Oxford University Press, 1992.
- Max Urchs, Klassische Logik: eine Einführung, Berlin (1993). – ISBN 3-05-002228-0. (dort im Kapitel: Theorien erster Ordnung, S. 137–149).
- Palle Yourgrau: *Gödel, Einstein und die Folgen. Vermächtnis einer ungewöhnlichen Freundschaft.* Beck, München 2005. ISBN 3-406-52914-3. (Original: *A World Without Time: The Forgotten Legacy of Gödel and Einstein.* Basic Books, Cambridge, Massuchusetts 2005).
- Norbert Domeisen: *Logik der Antinomien*. Bern 1990. ISBN 3-261-04214-1, Zentralblatt MATH [3].

Weblinks

- Die Grenzen der Berechenbarkeit [4]
- Einstieg in die Zusammenhänge des Unvollständigkeitssatzes [5]
- Christopher v. Bülow: Der erste Gödelsche Unvollständigkeitssatz. Eine Darstellung für Logiker in spe. [6] März 1992 (PDF, 355 kB)
- Eine englische Übersetzung von Gödels Aufsatz *Über formal unentscheidbare Aussagen.* [7] (PDF, 328 KB)

Referenzen

[1] http://dx.doi.org/10.1007%2FBF01700692
[2] http://www.zentralblatt-math.org/zbmath/search/?q=an:57.0054.02
[3] http://www.zentralblatt-math.org/zbmath/search/?q=an%3A0724.03003
[4] http://www.joergresag.privat.t-online.de/mybk3htm/start3.htm
[5] http://www.schulfach-ethik.de/ethik/Personen/goedelkurt.htm
[6] http://www.uni-konstanz.de/FuF/Philo/Philosophie/Spohn/vonBuelow/goedel.pdf
[7] http://www.research.ibm.com/people/h/hirzel/papers/canon00-goedel.pdf

Hoare-Kalkül

Der **Hoare-Kalkül** (auch **Hoare-Logik**) ist ein Formales System, entwickelt von dem britischen Informatiker C. A. R. Hoare und später verfeinert von Hoare und anderen Wissenschaftlern. Er wurde 1969 in einem Artikel mit dem Titel *An axiomatic basis for computer programming* veröffentlicht. Der Zweck des Systems ist es, eine Menge von logischen Regeln zu liefern, die es erlauben, Aussagen über die Korrektheit von imperativen Computer-Programmen zu treffen und sich dabei der mathematischen Logik zu bedienen. Hoare knüpft an frühere Beiträge von Robert Floyd an, der ein ähnliches System für Flussdiagramme veröffentlichte. Alternativ kann auch der wp-Kalkül benutzt werden, bei dem im Gegensatz zum Hoare-Kalkül eine Rückwärtsanalyse stattfindet.

Das zentrale Element des Hoare-Kalküls ist das Hoare-Tripel, das beschreibt, wie ein Programmteil den Zustand einer Berechnung verändert:

$$\{P\}\ S\ \{Q\}.$$

Dabei nennt man P und Q Zusicherungen (englisch *assertions*). S ist ein Programmsegment. P heißt die Vorbedingung (englisch *precondition*), Q heißt die Nachbedingung (englisch *postcondition*). Zusicherungen sind Formeln der Prädikatenlogik.

Ein Tripel kann auf folgende Weise verstanden werden: Falls P für den Programmzustand vor der Ausführung von S gilt, dann gilt Q danach. Falls S nicht terminiert, dann gibt es kein danach, also kann in diesem Fall Q jede beliebige Aussage sein. Tatsächlich kann man für Q die Aussage *false* wählen, um auszudrücken, dass S nicht terminiert. Man spricht hier von *partieller Korrektheit*. Falls S immer terminiert und danach Q wahr ist, spricht man von *totaler Korrektheit*. Die Terminierung muss unabhängig bewiesen werden.

Partielle Korrektheit

Der Hoare-Kalkül besteht aus Axiomen und Ableitungsregeln für alle Konstrukte einer einfachen imperativen Programmiersprache:

Zuweisungsaxiom

Das Zuweisungsaxiom besagt, dass nach einer Zuweisung jede Aussage für die Variable gilt, welche vorher für die rechte Seite der Zuweisung galt:

$$\{P[x/E]\}\ E := x\ \{P\}.$$

$P[x/E]$ ist die Aussage, die dadurch entsteht, dass man in P jedes freie Vorkommen von E durch x ersetzt.
Genau genommen ist das Zuweisungsaxiom kein einzelnes Axiom, sondern ein Schema für eine unendliche Menge von Axiomen, denn x, E und P können jede mögliche Form annehmen, und $P[x/E]$ kann daraus konstruiert werden.
Ein Beispiel für ein durch das Zuweisungsaxiom beschriebenes Tripel ist:

$$\{x + 1 = 43\}\ y := x + 1\ \{y = 43\}.$$

Kompositions- oder Sequenzregel

$$\frac{\{P\}\ S\ \{R\}\ ,\ \{R\}\ T\ \{Q\}}{\{P\}\ S;T\ \{Q\}}$$

Diese Regel kann auf folgende Weise angewendet werden: Wenn die über dem Strich stehenden Aussagen bewiesen worden sind, kann die unter dem Strich stehende Aussage auch als bewiesen angesehen werden.

Betrachtet man zum Beispiel die folgenden beiden Aussagen, die aus dem Zuweisungsaxiom folgen

$$\{x + 1 = 43\}\ y := x + 1\ \{y = 43\}$$

und

$$\{y = 43\}\ z := y\ \{z = 43\}$$

kann man die folgende Aussage daraus folgern:

$$\{x + 1 = 43\}\ y := x + 1; z := y\ \{z = 43\}.$$

Auswahlregel (if-then-else-Regel)

$$\frac{\{P \wedge B\}\ S\ \{Q\}\ ,\ \{P \wedge \neg B\}\ T\ \{Q\}}{\{P\}\ \text{if}\ B\ \text{then}\ S\ \text{else}\ T\ \{Q\}}$$

Die Regel beweist also sowohl den if-Zweig, als auch den else-Zweig.

Iterationsregel (while-Regel)

$$\frac{\{I \wedge B\}\ S\ \{I\}}{\{I\}\ \text{while}\ B\ \text{do}\ S\ \text{done}\ \{I \wedge \neg B\}}$$

Hierbei wird I als die Schleifeninvariante bezeichnet.

Konsequenzregel

$$\frac{P \Rightarrow P',\ \{P'\}\ S\ \{Q'\}\ ,\ Q' \Rightarrow Q}{\{P\}\ S\ \{Q\}}$$

Die Konsequenzregel erlaubt es, die Vorbedingung zu verstärken und die Nachbedingung abzuschwächen und so die Anwendung anderer Beweisregeln zu ermöglichen. Insbesondere kann man auch die Vor- oder Nachbedingung durch eine äquivalente logische Formel ersetzen. Beispiel:

$\{true\}\ x = 0\ \{x >= 0\}$ ist partiell korrekt, denn

$$\frac{true \Rightarrow 0 = 0, \{0 = 0\}\ x = 0\ \{x = 0\},\ x = 0 \Rightarrow x >= 0}{\{true\}\ x = 0\ \{x >= 0\}}.$$

Totale Korrektheit

Wie oben erläutert eignet sich das beschriebene Kalkül nur für den Beweis der partiellen Korrektheit. Zum Beweis der totalen Korrektheit kann es durch Erweiterung der while-Regel verwendet werden:

Iterationsregel für totale Korrektheit

$$\frac{\{I \wedge B\ \wedge\ t = z\}\ S\ \{I\ \wedge\ t < z\},\ I \Rightarrow t \geq 0}{\{I\}\ \text{while}\ B\ \text{do}\ S\ \text{done}\ \{I \wedge \neg B\}}$$

Hierbei ist t ein Term, I die Schleifeninvariante – also das, was in jedem Schleifendurchlauf gilt – und z eine Variable, die in B, t, S und I nicht (frei) vorkommt. Sie dient dazu, den Wert des Terms vor der Schleife mit dem nach der Schleife zu vergleichen. Die Bedingung $I \Rightarrow t \geq 0$stellt sicher, dass t nicht negativ wird. Die Idee hinter der Regel ist, dass, wenn t mit jedem Schleifendurchlauf abnimmt, aber nie kleiner als Null wird, die Schleife irgendwann enden muss. t muss dabei aus einer fundierten Menge sein.

Literatur

- C. A. R. Hoare: *An axiomatic basis for computer programming* [1] (pdf). In: *Communications of the ACM.* 12(10): 576–585, Oktober 1969.
- Robert D. Tennent: *Specifying Software.* (a recent textbook that includes an introduction to Hoare logic) ISBN 0-521-00401-2 [2]

Weblinks

- Das Project Bali [3] hat Regeln nach Art des Hoare-Kalküls für ein Subset von Java aufgestellt, zur Benutzung mit dem Theorembeweiser Isabelle
- Hoare Tutorial [4] Ein Tutorial, das den Umgang mit dem Hoare-Kalkül zur Programmverifikation erklärt (PDF-Datei; 493 kB)
- j-Algo-Modul Hoare Kalkül [5] Ein Visualisierung des Hoare-Kalküls im Rahmen des Algorithmenvisualisierungsprogramms j-Algo

Referenzen

[1] http://www.spatial.maine.edu/~worboys/processes/hoare%20axiomatic.pdf
[2] http://www.cs.queensu.ca/home/specsoft/
[3] http://isabelle.in.tum.de/Bali/
[4] http://wwwswt.informatik.uni-rostock.de/deutsch/Mitarbeiter/michael/lehre/pt_2001/Hoare_akt_008.pdf
[5] http://j-algo.binaervarianz.de/

Imperative Programmierung

Imperative Programmierung ist ein Programmierparadigma. Ein imperatives Programm beschreibt eine Berechnung durch eine Folge von Anweisungen, die den Status des Programms verändern. Im Gegensatz dazu wird unter dem deklarativen Programmierparadigma beschrieben, *was* berechnet werden soll, aber nicht *wie*.

Details

Der Programmstatus besteht aus allen Speicherzellen, auf die das Programm Zugriff hat, inklusive der deklarierten Variablen, den Prozessor-Registern und dem Programm-Zähler. Der Programm-Zähler gibt die Position in der Anweisungsfolge an, mit der die Programmausführung nach der aktuellen Anweisung fortgesetzt werden soll.

Beispiele für imperative Anweisungen sind eine Schleifenanweisung (bedingt wird der Programm-Zähler verändert) oder die Speicherung eines konstanten Wertes in einer Variablen.

Fortran, Pascal und C sind Beispiele für konkrete imperative Programmiersprachen.

Das abstrakte Ausführungsmodell, das dem imperativen Paradigma zugrunde liegt, ist nah verwandt mit der Ausführung von Maschinen-Code auf einem konkreten Computer, der die vorherrschende Von-Neumann-Architektur implementiert. Es existieren beispielsweise bedingte und unbedingte Sprunganweisungen, mit denen Schleifen realisiert werden können. Der Status des Rechners setzt sich aus dem Arbeitsspeicherinhalt und den Registerinhalten zusammen.

Beispiel

Ausgabe der Quadratzahlen von ungeraden Zahlen von 3 bis 11.

Imperativ

Imperative Version in C:

```
int i;
for (i=3; i<12; i=i+2) {
  int q = i * i;
  printf("%d\n", q);
}
```

Deklarativ

Deklarative Version in Haskell

```
show $ [ i*i | i <- [3..11], odd i ]
```

Geschichte

Die 1957 entwickelte Programmiersprache Fortran implementiert wie die zuvor verwendeten Assemblersprachen das imperative Paradigma. LISP, 1958 erschienen, ist ein Beispiel für eine deklarative Sprache, bei der der Computer selbst einen Handlungsablauf zur Berechnung konstruieren muss.

Abgrenzung

Der Begriff der prozeduralen Programmierung wird oft synonym gebraucht, setzt aber die Verwendung von Prozeduren voraus, was nicht für jede imperative Programmiersprache gelten muss. Beispielsweise kennen ältere BASIC-Varianten keine Prozeduren.

Das Prinzip der Datenkapselung (*Information Hiding*) wird in imperativen Sprachen oft dadurch umgesetzt, dass Prozeduren, die eine logische Einheit bilden, in Modulen oder Paketen zusammengefasst werden.

Einige imperative Programmiersprachen wie z.B. C++ und Java bieten neben prozeduralen Merkmalen auch zusätzliche Sprachmittel, um explizit die objektorientierte Programmierung zu unterstützen.

Literatur

- Terrence W. Pratt and Marvin V. Zelkowitz: *Programming Languages: Design and Implementation.* 4th Auflage. Prentice Hall, 2000, ISBN 978-0130276780.
- Robert W. Sebesta: *Concepts of Programming Languages.* 9th Auflage. Addison Wesley, 2009, ISBN 978-0136073475.

Keyword-Driven Testing

Keyword-Driven Testing (*auch* Table-Driven Testing, Action-Word Testing) ist eine Technik des automatischen Software-Testens.

Charakteristik

Obwohl man Keyword-Driven Testing auch für manuelles Testen verwenden kann, ist es eher für das automatische Testen geeignet[1] . Die hohe Abstraktionsebene von solchen schlüsselwort-gesteuerten Tests verbessert die Wiederverwendbarkeit und die Wartbarkeit automatischer Tests.

Methode

Im Keyword-Driven Testing findet die Testerstellung meist in zwei Etappen statt.

Planung

Zunächst werden die zu testenden Aktionen oder Operationen in der Anwendung (oder in den Anforderungen für die Anwendung) analysiert. Wiederkehrende Aktionen und Abläufe werden in Keywords (Schlüsselwörtern) gekapselt.

Beispiele für Keywords

- Ein einfaches Keyword (eine Aktion auf einem Objekt), z. B. Eingabe von einem Benutzernamen in ein Textfeld.

Objekt	Aktion	Daten
Textfeld (Benutzername)	Text eingeben	<Benutzername>

- Ein komplexeres Keyword (aus anderen Keywords zusammengestellt) z.B. Einloggen.

Objekt	Aktion	Daten
Textfeld (Domäne)	Text eingeben	<Domaene>
Textfeld (Benutzername)	Text eingeben	<Benutzername>
Textfeld (Passwort)	Text eingeben	<Passwort>
Button (einloggen)	Klicken	Einmal Klicken mit der linken Maustaste

Implementierung

Die Implementierung unterscheidet sich je nachdem, welches Tool oder Framework eingesetzt wird. Häufig müssen Testentwickler ein Framework implementieren, um Keywords wie „Prüfen“ oder „Eingeben“ bereitzustellen[1] . Ein bekanntes Open-Source-Framework ist das Framework for Integrated Test. Ein Tester ohne Programmierkenntnisse kann dann Testfälle gemäß der Planung anhand dieser fertig kodierten Keywords erstellen. Der daraus entstehende Test wird von einem Roboter ausgeführt. Der Roboter liest die Keywords ein und führt die entsprechenden Codezeilen aus.

Andere Ansätze trennen das Testdesign und die Keywordimplementierung nicht. Hier gibt es nur einen Schritt zur Implementierung – das Testdesign ist gleichzeitig die Testautomatisierung. Keywords wie „Prüfen“ oder „Eingeben“ werden anhand fertiger Bausteine erstellt, in denen der notwendige Code für die Keywords bereits vorhanden ist. Dadurch entfällt der Bedarf an zusätzlichen technischen Fachkräften zur Programmierung im Testprozess. Diesen Ansatz verwenden Werkzeuge wie GUIdancer und Worksoft Certify. Das Open-Source-Werkzeug Selenium stellt fertige Keywords für das Testen von Webanwendungen zur Verfügung, die in HTML-Tabellen zu Testfällen

zusammengestellt werden können. Darauf setzt u.a. das freie Firefox-Plugin Molybdenum [2] auf, welches die Zusammensetzung einzelner "Commands" zu parametrierbaren Testbausteinen ermöglicht.

Vorteile

Beim Keyword-Driven Testing erscheint der Aufwand zu Beginn höher als bei aufgenommenen Skripten. Jedoch macht sich sorgfältige Planung bei der folgenden Testerstellung und -wartung bezahlt. So fördert Keyword-Driven Testing eine stabile und übersichtliche Test-Struktur. Umso abstrakter die Keywords, desto einfacher sind sie wiederzuverwenden. Dadurch wird der Aufwand für Wartungsarbeiten gesenkt. Die modulare Struktur eines Keyword-Driven Tests erlaubt außerdem die bequeme Erstellung neuer Tests anhand schon vorhandener Keywords.

Ein weiterer Vorteil liegt darin, dass keine technischen Kenntnisse vorausgesetzt werden. Im ersten Ansatz müssen ausschließlich die Keyword-Entwickler programmieren können. Im zweiten Ansatz entfällt sogar diese Notwendigkeit. Somit können Tests ganz ohne Programmierkenntnisse automatisiert werden.

Einzelnachweise

[1] Danny R. Faught: *Keyword-Driven Testing, Sticky Minds.* (http://www.stickyminds.com/sitewide.asp?Function=edetail&ObjectType=COL&ObjectId=8186)
[2] http://www.molyb.org

Weblinks

- LogiGear (http://www.logigear.com/newsletter/key_success_factors_for_keyword_driven_testing.asp)
- Test automation Frameworks (http://safsdev.sourceforge.net/DataDrivenTestAutomationFrameworks.htm)

Kontrollflussorientierte Testverfahren

Die **kontrollflussorientierten Testverfahren**, auch **Überdeckungstests** genannt, gehören zu der Gruppe der strukturorientierten Testmethoden.

Die kontrollflussorientierten Testverfahren orientieren sich am Kontrollflussgraphen des Programms. Es handelt sich bei diesen Tests um White-Box-Testverfahren, d.h. die Struktur des Programms muss bekannt sein.

Die einzelnen Tests werden mit *Cx* bezeichnet was für "Coverage" steht, was so viel heißt wie die Abdeckung oder die Gesamtheit der ausgewerteten Informationen.

Nomenklaturen

Es gibt mehrere zueinander sehr ähnlich aussehende, aber in der Bedeutung unterschiedliche Bezeichnungsarten:

- Beginnt die Bezeichnung mit kleinen c → siehe Harry M. Sneed, Mario Winter "Testen objektorientierter Software", 3-446-21820-3, Hanser Verlag
- Beginnt die Bezeichnung mit einem großen C und steht die nun folgende Ziffer auf der Grundlinie wie das C, z. B. C4 → Ernest Wallmüller, „Software - Qualitätssicherung in der Praxis“, Hanser Verlag.
- Beginnt die Bezeichnung mit einem großen C und ist die folgende Ziffer ein Subskript, z.B. C_1, liegt also unterhalb der Grundlinie → siehe Standard DO-178B.

Die unterschiedliche Testarten

Zeilenüberdeckungstests

Noch vor der Hierarchie der C_x Testverfahren steht die von vielen Werkzeugen in der Softwareentwicklung bereitgestellte Zeilenüberdeckungskennzahl. Sie ist etwas unschärfer als C_0, orientiert sich auch nicht direkt am Kontrollflussgraph, kann aber oft direkt aus den Informationen gewonnen werden, die Debugger ohnehin liefern.

Bei der Zeilenüberdeckung werden nicht die Anweisungen betrachtet, sondern nur die ausführbaren Quellcodezeilen.

```
if(false){print "abgedeckt?";}
```

würde zu einer 100% Zeilenabdeckungskennzahl führen. Während das syntaktisch identische, aber anders formatierte Programm

```
if(false){
  print "abgedeckt?";
}
```

nur zu einer 50% Zeilenüberdeckungskennzahl führt.

In der Regel sind die Unterschiede in der praktischen Anwendung allerdings nicht relevant, da durch andere Maßnahmen in der Softwareentwicklung wie Style Guides eine weitgehende Homogenisierung der Quellcodeformatierung vorliegt.

Vorteile

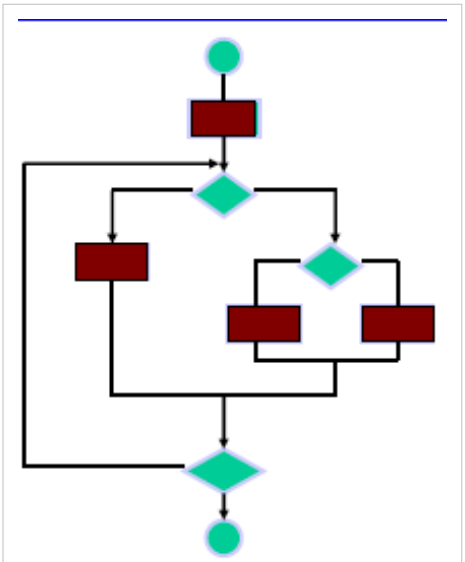
- siehe C_0
- einfachere technische Implementierung über die von Debuggern gelieferten Zeilennummern

Nachteile

- keine Standardmetrik wie C_0
- liefert bei syntaktisch identischen Programmen je nach Formatierung unterschiedliche Werte

C_0. Anweisungsüberdeckungstest (Statement Coverage)

Allgemein



Anweisungsüberdeckungstest

Anweisungsüberdeckungstests, auch C0-Test genannt, sind die am einfachsten anwendbaren kontrollflussorientierten Testmethoden. Mit den Anweisungsüberdeckungstests wird sichergestellt, dass kein “toter Code“, Anweisungen die niemals durchlaufen werden, im Programm existiert. Dies ist ein notwendiges Kriterium um sicherzugehen, dass jede Anweisung auf Fehler untersucht wird. Der Sinn des Anweisungsüberdeckungstests ist die mindestens einmalige Ausführung aller Anweisungen in einem Programm. Ist dies gewährleistet, spricht man von einer völligen Anweisungsüberdeckung. Wie in der unten stehenden Abbildung zu erkennen ist, werden alle Anweisungen mindestens einmal ausgeführt, wenn die While-Schleife einmal durchlaufen wird. Unser Testpfad enthält zwar alle Knoten, aber nicht alle Kanten. Die Kante (n3,n5) wird im optionalen Else-Teil nicht ausgeführt. Genau dieser Fall wird im Zweigüberdeckungstest betrachtet. Durch die Einführung der Zähler kann beim automatisierten Testen kontrolliert werden, ob jede Anweisung ausgeführt wurde, indem man sich die Werte nach dem Testdurchlauf ausgeben lässt. Anweisungsüberdeckungstests werden selten als Haupttestwerkzeug in einem Vollständigkeitstest eingesetzt, denn dafür sind sie i. d. R. zu schwach.

Metrik (Messung)

Der Anweisungsüberdeckungsgrad bestimmt sich wie folgt:

$$C_{\text{Anweisung}} = \frac{\text{Anzahl der ausgefuehrten Anweisungen}}{\text{Gesamtanzahl der Anweisungen}}$$

Vorteil

- Es werden nicht erreichbare Anweisungen im Quellcode aufgedeckt.

Nachteil

- jede Anweisung im Quellcode wird gleichgewichtig gewertet
- bei Steuerstrukturen (Schleifen, Bedingungen, …) werden die Datenabhängigkeiten nicht beachtet
- leere Zweige werden nicht entdeckt

Beispiel

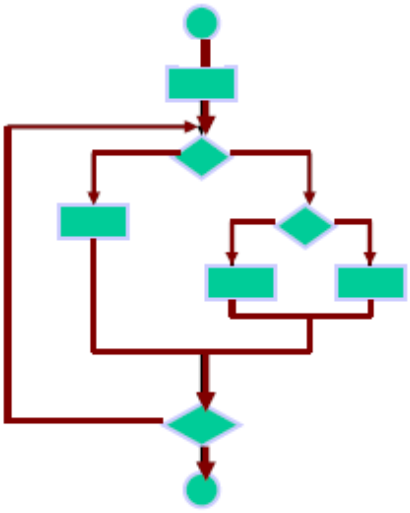
Gegeben sei folgender Quellcode:

```
/* z wird das doppelte des größeren Werts von x oder y zugewiesen */
int z = x;
if (y > x)
   z = y;
z *= 2;
```

In diesem Fall genügt ein einziger Testfall, um eine 100%ige Anweisungsüberdeckung zu erreichen: z. B. x = 0, y = 2 Es sind beliebig viele passende Testfälle denkbar, solange y größer als x ist ansonsten wird z = y nicht ausgeführt, und die Anweisungüberdeckung ist <100%).

C_1. Zweigüberdeckungstest (Branch Coverage)

Allgemein



Zweigüberdeckungstest

Der Zweigüberdeckungstest (C_1–Test) umfasst den Anweisungsüberdeckungstest vollständig. Für den C_1–Test müssen strengere Kriterien erfüllt werden als beim Anweisungsüberdeckungstest. Im Bereich des kontrollflussorientierten Testens wird der Zweigüberdeckungstest als Minimalkriterium angewendet. Mit Hilfe des Zweigüberdeckungstests lassen sich nicht ausführbare Programmzweige aufspüren. Anhand dessen kann man dann Softwareteile, die oft durchlaufen werden, gezielt optimieren.

Analog zum Anweisungsüberdeckungstest wird, um die Codeabdeckung messbar zu machen, der Code in unten stehender Abbildung, durch eine boolesche Hilfsvariable test instrumentiert.

Im Gegensatz zum Anweisungsüberdeckungstest durchläuft der Zweigüberdeckungstest alle Zweige. Der Zweigüberdeckungstest wird auch Entscheidungsüberdeckungstest genannt, da die Hilfsvariable mindestens einmal mit dem Wert true und false durchlaufen werden muss. In diesem Fall muss die While-Schleife mindestens zweimal durchlaufen werden. Mit dem Durchlaufen der Zweige wird auch sichergestellt, dass jeder Knoten (Anweisung) mindestens einmal ausgeführt wird. Somit wird auch das Kriterium für den Anweisungsüberdeckungstest erfüllt. Daher subsumiert der Zweigüberdeckungstest den Anweisungsüberdeckungstest. Schwierig ist es für den Zweigüberdeckungstest Testfälle zu generieren, wo Betriebssystemzustände oder Dateikonstellationen getestet werden müssen. Weiterhin ist diese Technik des Testens zum Testen von 'Schleifen' und zusammengesetzter Entscheidungen nicht geeignet, da weder Kombinationen von Zweigen, noch kompliziert aufgebaute Entscheidungen in Betracht gezogen werden können. Hierfür müssen Erweiterungen herangezogen werden.

Die Zyklomatische Komplexität gibt an, wieviele Testfälle höchstens nötig sind, um eine Zweigüberdeckung zu erreichen.

Weitaus problematischer erweist sich das Zweigüberdeckungsmaß. In dem Fall, dass alle Knoten gleich bewertet sind, verzichtet man auf die Betrachtung der Abhängigkeiten untereinander. Dadurch entsteht kein linearer Zusammenhang zwischen der erreichten Überdeckungsrate und dem Verhältnis zwischen der Anzahl der dazu benötigten Testfälle und der eigentlichen Anzahl der Testfälle, die für die 100 prozentige Zweigüberdeckung notwendig sind. Um den Zweigüberdeckungstest zu verbessern, wird ein Zweig, der abhängig von einem anderen Zweig ist, nicht weiter berücksichtigt. Die Zweige, die nicht abhängig sind, werden als primitiv bezeichnet.

Metrik

Daher ergibt sich für das Überdeckungsmaß: $C_{\text{primitiv}} = \frac{\text{Anzahl der ausgefuehrten primitiven Zweige}}{\text{Anzahl aller primitiven Zweige}}$.

Vorteile

- Deckt nicht erreichbare Zweige auf
- Fehlerentdeckungsrate bei ca. 33%. Ein Fünftel davon sind Berechnungsfehler, der Rest sind Steuerflussfehler.

Nachteile

- Abhängigkeiten zwischen Bedingungen werden nicht berücksichtigt
- Schleifen werden nur unzureichend getestet; siehe Pfadüberdeckungstest
- komplexe Verzweigungsbedingungen werden nur schwach getestet

Beispiel

Gegeben sei folgender Quellcode:

```
/* z wird das Doppelte des größeren Werts von x oder y zugewiesen */
int z = x;
if (y > x)
   z = y;
z *= 2;
```

Im Gegensatz zum Anweisungsüberdeckungstest sind nun mehr als ein Testfall notwendig, um eine 100%ige Zweigüberdeckung zu erreichen, da sowohl der Fall für den durchlaufenen If-Zweig, als auch der Fall für den nicht-durchlaufenen If-Zweig überprüft werden muss:

Testfall 1: x = 0, y = 2 Testfall 2: x = 2, y = 0

Wie auch im Anweisungsüberdeckungstest sind verschiedene Testfälle möglich, die das geforderte Kriterium erfüllen. Nach Ausführung stellt sich heraus, dass das Ergebnis bei beiden Testfällen der Spezifikation entspricht und der Test somit bestanden ist.

C_2. Pfadüberdeckungstest (Path Coverage)

Beim Pfadüberdeckungstest (auch C2-Test bzw. englisch path coverage) werden im Kontrollflussgraphen die möglichen Pfade vom Startknoten bis zum Endknoten betrachtet.

Übersicht

- C_2a - vollständiger Pfadüberdeckungstest
- C_2b - Boundary-Interior Pfadüberdeckungstest
- C_2c - Strukturierter Pfadüberdeckungstest
- Vorteil
- Nachteil

C_2a - vollständiger Pfadüberdeckungstest

Es werden alle möglichen Pfade getestet. Problem: Bei Programmen mit Schleifen kann es unendlich viele Pfade geben.

C_2b - Boundary-Interior-Pfadüberdeckungstest

Im Prinzip wie der C_2a-Test, nur dass nun die Schleifendurchläufe auf <=2 reduziert werden.

Für jede Schleife gibt es zwei Gruppen von Pfaden:

Boundary-Test

- Jede Schleife wird keinmal betreten.
- Jede Schleife wird genau einmal betreten und alle Pfade in dem Schleifenkörper werden einmal abgearbeitet.

Interior-Test

- Das Schleifeninnere gilt als getestet, wenn alle Pfade, die bei zweimaligem Durchlaufen möglich sind, abgearbeitet wurden.

C_2c - Strukturierter Pfadüberdeckungstest

Im Prinzip wie der C_2b-Test, nur dass nun die Anzahl der Schleifendurchläufe auf eine vorgegebene natürliche Zahl n reduziert wird.

Vorteil

- Hohe Fehlererkennungsrate

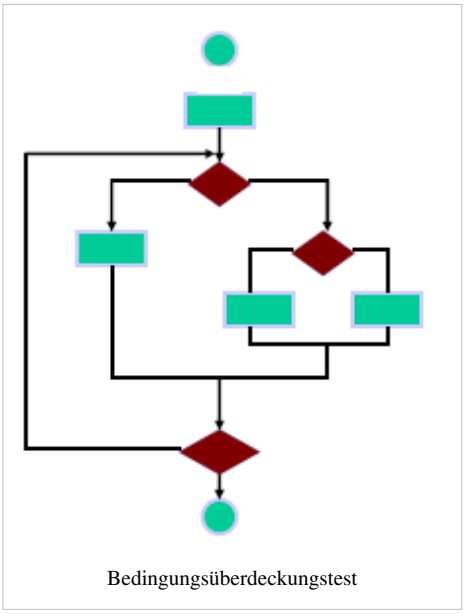
Nachteil

- nicht ansprechbare Pfade auf Grund von Bedingungen

C_3. Bedingungsüberdeckungstest (Condition Coverage)

Das Problem der bisherigen Überdeckungstests (C_1-Test, C_2-Test) ist, dass zusammengesetzte, hierarchische Bedingungen nicht ausreichend getestet werden.

- C_3a - Einfachbedingungsüberdeckungstest
- C_3b - Mehrfachbedingungsüberdeckungstest
- C_3c - minimaler Mehrfachbedingungsüberdeckungstest
- Bewertung



Bedingungsüberdeckungstest

C_3a - Einfachbedingungsüberdeckungstest

Jede atomare Bedingung einer Entscheidung muss einmal mit true und einmal mit false getestet werden. Beispiel:

```
boolean a,b;
if(a || b)
{
   ...
```

```
}
```

Testfall 1 wäre a=false und b=false. Testfall 2 wäre a=true und b=true.

C_3b - Mehrfachbedingungsüberdeckungstest

Dieser Test betrachtet alle atomaren Bedingungen einer Bedingung. Wenn n atomare Bedingungen in der Bedingung stehen, dann werden 2^n Kombinationen gebildet.

Das heißt für das obige Beispiel, dass 4 Testfälle gebildet werden.

C_3c - minimaler Mehrfachbedingungsüberdeckungstest

Diese Version erstellt mehr Testfälle als C_3a und weniger als C_3b, indem jede Bedingung (atomar und zusammengestellt) zu true und zu false evaluiert wird. Die logische Struktur wird hierbei berücksichtigt und der C_1-Test (Zweigüberdeckungstest) ist vollständig in diesem Test enthalten. Ein weiterer Punkt ist, dass der C_3c-Test berechenbar ist.

Vorteil

- Hohe Fehlererkennungsrate

Nachteil

- nicht ansprechbare Pfade auf Grund von Bedingungen

Bewertung

- Unvollständige Auswertung einer Bedingung durch eine Programmiersprache mit sog. short circuit evaluation wie z. B. C/C++, Java, C#.

Beispiel:

```
if (a && b)
{
    ...
}
else
{
   // Lese b aus
}
```

Wenn a false ist, dann ist die Belegung der Variable b egal. z. B. a=false und b=null, dann passiert ein Fehler im else-Zweig

Zusammenfassung

	Kurzname	erfüllte Bedingung	Durchführbarkeit
Anweisungsüberdeckungstest	C_0	jede Anweisung wird mindestens einmal ausgeführt	relativ einfach
Zweigüberdeckungstest	C_1	jede Kante im KFG wird mindestens einmal durchlaufen	realistische Mindestanforderung, vertretbarer Aufwand
Pfadüberdeckungstest	C_2		
Vollständig	C_2a	Alle möglichen Pfade werden durchlaufen	unmöglich bei Schleifen
Boundary-Interior	C_2b	wie C_2a, Schleifen werden jedoch nach speziellen Regeln durchlaufen	aufwändig
Strukturiert	C_2c	wie C_2b, Schleifen werden jedoch genau n-mal durchlaufen	aufwändig
Bedingungsüberdeckungstest	C_3		
Einfachbedingung	C_3a	jede atomare Bedingung wird einmal mit true und false getestet	
Mehrfachbedingung	C_3b	jede true/false Kombination der atomaren Bedingungen wird getestet	sehr hoher Aufwand
Minimale Mehrfachbedingung	C_3c	jede atomare Bedingung und die Gesamtbedingung wird mit true und false getestet	hoher Aufwand

Bewertung

Die Qualität eines Tests hängt entscheidend vom gewählten Test ab: Wurde nur nach C_0 mit Überdeckungsgrad 100 % getestet, so ist dies trotzdem kein verlässlicher Indikator für eine fehlerfreie Software. Wurde hingegen mit C_2 auf 100 % getestet, würde dies ein gutes Kriterium für eine fehlerfreie bzw. -arme Software darstellen. Leider wird dieser Test wegen der kombinatorischen Explosion in der Praxis nur für sicherheitskritische Software (z.B. Luftfahrt) durchgeführt.

Die zweite wichtige Größe ist der Überdeckungsgrad. Dieser ist aber nur bei Verwendung des gleichen Tests untereinander vergleichbar. Bei einem hohen Überdeckungsgrad werden mehr Fehler gefunden als bei einem niedrigen.

Literatur

- Helmut Balzert: *Lehrbuch der Software-Technik*, Spektrum Verlag 2008, ISBN 978-3-8274-1161-7
- Andreas Spillner, Tilo Linz: *Basiswissen Softwaretest*, dpunkt.Verlag 2005, ISBN 3-89864-358-1
- Harry M. Sneed, Mario Winter: *Testen objektorientierter Software*, Hanser Verlag 2002, ISBN 3-446-21820-3
- Ernest Wallmüller: *Software Qualitätssicherung in der Praxis*, Hanser Verlag 2001, ISBN 3-446-21367-8

Weblinks

- [1] Seminararbeit über die Thematik (Grundlagen dieser Seite; PDF-Datei; 511 kB)
- [2] PowerPoint-Vortrag über die Verfahren und theoretische Grundlagen
- [3]Prof. Dr Holger Schlingloff, Fraunhofer Ges., FIRST, "Überdeckungen" (PDF-Datei; 337 kB)
- [4] Steve Cornett, "Code Coverage Analysis"
- [5] DO-248B, Final Annual Report For Clarification Of DO-178B “Software Considerations In Airborne Systems And Equipment Certification”

Referenzen

[1] http://studium.christian-grafe.de/praesentationen/software-engineering/SEII_Kontrollflussorientierte%20Testverfahren.pdf
[2] http://studium.christian-grafe.de/praesentationen/software-engineering/Vortrag_SE2.ppt
[3] http://www2.informatik.hu-berlin.de/~hs/Lehre/2004-WS_SWQS/20041124_Ueberdeckung.pdf
[4] http://www.bullseye.com/coverage.html
[5] http://www.rtca.org/onlinecart/product.cfm?id=202

Korrektheit (Informatik)

Unter **Korrektheit** versteht man in der Informatik die Eigenschaft eines Computerprogramms, einer Spezifikation zu genügen (siehe auch Verifikation). Spezialgebiete der Informatik, die sich mit dieser Eigenschaft befassen, sind die Formale Semantik und die Berechenbarkeitstheorie.

Nicht abgedeckt vom Begriff *Korrektheit* ist, ob die *Spezifikation* die vom Programm zu lösende Aufgabe korrekt beschreibt (siehe dazu Validierung).

Ein Programmcode wird bezüglich einer Vorbedingung P und der Nachbedingung Q **partiell korrekt** genannt, wenn bei einer Eingabe, die die Vorbedingung P erfüllt, jedes Ergebnis die Nachbedingung Q erfüllt. Dabei ist es noch möglich, dass das Programm nicht für jede Eingabe ein Ergebnis liefert, also nicht terminiert.

Ein Code wird **total korrekt** genannt, wenn er partiell korrekt ist und zusätzlich für jede Eingabe, die die Vorbedingung P erfüllt, terminiert. Aus der Definition folgt sofort, dass total korrekte Programme auch immer partiell korrekt sind.

Der Nachweis partieller Korrektheit (Verifikation) kann z. B. mit dem wp-Kalkül erfolgen. Um zu zeigen, dass ein Programm total korrekt ist, muss hier der Beweis der Terminierung in einem gesonderten Schritt behandelt werden. Mit dem Hoare-Kalkül kann die totale Korrektheit in vielen Fällen nachgewiesen werden.

Der Nachweis der Korrektheit eines Programms kann jedoch nicht in allen Fällen geführt werden: das folgt aus dem Halteproblem bzw. aus dem Gödelschen Unvollständigkeitssatz. Auch wenn die Korrektheit für Programme, die bestimmten Einschränkungen unterliegen, bewiesen werden kann, so zählt die Korrektheit von Programmen allgemein zu den nicht-berechenbaren Problemen.

Die Durchführung einer Überprüfung auf Korrektheit bezeichnet man als Beweis. Dabei ist ein Beweis der totalen Korrektheit ein Spezialfall eines mathematischen Beweises, erlaubt also im Gegensatz zum umgangssprachlichen Beweisbegriff eine absolute Aussage.

Weblinks

- Gemeinsame Kriterien für die Prüfung und Bewertung der Sicherheit von Informationstechnik [1]
- IT-Sicherheitskriterien und Evaluierung nach ITSEC [2]
- Deutsche IT-Sicherheitskriterien [3]

Referenzen

[1] https://www.bsi.bund.de/DE/Themen/ZertifizierungundAnerkennung/ZertifizierungnachCCundITSEC/ITSicherheitskriterien/CommonCriteria/commoncriteria_node.html
[2] https://www.bsi.bund.de/DE/Themen/ZertifizierungundAnerkennung/ZertifizierungnachCCundITSEC/ITSicherheitskriterien/ITSEC/itsec_node.html
[3] https://www.bsi.bund.de/DE/Themen/ZertifizierungundAnerkennung/ZertifizierungnachCCundITSEC/ITSicherheitskriterien/DeutscheITSicherheitskriterien/deutscheitsicherheitskriterien_node.html

Prädikatenlogik

Prädikatenlogik oder **Quantorenlogik** ist eine Familie logischer Systeme, die es erlauben, einen weiten und in der Praxis vieler Wissenschaften und deren Anwendungen wichtigen Bereich von Argumenten zu formalisieren und auf ihre Gültigkeit zu überprüfen. Auf Grund dieser Eigenschaft spielt die Prädikatenlogik eine große Rolle in der formalen und nicht formalen Logik sowie in Mathematik, Informatik, Linguistik und Philosophie.

Prädikatenlogik ist eine Erweiterung der Aussagenlogik. In der Aussagenlogik werden zusammengesetzte Aussagen daraufhin untersucht, aus welchen einfacheren Aussagen sie mit Hilfe von aussageverknüpfenden Bindewörtern – in der Logik: Junktoren – zusammengesetzt sind. Zum Beispiel besteht die Aussage „Es regnet oder die Erde ist eine Scheibe“ aus den beiden Aussagen „Es regnet“ und „Die Erde ist eine Scheibe.“ Diese beiden Aussagen lassen sich ihrerseits nicht in weitere Teilaussagen zerlegen – sie werden deshalb atomar oder elementar genannt. In der Prädikatenlogik werden atomare Aussagen hinsichtlich ihrer inneren Struktur untersucht.

Das zentrale Konzept der Prädikatenlogik ist das **Prädikat**. Ein Prädikat in diesem prädikatenlogischen Sinn ist eine Folge von Wörtern mit klar definierten Leerstellen (Auslassungen), die zu einer – wahren oder falschen – Aussage wird, wenn in jede Leerstelle ein Eigenname eingesetzt wird. Zum Beispiel ist die deutsche Wortfolge „... ist ein Mensch“ ein Prädikat, weil durch Einsetzen eines Eigennamens – etwa „Sokrates“ – ein Aussagesatz, zum Beispiel „Sokrates ist ein Mensch“, entsteht. In diesem Sinn lässt sich die elementare Aussage „Die Erde ist eine Scheibe“ prädikatenlogisch in das Prädikat „... ist eine Scheibe“ und den Eigennamen „die Erde“ zerlegen. Anhand der Definition und der Beispiele wird klar, dass der Begriff „Prädikat“ in der Logik, speziell in der Prädikatenlogik, nicht dieselbe Bedeutung hat wie in der traditionellen Grammatik bzw. Schulgrammatik (auch wenn historisch und philosophisch ein Zusammenhang besteht).

Das charakteristische und wichtigste Sprachmittel der Prädikatenlogik ist der **Quantor** oder **Quantifikator**. Quantoren erlauben es, Aussagen darüber zu machen, auf wie viele Individuen ein Prädikat zutrifft. Der Allquantor oder Universalquantifikator sagt aus, dass ein Prädikat auf alle Individuen zutrifft; der Existenzquantor oder Existenzialquantifikator sagt aus, dass ein Prädikat auf mindestens ein Individuum zutrifft. Damit ermöglichen es die Quantoren, Aussagen wie „Alle Menschen sind sterblich“ oder „Es gibt (mindestens) einen rosa Elefanten“ (gleichbedeutend mit: „Es gibt rosa Elefanten“) in ihrer Struktur zu analysieren. Gelegentlich werden zusätzlich numerische Quantoren verwendet, mit denen ausgesagt werden kann, dass ein Prädikat zum Beispiel auf eine genaue Anzahl von Individuen zutrifft; numerische Quantoren lassen sich allerdings auf den All- und den Existenzquantor sowie auf das Identitätsprädikat (meist ausgedrückt durch das Zeichen „=“) zurückführen.

Gottlob Frege und Charles Sanders Peirce[1] entwickelten unabhängig voneinander die Prädikatenlogik. Frege entwickelt und formalisiert sein System in der 1879 erschienenen Begriffsschrift. Ältere logische Systeme, zum Beispiel die traditionelle Begriffslogik seit ihrer Begründung in der aristotelischen Syllogistik oder auch der modernere Relationenkalkül von Charles Peirce, sind hinsichtlich ihrer Ausdrucksstärke echte Teilmengen der Prädikatenlogik, d. h. sie lassen sich vollständig in dieser ausdrücken bzw. in diese übersetzen.

Prädikate

Die in der Einführung gegebene Definition eines Prädikats als Folge von Wörtern mit klar definierten Leerstellen, die zu einer Aussage wird, wenn in jede Leerstelle ein Eigenname eingesetzt wird, ist eine rein formale, inhaltsfreie Definition. Inhaltlich betrachtet können Prädikate ganz unterschiedliche Gegebenheiten ausdrücken, zum Beispiel Begriffe (z. B. „_ ist ein Mensch“), Eigenschaften (z. B. „_ ist rosa“) oder Relationen, d. h. Beziehungen zwischen Individuen (z. B. „$_1$ ist größer als $_2$“ oder „$_1$ liegt zwischen $_2$ und $_3$“). Da die genaue Natur und der ontologische Status von Begriffen, Eigenschaften und Relationen umstritten sind bzw. von unterschiedlichen philosophischen Richtungen unterschiedlich betrachtet werden und da auch die genaue Abgrenzung von Begriffen, Eigenschaften und Relationen untereinander unterschiedlich gesehen wird, ist diese formale Definition die anwendungspraktisch

günstigste, weil sie es erlaubt, Prädikatenlogik zu verwenden, ohne bestimmte ontologische bzw. metaphysische Voraussetzungen akzeptieren zu müssen.

Die Zahl der unterschiedlichen Leerstellen eines Prädikats wird seine Stelligkeit genannt. So ist ein Prädikat mit einer Leerstelle einstellig, eines mit zwei Leerstellen zweistellig usw. Gelegentlich werden Aussagen als nullstellige Prädikate, d. h. als Prädikate ohne Leerstellen betrachtet. Bei der Zählung der Leerstellen werden nur unterschiedliche Leerstellen berücksichtigt.

In formaler Prädikatenlogik werden Prädikate durch Prädikatbuchstaben ausgedrückt, meist Großbuchstaben vom Anfang des lateinischen Alphabets, zum Beispiel F_{-1-2} für ein zweistelliges Prädikat, G_{-1} für ein einstelliges Prädikat oder H_{-1-2-3} für ein dreistelliges Prädikat. Oft werden die Argumente eines Prädikats in Klammern gesetzt und durch Kommas getrennt, sodass die genannten Beispiele als $F(_{-1},_{-2})$ bzw. $G(_{-1})$ und $H(_{-1},_{-2},_{-3})$ geschrieben würden.

Eigennamen und Individuenkonstanten

In Sprachphilosophie und Sprachwissenschaft ist das Thema der Eigennamen ein durchaus komplexes. Für die Behandlung im Rahmen einer einleitenden Darstellung der Prädikatenlogik soll es ausreichen, solche Sprachausdrücke als Eigennamen zu bezeichnen, die genau ein Individuum bezeichnen; das Wort „Individuum“ wird hier in einem ganz allgemeinen Sinn verstanden und meint jedes „Ding“ (physikalischer Gegenstand, Zahl, Person,...), das in irgendeiner erdenklichen Weise von anderen Dingen unterschieden werden kann. Eigennamen im genannten Sinn werden meistens eigentliche Eigennamen (z. B. „Gottlob Frege“) oder Kennzeichnungen (z. B. „der gegenwärtige Bundeskanzler von Österreich“) sein.

Das Gegenstück zu den Eigennamen der natürlichen Sprache sind die Individuenkonstanten der Prädikatenlogik; meist wählt man Kleinbuchstaben vom Anfang des lateinischen Alphabets, zum Beispiel a, b, c. Im Gegensatz zu natürlichsprachlichen Eigennamen bezeichnet jede Individuenkonstante tatsächlich genau ein Individuum. Dies bedeutet keine impliziten metaphysischen Voraussetzungen, sondern legt lediglich fest, dass nur solche natürlichsprachlichen Eigennamen mit Individuenkonstanten ausgedrückt werden, die tatsächlich genau ein Individuum benennen.

Mit dem Vokabular von Prädikatbuchstaben und Individuenkonstanten lassen sich aussagenlogisch atomare Sätze wie „Sokrates ist ein Mensch“ oder „Gottlob Frege ist Autor der ‚Begriffsschrift‘“ bereits in ihrer inneren Struktur analysieren: Übersetzt man den Eigennamen „Sokrates“ mit der Individuenkonstante a, den Eigennamen „Gottlob Frege“ mit der Individuenkonstante b, den Eigennamen bzw. Buchtitel „Begriffsschrift“ mit der Individuenkonstante c und die Prädikate „$_{-}$ ist ein Mensch“ und „$_{-1}$ ist der Autor von $_{-2}$“ mit den Prädikatbuchstaben F_ bzw. G_{-1-2}, dann lässt sich „Sokrates ist ein Mensch“ als Fa und „Gottlob Frege ist der Autor der ‚Begriffsschrift‘“ mit Gbc ausdrücken.

Quantoren

Hauptartikel: Quantor

Quantoren ermöglichen es, Aussagen darüber zu machen, auf wie viele Individuen ein Prädikat zutrifft. Der Existenzquantor sagt aus, dass ein Prädikat auf mindestens ein Individuum zutrifft, beschreibt also die Existenz mindestens eines unter das Prädikat fallenden Gegenstandes. Der Allquantor sagt aus, dass ein Prädikat auf alle Individuen zutrifft.

Der Existenzquantor wird in halbformaler Sprache als „es gibt mindestens ein Ding, sodass...“ oder „es gibt mindestens ein Ding, für das gilt...“ ausgedrückt. In formaler Sprache werden die Zeichen $\exists$ oder $\bigvee$ verwendet.

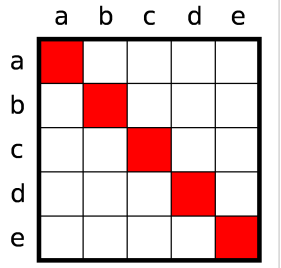
Der Allquantor wird in halbformaler Sprache als „Für jedes Ding gilt: ...“ ausgedrückt, in formaler Sprache durch eines der Zeichen $\forall$ oder $\bigwedge$.

Unmittelbar einsichtig ist die Verwendung von Quantoren bei einstelligen Prädikaten, zum Beispiel „_ ist ein Mensch.“ Die existenzquantifizierte Aussage würde lauten „Es gibt mindestens ein Ding, für das gilt: es ist ein Mensch,“ in formaler Sprache: $\exists x M x$. Dabei ist M_ die Übersetzung des einstelligen Prädikats „_ ist ein Mensch“ und $\exists$ ist der Existenzquantor. Der Buchstabe x ist keine Individuenkonstante, sondern erfüllt dieselbe Funktion, die in der halbformalen Formulierung das Wort „es“ erfüllt: Beide kennzeichnen die Leerstelle, auf die sich der Quantor bezieht. Im gewählten Beispiel erscheint das als redundant, weil es nur einen Quantor und nur eine Leerstelle enthält und daher keine Mehrdeutigkeit möglich ist. Im allgemeinen Fall, in dem ein Prädikat mehr als eine Leerstelle und ein Satz mehr als einen Quantor und mehr als ein Prädikat enthalten kann, wäre ohne die Verwendung geeigneter „Querverweiszeichen“ keine eindeutige Lesart vorgegeben.

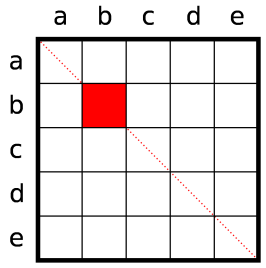
Zum Herstellen der Beziehung zwischen einem Quantor und der Leerstelle, auf die er sich bezieht, werden meist Kleinbuchstaben vom Ende des lateinischen Alphabets verwendet, zum Beispiel die Buchstaben x, y und z; sie werden als Individuenvariablen bezeichnet. Die Leerstelle, auf die sich ein Quantor bezieht, bzw. die Variable, die zum Herstellen dieser Verbindung verwendet wird, bezeichnet man als durch den Quantor gebunden.

Bindet man in einem mehrstelligen Prädikat eine Leerstelle durch einen Quantor, dann entsteht ein Prädikat von um eins niedrigerer Stelligkeit. Das zweistellige Prädikat $L_{\_1\_2}$, „$\__1$ liebt $\__2$“, das die Relation des Liebens ausdrückt, wird durch Binden der ersten Leerstelle durch den Allquantor zum einstelligen Prädikat $\forall x L x \_$, sozusagen zur Eigenschaft, von jedem geliebt zu werden (der Allquantor bezieht sich auf die erste Leerstelle, in der das Individuum steht, von dem die Liebe ausgeht). Durch Binden der zweiten Leerstelle wird daraus hingegen das einstellige Prädikat $\forall x L \_ x$, sozusagen die Eigenschaft, alles und jeden zu lieben (der Allquantor bindet die zweite Leerstelle, also jene, in der das Individuum steht, das die Rolle des oder der Geliebten innehat).

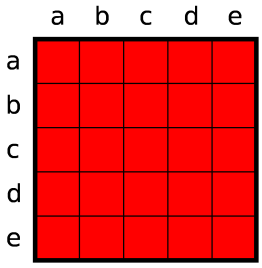
Interessant sind Sätze mit Prädikaten, in denen mehr als eine Leerstelle durch einen Quantor gebunden wird. Die Möglichkeit der Behandlung solcher Sätze macht die große Leistungsfähigkeit der Prädikatenlogik aus, ist aber zugleich der Punkt, an dem das System für den Neueinsteiger etwas kompliziert wird und intensiverer Auseinandersetzung und Übung bedarf. Als kleiner Einblick in die Möglichkeiten der Prädikatenlogik sollen für das einfache zweistellige Prädikat $L_{\_1\_2}$, das zum Beispiel wie oben gelesen werden kann als „$\__1$ liebt $\__2$“, alle Möglichkeiten aufgezählt werden, die Leerstellen durch Quantoren zu binden:



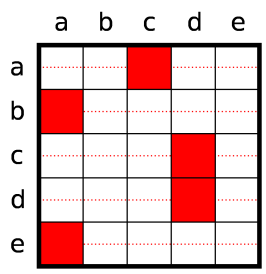
1. $\forall x L x x$:
Alle lieben sich selbst.
(Diagonale ist voll.)



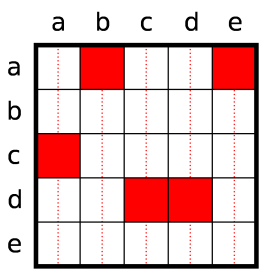
2. $\exists x L x x$:
Jemand liebt sich selbst.
(Diagonale ist nicht leer.)



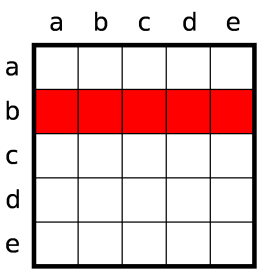
3. $\forall x \forall y L x y$:
Jeder liebt jeden.
4. $\forall x \forall y L y x$:
Jeder wird von jedem geliebt.
(Matrix ist voll.)



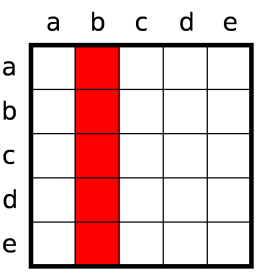
5. $\forall x \exists y Lxy$:
Jeder liebt jemanden.
(Keine Zeile ist leer.)



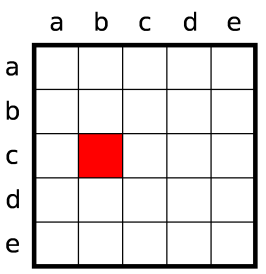
6. $\forall x \exists y Lyx$:
Jeder wird von jemandem geliebt.
(Keine Spalte ist leer.)



7. $\exists x \forall y Lxy$:
Jemand liebt alle.
(Eine Zeile ist voll.)

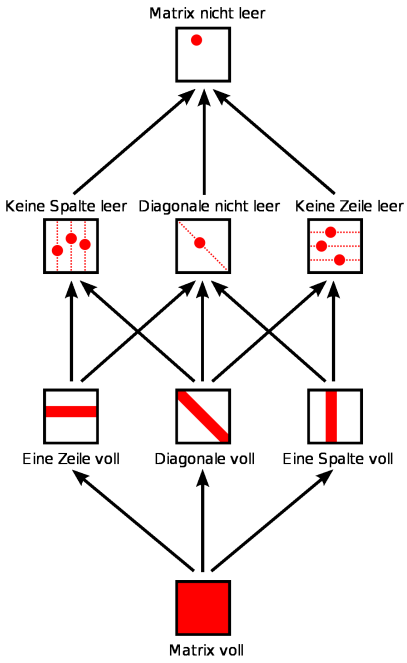


8. $\exists x \forall y Lyx$:
Jemand wird von allen geliebt.
(Eine Spalte ist voll.)



9. $\exists x \exists y Lxy$:
Einer liebt einen.
10. $\exists x \exists y Lyx$:
Einer wird von einem geliebt.
(Matrix ist nicht leer.)

Die Matrizen veranschaulichen die Formeln für den Fall, dass fünf Individuen als Liebende und Geliebte in Frage kommen. Abgesehen von den Sätzen 1 und 3/4 handelt es sich um Beispiele. Die Matrix zu Satz 2 steht z.B. für „b liebt sich selbst.“; die zu Satz 9/10 für „c liebt b.“



Hasse-Diagramm der Implikationen

Wichtig und instruktiv ist es, zwischen den Sätzen 6, $\forall x \exists y Lyx$, und 7, $\exists x \forall y Lxy$, zu unterscheiden: In beiden Fällen wird jeder geliebt; im ersten Fall jedoch wird jeder von irgendjemandem geliebt, im zweiten Fall wird jeder von ein und demselben Individuum geliebt.
Zwischen einigen dieser Sätze bestehen Folgerungszusammenhänge - so folgt etwa aus Satz 7 Satz 6, aber nicht umgekehrt. (Siehe Hasse-Diagramm)

Mit dreistelligen Prädikaten können Formeln wie $\exists x \ \forall y \ \exists z \ Pxyz$ gebildet werden. Mit dem Prädikat „x will, dass y z liebt.“ bedeutet diese Formel „Jemand wünscht allen jemanden zu lieben.“ Hier befindet sich eine Liste aller Formeln mit dreistelligen Prädikaten.
In natürlicher Sprache treten Quantoren in sehr unterschiedlichen Formulierungen auf. Oft werden Wörter wie „alle,“ „keine,“ „einige“ oder „manche“ verwendet, manchmal ist die Quantifizierung nur aus dem Zusammenhang erkennbar – zum Beispiel meint der Satz „Menschen sind sterblich“ in der Regel die Allaussage, dass alle Menschen sterblich sind.

Einige prädikatenlogische Äquivalenzen

Dieses Kapitel stellt exemplarisch einige häufiger gebrauchte prädikatenlogische Äquivalenzen, angezeigt durch den Doppelpfeil, dar.

- $\neg\forall x Px \leftrightarrow \exists x \neg Px$

  Sprich: die Verneinung der Aussage „Alle Autos sind grün“ lässt sich wahlweise als „Nicht alle Autos sind grün“ oder als „Es gibt mindestens ein Auto, das nicht grün ist“ formulieren.

- $\neg\exists x Px \leftrightarrow \forall x \neg Px$

  Diese Äquivalenz zeigt auf, dass die Verneinung der Aussage „Es gibt mindestens einen Ostfriesen, der die Wahrheit sagt“ wahlweise als „Es ist nicht der Fall, dass es mindestens einen Ostfriesen gibt, der die Wahrheit sagt“ oder als „Alle Ostfriesen sagen nicht die Wahrheit“ ausgedrückt werden kann.

- $\exists x(Px \vee Qx) \leftrightarrow (\exists x Px \vee \exists x Qx)$
- $\forall x(Px \wedge Qx) \leftrightarrow (\forall x Px \wedge \forall x Qx)$
- $\forall x(P \vee Qx) \leftrightarrow (P \vee \forall x Qx)$
- $\exists x(P \wedge Qx) \leftrightarrow (P \wedge \exists x Qx)$
- $(\forall x Px \rightarrow Q) \leftrightarrow \exists x(Px \rightarrow Q)$
- $(\exists x Px \rightarrow Q) \leftrightarrow \forall x(Px \rightarrow Q)$
- $(P \rightarrow \forall x Qx) \leftrightarrow \forall x(P \rightarrow Qx)$
- $(P \rightarrow \exists x Qx) \leftrightarrow \exists x(P \rightarrow Qx)$

Arten von Prädikatenlogik

Wenn – wie bisher skizziert – Quantoren die Leerstellen von Prädikaten binden, dann spricht man von **Prädikatenlogik erster Stufe** oder Ordnung, englisch: ***first order logic***, abgekürzt **FOL**; sie ist sozusagen das Standardsystem der Prädikatenlogik.

Eine naheliegende Variation der Prädikatenlogik besteht darin, nicht nur die Leerstellen von Prädikaten zu binden, also nicht nur über Individuen zu quantifizieren, sondern auch Existenz- und Allaussagen *über Prädikate* zu machen. Auf diese Weise kann man Aussagen wie „Es gibt ein Prädikat, für das gilt: es trifft auf Sokrates zu“ und „Für jedes Prädikat gilt: es trifft auf Sokrates zu, oder es trifft nicht auf Sokrates zu“ formalisieren. Zusätzlich zu den individuellen Leerstellen der Prädikate erster Stufe hätte man auf diese Weise Prädikatsleerstellen eingeführt, die zu Prädikaten zweiter Stufe führen, zum Beispiel eben zu „_ trifft auf Sokrates zu“. Von hier ist es nur ein kleiner Schritt zu Prädikaten dritter Stufe, in deren Leerstellen Prädikate zweiter Stufe eingesetzt werden können, und allgemein zu Prädikaten höherer Stufe. Man spricht in diesem Fall daher von **Prädikatenlogik höherer Stufe**, englisch ***higher order logic***, abgekürzt **HOL**.

Die formal einfachste Erweiterung der Prädikatenlogik erster Stufe besteht jedoch in der Ergänzung um Mittel zur Behandlung von Identität. Das entstehende System heißt **Prädikatenlogik der ersten Stufe mit Identität**. Zwar lässt sich Identität in der Prädikatenlogik höherer Stufe definieren, d. h. ohne Spracherweiterung behandeln, doch ist man bestrebt, möglichst lange und möglichst viel auf der ersten Stufe zu arbeiten, weil es für diese einfachere und vor allem vollständige Kalküle gibt, d. h. Kalküle, in denen alle in diesem System gültigen Formeln und Argumente hergeleitet werden können. Für die Prädikatenlogik höherer Stufe gilt das nicht mehr, d.h. es ist für die höhere Stufe nicht möglich, mit einem einzigen Kalkül alle gültigen Argumente herzuleiten.

Umgekehrt kann man Prädikatenlogik der ersten Stufe einschränken, indem man sich zum Beispiel auf einstellige Prädikate beschränkt. Das aus dieser Einschränkung entstehende logische System, die **monadische Prädikatenlogik**, hat den Vorteil, entscheidbar zu sein; das bedeutet, dass es mechanische Verfahren (Algorithmen) gibt, die für jede Formel bzw. für jedes Argument der monadischen Prädikatenlogik in endlicher Zeit feststellen können, ob sie bzw. ob es gültig ist oder nicht. Für einige Anwendungszwecke ist monadische Prädikatenlogik ausreichend; zudem lässt sich die gesamte traditionelle Begriffslogik, namentlich die Syllogistik, in monadischer

Prädikatenlogik ausdrücken.

Parallel zur bereits thematisierten Unterscheidung prädikatenlogischer Systeme nach ihrer Stufe bzw. Ordnung gibt es klassische und nichtklassische Ausprägungen. Von **klassischer Prädikatenlogik** bzw. allgemein von klassischer Logik spricht man genau dann, wenn die beiden folgenden Bedingungen erfüllt sind:

- das behandelte System ist zweiwertig, d. h. jede Aussage nimmt genau einen von genau zwei Wahrheitswerten an (Prinzip der Zweiwertigkeit); und
- der Wahrheitswert von Aussagen, die durch aussagenlogische Junktoren zusammengesetzt sind, ist durch die Wahrheitswerte der zusammengesetzten Aussagen eindeutig bestimmt (Extensionalitätsprinzip).

Weicht man von mindestens einem dieser Prinzipien ab, dann entsteht **nichtklassische Prädikatenlogik**. Selbstverständlich ist es auch innerhalb der nichtklassischen Prädikatenlogik möglich, sich auf einstellige Prädikate zu beschränken (nichtklassische monadische Prädikatenlogik), über Individuen zu quantifizieren (nichtklassische Prädikatenlogik der ersten Stufe), das System um Identität zu erweitern (nichtklassische Prädikatenlogik der ersten Stufe mit Identität) oder die Quantifikation auf Prädikate auszudehnen (nichtklassische Prädikatenlogik höherer Stufe). Ein häufig verwendetes nichtklassisches prädikatenlogisches System ist die modale Prädikatenlogik (siehe Modallogik).

Semantik der Prädikatenlogik

Hauptartikel: Interpretation (Logik)

Wie in formalen Sprachen üblich, wird für die Prädikatenlogik beziehungsweise für jedes prädikatenlogische System eine Interpretation festgelegt, das ist eine Funktion im mathematischen Sinn, die den Zeichen und Ausdrücken der formalen prädikatenlogischen Sprache eine Bedeutung zuordnet. Zunächst wird ein Diskursuniversum festgelegt, das ist die Gesamtheit der unterscheidbaren Gegenstände („Individuen“), auf die sich die zu interpretierenden prädikatenlogischen Aussagen beziehen sollen. Für die klassische Prädikatenlogik werden dann die einzelnen Sprachelemente folgendermaßen interpretiert:

Individuenkonstanten

Jeder Individuenkonstante wird genau ein Element aus dem Diskursuniversum zugeordnet, das heißt jede Individuenkonstante benennt genau ein Individuum.

Einstellige Prädikate

Jedem einstelligen Prädikat wird eine Menge von Individuen aus dem Diskursuniversum zugeordnet. Auf diese Weise wird festgelegt, auf welche Individuen das betroffene Prädikat zutrifft. Wird zum Beispiel dem einstelligen Prädikat F die Menge $\{a, b, c\}$ zugeordnet, dann ist damit festgelegt, dass F auf a, auf b und auf c zutrifft.

Mehrstellige Prädikate

Jedem n-stelligen Prädikat wird eine Menge von n-Tupeln von Individuen aus dem Diskursuniversum zugeordnet.

Aussagen

Um den Wahrheitswert von Aussagen bestimmen zu können, muss die Bewertungsfunktion die Menge aller wohlgeformten Aussagen in die Menge der Wahrheitswerte abbilden, also für jede Aussage der prädikatenlogischen Sprache festlegen, ob sie wahr oder falsch ist. Dies geschieht in der Regel rekursiv nach folgendem Muster (die Bewertungsfunktion wird hier mit **B** bezeichnet):

- **B**($\neg\varphi$) = wahr (φ ist hier eine prädikatenlogische Aussage), wenn **B**(φ) = falsch; andernfalls ist **B**($\neg\varphi$) = falsch. Mit anderen Worten: Die Verneinung einer falschen Aussage ist wahr, die Verneinung einer wahren Aussage ist falsch.

- **B**($\varphi \wedge \psi$) = wahr (φ, ψ sind hier prädikatenlogische Aussagen), wenn **B**(φ) = **B**(ψ) = wahr; andernfalls ist **B**($\varphi \wedge \psi$) = falsch. Mit anderen Worten: Eine Konjunktion ist genau dann wahr, wenn beide Konjunkte wahr sind; andernfalls ist sie falsch.
- Analoge Definitionen werden für alle anderen Junktoren aufgestellt.
- **B**($\varphi(\alpha)$), wobei φ ein einstelliger Prädikatbuchstabe und α eine Individuenkonstante ist, liefert den Wahrheitswert „wahr“, wenn die Interpretation von α ein Element der Interpretation von φ ist, mit anderen Worten: wenn das von α benannte Individuum unter das Prädikat φ fällt. Andernfalls liefert **B**($\varphi(\alpha)$) den Wahrheitswert „falsch“.
- **B**($\varphi(\alpha_1, ..., \alpha_n)$), wobei φ ein n-stelliger Prädikatbuchstabe ist und α_1 bis α_n Individuenkonstanten sind, liefert den Wahrheitswert „wahr“, wenn das n-Tupel $< \alpha_1, ..., \alpha_n >$ Element der Interpretation des Prädikatbuchstaben φ ist. Andernfalls liefert **B**($\varphi(\alpha_1, ..., \alpha_n)$) den Wahrheitswert „falsch“.
- **B**($\forall\chi\varphi(\chi)$), wobei χ eine Individuenvariable ist und $\varphi(\chi)$ ein einstelliges Prädikat, in dessen (ein- oder mehrfach vorkommender) Leerstelle χ eingetragen ist, liefert den Wahrheitswert „wahr“, wenn **B**($\varphi(\frac{\beta}{\chi})$) den Wahrheitswert „wahr“ liefert - unabhängig davon, für welches Individuum β steht. Dabei ist β eine Individuenkonstante, die nicht in $\varphi(\chi)$ vorkommt und $\varphi(\frac{\beta}{\chi})$ ist der Ausdruck, der entsteht, wenn man in $\varphi(\chi)$ jedes Vorkommnis der Individuenvariable χ durch die Individuenkonstante β ersetzt. andernfalls ist **B**($\forall\chi\varphi(\chi)$) = falsch. Mit anderen Worten: **B**($\forall\chi\varphi(\chi)$) ist genau dann wahr, wenn φ tatsächlich auf *alle* Individuen des Diskursuniversums zutrifft.
- **B**($\exists\chi\varphi(\chi)$), wobei χ eine Individuenvariable ist und $\varphi(\chi)$ ein einstelliges Prädikat, in dessen (ein- oder mehrfach vorkommender) Leerstelle χ eingetragen ist, liefert den Wahrheitswert „wahr“, wenn, wenn φ auf *mindestens ein* Individuum aus dem Diskursuniversum zutrifft, das heißt wenn es möglich ist, einer in φ nicht vorkommenden Individuenkonstante β ein Individuum aus dem Diskursuniversum derart zuzuordnen, dass **B**($\varphi(\frac{\beta}{\chi})$) den Wahrheitswert „wahr“ liefert.

Alternativen

Vor dem Aufblühen von Aussagenlogik und Prädikatenlogik dominierte die Begriffslogik in Gestalt der von Aristoteles entwickelten Syllogistik und darauf aufbauender relativ moderater Erweiterungen. Zwei in den 1960er-Jahren in der Tradition der Begriffslogik entwickelte Systeme werden von ihren Vertretern als der Prädikatenlogik gleichmächtig (Freytag) bzw. sogar überlegen (Sommers) bezeichnet, haben aber in der Fachwelt wenig Resonanz gefunden (siehe Artikel Begriffslogik).

Die Gesetze der Prädikatenlogik gelten nur dann, wenn der Bereich der untersuchten Individuen nicht leer ist, d. h. wenn es überhaupt mindestens ein Individuum (welcher Art auch immer) gibt. Eine Modifikation der Prädikatenlogik, die dieser Existenzvoraussetzung nicht unterliegt, ist die Freie Logik (engl. free logic).

Anwendung

Neben der Anwendung als Hilfsmittel vor allem für Informatik, Mathematik und Linguistik spielt die Prädikatenlogik insbesondere in der Konzeption und Programmierung von Expertensystemen und in der künstlichen Intelligenz eine Rolle. In den beiden letztgenannten Gebieten wird oft sogar eine Form angewandter Prädikatenlogik, Prolog („programming in logic“), als Programmiersprache verwendet.

Eine Form der Wissensrepräsentation kann mit einer Sammlung von Ausdrücken in Prädikatenlogik erfolgen.

Der Relationenkalkül, eine der theoretischen Grundlagen von Datenbankabfragesprachen wie etwa SQL, bedient sich ebenfalls der Prädikatenlogik als Ausdrucksmittel.

Quellen

[1] Eric M. Hammer: Semantics for Existential Graphs, *Journal of Philosophical Logic*, Volume 27, Issue 5 (Oktober 1998), Seite 489: „Development of first-order logic independently of Frege, anticipating prenex and Skolem normal forms“

Literatur

Einführungen

- Jon Barwise, John Etchemendy: *Sprache, Beweis und Logik*, *Band 1: Aussagen- und Prädikatenlogik.* Mentis 2005, ISBN 3-89785-440-6; *Band 2: Anwendungen und Metatheorie.* Mentis 2006, ISBN 3-89785-441-4
- Benson Mates: *Elementare Logik - Prädikatenlogik der ersten Stufe.* Vandenhoeck & Ruprecht Göttingen 1997, ISBN 3-525-40541-3
- Wesley C. Salmon: *Logik.* Stuttgart: Reclam 1983 (=Universal-Bibliothek), ISBN 3-15-007996-9

Zur Geschichte

- Karel Berka, Lothar Kreiser: *Logik-Texte. Kommentierte Auswahl zur Geschichte der modernen Logik.* Akademie-Verlag Berlin, 4. Auflage 1986
- William Kneale, Martha Kneale: *The Development of Logic.* Clarendon Press, 1962, ISBN 0-19-824773-7 – Standardwerk zur Geschichte der Logik, in englischer Sprache

Weblinks

- Uschi Robers: Formale Darstellung der Prädikatenlogik (http://www-ai.cs.uni-dortmund.de:8765/lexikon/theorie/logik/node3.html), Technische Universität Dortmund.
- Klaus Dethloff / Christian Gottschall: Einführung in die Prädikatenlogik (http://logik.phl.univie.ac.at/~chris/skriptum/skriptum.html), Universität Wien.

Softwaremetrik

Eine **Softwaremetrik**, oder kurz **Metrik**, ist eine (meist mathematische) Funktion, die eine Eigenschaft von Software in einen Zahlenwert, auch **Maßzahl** genannt, abbildet. Hierdurch werden formale Vergleichs- und Bewertungsmöglichkeiten geschaffen.

Hintergrund

Formell spricht man davon, die Metrik auf eine *Software-Einheit* anzuwenden. Das Ergebnis ist die Maßzahl. Mit Software-Einheit ist in der Mehrheit der Fälle der zugrundeliegende Quellcode gemeint. Da der Quellcode üblicherweise auf eine oder mehrere einzelne Dateien verteilt wird, kann die Metrik je nach Art auf den ganzen Quellcode oder Teile davon angewendet werden. Es gibt zudem Metriken, wie etwa die Function-Point-Analyse, die bereits auf der Spezifikation von Software angewendet werden können, um im Vorfeld den Aufwand zur Entwicklung der Software zu bestimmen.

In der Form des Zahlenwerts, der Maßzahl, dient die Metrik als Maß für eine Eigenschaft, ein Qualitätsmerkmal, von Software. Sie kann einen funktionalen Zusammenhang repräsentieren oder auch aus einer Checkliste abgeleitet werden. Einfache Metriken zeigen die Größe des Quellcode in Zeilen oder Zeichen auf, komplexere Metriken versuchen die Verständlichkeit des Quellcodes zu beurteilen. Mit einer geeigneten Zahl verschiedener Metriken kann beurteilt werden, wie aufwändig (sprich personal- und kostenintensiv) die Wartung, Weiterentwicklung und anschließende Tests der Software werden.

Von einem neu entwickelten Programm werden oft nicht nur bestimmte Funktionen gefordert, sondern auch Qualitätsmerkmale wie zum Beispiel Wartbarkeit, Erweiterbarkeit oder Verständlichkeit. Softwaremetriken können dabei keine korrekte Umsetzung der Funktionen bewerten, sie können allenfalls vorherbestimmen, welchen Aufwand die Erstellung der Software etwa bereiten wird und wie viele Fehler auftreten werden.

Werden während der langfristigen Weiterentwicklung einer Software regelmäßig Metriken angewendet, können negative Trends, also Abweichungen vom Qualitätsziel, frühzeitig entdeckt und korrigiert werden.

Die Interpretation der Daten einer Softwaremetrik ist Aufgabe der Disziplin der Softwaremetrie, dort stellen die Softwaremetriken einen Teil der Basisdaten für die Interpretation dar.

Definition nach IEEE Standard 1061

*Eine **Softwaremetrik** ist eine Funktion, die eine Software-Einheit in einen Zahlenwert abbildet. Dieser berechnete Wert ist interpretierbar als der Erfüllungsgrad einer Qualitätseigenschaft der Software-Einheit* (IEEE Standard 1061, 1992)

Anmerkung: Mit *Software-Einheit* ist dabei in der Regel der zugrundeliegende Quellcode gemeint. Da der Quellcode üblicherweise auf eine odere mehrere einzelne Dateien verteilt wird, kann die Metrik (je nach Art) auf den ganzen Quellcode oder Teile davon angewendet werden. Es gibt zudem Metriken, wie etwa die Function-Point-Analyse, die bereits auf der Spezifikation der Software angewendet werden können.

Ordnung von Softwaremetriken

Metriken bedienen verschiedene Aspekte der entstehenden Software, des angewendeten Vorgehensmodells und der Bewertung der Erfüllung der Anforderungen.

Nutzung

Der Einsatz von Metriken erstreckt sich von der Beurteilung der Entwicklungsphasen über die Beurteilung der Phasenergebnisse bis hin zur Beurteilung der eingesetzten Technologien. Das Ziel der Anwendung einer Metrik in der Softwareentwicklung ist die Fehlerprognose und die Aufwandschätzung, wobei zwischen vorlaufendem, mitlaufendem und retrospektivem Einsatz unterschieden wird.

Beschränkung

Grundsätzlich sind Metriken, die überschaubar bleiben, eindimensional. Damit zwingen sie zur Vereinfachung. In der Regel wird das erreicht, indem jede Metrik auf eine Sicht eingeengt wird. Das bedeutet dann zwingend, dass andere Sichten nicht gleichzeitig in gleicher Qualität bedient werden.

1. Sicht des *Managements*
   - Kosten der Software-Entwicklung (Angebot, Kostenminimierung)
   - Produktivitätssteigerung (Prozesse, Erfahrungskurve)
   - Risiken (Marktposition, Time2Market)
   - Zertifizierung (Marketing)
2. Sicht des *Entwicklers*
   - Lesbarkeit (Wartung, Wiederverwendung)
   - Effizienz und Effektivität
   - Vertrauen (Restfehler, MTBF, Tests)
3. Sicht des *Kunden*
   - Abschätzungen (Budgettreue, Termintreue)
   - Qualität (Zuverlässigkeit, Korrektheit)
   - Return on Investment (Wartbarkeit, Erweiterbarkeit)

Klassifikation

Für die verschiedenen Aspekte der Bewertung gibt es *Entwurfsmetriken*, *wirtschaftliche Metriken*, *Kommunikationsmetriken* usw. Metriken können verschiedenen Klassen zugeordnet werden, die den Gegenstand der Messung oder Bewertung bezeichnen:

1. *Prozess-Metrik*
   - Ressourcenaufwand (Mitarbeiter, Zeit, Kosten)
   - Fehler
   - Kommunikationsaufwand
2. *Produkt-Metrik*
   - Umfang (Lines of Code, Wiederverwendung, Prozeduren, ...)
   - Komplexität
   - Lesbarkeit (Stil)
   - Entwurfsqualität (Modularität, Kohäsion, Kopplung, ...)
   - Produktqualität (Testergebnisse, Testabdeckung, ...)
3. *Aufwands-Metrik*
   - Aufwandsstabilität

- Aufwandsverteilung
- Produktivität
- Aufwand-Termin-Treue

4. *Projektlaufzeit-Metrik*
   - Entwicklungszeit
   - Durchschnittliche Entwicklungszeit
   - Meilenstein-Trend-Analyse
   - Termintreue
5. *Komplexitäts-Metrik*
   - Softwaregröße
   - Fertigstellungsgrad
6. *Anwendungs-Metrik*
   - Schulungsaufwand
   - Kundenzufriedenheit

Gütekriterien

Eine Metrik aus der Produktionsphase der Software allein ist noch kein Gütekriterium. In der Regel werden Gütemerkmale an der Erfüllung der Anforderungen des Kunden und seiner Anwendung gemessen. Dabei sind die Übertragbarkeit der Ergebnisse und die Repräsentanz der Messwerte für den Kundennutzen von Bedeutung:

- *Objektivität*: keine subjektiven Einflüsse des Messenden
- *Zuverlässigkeit*: bei Wiederholung gleiche Ergebnisse
- *Normierung*: Messergebnisskala und Vergleichbarkeitsskala
- *Vergleichbarkeit*: Maß mit anderen Maßen in Relation setzbar
- *Ökonomie*: minimale Kosten
- *Nützlichkeit*: messbare Erfüllung praktischer Bedürfnisse
- *Validität*: von messbaren Größen auf andere Kenngrößen zu schließen (schwierig)

Metriken

Einige der bekannteren Metriken sind:

- Anzahl der Codezeilen, engl. Lines Of Code, kurz LOC. Diese trivialste Metrik ist insb. von der Formatierung des Quellcodes und dem tatsächlichen enthaltenen Kommentarzeilen abhängig. LOC kann daher bei verschiedenen Quellcodes nicht vergleichbare Ergebnisse liefern. Dem kann aber mittels automatischer Formatierung weitgehend entgegen gewirkt werden.
- das Function-Point-Verfahren zur Aufwandsabschätzung in der Analysephase
- COCOMO
- die zyklomatische Komplexität (nach McCabe) zur Komplexitätsbestimmung eines Programmodules
- die Halstead-Metrik zur Implementierungsabschätzung in der Entwurfsphase
- Kontrollflussorientierte Metriken sind im Artikel Kontrollflussorientierte Testverfahren ausgeführt
  - Verhältnis von ausgeführten ELOC zu vorhandenen ELOC, wobei ELOC (executable line of code) eine ausführbare Quellcodezeile ist
  - C_0 Anweisungsüberdeckungskennzahl
  - C_1 Zweigüberdeckungskennzahl
  - C_2 Pfadüberdeckungskennzahl
  - C_3 Bedingungsüberdeckungskennzahl

Durch Kombination vorhandener Metriken werden immer wieder neue Metriken entwickelt, die zum Teil neue Entwicklungen im Software Engineering widerspiegeln. Ein Beispiel hierfür ist die 2007 vorgestellte C.R.A.P. (Change Risk Analysis and Predictions) Metrik zur Beurteilung der Wartbarkeit von Code.

Auswahl geeigneter Metriken

Zur Identifikation geeigneter Maße kann das Goal Question Metric (GQM) Verfahren eingesetzt werden.

Vorgehen

1. *Phasen- und Rollenmodell* festlegen
2. *Ziele* bestimmen
3. *Metrik-Maske* definieren
4. *Messplan* aufstellen
5. *Daten* sammeln
6. Daten validieren
7. Daten analysieren und interpretieren
8. Daten sichern und visualisieren

Literatur

- Christof Ebert und Reiner Dumke: *Software Measurement - Establish, Extract, Evaluate, Execute*. Springer-Verlag, 2007, ISBN 978-3-540-71648-8
- Georg E. Thaller: *Software-Metriken einsetzen - bewerten - messen*. Verlag Technik, 2000, ISBN 3-341-01260-5
- M. Rezagholi: *Prozess- und Technologie Management in der Softwareentwicklung*. Oldenbourg Verlag München Wien, 2004, ISBN 3-486-27549-6
- Ch. Bommer, M. Spindler, V. Barr: *Softwarewartung - Grundlagen, Management und Wartungstechniken*, dpunkt.verlag, Heidelberg 2008, ISBN 3-89864-482-0

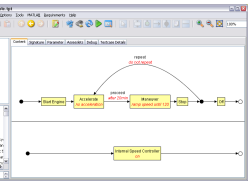
Weblinks

- Softwarequalitätsmanagement [1]
- Präsentation "Metrik basierte Technologie- und Prozessbewertung" [2] (PDF-Datei; 1,33 MB)
- Berechnung von McCabe- und Halstead-Metriken anhand eines Beispielprojekts [3] (PDF-Datei; 737 kB)

Referenzen

[1] http://www-ivs.cs.uni-magdeburg.de/~dumke/ST2/qbegriffe.html
[2] http://www.m-boehmer.de/pdf/Metrik_basierte_Technologie-_und_Prozessbewertung.pdf
[3] http://www.verifysoft.com/de_cmtpp_mscoder.pdf

TPT (Software)

Time Partition Testing (TPT)

Basisdaten

Entwickler	PikeTec GmbH
Aktuelle Version	3.4 (Februar 2011)
Betriebssystem	Windows
Kategorie	Testsoftware
Lizenz	proprietär
Deutschsprachig	nein

TPT [1]

TPT (*Time Partition Testing*) ist eine Methode und ein Software-Werkzeug der Firma PikeTec für den automatisierten Softwaretest oder die Softwareverifikation eingebetteter Systeme. Meist werden eingebettete Systeme mit Hilfe von Testskripten getestet – bei TPT werden Testfälle grafisch modelliert. TPT ist ein modellbasiertes Testwerkzeug für den Modultest, Integrationstest, Systemtest und Regressionstest . TPT unterstützt unter anderem auch Tests von Regelungssystemen und Systemen mit kontinuierlichem Verhalten (Echtzeitsysteme, die mit ihrer Umgebung physikalische Werte bzw. Signale austauschen, die in einem Zeitraster zeitdiskret empfangen oder versandt werden). Die meisten Steuerungs- und Regelungssysteme gehören zu dieser Systemklasse.

TPT umfasst die folgenden Aufgabenbereiche:

- Testfallmodellierung,
- Testdurchführung (vollautomatisch) auf verschiedenen Plattformen mit z. B. MATLAB/Simulink, ASCET, TargetLink, C-Code oder unter Nutzung von Kommunikationsstandards wie CAN, LIN, EtherCAT
- Testauswertung (vollautomatisch)
- Testdokumentation (vollautomatisch)
- Testverwaltung
- Nachverfolgbarkeit von Anforderungen aus Telelogic DOORS und Testfällen

Reaktive Tests

Mit TPT kann jeder Testfall während des Testablaufs gezielt auf das Systemverhalten reagieren, um beispielsweise genau bei Eintreten eines bestimmten Zustands eingreifen und weitere, testrelevante Systemzustände provozieren zu können. Soll beispielsweise für eine Motorsteuerung bei Überschreiten der Leerlaufdrehzahl ein Sensorausfall simuliert werden, um das Verhalten der Motorsteuerung bei dieser Situation zu testen, muss in der Beschreibung des Testfalls auf das Ereignis „Leerlaufdrehzahl überschritten“ reagiert werden können.

Die Reaktivität zusammen mit der Echtzeitfähigkeit der Testausführung erlaubt die Programmierung von zeitdiskreten dynamischen Systemen wie beispielsweise die Implementierung von Filterfunktionen und

Regelalgorithmen mit TPT.

Grafische Testfallmodellierung

Der Ablauf von Testfällen wird bei TPT grafisch mit Hilfe spezieller Zustandsautomaten modelliert. Diese Beschreibungstechnik für Testfälle ist für das Einsatzfeld eingebetteter Systeme besonders naheliegend, weil Testfälle immer aus einzelnen, zeitlich aufeinander folgenden Schritten bestehen. Die Testfälle sind dadurch intuitiv lesbar, auch wenn sie sehr komplex sind.

Einfache Sequenzen (Test Step Listen)

Einfache Abfolgen von Testschritten wie beispielsweise Signal setzen (set channel), Signalrampe (ramp channel), Parameter setzen (set parameter), Warten (wait) können mit Testschritten einfach modelliert werden. In die Sequenzen können Abfragen für das erwartete Testergebnis zur Testbewertung als Testorakel eingefügt werden. Die Testsequenzen können auch mit anderen Modellierungsmethoden kombiniert werden.

Systematische Testfälle

TPT wurde speziell für den Test des kontinuierlichen und reaktiven Verhaltens eingebetteter Systeme entwickelt. Selbst für sehr komplexe Systeme, deren gründlicher Test eine große Menge an Testfällen erfordert, gewährleistet TPT durch sein systematisches Vorgehen bei der Testfallermittlung den Überblick und ermöglicht es so, Schwachstellen im zu testenden System mit einer optimalen Menge von Testfällen aufzudecken. Die zugrunde liegende Idee der Systematik von TPT ist die Separierung von Gemeinsamkeiten und Unterschieden zwischen den Testfällen: Die meisten Testfälle sind einander in ihrem strukturellen Ablauf sehr ähnlich und unterscheiden sich „nur“ in wenigen, aber entscheidenden Details. TPT macht sich diese Tatsache zunutze, indem gemeinsame Strukturen auch gemeinsam modelliert und genutzt werden. Dadurch werden zum einen Redundanzen vermieden. Zum anderen wird sehr klar herausgestellt, worin sich die Testfälle tatsächlich unterscheiden – das heißt, welche spezifischen Aspekte sie jeweils testen. Durch diesen Ansatz wird die Vergleichbarkeit der Testfälle und damit die Übersicht deutlich verbessert und das Hauptaugenmerk des Testers auf das Wesentliche – die differenzierenden Merkmale der Testfälle – gelenkt. Durch die hierarchische Struktur der Testfälle lassen sich komplexe Testprobleme in Teilprobleme zerlegen, was die Übersichtlichkeit und dadurch die Qualität des Tests ebenfalls verbessert. Mit diesen Modellierungstechniken wird der Tester dabei unterstützt, die tatsächlich relevanten Fälle zu finden, Redundanzen zu vermeiden und selbst bei einer großen Menge an Testfällen den Überblick zu bewahren.

Testausführung

TPT-Testfälle sind unabhängig von ihrer Ausführung. Die Testfälle sind mittels einer sogenannten Virtuellen Maschine (VM) quasi auf jeder Plattform automatisch und wenn nötig in Echtzeit ausführbar. Beispiele für die Anwendung sind Model in the Loop (MiL) mit MATLAB/Simulink, TargetLink, ASCET, C-Code, CAN, LIN (mit Restbussimulation), EtherCAT, AUTOSAR-Komponenten[2] ., INCA, LABCAR, SiL und HiL.

TPT kann als Signalgenerator auch zum "Ausprobieren" der Implementierung während der Entwicklungsphase benutzt werden. Durch die Möglichkeit beliebige Signale zu generieren kann TPT auch als Signalgenerator benutzt werden. In MATLAB/Simulink können damit Schalter und Signalgeneratoren komfortabel ersetzt werden. Die Ergebnisse sind wiederholbar und führen zu einer verbesserten Qualität der Entwicklung.

Das testsynchrone Messen steuergeräteinterner Messgrößen erfolgt über die ASAM (Automobilbau) MCD-3 Schnittstelle. Damit können Tools wie INCA oder CANape angesprochen werden und die Messergebnisse stehen zur Testfallbewertung zur Verfügung[3] .

Bei der Testdurchführung steht eine konfigurierbare grafische Benutzeroberfläche mit Anzeigen und Steuerelementen zur Verfügung. Damit können in Echtzeit parallel zur Testausführung Werte vom Benutzer

stimuliert oder angezeigt werden.

Programmierte Testauswertung

Neben der oben angesprochenen Testfallmodellierung kann mit Hilfe einer eigenen Programmiersprache auf Python-Basis und/oder mit MATLAB-Skripten die Testfallprüfung programmiert werden oder mit einem graphischen Interface erfolgen. Zeitliches und funktionales Verhalten des Testobjekts kann daher nicht nur strikt quantitativ, sondern auch qualitativ einfacher beurteilt werden. Die visuelle Inspektion nach jedem Testdurchlauf entfällt.

Eine Auswertung anderweitig erhaltener Messergebnisse ist gleichfalls möglich. Die Messdaten modellinterner TargetLink- oder Simulink-Signale sowie Daten anderer Messgeräte sind automatisch auswertbar.

Einsatzgebiete

TPT wird vorrangig in der Automobilindustrie eingesetzt. Die ursprüngliche Idee ist bei der Daimler AG und Mercedes Benz für die eigene Fahrzeugentwicklung entstanden. Mit den ersten Versionen des Testwerkzeugs wurde dort schon im Jahr 2000 gearbeitet. Inzwischen arbeiten auch andere Automobilfirmen und -zulieferer wie Bosch, Hella und Conti Temic mit dem Testwerkzeug. Daimler hat die Weiterentwicklung von TPT jahrelang selbst koordiniert und TPT dabei für den Automobilsoftwarebereich optimiert.

Nachverfolgbarkeit von Anforderungen

Internationalen Standards für sichere Systeme wie IEC 61508, DO-178B, EN 50128 und ISO 26262 fordern Nachverfolgbarkeit von Anforderungen und Tests. Mit TPT können Requirements aus Telelogic DOORS importiert werden, Testfälle mit den Anforderungen verlinkt werden und die Daten miteinander synchronisiert werden.

Literatur

[1] http://www.piketec.com/products/tpt.php

[2] Jens Lüdemann: *AUTOSAR-Komponententest mit TPT*, In: *2. Elektronik automotive congress* in Ludwigsburg, Germany, 2010. PDF-Artikel (http://www.piketec.com/downloads/papers/Luedemann-2010-AUTOSAR-testing-using-TPT.pdf)

[3] Jens Lüdemann: *Automatic ASAM MCD-3 supported Test*, In: *Open Technology Forum at the Testing Expo* in Stuttgart, Germany, 2009. PDF-Artikel (http://www.piketec.com/downloads/papers/automatic-ASAM-MCD-3-supported-test.pdf)

Weblinks

- Lehmann, TPT - Dissertation, 2003 (http://deposit.ddb.de/cgi-bin/dokserv?idn=970190239)
- PikeTec (http://www.piketec.com/products/tpt.php)
- IX-Studie zu modellbasierten Testwerkzeugen (http://www.heise.de/kiosk/special/ixstudie/09/01/)
- Modell- und Softwareverifikation vereinfacht. Elektronik Automotive, Heft 4.2009 (http://www.piketec.com/downloads/papers/Effizient-Testen-Elektronik-Automotive-04-09.pdf) (PDF-Datei; 314 kB)

Validierung (Informatik)

Validierung in der Informatik und Softwaretechnik ist die dokumentierte Beweisführung, dass ein System die Anforderungen in der Praxis erfüllt.

Validierung als Plausibilitätsprüfung

In der Softwaretechnik bezeichnet **Validierung** (auch *Plausibilisierung*, als Test auf Plausibilität, oder engl. Sanity Check genannt) die Kontrolle eines konkreten Wertes darauf, ob er zu einem bestimmten Datentyp gehört oder in einem vorgegebenen Wertebereich oder einer vorgegebenen Wertemenge liegt. Die meisten Programmfehler und Sicherheitsprobleme sind auf fehlende Plausibilisierung von Eingabewerten zurückzuführen.

Für die Validierung gilt die goldene Regel: *never trust the user* (traue niemals dem Benutzer) - wobei der „Benutzer“ auch ein Programmierer sein kann, der die fraglichen Funktionen und Module verwendet. Die Validierung von Werten kann und soll also an verschiedenen Punkten der Lebenszeit einer Software stattfinden:

- Im Entwicklungsprozess: Während das Programm entsteht, sollten regelmäßig die einzelnen Funktionen und Module so genannten Unit-Tests unterzogen werden, die den Quellcode flächendeckend (*Code Coverage Analysis*) auf korrektes Verhalten überprüfen .
- Bei der Übersetzung des Programmes: einige Arten der Validierung können bereits von Compiler vorgenommen werden, insbesondere die Typprüfung.
- Durch die Laufzeitumgebung: Viele Programmiersprachen haben ein Laufzeitsystem, das bestimmte Arten von Fehlern selbständig erkennt; insbesondere der Zugriff auf nicht vorhandene Objekte wird von vielen modernen Systemen erkannt.
- Zur Laufzeit: Alle Funktionen und Module sollten defensiv umgesetzt sein, sich also nicht darauf verlassen, dass sie korrekt verwendet werden. Das heißt sie sollten, wenn sie mit falschen Parametern verwendet werden, sofort einen Fehler melden statt komplizierte Folgefehler zu riskieren (es gilt die Faustregel: *fail fast* - schneller Abbruch). Hierfür eignet sich vor allem das Konzept der Ausnahmebehandlung. Bei falschen Parameterwerten, die nach Ansicht des Programmieres "eigentlich nie vorkommen dürfen", kommen Assertions zum Einsatz.
- Bei Benutzereingaben: hier gilt der Grundsatz *was sich überprüfen lässt, wird auch überprüft*. Bei ungültigen Eingaben wird eine Fehlermeldung ausgegeben und die Verarbeitung abgelehnt. Bei zweifelhaften Eingaben kann eine Warnung, d. h. eine Bitte um Überprüfung durch den Anwender ausgegeben werden.
- Bei XML Daten: Elemente und Attribute werden gegenüber einem Schema überprüft. Schlägt diese Überprüfung fehl, gelten die XML Daten als nicht *valide*.

Validierung in der Softwarequalitätssicherung

Im Bereich der Softwarequalitätssicherung wird unter **Validierung** (Validation) die Prüfung der Eignung beziehungsweise der Wert einer Software bezogen auf ihren Einsatzzweck verstanden. Die Eignungsprüfung erfolgt auf Grundlage vorher aufgestellten Anforderungsprofils und kann sowohl technisch als auch personell geschehen.

Umgangssprachlich formuliert wird die Frage "Wird das richtige Produkt entwickelt?" beantwortet. (Lit.: Balzert S. 101) Es wird also die Effektivität der Entwicklung sichergestellt.

Im Zusammenhang mit dem V-Modell wird die Validierung der Anforderungsdefinition bzw. dem Abnahmetest zugeordnet. Insofern ist unter der Validierung die Überprüfung der Eignung der Anforderungsdefinition mit den ursprünglichen Zielen des Kunden zu verstehen. Methoden der Validierung umfassen:

- Reviews mit dem Kunden zur Aufdeckung von Unklarheiten und irrtümlichen Annahmen
- Prototyping von Benutzeroberflächen als Kommunikationsgrundlage mit dem Anwender
- Inkrementelle Entwicklung für schnelles Kundenfeedback

In agilen Entwicklungsprozessen wie XP wird Validierung durch die

- permanente Anwesenheit des Kunden
- Nutzerakzeptanztests
- kundennutzenbezogene Releaseplanung im XP-Planspiel
- kurzen Releasezyklen

sichergestellt.

Abgrenzung zur Verifikation

Verifikation als Beweis der Korrektheit

Im Gegensatz zur Validierung bezeichnet die Verifikation in der Softwaretechnik den Nachweis, dass ein Softwareartefakt (Methode, Klasse, Modul, …) die korrekten Ergebnisse liefert. Wurde ein Softwareartefakt verifiziert, so ist sichergestellt, dass es korrekt funktioniert, d. h. das spezifizierte Verhalten aufweist.

Verifikation in der Softwarequalitätssicherung

In der Verifikation wird dagegen überprüft, ob ein System seiner formalen Spezifikation genügt. Die umgangssprachliche Formulierung lautet: "Bauen wir das Produkt richtig?". Es ist durchaus denkbar, dass eine Software ihre Spezifikation erfüllt, jedoch für den Kunden nur geringen Nutzen hat.

Methoden zur Verifikation umfassen:

- Codereviews
- formale Verifikation besonders sensibler Bereiche
- Verfahren des Softwaretests

Literatur

- H. Balzert: Lehrbuch der Software-Technik. Bd.2. Software-Management, Software-Qualitätssicherung, Unternehmensmodellierung. Spektrum Akademischer Verlag, Heidelberg 1998. ISBN 3-8274-0065-1

Weblinks

- Software-Engineering-Kompetenzzentrum [1] (Ein vom Bundesministerium für Bildung und Forschung/BMBF gefördertes Portal)

Referenzen

[1] http://www.software-kompetenz.de/

Verifizierung

Verifizierung oder **Verifikation** (von lat. *veritas* ‚Wahrheit' und *facere* ‚machen') ist der Nachweis, dass ein vermuteter oder behaupteter Sachverhalt wahr ist. Der Begriff wird unterschiedlich gebraucht, je nachdem, ob man sich bei der Wahrheitsfindung nur auf einen geführten Beweis stützen mag oder aber auch die in der Praxis leichter realisierbare bestätigende Überprüfung und Beglaubigung des Sachverhaltes durch Argumente einer unabhängigen Instanz als Verifizierung betrachtet.

Wissenschaftstheorie

In der Wissenschaftstheorie versteht man unter der Verifizierung einer Hypothese den Nachweis, dass diese Hypothese richtig ist. Logischer Empirismus und Positivismus gehen davon aus, dass solche Nachweise führbar seien. Im Rahmen des kritischen Rationalismus (K. Popper) wird argumentiert, dass es Verifikation nicht gibt. Allgemeine Gesetzesaussagen können nur wahr, aber unverifiziert sein, oder mit Beschreibungen von Sachverhalten, die der Aussage widersprechen, falsifiziert werden, sich also als ungültig herausstellen. Zum Verständnis ein Beispiel, das Karl Popper anführt: Angenommen, die Hypothese lautet: , so trägt das Finden zahlreicher weißer Schwäne nur dazu bei, dass die Hypothese beibehalten werden darf. Es bleibt stets die Möglichkeit bestehen, einen andersfarbigen Schwan zu finden. Tritt dieser Fall ein, so ist die Hypothese widerlegt. Solange aber kein andersfarbiger Schwan gefunden wurde, kann die Hypothese weiterhin als nicht widerlegt betrachtet werden.

Auch bei lokalisierenden Existenzhypothesen (auch bestimmte Existenzhypothesen genannt) gilt der Falsifikationismus: Die (allgemeine) Hypothese „Es gibt weiße Schwäne" kann für sich genommen nicht überprüft werden. Hat man jedoch den Aufenthaltsort eines weißen Schwans in einem Raum-Zeit-Gebiet, so ist die Falsifikation möglich, und zwar umso leichter, je eingeschränkter dieses Gebiet ist. Findet sich dort nämlich kein weißer Schwan, so kann die Hypothese als widerlegt betrachtet werden. Umgekehrt folgt die unfalsifizierbare Aussage „Es gibt weiße Schwäne" aus einer solchen bestimmten Existenzhypothese wie „Am 25. August ist ein weißer Schwan im Augsburger Zoo". Sie wird aber nicht bereits dadurch verifiziert, dass die Falsifikation (einstweilen) ausbleibt, so ist es denkbar, dass das beobachtete Tier nur aus der Ferne aussah wie ein Schwan.

Weitere Formen von wissenschaftlichen Hypothesen sowie deren Prüfbarkeit finden sich bei Groeben und Westmeyer.[1]

Informatik

In der Informatik und Softwaretechnik versteht man unter formaler Verifikation den mathematischen Beweis, dass ein Programm (also eine konkrete Implementation) der vorgegebenen Spezifikation entspricht (siehe Korrektheit (Informatik)). Solche Beweise werden mit Hilfe der Methoden der formalen Semantik geführt. Die Verifikation ist jedoch grundsätzlich nicht in jedem Fall möglich, wie das Halteproblem und der Gödelsche Unvollständigkeitssatz zeigen.

Da Beweise zur Verifikation zumeist außerordentlich groß und oft für den Menschen nicht intuitiv sind, werden interaktive oder automatisierte Theorembeweiser eingesetzt. Erstere basieren auf symbolischer Deduktion, während letztere spezielle Datenstrukturen verwenden. Während erstere zur Lösung sehr allgemeiner Probleme verwendet werden können, sind letztere nur in speziellen Bereichen (dann aber mit geringem Aufwand und geringen Vorkenntnissen) anwendbar.

Zur automatisierten Verifikation werden häufig Automatenmodelle eingesetzt. Für kleine Systeme mit endlicher Zustandsmenge (zum Beispiel im Hardwaredesign) werden dafür gerne Endliche Automaten eingesetzt (Modellprüfung), für parallele Prozesse finden Petri-Netze Verwendung. Aber auch andere Automaten können eingesetzt werden. Hintergrund ist die Möglichkeit, formale Spezifikationen in äquivalente Automaten zu überführen (so zeigt der Satz von Büchi-Elgot-Trakhtenbrot die Äquivalenz von endlichen Automaten und Formeln der

monadischen Logik 2. Stufe, siehe MSO), wobei das Problem des Erfüllens einer Spezifikation auf ein äquivalentes Problem der Analyse einer Eigenschaft des Automaten überführt wird. Automaten sind die geeignetere Repräsentation der Problemstellung zum Zwecke der Analyse, da hier gute Algorithmen bekannt sind.

Siehe auch: Validierung (Informatik) und Korrektheit (Informatik)

Raumfahrt

In der von der NASA geprägten Raumfahrt unterscheidet man unter dem Oberbegriff *Verifikation* zwei Tätigkeiten.

Qualifikation

> formeller Nachweis, dass der Entwurf alle Anforderungen des Lastenheftes (specification) einschl. Toleranzen durch Fertigung, Lebensdauer, Fehler usw und die im Schnittstellen-Kontroll-Dokument (Interface Control Document *ICD*) festgelegten Parameter erfüllt. Der Abschluss der Qualifikation ist die Unterschrift des Auftraggebers unter das COQ (Certificate of Qualification). Qualifikations-Verifikationsmethoden sind

- Entwurfsüberprüfung anhand von Zeichnungen (Review of Design / RoD). Für Software wird ein Code-Review durchgeführt.
- Analyse
- Test (Funktionen, Masse, Abmessungen usw)
- Inspektion

Jedes Software-Element wird einzeln und als Teil der Gesamtkonfiguration durch Test qualifiziert wie jedes andere Element des Systems.

Abnahme

> formeller Nachweis, dass das abzuliefernde Produkt alle Anforderungen des Lastenheftes (specification) erfüllt (bezogen auf die Seriennummer) und keine Material- oder Fertigungsfehler hat. Die Abnahme basiert auf dem Nachweis der erfolgreichen, vorhergehenden Qualifikation (einschließlich. Identität der Bauunterlagen zum Qualifikationsmodell). Abschluss der Abnahme ist die Unterschrift des Auftraggebers unter das COA (Certificate of Acceptance). Abnahme-Verifikationsmethoden sind:

- Test
- Inspektion

Die Verifikationstätigkeiten sind die Ursache für die hohen Kosten für Raumfahrtgeräte, verglichen mit einem gleichen technischen Produkt, das unter normalen Industriebedingungen entwickelt wurde. Alle dabei anfallenden Ergebnisse werden dokumentiert und bleiben verfügbar für eventuell später notwendige Fehleruntersuchungen. Während früher diese Regeln für alle Ebenen bis zum Bauelement galten, versucht man heute die Kosten durch Einsatz kommerzieller Bauelemente für nicht sicherheitsrelevante Geräte zu reduzieren.

Während vor einigen Jahren die Raumfahrt der Vorreiter für die Entwicklung miniaturisierter elektronischer Bauelemente war, sind die verfügbaren, extrem komplexen Chips nicht ohne weiteres für die Raumfahrt einsetzbar. Ihr Verhalten unter Weltraumstrahlungsbedingungen (Zerstörung oder zeitweiliges Fehlverhalten) ist meistens nicht bekannt oder kann sogar am Boden nicht getestet werden. Daher hat die *Hardware / Software Interaction Analysis*, die die Reaktion von Hardware - Fehlern auf die im Processor laufende Software untersucht, speziell für stochastische Fehler große Bedeutung erlangt. Bei der NASA wurden bis heute große Summen aufgewendet, um einen kommerziellen Laptop zu finden, der auf der ISS einsetzbar ist.

Ein weiteres, hohe Kosten verursachendes Gebiet ist die Qualifikation des Langzeitverhaltens von Materialien im Weltraum wegen des atomar vorkommenden Sauerstoffs. In vielen Raumfahrtprogrammen wird die Qualifikation der Lebensdauer von Geräten und Materialien stark vereinfacht, um im Kostenrahmen zu bleiben; zum Beispiel gibt es keine Kabel, die für mehr als zwölf Jahre zertifiziert sind. Neuerdings werden von der *European Cooperation on Space Standards* (*ECSS*) die oben unter *Qualifikation* definierten Tätigkeiten mit dem Begriff *Verifikation* bezeichnet; der Oberbegriff *Verfikation* entfällt damit.

Qualitätssicherung

Die DIN EN ISO 8402 vom August 1995, Ziffer 2.17 versteht unter *Verifizierung* „das Bestätigen aufgrund einer Untersuchung und durch Bereitstellung eines Nachweises, daß festgelegte Forderungen erfüllt worden sind." Diese Norm bezieht sich auf die Qualitätssicherung von organisatorischen und betrieblichen Abläufen. Verifizierung wird hier also verstanden als eine „Bestätigung im Nachhinein", ob vorhandene Abläufe die gewünschten Ergebnisse erzielen.

Diese DIN 8402 ist zurückgezogen, aber alle Begriffe des Qualitätsmanagements findet man seit 2000 in der ISO 9000:2000. Die letzte Version ISO 9000:2005, Abschnitt 3.8.4, definiert Verifizierung als „Bestätigung durch einen objektiven Nachweis, dass Anforderungen erfüllt werden". Im Gegensatz dazu beschreibt ISO 9000:2005, Abschnitt 3.8.5, *Validierung* als „die Bestätigung durch objektiven Nachweis, dass die Anforderungen für eine bestimmte Anwendung oder einen bestimmten Gebrauch erfüllt sind".

Beispiel:

- Bestätigung, dass ein Produkt ein unternehmenseigenes, internes Pflichtenheft erfüllt, führt zu einem verifizierten Produkt.
- Bestätigung, dass ein Produkt ein vom Kunden erstelltes Lastenheft und damit so weit die Anforderungen an den Gebrauch durch den Kunden erfüllt, führt zu einem validierten Produkt.

Im Normalfall erfolgt in einem Unternehmen immer zuerst die Verifizierung und dann die Validierung. Dies ist vor allem deshalb richtig, sofern man die EN ISO 9001:2008 befolgt und im Unternehmen die Kundenanforderungen ermittelt und in einem internen Lastenheft festgeschrieben hat. Die Verifizierung ist eine Überprüfung der Konformität zur formali im internen Lastenheft festgehaltenen Kundenanforderungen. Die Validierung ist hingegen eine Art Feldversuch um zu überprüfen, ob das Produkt in der Anwendung wirklich das leistet, was der Kunde haben will und ist somit unter anderem eine Verifizierung des Lastenhefts. Ein Produkt, dass zwar verifiziert aber nicht validiert wurde birgt große Gefahr, dass der Anwender oder Kunde ein Produkt erhält, dass zwar sehr gute Eigenschaften haben kann und dem internen Lastenheft gerecht wird, aber nicht den Anforderungen des Kunden in der Anwendung entspricht.

In der Informatik wird diese Art der Überprüfung der Validierung gegenübergestellt.

Authentifizierung

Die Verifizierung von Personendaten oder Protokollen ist als Vorgang einer gemeinsamen Unterschrift oder als hoheitlicher Akt der Beglaubigung bekannt. Hier findet auch der verwandte Begriff der Authentifizierung als Synonym für einen Identitätsnachweis Verwendung. Umgangssprachlich wird hier oft auch in technischen Dokumentationen von Verifizierung gesprochen.

Beispiele für Verifizierungen

- Nachweis einer genormten Vorgehensweise in einer Projektorganisation.
- Betrieblicher Abgleich von EDV-Protokollen.
- Empirischer Beleg der Wirksamkeit eines Medikamentes.
- Notarielle Beglaubigung einer Unterschrift.
- Überprüfung von Firmenadressen in einem Telefonverzeichnis.
- Der Abgleich von hinterlegten biometrischen Daten bei einer Zugangskontrolle.
- Nachweis von in Simulationen ermittelten Eigenschaften eines Produktes durch Experimente.
- Überprüfung von Online-Bestellungen durch Rückfrage beim Kunden.

Militärwesen

Verifikation ist die Bezeichnung für alle diejenigen Maßnahmen, die der Überwachung der Einhaltung von Abrüstungs- beziehungsweise Rüstungskontrollabkommen dienen. Mittel der Verifikation sind technische Systeme (wie die Satellitenüberwachung), Manöverbeobachter und Inspektoren.

Zusammenfassung

Die frühzeitige Verifizierung beziehungsweise Verifikation eines Prozesses oder einer Aussage hilft, Fehler rechtzeitig zu erkennen und technische, menschliche oder prozessuale Kommunikationsverluste zu vermeiden. Verifizierung ist nicht zu verwechseln mit Validierung.

Die inhaltliche Beurteilung der überprüften Aussagen oder Daten auf Plausibilität oder Wirkung ist nicht Aufgabe der Verifizierung. Es handelt sich hierbei also nur um den Nachweis einer gewissen Authentizität der Aussage an sich. Ein verifizierter Ausdruck (das Ergebnis eines Experimentes) ist somit von Dritter Stelle überprüft, seine wissenschaftliche Aussagekraft ist damit jedoch noch nicht belegt. Die verifizierte Aussage hat somit zwar einen höheren Stellenwert als die unbelegte Behauptung, jedoch einen niedrigeren Stellenwert als der schlüssige Beweis. Der Beweis gehört allerdings nicht mehr zum Bereich der synthetischen (*empirischen*), sondern der analytischen (*theoretischen*) Wahrheit.

Literatur

- Elliott Sober: *Testability* [2]. In: *Proceedings and Addresses of the American Philosophical Association* 73, 1999, S. 47–76 (PDF-Fassung; Verteidigung des Kriteriums).

Weblinks

- Verifizierung von Lexikoneinträgen [3]
- VeriFun [4], ein semiautomatischer Beweiser für funktionale Programme

Einzelnachweise

[1] N. Groeben und H. Westmeyer: *Kriterien psychologischer Forschung*. Juventa, München 1975, S. 107-133
[2] http://philosophy.wisc.edu/sober/test.pdf
[3] http://www.linguistik-online.de/2_99/retti.html
[4] http://www.VeriFun.de/

Versionsnummer

Versionsnummern unterscheiden einzelne Versionen einer Software um die Ergebnisse sich anreihender Entwicklungszyklen gegeneinander abzugrenzen.

Die Versionsnummer ist die Grundlage für die Versionsverwaltung. Den Prozess der Vergabe der Versionsnummer nennt man Versionierung.

Aufbau und Bedeutung

Eine klassische Versionsnummer setzt sich häufig zusammen:

Hauptversionsnummer

(englisch: *major release*) indiziert meist äußerst signifikante Änderung am Programm – zum Beispiel wenn das Programm komplett neu geschrieben wurde (zum Beispiel GIMP 1.x nach 2.x) oder sich bei Bibliotheken keine Schnittstellenkompatiblität aufrechterhalten lässt.

Nebenversionsnummer

(englisch: *minor release*) bezeichnet meistens die funktionelle Erweiterung des Programms.

Revisionsnummer

(englisch: *patch level*) enthält meist Fehlerbehebungen.

Buildnummer

(englisch: *build number*) kennzeichnet in der Regel den Fortschritt der Entwicklungsarbeit in Einzelschritten, wird also zum Beispiel bei 0001 beginnend mit jedem Kompilieren des Codes um eins erhöht. Version 5.0.0-3242 stünde also für das 3242. Kompilationsprodukt einer Software. Verwendet man Versionskontrollsysteme, so wird an Stelle der Build-Nummer gerne eine Nummer verwendet, die die Quellen zum Kompilat innerhalb des Versionskontrollsystems eindeutig identifiziert. Das erleichtert bei Vorkommen eines Bugs die betreffenden Quellen zu finden.

Beispiel für die 2. Version eines Programms, in der 3. Nebenversion und in der 5. Fehlerkorrektur, Build 0041:

```
2.3.5-0041
| | |  └──────── Buildnummer
| | └──────────── Revisionsnummer
| └────────────── Nebenversionsnummer
└──────────────── Hauptversionsnummer
```

Jede dieser Versionsnummern kann auch aus mehreren Ziffern bestehen. Zum Beispiel folgt nach Version 0.9, wenn sich nur die Nebenversion erhöht, 0.10 und nicht 1.0. Bei manchen Programmen ist daher die Nebenversionsnummer zweistellig oder enthält eine führende Null wenn mit mehr als zehn Versionen dieser Art zu rechnen ist (Beispiel: 0.09).

Grundsätzlich gibt es für die Bedeutung der einzelnen Werte jedoch keine festen Vorgaben, vielmehr haben sich Quasi-Standards etabliert: Unter .NET folgt man z. B. dem gegenüber abweichenden Schema <Hauptversionsnummer>.<Nebenversionsnummer>.<Buildnummer>.<Revisionsnummer>[1] (vertauschte Position für Revisionsnummer und Buildnummer).

Zu einem Versionsstand an einem beliebigen Zeitpunkt sagt man auch Build. Die Build-Nummer wird in vielen Projekten unabhängig von den anderen Nummern erhöht und nicht zurückgesetzt. Zum Beispiel gibt es bei Microsoft Windows zwei Produktreihen: in der Reihe Windows 95 sind die Builds 950 (Windows 95) und 2222 (Windows 98 SE) bekannt, in der Reihe Windows NT sind dies die Builds 1381 (Windows NT 4.0 Service Pack 6), 2600 (Windows XP), 6000 (Windows Vista) und 7600 (Windows 7).

Oftmals ist es – vor allem bei Open-Source-Software – der Fall, dass sich die Versionsnummern von Programmen oder Systemen noch vor der Version 1.x befinden. Dies deutet jedoch nicht zwingend darauf hin, dass die Entwicklung noch nicht weit fortgeschritten ist, sondern eher, dass die Version noch nicht das von den Entwicklern gesteckte Ziel erreicht hat und sich weiterhin in der Entwicklung befindet. Teilweise gibt es sogar Open-Source-Programme, die - obwohl sie den Alpha- und Beta-Status längst verlassen haben - weiterhin noch unterhalb der Version 1.0 versioniert sind.

Eine Versionsnummer wird oft nach dem Programmnamen angeführt und manchmal durch „v.“ oder „V“ (für *Version*) speziell gekennzeichnet.

Marketingaspekte

Für die Software-Entwicklung stellen Versionsnummern eine weitaus wichtigere Information als für den Kunden dar. Mit Hilfe der Versionsnummern kann unter anderem sichergestellt werden, dass in Entwicklergruppen neue Programmteile nicht mit älteren überschrieben werden. (siehe Versionsverwaltung).

In größeren Softwareprojekten wird unter Umständen aus Marketingaspekten von der internen, eher technisch motivierten Versionierung abgewichen, was dann zu Versionsnamen führt („Windows XP“ entspricht beispielsweise „Windows NT 5.1“), aus denen sich die Versionsfolge nicht mehr ohne weiteres erkennen lässt. Aus Marketinggründen kann es auch zum Überspringen von Versionsnummern kommen, um keine niedrigere Version (welche als „älter“ interpretiert werden kann) als Mitwettbewerber zu haben. Dies war beispielsweise bei WinWord, dessen Version von 2.0 auf 6.0 sprang, und Slackware der Fall. Auch bei verschiedener Software vom gleichen Hersteller kann es ähnliche Fälle geben, so wurde z.B. die erste Version von Windows NT Windows NT 3.1 genannt, da sie nach Windows 3.1 auf den Markt kam. Bei „Windows 7“ wurde von der internen Versionsnummer („Windows NT 6.1“) aus technischen und psychologischen Gründen abgewichen.

Auch werden andere Arten, Programmversionen voneinander zu unterscheiden, verwendet, da sie leichter zu merken sind:

- Jahreszahlen, zum Beispiel: Windows *95;*
- alphanumerische Bezeichner, zum Beispiel: Adobe Photoshop *CS2,* Adobe Flash *MX;*
- Codenamen, zum Beispiel: Mac OS X *Tiger;*
- Jahre und Monate; so wird zum Beispiel die Versionsnummer der Ubuntu-Distribution aus der letzten Ziffer der Jahreszahl und dem Monat des Erscheinens gebildet. Zum Beispiel Ubuntu 6.04 (für 2006.04 = April 2006).
- Die Versionsnummer von TeX nähert sich π an; die von METAFONT der Eulerschen Zahl e.

Ergänzungen

Je nach Entwicklungsstadium der Software gibt es noch Ergänzungen:

Logo einer Beta-Version

Alpha
: während der Entwicklung der Software, sehr frühes Stadium

Beta
: zum Testen vorgesehen, begrenzter Anwenderkreis

RC
: Veröffentlichungskandidat (release candidate *rc*), abschließende Testversion

Release (final)
: endgültige Version

Patch

auch patchlevel *pl* – Korrekturlevel

Einzelnachweise

[1] Versionsnummern in .NET (http://msdn.microsoft.com/en-us/library/ms973869.aspx#managevers_topic1) (englisch)

Programmierschnittstelle

Eine **Programmierschnittstelle** (engl. *application programming interface* (*API*), dt. *„Schnittstelle zur Anwendungsprogrammierung“*) ist ein Programmteil, der von einem Softwaresystem anderen Programmen zur Anbindung an das System zur Verfügung gestellt wird. Im Gegensatz zu einer Binärschnittstelle (ABI) definiert ein API nur die Programmanbindung auf Quelltextebene.[1]

Neben dem Zugriff auf Datenbanken oder Hardware wie Festplatte oder Grafikkarte, kann eine API auch das Erstellen von Komponenten der grafischen Benutzeroberfläche ermöglichen oder vereinfachen.

Heutzutage stellen auch viele Internetdienste APIs zur Verfügung. Folgende APIs gibt es unter anderem für/von soziale/n Netzwerke/n: Connect (Facebook), Direct (YouTube) sowie Sitecore (Facebook, Twitter, StudiVZ)

Im weiteren Sinne wird die Schnittstelle jeder Bibliothek (*Library*) als API bezeichnet.

Einteilung

Programmierschnittstellen lassen sich in folgende Klassen einteilen:

- funktionsorientiert (z. B. Dynamic Link Library)
- dateiorientiert (z. B. Gerätedateien unter UNIX)
- objektorientiert (z. B. ActiveX-DLLs)
- protokollorientiert (z. B. FTP)

Funktionsorientierte Programmierschnittstellen

kennen nur Funktionen mit oder ohne Rückgabewert als Mittel der Kommunikation. Dabei wird fast immer das Konzept der Handles verwendet. Man ruft eine Funktion auf und bekommt ein Handle zurück. Mit diesem Handle lassen sich weitere Funktionen aufrufen, bis abschließend das Handle wieder geschlossen werden muss. Das BIOS eines Personal Computer ist die älteste Programmierschnittstelle für diesen Rechnertyp.

Dateiorientierte Programmierschnittstellen

werden über die normalen Dateisystemaufrufe `open`, `read`, `write` und `close` angesprochen. Sollen Daten an ein Objekt gesendet werden, werden diese mit `write` geschrieben, sollen welche empfangen werden, werden sie mit `read` gelesen. Unter UNIX ist dieses Prinzip bei der Ansteuerung von Gerätetreibern weit verbreitet.

Objektorientierte Programmierschnittstellen

verwenden Schnittstellenzeiger und sind damit deutlich flexibler als die funktionsorientierten Programmierschnittstellen. Häufig wird eine Typbibliothek mitgegeben.

Protokollorientierte Programmierschnittstellen

sind unabhängig vom Betriebssystem und der Computerhardware. Allerdings muss das Protokoll stets neu implementiert werden. Um diesen Aufwand zu minimieren, wird die protokollorientierte Schnittstelle durch eine funktions- oder interfaceorientierte Schnittstelle gekapselt. Man kann hier weiterhin zwischen allgemeinen (z. B. SOAP) und anwendungsspezifischen (z. B. SMTP) Protokollen unterscheiden.

Einzelnachweise

[1] Oliver Thoma (2006). *Mac OS X 10.4 Tiger* (http://books.google.at/books?id=xXHNB1Xm3UEC&pg=PA57&lpg=PA57&dq="was+ist+eine+programmierschnittstelle"&source=bl&ots=e72aNoPR9m&sig=2-q0V54ha2OQnvyN_0i4AezUDAk&hl=de&ei=hVmPTf_YLZK5hAf6pOC7Dg&sa=X&oi=book_result&ct=result&resnum=1&ved=0CBoQ6AEwAA#v=onepage&q="was ist eine programmierschnittstelle"&f=false). Google Bücher. Abgerufen am 27. März 2011.

Literatur

- E. Gamma, R. Helm, R. Johnson, J. Vlissides: *Design Patterns – Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995, ISBN 0-201-63361-2.
- Joshua Bloch: *How to Design a Good API and Why it Matters* (http://static.scribd.com/docs/908bil5xonqxe.pdf).

Versionsverwaltung

Eine **Versionsverwaltung** ist ein System, das zur Erfassung von Änderungen an Dokumenten oder Dateien verwendet wird. Alle Versionen werden in einem Archiv mit Zeitstempel und Benutzerkennung gesichert und können später wiederhergestellt werden. Versionsverwaltungssysteme werden typischerweise in der Softwareentwicklung eingesetzt um Quelltexte zu verwalten. Versionsverwaltung kommt auch bei Büroanwendungen oder Content-Management-Systemen zum Einsatz.

Ein Beispiel ist auch die Wikipedia, hier erzeugt die Software nach jeder Änderung eines Artikels eine neue Version. Da zu jedem Versionswechsel die grundlegenden Angaben wie Verfasser und Uhrzeit festgehalten werden, kann genau nachvollzogen werden, wer wann was geändert hat. Bei Bedarf – beispielsweise bei versehentlichen Änderungen – kann man zu einer früheren Version zurückkehren.

Die Versionsverwaltung ist eine Form des Variantenmanagements. Für Versionsverwaltungssysteme ist die Abkürzung **VCS** (Version Control System) gebräuchlich.

Hauptaufgaben eines Versionsverwaltungssystems

- Protokollierungen der Änderungen: Es kann jederzeit nachvollzogen werden, wer wann was geändert hat.
- Wiederherstellung von alten Ständen einzelner Dateien: Somit können versehentliche Änderungen jederzeit wieder rückgängig gemacht werden.
- Archivierung der einzelnen Stände eines Projektes: Dadurch ist es jederzeit möglich, auf alle Versionen zuzugreifen.
- Koordinierung des gemeinsamen Zugriffs von mehreren Entwicklern auf die Dateien.
- Gleichzeitige Entwicklung mehrerer *Entwicklungszweige* (engl. *Branches*) eines Projektes.

Terminologie

Ein *Branch*, zu Deutsch Zweig, ist eine Abspaltung von einer anderen Version, so dass unterschiedliche Versionen parallel weiterentwickelt werden können. Änderungen können dabei von einem Branch auch in einem anderen wieder einfließen, das wird dann als *Merging*, zu deutsch verschmelzen, bezeichnet. Oft wird der Hauptentwicklungszweig als *Trunk* bezeichnet. Branches können zum Beispiel für neue Hauptversionen einer Software erstellt werden oder für Entwicklungszweige für unterschiedliche Betriebssysteme oder aber auch um experimentelle Versionen zu haben. Forks sind in der Regel Branches. Ein *Tag* kennzeichnet einen konkreten Versionsstand eines Branches, z. B. eine zur Auslieferung bestimmte Version. Er erhält einen Namen, über den er später noch verfügbar ist.

Funktionsweise

Damit die eingesetzten Programme wie z. B. Texteditoren oder Compiler mit den im *Repository* (engl. Behälter, Aufbewahrungsort) abgelegten Dateien arbeiten können, ist es erforderlich, dass jeder Entwickler sich den aktuellen (oder einen älteren) Stand des Projektes in Form eines Verzeichnisbaumes aus herkömmlichen Dateien erzeugen kann. Ein solcher Verzeichnisbaum wird als *Arbeitskopie* bezeichnet. Ein wichtiger Teil des Versionsverwaltungssystems ist ein Programm, das in der Lage ist, diese *Arbeitskopie* mit den Daten des Repositorys zu synchronisieren. Das Übertragen einer Version aus dem Repository in die Arbeitskopie wird als *Checkout* oder *Aktualisieren* bezeichnet, während die umgekehrte Übertragung *Check-in* oder *Commit* genannt wird. Solche Programme sind entweder kommandozeilenorientiert, mit grafischer Benutzeroberfläche oder als Plugin für integrierte Softwareentwicklungsumgebungen ausgeführt. Häufig werden mehrere dieser verschiedenen Zugriffsmöglichkeiten wahlweise bereitgestellt.

Es gibt drei Arten der Versionsverwaltung, die älteste funktioniert lokal, also nur auf einem Computer, die nächste Generation funktionierte mit einem zentralen Archiv und die neueste Generation arbeitet verteilt, also ohne zentrales Archiv. Allen gemein ist, dass die Versionsverwaltungssoftware dabei üblicherweise nur die Unterschiede zwischen zwei Versionen speichert, um Speicherplatz zu sparen. Die meisten Systeme verwenden hierfür ein eigenes Dateiformat oder eine Datenbank. Dadurch kann eine große Zahl von Versionen archiviert werden. Durch dieses Speicherformat kann jedoch nur mit der Software des Versionsverwaltungssystems auf die Daten zugegriffen werden, die die gewünschte Version bei einem Abruf unmittelbar aus den archivierten Versionen rekonstruiert.

Lokale Versionsverwaltung

Bei der lokalen Versionsverwaltung wird oft nur eine einzige Datei versioniert, diese Variante wurde mit Werkzeugen wie SCCS und RCS umgesetzt. Sie findet auch heute noch Verwendung in Büroanwendungen, die Versionen eines Dokumentes in der Datei des Dokuments selbst speichern. Auch in technischen Zeichnungen werden Versionen zum Beispiel durch einen Änderungsindex verwaltet.

Zentrale Versionsverwaltung

Diese Art ist als Client-Server-System aufgebaut, sodass der Zugriff auf ein Repository auch über Netzwerk erfolgen kann. Durch eine Rechteverwaltung wird dafür gesorgt, dass nur berechtigte Personen neue Versionen in das Archiv legen können. Die Versionsgeschichte ist hierbei nur im Repository vorhanden. Dieses Konzept wurde vom Open Source Projekt Concurrent Versions System (CVS) populär gemacht, mit Subversion (SVN) neu implementiert und von vielen kommerziellen Anbietern verwendet.

Verteilte Versionsverwaltung

Die verteilte Versionsverwaltung verwendet kein zentrales Repository mehr. Jeder, der an dem verwalteten Projekt arbeitet, hat sein eigenes Repository und kann dieses mit jedem beliebigen anderen Repository abgleichen. Die Versionsgeschichte ist dadurch genauso verteilt. Änderungen können lokal verfolgt werden, ohne eine Verbindung zu einem Server aufbauen zu müssen.

Im Gegensatz zur zentralen Versionsverwaltung kommt es nicht zu einem Konflikt, wenn mehrere Benutzer dieselbe Version einer Datei ändern. Die sich widersprechenden Versionen existieren zunächst parallel und können weiter geändert werden. Sie können später in eine neue Version zusammengeführt werden. Dadurch entsteht ein gerichteter azyklischer Graph (Polyhierarchie) anstatt einer Kette von Versionen. In der Praxis werden bei der Verwendung in der Softwareentwicklung meist einzelne Features oder Gruppen von Features in separaten Versionen entwickelt und diese bei größeren Projekten von Personen mit einer Integrator-Rolle überprüft und zusammengeführt.

Systembedingt bieten verteilte Versionsverwaltungen keine Locks. Da wegen der höheren Zugriffsgeschwindigkeit die Granularität der gespeicherten Änderungen viel kleiner sein kann, können sie sehr leistungsfähige, weitgehend automatische Merge-Mechanismen zur Verfügung stellen.

Eine Unterart der Versionsverwaltung bieten einfachere Patchverwaltungssysteme, die Änderungen nur in eine Richtung in Produktivsysteme einspeisen.

Konzepte der Versionsverwaltung

Lock Modify Write

Diese Arbeitsweise eines Versionsverwaltungssystems wird auch als Lock Modify Unlock bezeichnet. Die zugrunde liegende Philosophie wird **pessimistische Versionsverwaltung** genannt. Einzelne Dateien müssen vor einer Änderung durch den Benutzer gesperrt und nach Abschluss der Änderung wieder freigegeben werden. Während sie gesperrt sind, verhindert das System Änderungen durch andere Benutzer. Der Vorteil dieses Konzeptes ist, dass kein Zusammenführen von Versionen erforderlich ist, da nur immer ein Entwickler eine Datei ändern kann. Der Nachteil ist, dass man unter Umständen auf die Freigabe eines Dokuments warten muss, um eine eigene Änderung einzubringen. Zu beachten ist, dass Binärdateien (im Gegensatz zu Quelltextdateien) in der Regel diese Arbeitsweise erfordern, da das Versionsverwaltungssystem verteilte Änderungen nicht automatisch synchronisieren kann.

Ältester Vertreter dieser Arbeitsweise ist das Revision Control System. Bekannter ist aber Visual SourceSafe, das den Entwicklungswerkzeugen von Microsoft kostenfrei beilag und inzwischen weitgehend durch den Team Foundation Server abgelöst wurde.

Verteilte Versionsverwaltungssysteme kennen systembedingt diese Arbeitsweise nicht.

Copy Modify Merge

Ein solches System lässt gleichzeitige Änderungen durch mehrere Benutzer an einer Datei zu. Anschließend werden diese Änderungen automatisch oder manuell zusammengeführt (Merge). Somit wird die Arbeit des Entwicklers wesentlich erleichtert, da Änderungen nicht im Voraus angekündigt werden müssen. Insbesondere, wenn viele Entwickler räumlich getrennt arbeiten, wie es beispielsweise bei Open-Source-Projekten häufig der Fall ist, ermöglicht dies erst effizientes Arbeiten, da kein direkter Kontakt zwischen den Entwicklern benötigt wird. Problematisch bei diesem System sind Binärdateien, da diese nicht automatisch zusammengeführt werden können. Manche Vertreter dieser Gattung unterstützen daher auch alternativ das Lock-Modify-Write-Konzept für bestimmte Dateien.

Die zugrunde liegende Philosophie wird als **optimistische Versionsverwaltung** bezeichnet und wurde entwickelt, um die Schwächen der pessimistischen Versionsverwaltung zu beheben. Alle modernen zentralen und verteilten Systeme setzen dieses Verfahren um.

Objekt-basierte Versionierung

Über die Grenze des abspeichernden Mediums, der Datei, hinaus geht die Objekt-basierte Versionierung. Objekte werden in der Informatik als sogenannte Instanzen, also Ausprägungen eines Schemas, erzeugt. Auch diese können versioniert gespeichert werden. Die Versionsverwaltung, welche die unterschiedlichen Objektversionen abspeichert, muss mit den entsprechenden Objekttypen umgehen können. Aus dem Grund liest ein solches System nicht allein die Datei und überprüft diese auf Veränderungen, sondern kann die darin enthaltene Semantik interpretieren. Üblicherweise werden Objekte dann nicht dateibasiert, sondern in einer Datenbank abgespeichert. Produktdatenmanagement-Systeme (PDM-Systeme) verwalten ihre Daten nach diesem Prinzip.

Beispiele

Die folgende Tabelle enthält einige Versionsverwaltungssysteme als Beispiele für die verschiedenen Ausprägungsarten.

	Open-Source-Systeme	**Proprietäre Systeme**
Zentrale Systeme	• CVS • Subversion	• Alienbrain • Perforce • Team Foundation Server • Visual SourceSafe • ClearCase
Verteilte Systeme	• Bazaar • Darcs • Git • GNU arch • Mercurial • Monotone	• BitKeeper

Weblinks

- Vergleich von über 25 Versionsverwaltungen [1] (englisch, 2010)
- Lange Liste von Versionverwaltungssystemen speziell für Linux [2] (englisch, 2004)
- Versionskontrollsysteme in der Softwareentwicklung [3] (PDF-Datei; 790 kB, 2005)
- Verteilte Versionskontrollsysteme [4] (2009), Chaosradio Express
- Einführung in Verteilte Versionsverwaltung [5] (englisch)

Referenzen

[1] http://better-scm.berlios.de/comparison/comparison.html
[2] http://linuxmafia.com/faq/Apps/vcs.html
[3] http://www.gesis.org/fileadmin/upload/forschung/publikationen/gesis_reihen/iz_arbeitsberichte/ab_36.pdf
[4] http://chaosradio.ccc.de/cre130.html
[5] http://betterexplained.com/articles/intro-to-distributed-version-control-illustrated/

wp-Kalkül

Der **wp-Kalkül** ist ein Kalkül in der Informatik zur Verifikation eines imperativen Programmcodes. Die Abkürzung wp steht für *weakest precondition*, auf deutsch schwächste Vorbedingung. Bei der Verifikation geht es nicht darum, die Funktion mit einer bestimmten Menge an Eingabedaten auf korrekte Ergebnisse zu testen, sondern darum, eine allgemeingültige Aussage über das korrekte Ablaufen des Programms zu erhalten.

Die Überprüfung der Korrektheit geschieht durch Rückwärtsanalyse des Programmcodes. Ausgehend von der Nachbedingung wird geprüft, ob diese durch die Vorbedingung und den Programmcode garantiert wird.

Alternativ kann auch der Hoare-Kalkül benutzt werden, bei dem im Gegensatz zum wp-Kalkül eine Vorwärtsanalyse stattfindet.

Einleitung

Der wp-Kalkül hilft gewisse Zusicherungen im Programm zu machen. Eine Zusicherung ist eine prädikatenlogische Aussage über den Inhalt der Variabelen an der bestimmten Stelle. Eine Zusicherung vor einem Programmtext heißt Vorbedingung P, eine Zusicherung danach Nachbedingung Q.

```
// x ist gerade
// P: (x % 2) = 0
x = x + 1;
// x ist ungerade
// Q: (x % 2) = 1
```

Der wp-Kalkül erlaubt es nun anhand bestimmter Regeln aus einer Nachbedingung die nötige Vorbedingung, und zwar die *schwächste Vorbedingung*, zu schließen, die erfüllt sein muss damit nach Ausführung des Programmcodes die Nachbedingung erfüllt ist.

Transformationen

Um die schwächste Vorbedingung P für eine Nachbedingung Q zu erhalten schreibt man *P = wp("Anweisung", Q)*. Für diese Funktion gelten nun noch einige Definitionen:

1. wp("", Q) = Q - Nichts passiert, die Vorbedingung bleibt gleich
2. wp("Fehler", Q) = falsch - Fehler dürfen nicht auftreten
3. $wp(A,Q) \wedge wp(A,R) = wp(A, Q \wedge R)$ - Distributivität der Konjunktion
4. $wp(A,Q) \vee wp(A,R) = wp(A, Q \vee R)$ - Distributivität der Disjunktion

Sequenzregel

Zwei Programmstücke C_1 und C_2 können zu einem Programmstück $C_1; C_2$ zusammengefügt werden, wenn die Vorbedingung von C_2 aus der Nachbedingung von C_1 folgt.
In der konkreten Analyse eines Programms kommt man dadurch dem Ziel, einer Vorbedingung für das gesamte Programm dadurch näher, indem man die Sequenzregel anwendet und die Nachbedingung Q in eine Nachbedingung Q' überführt die eine Zeile, oder logische Einheit, weiter oben steht. Man schiebt also, bildlich gesprochen, die Zusicherung am Ende eine Zeile nach oben, indem man die Vorbedingung dieser einen Zeile ermittelt. Dazu ein kleines Beispiel (man sollte von unten nach oben lesen):

```
// P = wp("x = x * 2 + y", Q')
x = x * 2 + y;
// Q' = wp("x = x + 1", Q)
x = x + 1;
// Q: x < 100
```

Zuweisungsregel

Ist x eine Variable und e ein Ausdruck, so erhält man die schwächste Vorbedingung, indem man jedes Auftreten der Variable x in Q durch den Ausdruck e ersetzt.

$$wp("x := e", Q) = Q[x/e]$$

Diese Ersetzung führt dazu, dass man die Auswirkungen der Zuweisungen quasi innerhalb der Nachbedingung simuliert. Diese Ersetzung kann man allerdings nur vornehmen, wenn e seiteneffektfrei bezüglich Q ist, diese also nicht verändert. Dazu ein Beispiel:

```
// wp("x = x + 1", x > 100) = (x + 1) > 100 = x > 99
x = x + 1;
// Q: x > 100
```

Schleifen

Die Behandlung von Schleifen ist etwas schwieriger als die von anderen Konstrukten, da die Variablen innerhalb jedes einzelnen Schleifendurchgangs verändert werden. Daher ist es nicht einfach möglich eine starre Ersetzung vorzunehmen. Anstattdessen verwendet man eine Art Vollständige Induktion um die Funktion der Schleife nachzuweisen.

Um die schwächste Vorbedingung eines Ausdrucks der Form „while b { A }“ zu finden verwendet man eine Schleifeninvariante. Sie ist ein Prädikat für das

$$\{I \wedge b\} A \{I\}$$

gilt. Die Schleifeninvariante gilt also sowohl vor, während und nach der Schleife. Das Schema einer Schleife sieht dann wie folgt aus:

```
// { I } - Invariante gilt vor der Schleife
while ( b ) {
      // { I AND b} - Invariante gilt vor dem Schleifenkörper
      A;
      // { I } - Invariante gilt nach dem Schleifenkörper
}
// { I AND (NOT b) }
```

Nun gilt es nur noch folgende Schritte zu beweisen:

1. Die Invariante gilt vor Schleifeneintritt
2. $\{I \wedge b\} A \{I\}$, dass also I wirklich eine Invariante ist
3. $(I \wedge \neg b) \Rightarrow Q$, dass also bei der Terminierung auch die Nachbedingung aus der Invariante folgt.
4. Dass die Schleife terminiert (mittels Schleifenvariante/Terminierungsfunktion)

Dazu ein Beispiel, dass die Fakultät einer Variable x ausrechnet und in der Variable s ausgibt

```
i = 1;
s = 1;
// I: 0! = 0
while (i < x) {
      // I: s * (i + 1) = (i + 1)! AND i < x
      i = i + 1;
      // I: s * i = i!
      s = s * i;
      // I: s = i!
}
// I: s = i! AND x = i
```

Die Schleifenvariante ist hier (x - i). Dieser Ausdruck fällt streng monoton während der Schleifenausführung gegen 0 und ist die Abbruchbedingung.

Literatur

- Edsger W. Dijkstra: *A Discipline of Programming*, Prentice-Hall, 1976.
- David Gries: *The Science of Programming*, Springer, 1981.

FURPS

FURPS ist ein Akronym aus dem Bereich der Softwarequalität. Es entstammt aus dem Englischen und fasst Qualitätsmerkmale bei Software zusammen.

Bedeutung

Die Buchstaben von FURPS bedeuten im Einzelnen folgendes:

- **F**unctionality (*Funktionalität*)
- **U**sability (*Benutzbarkeit*)
- **R**eliability (*Zuverlässigkeit*)
- **P**erformance (*Effizienz*)
- **S**upportability (*Änderbarkeit*)

Die Übersetzungen in Klammern sind die Begriffe, wie sie auch in der ISO 9126 zu finden sind.

Das **P** kann auch für Portability (*Übertragbarkeit*) stehen. Eine traditionellere und sinngemäßere Übersetzung von **S**upportability ist *Wartbarkeit*.

Weblinks

- Furps+ [1] bei IBM

Referenzen

[1] http://www-128.ibm.com/developerworks/rational/library/4706.html

Schnittstellentest

Der **Schnittstellentest** ist ein Begriff aus der elektronischen Datenverarbeitung (EDV) und dient der Überprüfung einer korrekten Installierung zuvor noch getrennter Einheiten, oder der Neuprüfung nach einer Inbetriebnahme, wie auch der Korrespondenz zwischen verschiedenen Programmen. Hierbei werden die Daten zwischen den (Software-) Komponenten überprüft und etwaige Fehler lassen auf einen Mangel schließen.

Ein weiterer Punkt eines Schnittstellentests ist die Überprüfung der Grenzwerte. Damit sollten Korrelationen und "Überläufer" verhindert werden. Darunter versteht man, dass die Datengrößen möglicherweise nicht kompatibel sind, und der Wert der Komponente "A" größer ist als die Initalisierungsdatengröße einer anderen Komponente, die dadurch diesen zu großen Wert nicht verarbeiten kann. Dadurch kann es zu verschiedenen Fehlerbildern kommen. Hierbei kann es entweder in einem einfachen Falle zu einer Fehlermeldung, einem mit einer Zahl bezeichneten kategorisiertem Error kommen, oder bei älteren Systemen zu falschen Berechnungen. Ein Ausfall einer einzelnen Komponente könnte ebenso das gesamte System zum Absturz bringen.

Daher sind Schnittstellentests unbedingt notwendig und sollten bei keiner Funktionsüberprüfung, keinem Systemtest oder Integrationstest fehlen. Bei neueren Systemen werden diese automatisch durchgeführt, und es kommt bei einem Fehler zur Ausgabe eines Fehlercodes, bei einer Internetanbindung wird diese Fehlermeldung automatisch zur Bearbeitung weitergeleitet. In Firmen mit eigener Computerabteilung wird dieser Fehler dem zuständigen Admin gemeldet.

Lasttest (Computer)

Unter einem **Lasttest** (Lehnübersetzung von Performancetest) versteht man einen Softwaretest, mit dem eine zu erwartende, auch extreme Last auf dem laufenden System erzeugt und das Verhalten desselbigen beobachtet und untersucht wird. Dazu kann eine Simulation eingesetzt werden. Ziel dabei ist es

1. Fehler aufzudecken, die im funktional orientierten Systemtest/Integrationstest nicht gefunden wurden.
2. Erfüllung nichtfunktionaler Anforderungen, wie z. B. geforderte Antwortzeiten sowie Mengenverarbeitungen, für den Produktivbetrieb nachzuweisen.
3. Die Dimensionierung der Hardwareausstattung zu überprüfen.

Der Lasttest ist demnach dem funktionalen Test nachgelagert, d. h. das (Teil-)System muss in einem funktional stabilen Zustand sein, um überhaupt auf Lastbewältigung getestet werden zu können.

Ausprägungen

Die Last kann darin bestehen, dass Funktionen sehr schnell hintereinander ausgeführt werden, oder dass parallele Aktivitäten von virtuellen Benutzern (Multiuser, vUser) ausgeführt werden. In der Regel wird dabei direkt auf Protokollebene (Netzwerkprotokoll) gearbeitet.

Grundsätzlich lässt sich unterscheiden zwischen (1) Performancemessungen und (2) Lasttests. Performancemessungen wiederholen ausgewählte Testfälle bzw. Einzelprozesse aus dem Systemtest unter einer Grundlast: dadurch werden einzelne Funktionen auf ihre Performanzeigenschaften geprüft, d. h. sämtliche User führen den gleichen Prozess aus, wodurch die Skalierbarkeit für die Einzelfunktion(en) getestet wird. Man spricht in dem Zusammenhang auch von Transaktionen. Lasttests im engeren Sinne testen gesamte Prozessketten sowie den Prozessmix auf Performanz, d. h. die Verknüpfungen der Einzelprozesse; damit simulieren sie konkrete Vorgänge aus dem tatsächlichen Wirkbetrieb und stellen einen nicht zu unterschätzenden Schritt zur Erreichung der Wirkbetriebstauglichkeit dar. Auch hier ist die Skalierbarkeit von entscheidender Bedeutung, jedoch jetzt für den gesamten Prozessmix. Eine dabei häufige auftretende Fehlerwirkung sind Deadlocks beim Datenbankzugriff, die

sonst nur schwer testbar sind.

Wird das System bewusst über die definierte Lastgrenze hinaus beansprucht, spricht man vom *Stresstest*. Dabei sollte die Last (Anzahl der virtuellen User) schrittweise bis über die definierte Lastgrenze hinaus erhöht werden.

Damit werden folgende Fragestellungen untersucht:

- Wie ändert sich das Antwortzeitverhalten in Abhängigkeit von der Last?
- Kann mit dem System auch unter hoher Last noch akzeptabel gearbeitet werden?
- Zeigt das System undefiniertes Verhalten (z. B. Absturz)?
- Kommt es zu Dateninkonsistenz?
- Geht das System nach Rückgang der Überlast wieder in den normalen Bereich zurück?

Im Gegensatz dazu dient der *Niederlasttest*, der absichtlich mit einer geringen Intensität betrieben wird, der Untersuchung des Interaktionsverhaltens der virtuellen Benutzer und des von ihnen erzeugten Nachrichtenverkehrs auf dem System.

Einen Lasttest über einen längeren Zeitraum (z. B. 48–72 Stunden) nennt man *Dauerlasttest*; er dient in erster Linie zur Aufdeckung von Speicherlecks.

Die destruktivste Form eines Lasttests ist der *Fail-Over-Test*. Dabei geht es um die Überprüfung des Systemverhaltens unter Last bei Ausfall von Systemkomponenten. Im Idealfall werden damit Notfallszenarien überprüft, wie z. B. das rechtzeitige Zuschalten von Zusatzressourcen, um einen totalen Systemausfall zu verhindern.

Durchführung

Generierung der Testdaten

Das Testverhalten wird meist über eine Skriptsprache definiert, bei vielen Tools kann man es auch - ähnlich einem Makro - über einen Webbrowser „aufnehmen“. Dies wird zumeist über einen Proxy realisiert, welcher die Requests, etc. in die Skriptsprache übersetzt. Ein wichtiges Kriterium ist hier die Benutzerfreundlichkeit bei der Testerstellung, aber auch die Variabilität und die unterstützten Protokolle (HTTP, HTTPS etc.). Vor allem in Bereichen wo die Quantität der Daten wichtiger als deren genauer Inhalt ist, werden auch so genannte *Testdatengeneratoren* eingesetzt. Dies sind Programme, die eine große Datenmenge nach einem vorbestimmten Muster erzeugen, wobei die genaue Größe der Datenmenge in der Regel konfiguriert werden kann. Ein häufiger Anwendungsfall ist hier die Geschwindigkeitsmessung von Datenbanken.

Testlauf

Im Testlauf wird mittels des erstellten Skriptes das aufgezeichnete Verhalten (eventuell ergänzt durch zufällige Elemente bzw. zählerabhängigen Variablen) in beliebig hoher Anzahl (Virtual Users) nebenläufig ausgeführt und somit die Anwendung unter Last gesetzt. Ein wichtiges Kriterium ist hierbei die maximal erzeugbare Last, sowie die Hardwareanforderungen, die damit einhergehen.

Sinnvoll ist auch die Möglichkeit, die Lasterzeugung auf mehrere Rechner zu verteilen, welches einige Tools anbieten. Hierdurch kann der Einfluss der Netzwerk-Kapazität, sowie der Hardware-Beschränkungen des lasterzeugenden Rechners, minimiert werden. In letzter Zeit integrieren einige kommerzielle Tools die Möglichkeit, zusätzliche Lastgeneratoren in einer Cloud einzubinden.

Während des Testlaufs sammelt das Tool möglichst viele Daten. Grundsätzlich geschieht dies direkt auf der Seite der lasterzeugenden Anwendung (Antwortzeiten, Errorcodes, etc.). Einige Tools bieten auch zusätzliche Möglichkeiten, um bestimmte Web-/Datenbank-Server (z. B. IIS, Apache, MSSQL) oder Application-Server (Tomcat, etc.) zu überwachen, um direkt Zusammenhänge (z. B. hohe Antwortzeit vs. Datenbankzugriffe) zu analysieren. Es kann jedoch auch modularisiert stattfinden (Hilfsprogramme z. B. auf dem Server der zu testenden Anwendung). Wichtigstes Kriterium ist hier, dass möglichst viele Möglichkeiten zur Sammlung verschiedener Daten geboten

werden.

Auswertung

Zur Auswertung stehen meist gewisse Kennzahlen (z. B. Antwortzeit vs. Zeit, Timeouts vs. Benutzerzahl, etc.) in Logdateien bzw. zeitabhängigen Graphen zur Verfügung. Gute (meist kommerzielle) Tools bieten auch Möglichkeiten, z. B. über (Auto-)Korrelationsfunktionen, die Abhängigkeiten im Verhalten zu analysieren (z. B. hohe Antwortzeit vs. Aufruf einer bestimmten Seite, etc.).

Normen

Als Orientierung für die Planung eines Last- und Performancetests ist die DIN 66273 ein geeigneter Ausgangspunkt. Diese ist in der internationalen Norm ISO 14756 enthalten und standardisiert Begriffe sowie Mess- und Bewertungsverfahren der Leistung von komplexen DV-Systemen.

Für die Instrumentierung von Anwendungen zur Performance- bzw. Antwortzeitmessung wurde innerhalb der Open Group der Application Response Measurement (ARM) Standard verabschiedet. Dieser Standard definiert eine Programmierschnittstelle für die Programmiersprachen C und Java.

Softwaretools

Zur Durchführung von Lasttests bieten sich sog. Lasttesttools an. Im Allgemeinen wird ein Lastserver installiert, der die Last auf das zu testende System erzeugt. Die Lasttesttools können entweder selbst hergestellt werden, oder man verwendet Standardsoftware, die eine Fülle an Funktionen und Auswertungsmöglichkeiten bietet.

Kommerzielle Anbieter

- Compuware
- Empirix
- Scapa Technologies
- IBM Rational
- Hewlett-Packard
- Quotium
- Borland
- Proxy-Sniffer
- WST
- OPNET
- Moniforce
- Zott+Co GmbH - s_aturn
- Xceptance GmbH
- C1 SetCon GmbH - TAPE
- Neotys SAS (Neoload)
- GFB Softwareentwicklungsgesellschaft mbH (Q-up)
- Keynote Systems

Freie Software

- OpenSTA
- The Grinder
- BadBoy
- JMeter
- JaMonAPI - Java Performance Tuning und Scalability Measuring API
- Selenium, Testsoftware für Web-Anwendungen
- funkLoad Auf PyUnit basiertes Performance Framework für Web-Anwendungen
- soapUI (Web Services, existiert in einer freien und einer kommerziellen Version)

Weblinks

- Übersicht über freie und kommerzielle Tools mit Kurzbeschreibung [1] (englisch)
- Homepage mit Artikeln und News zum Thema Performance Testing [2] (englisch)
- Diplomarbeit über Testmethoden und Tools für Performancetests [3] (deutsch)

Literatur

- Röhrle, Jörg: *Ein regelbasierter Testdatengenerator für das Rapid Prototyping von Datenbankanwendungen*, Hamburg : Kovač, 1995
- Stefan Asböck: *Load Testing for eConfidence*. Segue Software Deutschland GmbH, Hamburg 2001.
- Mike Loukides, Gian-Paolo Musumeci: *System Performance Tuning*. 2. Auflage. O'Reilly & Associates, Sebastopol 2002.
- Sneed, Harry M: *Der Systemtest : von den Anforderungen zum Qualitätsnachweis*, 2., aktualisierte und erw. Aufl. - München : Hanser, 2009

Referenzen

[1] http://www.softwareqatest.com/qatweb1.html#LOAD

[2] http://www.loadtester.com/

[3] http://www.performance-test.de/

Sicherheitstest (Software)

Sicherheitstests sind Softwaretests, welche die Sicherheit einer Software testen. Sie stellen eine Möglichkeit zur Erhöhung der Informationssicherheit dar. Die Tests können beginnen, sobald die erste Zeile Quelltext geschrieben wurde, damit Fehler so früh wie möglich entdeckt werden.

Besonderheiten

Sicherheitstests haben eine andere Fragestellung als die meisten der übrigen, allgemeinen Tests, weil sie den Nachweis erbringen sollen, dass eine Software keine Funktionen enthält, die sie nicht enthalten soll. Daher handelt es bei Sicherheitstests meistens um so genannte Negativtests. Weiterhin sollen Sicherheitstests den Beweis erbringen, dass keine unsicheren Nebeneffekte in einem Programm vorhanden sind, weil bereits ein einzelner Fehler ausreicht, um das gesamte Programm zu kompromittieren. Die Formulierung eines Sicherheitstests ist in der Regel problematisch, weil die Muster der Schwachstellen nicht präzise genug definierbar sind. Sicherheitstests können allerdings nie den Beweis erbringen, dass eine Software zu einhundert Prozent sicher ist.

Ziel

Das Ziel von Sicherheitstests ist das Auffinden von sicherheitskritischen Schwachstellen in Programmen. Auf diese Weise wird versucht, die Abwesenheit von Schwachstellen in einer Software zu erbringen. Wichtig ist, dass sich die Tests über das gesamte Programm erstrecken, weil, wie oben bereits angeführt, ein einzelner Fehler ausreichend ist, um das gesamte Programm zu kompromittieren.

Programmierfehler

Sicherheitstests werden entwickelt, um alle Fehler innerhalb eines Programms zu finden. Die meisten dieser sicherheitskritischen Fehler lassen sich auf wenige Ursachen zurückführen. Die häufigste Ursache sind Programmierfehler. Zur Kategorisierung dieser Programmierfehler existieren verschiedene Schemata, um die einzelnen Fehler sauber von einander abzugrenzen. Die häufigsten immer wieder genannten Kategorien sind die folgenden:

- Cross-Site Scripting
- SQL-Injection
- Failing to Protect Data
- Buffer Overflow
- Improper Error Handling
- Information Leakage
- Integer Over- und Underflow
- unsichere Handhabung von File-Links
- Race Conditions
- Formatstring-Angriffe
- etc.

Im Rahmen des Open Web Application Security Project (OWASP) wurde eine Applikation (WebGoat) entwickelt, die dem Nutzer hilft, die verschiedenen Programmierfehler zu verstehen und nachzuvollziehen. Diese Applikation ist dadurch auch eine gute Ausgangsbasis für die Entwicklung von Sicherheitstests.

Werkzeuge

Bei den nachfolgend aufgelisteten Werkzeugen handelt es sich zumeist um Sicherheits-Testwerkzeuge. Einige der Werkzeuge sind Open Source.

- Jlint (Java)
- PQL/bddbddb (Java)
- Nikto (Perl)
- RATS (C, C++)
- WebScarab
- ITS4 (C, C++)
- Virtual Forge CodeProfiler (SAP ABAP)
- Fortify SCA Suite (Java, C, C++, etc.)
- Beyond Security beSTORM (Fuzzing Tools für über 140 verschiedene Protokolle, z.B. IPv4, IPv6, SIP, DHCP)
- Microsoft Security Development Lifecycle (SDL) (8 frei verfügbare Anwendungen)

Links

- Open Web Application Security Project [1]
- Bundesamt für Sicherheit in der Informationstechnik (BSI): Maßnahmenkatalog und Best Practices für die Sicherheit von Webanwendungen [2] (PDF-Datei; 989,71 kB)
- Sicherheitsrelevante Programmierfehler [3] (PDF-Datei; 1,09 MB)
- MSDN Security Developer Center Security Developer Center [4] im MSDN

Literatur

- Michael Howard, David LeBlanc: *Writing secure code. Practical strategies and proven techniques for building secure applications in a networked world.* 2nd edition. Microsoft Press, Redmond, WA 2003, ISBN 0-7356-1722-8

Referenzen

[1] http://www.owasp.org
[2] https://www.bsi.bund.de/cae/servlet/contentblob/476464/publicationFile/30642/WebSec_pdf.pdf
[3] http://www.suse.de/~thomas/papers/SecProg/Sicherheitsrelevante%20Programmierfehler.pdf
[4] http://msdn.microsoft.com/de-de/security/

Äquivalenzklassentest

Ein **Äquivalenzklassentest** dient der Qualitätsprüfung von Software.

Ziel der Äquivalenzklassenbildung ist es, Äquivalenzklassen zu bilden und so eine hohe Fehlerentdeckungsrate mit einer möglichst geringen Anzahl von Testfällen zu erreichen. Die Äquivalenzklassen sind also bezüglich Ein- und Ausgabedaten ähnliche Klassen bzw. Objekte, bei denen erwartet wird, dass sie sich gleichartig verhalten. Bspw. im Programm zur Verwaltung eines Fuhrparks sind äquivalente Klassen Fahrzeuge (Ferrari - BMW nicht jedoch Ferrari - Mitarbeiter). Das Wesen der Äquivalenzklassenbildung besteht darin, die gesamten Eingabedaten und Ausgabedaten eines Programms in Gruppen von Äquivalenzklassen zu unterteilen, so dass man annehmen kann, dass mit jedem beliebigen Objekt einer Klasse die gleichen Fehler wie mit jedem anderen Objekt dieser (Äquivalenz-)Klasse gefunden werden (Bspw. Ferrari ENZO - BMW M3). Die Bildung von Testfällen zu Äquivalenzklassen folgt dieser Abfolge:

- Analyse und Spezifikation der Eingabedaten, der Ausgabedaten und der Bedingungen gemäß den Spezifikationen
- Bildung der Äquivalenzklassen durch Klassifizierung der Wertebereiche für Ein- und Ausgabedaten
- Bestimmung der Testfälle durch Werteauswahl für jede Äquivalenzklasse

Die erstellten Testfälle gelten somit für alle Objekte der erstellten Äquivalenzklasse, sodass nicht für jede Ausprägung ein eigener Testfall erstellt werden muss.

Es werden zwei Arten von Äquivalenzklassen unterschieden:

- *gültige Äquivalenzklassen*
- *ungültige Äquivalenzklassen*

Bei gültigen Äquivalenzklassen werden gültige Eingabedaten, bei ungültigen Äquivalenzklassen ungültige Eingabedaten verwendet.

Äquivalenzklassen werden allgemein unter logischen Gesichtspunkten erstellt, indem insbesondere auf die Gleichartigkeit der Klassen- bzw. Objekteigenschaften sowie deren Ermittlung geachtet wird. Kandidaten für die Bildung von Äquivalenzklassen sind in der Welt der objektorientierten Programmierung insbesondere Kind- und Superklassen. Bei abstrakten Klassen ist auf die unterschiedliche Ausimplementierung der vererbten Methoden/Prozeduren insbesondere bezüglich Ein- und Ausgabeparameter zu achten. Stark unterschiedliche Implementierungen bei Kindklassen einer abstrakten Klasse KFZ (z. B.: LKW; PKW) können für die Bildung von Äquivalenzklassen nicht geeignet sein (bspw. Methode: ermittleGueltigeAnzahlAchsen() → PKW 2; LKW: 3 (nur Zugmaschine)).

Ein Beispiel für eine Grenzwertanalyse nach gültigen und ungültigen Äquivalenzklassen:

Die gültigen Werte einer Ein/Ausgabe bei gebildeten Äquivalenzklassen liegen zwischen 100 und 1000. Es wird nun ein Testfall definiert, welcher verifiziert, dass:

- die Ein/Ausgabe minus unendlich bis 99 als ungültig zurückgewiesen wird, (ungültige Äquivalenzklassen: - unendlich bis inklusive 99),
- die Ein/Ausgabe von 100 bis 1000 als gültig akzeptiert (gültige Äquivalenzklassen: 100 bis inklusive 1000), und
- die Ein/Ausgabe von 1001 bis unendlich als ungültig zurückgewiesen wird (ungültige Äquivalenzklassen: 1001 bis inklusive unendlich)

Smoke testing

Smoke testing ist ein Begriff aus dem Englischen, gebräuchlich im handwerklichen Bereich (z. B. in der Klempnerei, Elektronik oder beim Bau von Holzblasinstrumenten) wie auch in der Softwareentwicklung. Es bezeichnet den ersten Probelauf nach einer Reparatur oder der ersten Implementierung eines neuen Algorithmus um sicherzugehen, dass das Gerät oder die Programmfunktion nicht schon ansatzweise fehlschlägt. Nachdem der *smoke test* zeigt, dass die Rohre nicht lecken, die Stromkreise nicht durchbrennen bzw. das Programm nicht gleich abstürzt, kann die Apparatur eingehender geprüft werden.

- Beim Klempnern wird Rauch durch neu gelegte Rohre geführt um evtl. Lecks zu finden, bevor Wasser in die Rohre gelassen wird.
- Bei der Reparatur von Holzblasinstrumenten wurde früher ein Ende des Instruments verstopft und auch hier Rauch hineingeblasen, um Löcher oder Risse im Material zu finden.
- In der Elektronik ist der *smoke test* das erste Mal, dass der Stromkreis an eine Spannungsquelle angeschlossen wird. Dies kann manchmal Rauch produzieren, wenn ernsthafte Fehler bei der Verkabelung gemacht wurden.
- In der Programmierung bezeichnet *smoke testing* den ersten grundlegenden Probelauf einer Software, der simple Probleme offenlegen soll, die ernst genug sind um das Programm nochmals zu überarbeiten und ein mögliches Release zu vereiteln.

Smoketest bei sanitären Anlagen

Smoketests im Haushalt, aber auch in chemischen Fabrikanlagen, werden benutzt, um Stellen zu finden, an denen Flüssigkeit austreten kann. In Abflusssystemen wird dies gemacht, um zu sehen, wo Grund- oder Regenwasser in die Rohre gelangen kann. Der ungiftige Rauch tritt an Leckstellen aus. Smoke testing ist besonders nützlich wo eine komplette, permanente Versiegelung des gesamten Systems unerwünscht oder unsinnig ist, beispielsweise bei belüfteten Abwassersystemen.

Beim Smoketest von Sanitäranlagen ist es hilfreich, den zu prüfenden Teil des Systems zu isolieren, zum Beispiel mit Sandsäcken, die in den Kanalisationsschacht hineingelassen und dann in Position gebracht werden. Eine komplette Blockierung aller Rohre allerdings könnte einen Stau von verbliebenem Wasser verursachen, welches wiederum den Rauch am Austreten hindern kann.

Große Gebläse werden auf offene Kanalschächte gelegt, jeweils an beiden Enden des zu prüfenden Rohrabschnittes. Dann wird Rauch, entweder von einer Rauchbombe oder flüssiger Rauch, über die Gebläse in die Rohre gesaugt. Wenn alle Siphons in den angeschlossenen Haushalten funktionieren, sollte der Rauch über andere Kanaldeckel oder über die Belüftungsschächte der Abwasserrohre entweichen. Defekte Abwasserrohre oder trockene Siphons aber lassen den Rauch ins Haus eintreten.

Wenn Rauchwolken woanders als in den Abwasserbelüftungen der Häuser oder an den Rändern von Kanaldeckeln auftauchen, werden diese Stellen markiert, vermessen, notiert und ggf. fotografiert.

Überprüfungen bei der Reparatur von Holzblasinstrumenten

Beim smoke test wird ein Ende des Blasinstruments verschlossen und Rauch, üblicherweise Tabakrauch, in das andere hineingeblasen. Entweichender Rauch ist ein sicheres Zeichen für unsauber gesetzte Klappen oder schlechte Verbindungsstücke. Nach diesem Test wird das Instrument gereinigt, um Rauchrückstände wie Nikotin zu entfernen. Natürlich ist dieser Brauch gesundheitsschädigend für den Reparateur und den Spieler, und seit Anfang des Jahrhunderts wird die Methode kaum noch benutzt.

Tests in der Elektronik und im Ingenieurshandwerk

Die Begriffe *smoke test* oder *power on test* werden in der englischsprachigen Elektronik verwendet, wenn ein sich in der Entwicklung befindender Stromkreis zum ersten Mal an eine Stromquelle angeschlossen wird. Dies kann durchaus geschehen bevor die Arbeit fertiggestellt ist, einfach um sicherzugehen dass keine größeren Fehler gemacht wurden, die alle weitere Arbeit nutzlos machen würden. Bei größeren Leistungen (bei SMD-Komponenten können wenige Watt genügen) kann in einem fehlerhaften Stromkreis tatsächlich Rauch entstehen, z. B. wenn Widerstände durchbrennen.

Smoke testing in der Softwareentwicklung

Smoke testing wird von den Entwicklern vor einem ersten Release durchgeführt, oder auch von Testern, bevor sie einen Build für weitere Tests akzeptieren.

Im Allgemeinen besteht ein *smoke test* aus einer Sammlung von Tests, die neue oder reparierte Software durchlaufen muss. Diese Tests werden manchmal schon automatisch durchgeführt, z. B. vom Entwicklungssystem welches auch die Software kompiliert. In diesem Zusammenhang ist *smoke testing* also die Überprüfung von Änderungen im Quellcode bevor diese in die offiziellen Repositories eingehen.

Ein *smoke test*, etwas formeller auch *build verification test*, ist eine oberflächliche Überprüfung der Programmfunktionen. Der Tester geht durch das ganze Programm und probiert Dinge wie „Kann man diese Funktion überhaupt benutzen?“, „Öffnet dies hier ein Fenster?“, „Tun alle Buttons auf diesem Fenster etwas?“. Sobald eine fundamentale Frage wie diese hier mit „nein“ beantwortet werden kann, muss nicht weiter getestet werden. Das Programm ist so unfertig oder kaputt, dass ein weiteres Testen unsinnig wäre. Diese protokollierten Tests können manuell oder automatisiert stattfinden. Die automatischen Testvorgänge werden meist schon bei der Kompilierung durchgeführt.

Testgetriebene Entwicklung

Testgetriebene Entwicklung (auch *testgesteuerte Programmierung*, engl. *test first development* oder *test-driven development* (*TDD*)) ist eine Methode, die häufig bei der agilen Entwicklung von Computerprogrammen eingesetzt wird. Bei der testgetriebenen Entwicklung erstellt der Programmierer Software-Tests konsequent vor den zu testenden Komponenten. Die dazu erstellten Unit-Tests sind Grey-Box-Tests statt klassischer White-Box-Tests.

Gründe für die Einführung einer testgetriebenen Entwicklung

Nach klassischer Vorgehensweise, beispielsweise nach dem Wasserfall- oder dem V-Modell, werden Tests parallel zum und unabhängig vom zu testenden System entwickelt oder sogar nach ihm. Dies führt oft dazu, dass nicht die gewünschte und erforderliche Testabdeckung erzielt wird. Mögliche Gründe dafür sind unter anderem:

- Fehlende oder mangelnde Testbarkeit des Systems (monolithisch, Nutzung von Fremdkomponenten, …).
- Verbot der Investition in nicht-funktionale Programmteile seitens der Unternehmensführung. („Arbeit, von der man später im Programm nichts sieht, ist vergeudet.“)
- Erstellung von Tests unter Zeitdruck.
- Nachlässigkeit und mangelnde Disziplin der Programmierer bei der Testerstellung.

Ein weiterer Nachteil klassischer White-Box-Tests ist, dass der Entwickler das zu testende System und seine Eigenheiten selbst kennt und dadurch aus Betriebsblindheit unversehens „um Fehler herum“ testet.

Die Methode der testgetriebenen Entwicklung versucht den Gründen für eine nicht ausreichende Testabdeckung und einigen Nachteilen der White-Box-Tests entgegenzuwirken.

Vorgehensweise

Bei der testgetriebenen Entwicklung ist zu unterscheiden zwischen dem Testen im Kleinen (Unit-Tests) und dem Testen im Großen (Systemtests, Akzeptanztests), wobei Becks Methode auf Unit-Tests ausgelegt ist.

Testgetriebene Entwicklung mit Unit-Tests

Unit-Tests und mit ihnen getestete Units werden stets parallel entwickelt. Die eigentliche Programmierung erfolgt in kleinen und wiederholten Mikroiterationen. Eine solche Iteration, die nur wenige Minuten dauern sollte, hat drei Hauptteile:

1. Schreibe Tests für das erwünschte fehlerfreie Verhalten, für schon bekannte Fehlschläge oder für das nächste Teilstück an Funktionalität, das neu implementiert werden soll. Diese Tests werden vom bestehenden Programmcode erst einmal *nicht* erfüllt bzw. es gibt diesen noch gar nicht.
2. Ändere/schreibe den Programmcode mit möglichst wenig Aufwand, bis nach dem anschließend angestoßenen Testdurchlauf alle Tests bestanden werden.
3. Räume dann im Code auf (Refactoring): Entferne Wiederholungen (Code-Duplizierung), abstrahiere wo nötig, richte ihn nach den verbindlichen Code-Konventionen aus etc. Natürlich wieder mit abschließendem Testen. Ziel des Aufräumens ist es, den Code *schlicht* und *verständlich* zu machen.

Diese drei Schritte werden so lange wiederholt, bis die gewünschte Funktionalität erreicht oder der bekannte Fehler bereinigt ist und dem Entwickler keine sinnvollen weiteren Tests mehr einfallen, die vielleicht noch scheitern könnten. Die so behandelte programmtechnische Einheit (Unit) wird dann als (vorerst) fertig angesehen. Die gemeinsam mit ihr geschaffenen Tests bleiben erhalten, um auch zukünftige Umsetzungen daraufhin testen zu können, ob das erwünschte Verhalten fortbesteht.

Die konsequente Befolgung dieser Vorgehensweise läuft auf evolutionären Entwurf hinaus, weil die ständige Änderung die Weiterentwicklung eines Systems in den Vordergrund rückt.

Da der einzelne Unit-Test sowohl Züge eines White-Box-Tests als auch eines Black-Box-Tests aufweist, bezeichnet man ihn auch als Grey-Box-Test.

Testgetriebene Entwicklung mit Systemtests

Systemtests werden immer vor dem System selbst entwickelt oder doch wenigstens spezifiziert. Aufgabe der Systementwicklung ist bei *testgetriebener Entwicklung* nicht mehr, wie klassisch, schriftlich formulierte Anforderungen zu erfüllen, sondern spezifizierte Systemtests zu bestehen.

Gemeinsamkeiten zwischen Testgetriebener Entwicklung mit Systemtests und Unit-Tests

Bei beiden Arten von Tests wird eine möglichst vollständige Testautomatisierung angestrebt. Für *testgetriebene Entwicklung* müssen alle Tests einfach („per Knopfdruck“) und möglichst schnell ausgeführt werden können. Für Unit-Tests bedeutet das eine Dauer von wenigen Sekunden, für Systemtests von maximal einigen Minuten, bzw nur in Ausnahmen länger.

Die großen Vorzüge der testgetriebenen Methodik gegenüber der klassischen sind:

- Man hat eine triviale Metrik für die Erfüllung der Anforderungen: die Tests werden bestanden oder nicht.
- Das Refactoring, also das Aufräumen im Code, führt zu weniger Fehlern; weil man dabei in kleinen Schritten vorgeht und stets entlang bestandener Tests, entstehen dabei wenige neue, und sie sind besser lokalisierbar.
- Weil einfach und ohne großen Zeitaufwand getestet werden kann, arbeiten die Programmierer die meiste Zeit an einem korrekten System und also mit Zutrauen und konzentriert auf die aktuelle Teilaufgabe hin. (Keine „Durchquerung der Wüste“, kein „Alles hängt mit allem zusammen“)
- Der Bestand an Unit-Tests dokumentiert das System zugleich. Man erzeugt nämlich zugleich eine „ausführbare Spezifikation“ – was das Softwaresystem leisten soll, liegt in Form sowohl lesbarer wie auch jederzeit lauffähiger

Tests vor.

- Ein testgetriebenes Vorgehen führt in der Tendenz zu Programmcode, der stärker modularisiert ist sowie leichter zu ändern und zu erweitern. Denn aus Entwicklersicht entsteht dabei das geplante System aus kleinen Arbeitseinheiten, die unabhängig geschrieben und getestet, aber erst später integriert werden. Die korrespondierenden Softwareeinheiten (Klassen, Module, …) werden damit kleiner, spezifischer, ihre Kopplung wird lockerer und ihre Schnittstellen schlichter. Nutzt man auch Mock-Objekte, zwingt dies ebenfalls dazu, Abhängigkeitsstrukturen einfach zu halten, weil sonst der dabei essentielle schnelle und umstandslose Austausch von Modulen für Test und für Produktionscode nicht möglich wäre.

Einsatzgebiete

Testgetriebene Entwicklung ist wesentlicher Bestandteil des Extreme Programming (XP) und anderer agiler Methoden. Auch außerhalb dieser ist sie anzutreffen, häufig in Verbindung mit der Paarprogrammierung .

Werkzeuge

Die testgetriebene Entwicklung braucht vordringlich

- ein Werkzeug zur Build-Automatisierung wie etwa CruiseControl
- einen Rahmen und ein Werkzeug zu Testentwicklung und -automatisierung,

damit die Iterationen schnell und unkompliziert durchlaufen werden können.

Bei der Java-Entwicklung kommen dafür meist Ant oder Maven und JUnit zum Einsatz. Für die meisten anderen Programmiersprachen existieren ähnliche Werkzeuge, wie z. B. für PHP PHPUnit.

Für komplexe Systeme müssen mehrere Teilkomponenten unabhängig voneinander entwickelt werden und es finden dazu auch noch Fremdkomponenten Verwendung, etwa ein Datenbanksystem zwecks persistenter Datenhaltung. Die korrekte Zusammenarbeit und Funktion der Komponenten im System muss dann auch getestet werden. Um nun die Einzelkomponenten dabei separat testen zu können, die doch aber zu ihrer korrekten Funktion wesentlich von anderen Komponenten abhängen, verwendet man Mock-Objekte als deren Stellvertreter. Die Mock-Objekte ersetzen und simulieren im Test die benötigten anderen Komponenten in einer Weise, die der Tester ihnen einprogrammiert.

Ein Werkzeug für Akzeptanztests und Systemtests ist beispielsweise Framework for Integrated Test. Eine beliebte FIT-Variante ist *Fitnesse*, ein Wiki-Server mit integrierter Testerstellungs- und Testausführungsumgebung.

Kritik

Konsequenz ist nötig

Auch die Methode der *testgetriebenen Entwicklung* kann falsch eingesetzt werden und dann scheitern. Programmierern, die noch keine Erfahrung dabei besitzen, erscheint sie manchmal schwierig oder gar unmöglich. Sie fragen sich, wie man etwas testen soll, das doch noch gar nicht vorhanden ist. Auswirkung kann sein, dass sie die Prinzipien dieser Methode vernachlässigen, was insbesondere bei agilen Methoden wie dem Extreme Programming Schwierigkeiten beim Entwicklungsprozess oder sogar dessen Zusammenbruch zur Folge haben kann. Ohne ausreichende Unit-Tests wird keine ausreichende Testabdeckung für das Refactoring und die gewünschte Qualität erreicht. Dem kann man mit Paarprogrammierung und Schulung entgegenwirken.

Kein Ersatz für Systemtests

Auch diese stark auf Tests setzende Art der Programmierung kann nicht jeden Fehler aufdecken, insbesondere nicht Fehler, die programmextern entstehen: Timingfehler wie Thread-Deadlocks, Fehler bei Schnittstellenkommunikation usw. Ebenfalls kann es an der nötigen Testabdeckung mangeln, wenn nicht alle potentiellen Eingaben einer Funktion getestet werden können. Dies ist etwa der Fall, wenn etwa die Eingabe aus sehr vielen Einzeldaten besteht; dann kann aufwandshalber nicht mehr jede mögliche Kombination von Eingabedaten getestet werden. Fehler bei Benutzungsoberfächen sind schlecht testbar, weil natürlich kein automatisierter, prüfender „Testblick“ auf die Folge der Bildschirmdarstellungen und die Verständlichkeit von Darstellung und Textelementen möglich ist. Um solche Fehler zu finden, sind Integrationstests und Systemtests erforderlich.

Literatur

- Kent Beck: *Test Driven Development by Example*, Addison-Wesley Verlag, ISBN 0-321-14653-0
- Johannes Link: Softwaretests mit JUnit – Techniken der testgetriebenen Entwicklung [1], ISBN 3-89864-325-5
- Frank Westphal: *Testgetriebene Entwicklung mit JUnit und FIT* [2], ISBN 3-89864-220-8
- Klaus Meffert: *JUnit Profi-Tipps – Software erfolgreich testen* [3], ISBN 3-935042-76-0
- Lasse Koskela: *Test driven: TDD and Acceptance TDD for Java developers* [4], ISBN 1-932394-85-0

Weblinks

- JUnit, Framework zur testgetriebenen Entwicklung in Java (Englisch) [5]
- TDD Community (Englisch) [6]
- Einführende Artikel [7]
- Bessere Kodequalität und Kosteneinsparungen durch Einheittests/Modultests [8]
- Eine kurze Einführung in die Prinzipien der Test gesteuerten Programmierung [2]
- Fitnesse [9]
- Framework for Integrated Tests [10]
- EMOS Framework Open Source Code Library für Funktionale Softwaretests mit Winrunner [11]
- Test Driven Development in .NET [12] example for TDD in Visual Studio and .NET including WatiN test framework for web applications
- Video-Crashkurs 'Malen nach Zahlen': Test Driven Development Einführung für Software-Entwickler (Deutsch) [13]

Referenzen

[1] http://stmj.developertests.de
[2] http://www.frankwestphal.de/TestgetriebeneEntwicklungmitJUnitundFIT.html
[3] http://www.junit-buch.de
[4] http://manning.com/koskela
[5] http://junit.org
[6] http://testdriven.com
[7] http://www.frankwestphal.de/TestgetriebeneEntwicklung.html
[8] http://www.verifysoft.de/de_unittest.html
[9] http://www.fitnesse.org
[10] http://fit.c2.com
[11] http://emos-framework.sourceforge.net/
[12] http://www.software-architects.com/TechnicalArticles/TestDrivenDevelopment/tabid/83/language/en-US/Default.aspx
[13] http://www.makandra.de/malennachzahlen

Regressionstest

Unter einem **Regressionstest** (v. lat. regredior, regressus sum = zurückschreiten) versteht man in der Softwaretechnik die Wiederholung aller oder einer Teilmenge aller Testfälle, um *Nebenwirkungen* von *Modifikationen* in bereits getesteten Teilen der Software aufzuspüren. Solche Modifikationen entstehen regelmäßig z. B. aufgrund der Pflege, Änderung und Korrektur von Software. Der Regressionstest gehört zu den *dynamischen Testtechniken*.

Aufgrund des Wiederholungscharakters und der Häufigkeit dieser Wiederholungen ist es sinnvoll, wenn für Regressionstests Testautomatisierung zum Einsatz kommt.

In der Praxis steht der Begriff des Regressionstests für die reine Wiederholung von Testfällen. Die Testfälle selbst müssen spezifiziert und mit einem Soll-Ergebnis versehen sein, welches mit dem Ist-Ergebnis eines Testfalls verglichen wird. Ein direkter Bezug auf die Ergebnisse eines vorherigen Testdurchlaufs findet nicht statt.

Im Gegensatz dazu ordnet Liggesmeyer (Lit.: Liggesmeyer) den Regressionstest in die Gruppe der Diversifizierenden Tests ein. Dadurch wird im Unterschied zu Funktionsorientierten Testtechniken die Korrektheit der Testergebnisse nicht anhand der *Spezifikation* entschieden, sondern durch Vergleich der *Ausgaben* der aktuellen Version mit den Ausgaben des Vorgängers. Ein *Testfall* gilt beim Regressionstest als erfolgreich absolviert, wenn die Ausgaben identisch sind.

Regressionstests in Echtzeitsystemen

Der Regressionstest stellt insbesondere bei Echtzeitsystemen ein wesentliches Problem dar, da in diesen Systemen eine Wiederholung des Tests streng genommen nicht möglich ist. Zum einen führen typischerweise bereits geringfügige Änderungen an der Hardware des Systems zu einem veränderten Verhalten, andererseits sind hier im allgemeinen manuelle Eingriffe (zum Beispiel bei Telefonanlagen oder Flugüberwachungssystemen) notwendig, die wegen des menschlichen Zeitverhaltens nicht „regressionstestgerecht“ erfolgen können. Eine Lösung dieses Problems liegt in der Implementierung eines automatischen Testsystems. Der Aufwand hierfür wird jedoch aus folgenden Gründen meistens gescheut:

1. das automatische Testsystem muss alle Funktionen des Prüflings abdecken
2. das automatische Testsystem muss parallel zum Prüfling entwickelt werden
3. das automatische Testsystem muss parallel zum Prüfling angepasst werden
4. eine Hardware-Änderung führt zu einem Neu-Aufsetzen der Testergebnisse, gegen die verglichen werden soll.

Literatur

- Peter Liggesmeyer: *Software-Qualität: Testen, Analysieren und Verifizieren von Software*, Spektrum, Akademischer Verlag, Heidelberg, Berlin, 2002, ISBN 3-8274-1118-1

Review (Softwaretest)

Mit dem **Review** werden Arbeitsergebnisse der Softwareentwicklung manuell geprüft. Jedes Arbeitsergebnis kann einer Durchsicht durch eine andere Person unterzogen werden. Das Review ist eine statische Testmethode und gehört in die Kategorie der analytischen Qualitätssicherungsmaßnahmen.

In Anlehnung an die IEEE-Norm 729 ist das Review ein mehr oder weniger formal geplanter und strukturierter Analyse- und Bewertungsprozess, in dem Projektergebnisse einem Team von Gutachtern präsentiert und von diesem kommentiert oder genehmigt werden.

Der untersuchte Gegenstand eines Reviews kann verschieden sein. Es wird vor allem zwischen einem **Code-Review** (Quelltext) und einem **Architektur-Review** (Softwarearchitektur) unterschieden.

Beim Codereview wird ein Programmabschnitt, nach oder während der Entwicklung, von einem oder mehreren Gutachtern korrekturgelesen, um mögliche Fehler, Vereinfachungen oder Testfälle zu finden. Dabei kann der Gutachter selbst ein Software-Entwickler sein. Für unerfahrene Entwickler bietet der Codereview durch einen erfahrenen Programmierer eine gute Möglichkeit, sich schnell und praxisorientiert weiterzubilden.

Nutzen von Reviews

Fehler, die im Review auffallen, können häufig bedeutend kostengünstiger behoben werden, als wenn diese erst während der Testdurchführung gefunden werden. Untersuchungen haben gezeigt, dass mit Reviews 85% der Fehler bei einem Aufwand von 25% gefunden werden können.

Dabei laufen verschiedene Qualitätsprozesse ab:

- Der Programmierer entdeckt selbst eine Verbesserungsmöglichkeit.
- Der Rezensent stellt Verständnisfragen und der Programmierer kann die Antwort auf diese Fragen als Kommentare in den Programmcode einbauen und so die Verständlichkeit und die Dokumentation verbessern.
- Der Rezensent entdeckt Verbesserungsmöglichkeiten und empfiehlt diese dem Programmierer.

Zu den typischen Schwächen, die mit Reviews entdeckt werden können, gehören:

- Abweichungen von Standards, z. B. Verletzung von Namenskonventionen
- Fehler gegenüber (oder auch in) den Anforderungen
- Fehler im Design
- Unzureichende Wartbarkeit
- Falsche Schnittstellenspezifikation

Resultate aus Code-Reviews sind verbesserter Code; Beispiele: Effizienz; Robustheit; Wartbarkeit, z. B. durch verbesserte Code-Kommentare.

Reviewprozess

Ein typisches Review besteht aus folgenden Hauptphasen:

- **Planung**
  - Auswahl der beteiligten Personen und Besetzung der Rollen
  - Festlegung der Vor- und Nachbedingungen
- **Kick-Off**
  - Verteilung der Dokumente
  - Erläuterung der Ziele und des Prozesses
  - Prüfung der Vorbedingungen
- **Individuelle Vorbereitung**

  - Notierung von potentiellen Fehlern, Fragen und Kommentaren
- **Reviewsitzung**
  - Diskussion und Protokollierung der Ergebnisse
  - Empfehlungen geben oder Entscheidungen über Fehler treffen
- **Überarbeitung (rework)**
  - Beheben der gefundenen Fehler, typischerweise durch den Autor
- **Nachbearbeitung (follow up)**
  - Überprüfung der Überarbeitung
  - Prüfung von Testendekriterien

Reviewarten

Reviews variieren zwischen sehr informell (unstrukturiert) und sehr formal (d. h. tief strukturiert und geregelt). Die Art und Weise, wie ein Review durchgeführt wird, ist abhängig von den festgelegten Zielen des Reviews (z. B. das Finden von Fehlern, dem Erwerb von Verständnis oder einer Diskussion mit Entscheidung durch Konsens).

Nach Norm IEEE 1028 gibt es die folgenden vier Reviewarten:

- **Technisches Review**
  - Fachliche Prüfung eines wesentlichen Dokumentes (z. B. Architekturentwurf) auf Übereinstimmung mit Spezifikation
  - Zweck: Diskussion, Entscheidungen treffen, Alternativen bewerten, Fehler finden, technische Probleme lösen
- **Informelles Review**
  - Entspricht inhaltlich dem technischen Review, es soll ihm gegenüber aber Zeit gespart werden und daher wird es als nicht formaler Prozess durchgeführt.
  - Das Informelle Review ist nicht im IEEE Standard für Software Reviews enthalten. Demnach wird hier auf strukturierte Protokollierung/Dokumentierung verzichtet.
  - Es ist eine einfache Art eines Reviews, bei dem meistens „Gegenlesen unter Kollegen“ durchgeführt wird.
  - Inhaltlich können dieser Art folgende, praxisbezogene Reviewarten zugeordnet werden (Begriffe je nach Firmenkultur unterschiedlich):
    - Pulttest (Programm-Autor spielt den Code anhand von einfachen Testfällen gedanklich durch.)
    - Peer Rating (Gutachten, das von gleichgestellten Programmierern anonym über ein Programm erstellt wird.)
    - Stellungnahmeverfahren (Autor verteilt Arbeitsergebnis an ausgewählte Gutachter zur Beurteilung.)
- **Walkthrough**
  - Diskussion von Szenarien, Probeläufen und Alternativen im Kreis gleichgestellter Mitarbeiter mit möglichst niedrig gehaltenem Aufwand
  - Zweck: Lernen, Verständnis erzielen und Fehler finden
- **Inspektion**
  - Formalste Reviewtechnik mit einem dokumentierten Vorgehen nach IEEE 610, IEEE 1028.
  - Zweck: Sichtüberprüfung von Dokumenten um Mängel zu finden (z. B. Nichteinhaltung von Entwicklungsstandards, Nicht-Konformität gegenüber Spezifikationen, usw.).

Erfolgsfaktoren

Damit Reviews erfolgreich durchgeführt werden, müssen verschiedene Bedingungen erfüllt sein:

- Definition von klaren Zielen
- Auswahl von geeigneten Personen
- Konstruktive Kritikfähigkeit (gefundene Fehler werden objektiv zur Sprache gebracht und positiv aufgenommen)
- Psychologische Aspekte (insbesondere Sicherstellung einer positiven Erfahrung für den Autor)
- Auswahl der geeigneten Reviewtechnik
- Unterstützung Reviewprozess durch das Management
- Existenz einer Kultur von Lernen und Prozessverbesserung

Anforderungen an Rezensenten

- Er darf den Code nicht selbst geschrieben haben.
- Er muss Taktgefühl haben: Codereviews können für den Programmierer unangenehm sein, da er den Eindruck bekommen könnte, der eigene Code werde kritisiert. Wenn der Rezensent nicht taktvoll vorgeht, wird Widerstand und Ablehnung gegen die Durchsicht der Quelltexte aufgebaut.

Reviews als Philosophie

Kontinuierliches Inspizieren des Quelltextes wie bei der Paarprogrammierung ist auch eine der Methoden des Extreme Programming. Die im Extreme Programming (XP) eingesetzte Paarprogrammierung wird auch als „Codereview während der Entwicklung“ bezeichnet.

Ein öffentliches Review ist ebenfalls eine Motivation der Open Source-Software.

Online-Software Repositories wie CVS erlauben es Gruppen von Individuen, gemeinschaftlich Codereviews durchzuführen, und damit Sicherheit und Qualität des Programmcodes zu verbessern.

Literatur

- M. E. Fagan: Design and code inspections to reduce errors in program development, 1976 [1]
- Hansruedi Tremp: *IT-Systeme prüfen*. Compendio Verlag. 2. Auflage 2007. ISBN: 978-3-7155-9304-3

Referenzen

[1] http://www.research.ibm.com/journal/sj/382/fagan.pdf

Statische Code-Analyse

Statische Code-Analyse oder kurz **statische Analyse** ist ein statisches Software-Testverfahren. Der Quelltext wird hierbei einer Reihe formaler Prüfungen unterzogen, bei denen bestimmte Sorten von Fehlern entdeckt werden können, noch bevor die entsprechende Software (z. B. im Modultest) ausgeführt wird. Die Methodik gehört zu den falsifizierenden Verfahren, d. h. es wird die Anwesenheit von Fehlern bestimmt.

In Anlehnung an das klassische Programm *Lint*, wird der Vorgang auch als *linten* (engl. *linting*) bezeichnet.

Methodischer Zusammenhang

Einordnung

Im Rahmen der Softwaretestverfahren ist die Statische Code-Analyse den White-Box-Test-Verfahren zuzuordnen (man benötigt den Quellcode). Die Analyse kann durch manuelle Inspektion erfolgen, aber auch automatisch durch ein Programm. Man spricht dann von *statischer* Analyse, da die zu testende Software in Form von Algorithmen und Daten in ihrer Formulierung und Beschaffenheit (*statisch*) dem Prüfer (oder Werkzeug) vor*liegt*. Das ist in etwa vergleichbar mit statischen Berechnungen im Ingenieurbau.

Abgrenzung

Vorläufer der statischen Analyse sind die Prüfverfahren der normierten Programmierung und die Werkzeuge zur Erkennung von Code-Mustern, die sogenannten Style Checker.

Dynamische Code-Analyse setzt im Gegensatz zur statischen Analyse ein laufendes Programm voraus.

Verfahren

Neben dem gewissenhaften Studium von Quelltext durch Entwickler ist es möglich, viele inhaltliche Fehler werkzeuggestützt oder automatisch zu erkennen. Die Bandbreite reicht von der Sicherstellung von einfachen Coding-Standards (z. B. ein `return`-Statement pro Funktion) über die Prüfung von Typumwandlungen und Bereichsgrenzen über die Suche nach bestimmten Arten von Speicherlecks bis hin zur technischen Verifikation von Quelltext.

Einfache Analysen sind häufig bereits im Compiler (Übersetzer) einer Programmiersprache integriert, z. B. die Prüfung auf Initialisierung einer Variablen. Darüber hinaus gibt es Methoden, die den Programmierstil auf Ästhetik und Pragmatik prüfen, nämlich die stilistischen Methoden. Allerdings werden häufig nur Warnmeldungen angezeigt, die ignoriert werden können. Bei sogenannten Profilern wird zusätzlicher Objektcode generiert, welcher Aussagen über Codeabdeckung und Codefrequentierung generiert. Echte statische Analysierer gibt es nur wenige.

Automatisierte Codereview-Software vereinfacht die Aufgabe der Durchsicht großer Programmteile durch systematische Überprüfungen auf angreifbare Stellen wie:

- Race Conditions
- Formatstring-Angriffe
- Pufferüberläufe
- Speicherlecks

Werkzeuge

Als „Klassiker“ auf diesem Gebiet ist Lint zu nennen (siehe hierzu auch Splint). Neuere Werkzeuge beschränken sich nicht nur auf funktionale Fehler, sondern erkennen auch qualitative Schwachstellen im Code (so genannte *Bad Smells*), wie zum Beispiel duplizierten Code (auch Software-Klone genannt). Sie gehen auch über den Quellcode hinaus und prüfen Konformität des Codes mit der Architekturspezifikation. Ebenso sind durch die neueren Werkzeuge auch Programmierregeln wie die MISRA-C-Regeln überprüfbar. *Flawfinder* und *Rough Auditing Tool for Security (RATS)* sind zwei bekannte Werkzeuge für den Codereview.

Weiterhin besteht eine Abgrenzung zu den Style Checkern.

Weblinks

- Episode 59: Static Code Analysis [1] Interview (Podcast-Beitrag) auf Software Engineering Radio

Literatur

- Ch. Bommer, M. Spindler, V. Barr: *Softwarewartung - Grundlagen, Management und Wartungstechniken*, dpunkt.verlag, Heidelberg 2008, ISBN 3-89864-482-0

Referenzen

[1] http://www.se-radio.net/index.php?post_id=220531

IEEE 829 Standard for Software Test Documentation

Software Quality Assurance Plan

Definitionen von IEEE
• SQAP – Software Quality Assurance Plan IEEE 730 • SCMP – Software Configuration Management Plan IEEE 828 • STD – Software Test Documentation IEEE 829 • SRS – Software Requirements Specification IEEE 830 • SVVP – Software Validation & Verification Plan IEEE 1012 • SDD – Software Design Description IEEE 1016 • SPMP – Software Project Management Plan IEEE 1058

Die Definition **IEEE 730** für den **Software Quality Assurance Plan** (SQAP) beschreibt den Aufbau eines Qualitätssicherungs-Plans. In einem SQAP werden die Entwicklungs-, Test- und Schulungsabläufe innerhalb eines Software-Projektes oder allgemein gültige Richtlinien einer Firma beschrieben.

Literatur

- Eric J. Braude: „Software Engineering *An Object-Oriented Perspective*“, ISBN 0-471-32208-3

Software Configuration Management Plan

Definitionen von IEEE
• SQAP – Software Quality Assurance Plan IEEE 730 • SCMP – Software Configuration Management Plan IEEE 828 • STD – Software Test Documentation IEEE 829 • SRS – Software Requirements Specification IEEE 830 • SVVP – Software Validation & Verification Plan IEEE 1012 • SDD – Software Design Description IEEE 1016 • SPMP – Software Project Management Plan IEEE 1058

Die Definition **IEEE 828 Standard for Software Configuration Management Plans** beschreibt die Mindestanforderungen an einen Software Konfigurationsmanagement Plan. Er wird ergänzt durch den Standard IEEE 1042 (Reaff 1993), IEEE Guide to Software Configuration Management, der Planung und Durchführung des Software Konfigurationsmanagements beschreibt.

Der Standard bezieht sich auf den gesamten Software-Lebenszyklus von kritischer Software, also bei solcher Software, bei der ein Fehlverhalten zu großen finanziellen oder zu Schäden an Leib und Leben führen würde. Er ist ebenso anwendbar bei unkritischer Software und bereits entwickelter Software. Seine Anwendung ist nicht durch die Art der Software beschränkt.

Ein Software-Konfigurationsmanagement-Plan dokumentiert welche Aktivitäten im Rahmen des Konfigurationsmanagements durchzuführen sind, wie diese auszuführen sind, wer für ihre Durchführung verantwortlich ist, wann sie durchzuführen sind und welche Mitarbeiter und Ressourcen dafür erforderlich sind.

Literatur

- Eric J. Braude : "Software Engineering *An Object-Oriented Perspective*", ISBN 0-471-32208-3
- The Institute of Electrical and Electronics Engineers, Inc : IEEE Standard for Software Configuration Management Plans, Print: ISBN 0-7381-0331-4

Software Test Documentation

Definitionen von IEEE
• SQAP – Software Quality Assurance Plan IEEE 730 • SCMP – Software Configuration Management Plan IEEE 828 • STD – Software Test Documentation IEEE 829 • SRS – Software Requirements Specification IEEE 830 • SVVP – Software Validation & Verification Plan IEEE 1012 • SDD – Software Design Description IEEE 1016 • SPMP – Software Project Management Plan IEEE 1058

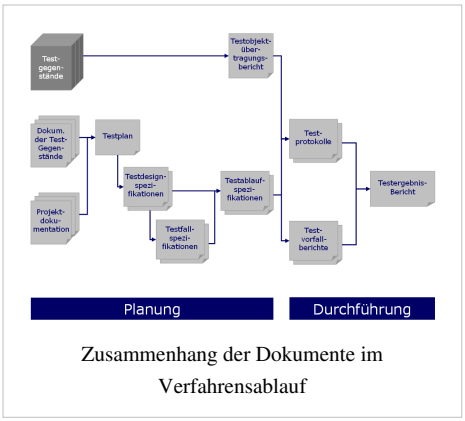
Die Definition **IEEE 829 Standard for Software Test Documentation** ist ein vom IEEE (Institute of Electrical and Electronic Engineers) veröffentlichter Standard, der einen Satz von acht Basis-Dokumenten zur Dokumentation von Software-Tests beschreibt. Der Standard beschreibt Form und Inhalt der jeweiligen Dokumente. Er schreibt jedoch nicht vor, welche der jeweiligen Dokumente zwingend verwendet werden müssen.

Allgemein

Der Standard beschreibt acht Dokumente, die sich wie folgt in drei Kategorien unterteilen lassen.

- ***Übersicht***

1. **Testkonzept** (test plan): Das Testkonzept bestimmt Abgrenzung, Vorgehensweise, Mittel und Ablaufplan der Testaktivitäten. Es bestimmt die Elemente und Produktfunktionen, die getestet werden sollen, die Testaufgaben, die durchgeführt werden müssen, das verantwortliche Personal für jede Aufgabe und das Risiko, das mit dem Konzept verbunden ist.



Zusammenhang der Dokumente im Verfahrensablauf

- ***Test-Spezifikation*** (test specification)

1. **Test-Design-Spezifikation** (test design specification): Die Test-Design-Spezifikation verfeinert die Beschreibung der Vorgehensweise für das Testen der Software. Sie identifiziert die Produktfunktionen, die von den Tests abgedeckt werden müssen. Sie beschreibt weiterhin die Testfälle und Testabläufe, die benötigt werden, um Tests zu bestehen und spezifiziert die Bestehens- oder Verfehlenskriterien der einzelnen Produktfunktionen.
2. **Testfall-Spezifikation** (test case specification): Die Testfall-Spezifikation dokumentiert die zu benutzenden Eingabewerte und erwarteten Ausgabewerte. Testfälle sind vom Test-Design getrennt. Dies erlaubt die Verwendung der Testfälle in mehreren Designs und die Wiederverwendung in anderen Situationen.
3. **Test-Ablauf-Spezifikation** (test procedure specification): Beschreibung aller Schritte zur Durchführung der spezifizierten Testfälle und Implementierung des zugehörigen Test-Designs.

- ***Test-Beschreibung*** (test reporting)

1. **Testfall-Übertragungsbeschreibung** (test item transmittal report): Die Testfall-Übertragungsbeschreibung beschreibt die Übertragung der Testfälle für den Fall, dass getrennte Entwicklungs- und Testteams eingebunden sind oder für den Fall, dass ein offizieller Zeitpunkt für den Beginn einer Testausführung erwünscht ist.
2. **Test-Protokoll** (test log): Das Test-Protokoll dient zur Aufzeichnung der Ereignisse während einer Testausführung.

3. **Test-Störfall-Beschreibung** (test incident report): Es beschreibt alle Ereignisse, die während einer Testausführung auftreten und weitere Nachprüfungen erfordern.
4. **Test-Zusammenfassung** (test summary report): Fasst die Testaktivitäten zusammen, die mit einem oder mehreren Test-Design-Spezifikationen zusammenhängen.

Literatur

- IEEE-SA Standards Board: *IEEE Standard for Software Test Documentation*. Piscataway NJ, 1998: The Institute of Electrical and Electronic Engineers Inc. ISBN 0-7381-1443-X
- Eric J. Braude : *Software Engineering. An Object-Oriented Perspective*. Hoboken NJ, 2000: Wiley Inc. ISBN 0-471-32208-3

Weblinks

- IEEE Software Engineering Collection [1]

Referenzen

[1] http://standards.ieee.org/

Software Requirements Specification

Definitionen von IEEE
• SQAP – Software Quality Assurance Plan IEEE 730
• SCMP – Software Configuration Management Plan IEEE 828
• STD – Software Test Documentation IEEE 829
• SRS – Software Requirements Specification IEEE 830
• SVVP – Software Validation & Verification Plan IEEE 1012
• SDD – Software Design Description IEEE 1016
• SPMP – Software Project Management Plan IEEE 1058

Die **Software Requirements Specification** (SRS) ist ein von IEEE (Institute of Electrical and Electronic Engineers) erstmals unter (ANSI/IEEE Std 830-1984) veröffentlichter Standard zur Spezifikation von Software. Das IEEE hat die Spezifikation mehrmals überarbeitet und die momentan neueste Version ist Std 830-1998.

Die SRS umfasst das Lastenheft wie auch das Pflichtenheft.

Qualität

Die IEEE Kap. 4.3 definiert 8 Charakteristika guter SRS:

- Korrekt
- Unzweideutig
- Vollständig
- Konsistent
- Bewertet nach Wichtigkeit und/oder Stabilität
- Verifizierbar
- Modifizierbar
- Verfolgbar (Traceable)

Korrekt und Vollständig bezieht sich dabei auf die SRS bezüglich der tatsächlichen Anforderungen (externer Bezug). Konsistenz bezieht sich auf die Anforderungen in Form der SRS alleine (interner Bezug). Unmehrdeutigkeit lässt genau eine Interpretation zu, Verifizierbarkeit begrenzt die Komplexität einer Anforderungsbeschreibung zusätzlich auf ein effizient prüfbares Maß. Modifizierbarkeit setzt insbesondere Redundanzfreiheit voraus. *Traceability* umfasst die vor- und rückwärtige Richtung.

Dokumentation

Die IEEE hat mit dieser Definition festgelegt, wie das Dokument aufgebaut werden soll. Die Kapitel, die in diesem Dokument vorkommen sollen, stehen somit fest. Dabei ist das Dokument grundsätzlich in 2 Bereiche aufgeteilt:

- C-Requirement (*Customer-Requirement*): Bereich ist mit Lastenheft vergleichbar
- D-Requirement (*Development-Requirement*): Bereich ist mit Pflichtenheft vergleichbar

Unter C-Requirement sind die Anforderungen aus Sicht des Kunden und/oder des End-Anwenders zu erfassen. Unter D-Requirement versteht man die Entwicklungsanforderungen. Dies ist die Sicht aus den Augen des Entwicklers, der technische Aspekte in den Vordergrund stellt, im Gegensatz zum Kunden.

Mit *Requirements* (deutsch: ‚Anforderungen') ist sowohl die qualitative als auch die quantitative Definition eines benötigten Programms aus der Sicht des Auftraggebers gemeint. Im Idealfall umfasst eine solche Spezifikation ausführliche Beschreibung von Zweck, geplantem Einsatz in der Praxis sowie dem geforderten Funktionsumfang einer Software.

Hierbei sollte fachlichen – „Was soll die Software können?" – wie auch technischen Aspekten – „In welchem Umfang und unter welchen Bedingungen wird die Software eingesetzt werden?" – Rechnung getragen werden.

Eine SRS enthält nach IEEE Standard mindestens drei Hauptkapitel. Die vorgeschlagene Gliederung sollte zwar in den Kernpunkten eingehalten werden. In der Praxis wird diese jedoch häufig im Detail modifiziert. Eine exemplarische Gliederung könnte wie folgt aussehen:

- Name des Softwareprodukts
- Name des Herstellers
- Versionsdatum des Dokuments und / oder der Software

1. Einleitung
    1. Zweck (des Dokuments)
    2. Umfang (des Softwareprodukts)
    3. Erläuterungen zu Begriffen und / oder Abkürzungen
    4. Verweise auf sonstige Ressourcen oder Quellen
    5. Übersicht (Wie ist das Dokument aufgebaut?)
2. Allgemeine Beschreibung (des Softwareprodukts)
    1. Produktperspektive (zu anderen Softwareprodukten)
    2. Produktfunktionen (eine Zusammenfassung und Übersicht)
    3. Benutzermerkmale (Informationen zu erwarteten Nutzern, z.B. Bildung, Erfahrung, Sachkenntnis)
    4. Einschränkungen (für den Entwickler)
    5. Annahmen und Abhängigkeiten (nicht Realisierbares und auf spätere Versionen verschobene Eigenschaften)
3. Spezifische Anforderungen (im Gegensatz zu 2.)
    1. funktionale Anforderungen (Stark abhängig von der Art des Softwareprodukts)
    2. nicht-funktionale Anforderungen
    3. externe Schnittstellen
    4. *Design Constraints*
    5. Anforderungen an Performance
    6. Qualitätsanforderungen

7. Sonstige Anforderungen

Die Schwierigkeiten, die sich in der Praxis bei einer solchen Anforderungsanalyse ergeben, sind

- mögliche Interessenkonflikte, also unterschiedliche Ziele seitens der Nutzer
- unklare oder sogar unbekannte technische Rahmenbedingungen
- sich ändernde Anforderungen oder Prioritäten schon während des Entwurfsprozesses

Literatur

- *IEEE Guide to Software Requirements Specification.* ANSI/IEEE Std 830-1984. IEEE Press, Piscataway/New Jersey 1984.
- Colin Hood, Susanne Mühlbauer, Chris Rupp, Gerhard Versteegen (Hrsg.): *iX-Studie Anforderungsmanagement.* Methoden und Techniken, Einführungsszenarien und Werkzeuge im Vergleich. 2. Auflage. Heise, Hannover April 2007, OCLC 255168117 [1] (Übersicht bei Heise [2]).

Weblinks

- IEEE Software Engineering Collection via the IEEE Shop [3]

Referenzen

[1] http://worldcat.org/oclc/255168117
[2] http://www.heise.de/kiosk/special/ixstudie/05/01/
[3] https://sbwsweb.ieee.org/ecustomercme_enu/start.swe?SWECmd=GotoView&SWEView=Catalog+View+(eSales)_Standards_IEEE&mem_type=Customer&SWEHo=sbwsweb.ieee.org&SWETS=1192713657

Software Validation & Verification Plan

Definitionen von IEEE
• SQAP – Software Quality Assurance Plan IEEE 730 • SCMP – Software Configuration Management Plan IEEE 828 • STD – Software Test Documentation IEEE 829 • SRS – Software Requirements Specification IEEE 830 • SVVP – Software Validation & Verification Plan IEEE 1012 • SDD – Software Design Description IEEE 1016 • SPMP – Software Project Management Plan IEEE 1058

Die Definition IEEE 1012 für den Software Validation & Verification Plan (SVVP) beschreibt den minimalen Standard, wie eine Validierung und Verifikation einer Software dokumentiert werden soll. Dabei soll die Frage, ob das System richtig gebaut wurde (Verification) sowie die Frage ob das richtige System gebaut wurde (Validation) ausreichend beantwortet werden.

Literatur

- Eric J. Braude: *Software Engineering. An Object-Oriented Perspective*. Wiley, New York, NY u. a. 2001, ISBN 0-471-32208-3.

Software Design Description

Definitionen von IEEE
• SQAP – Software Quality Assurance Plan IEEE 730 • SCMP – Software Configuration Management Plan IEEE 828 • STD – Software Test Documentation IEEE 829 • SRS – Software Requirements Specification IEEE 830 • SVVP – Software Validation & Verification Plan IEEE 1012 • SDD – Software Design Description IEEE 1016 • SPMP – Software Project Management Plan IEEE 1058

Die **Software Design Description** (SDD) ist ein von IEEE (Institute of Electrical and Electronic Engineers) unter (ANSI/IEEE Std 1016-1998) veröffentlichter Standard, welcher festlegt, wie ein Programm-Design spezifiziert werden soll.

Allgemein

Das SDD Dokument ist eine Repräsentation eines Software-Systems, die verwendet wird, um Software-Design-Information zu kommunizieren. Es umfasst wesentliche Teile der Development-Requirements (D-Requirements). Es beschreibt die Architektur der Software beziehungsweise des Gesamtsystems und der einzelnen Komponenten. Ein SDD wird in der Regel nur bei größeren Projekten erstellt, beziehungsweise da, wo die System-Architektur entscheidenden Einfluss auf die Software hat. Bei kleineren Projekten ist es üblich, das Design direkt im D-Requirement Teil der Software Requirements Specification (SRS) vorzunehmen. Dabei sollte die Beschreibung des Designs nicht mehr als 3 Seiten umfassen, ansonsten empfiehlt es sich ein SDD zu erstellen.

Das Dokument sollte im wesentlichen folgende Kapitel enthalten:

1. Einleitung
    1. Design-Übersicht
    2. Anforderungs-Nachvollziehbarkeits-Matrix
2. System-Architektur
    1. Gewählte Systemarchitektur
    2. Diskussion alternativer Architekturen
    3. Beschreibung der Schnittstellen des Systems
3. Detailbeschreibung der Komponenten
    1. Komponente n
    2. Komponente n+1
4. Benutzerschnittstelle (UI)
5. Zusätzliches Material (Appendix)

Software Project Management Plan

Definitionen von IEEE
• SQAP – Software Quality Assurance Plan IEEE 730 • SCMP – Software Configuration Management Plan IEEE 828 • STD – Software Test Documentation IEEE 829 • SRS – Software Requirements Specification IEEE 830 • SVVP – Software Validation & Verification Plan IEEE 1012 • SDD – Software Design Description IEEE 1016 • SPMP – Software Project Management Plan IEEE 1058

Die Definition IEEE 1058 bestimmt die Aufmachung und den Inhalts eines Software Project Management Plans (SPMP). Dieser ist das maßgebliche Dokument für die Organisation eines Software Projekts; er beschreibt die formalen und organisatorischen Prozesse, die für die Software Entwicklung nötig sind, um den Produkt-Anforderungen entsprechen zu können.

Anhang

AUTOSAR

AUTOSAR-Logo

AUTOSAR (*AUTomotive Open System ARchitecture*) ist eine Entwicklungspartnerschaft aus Automobilherstellern, Steuergeräteherstellern sowie Herstellern von Entwicklungswerkzeugen, Steuergeräte-Basis-Software und Mikrocontrollern. Ziel von AUTOSAR ist es, den Austausch von Software auf verschiedenen Steuergeräten zu erleichtern. Dazu wurde eine einheitliche Softwarearchitektur mit einheitlichen Beschreibungs- und Konfigurationsformaten für Embedded Software im Automobil erarbeitet. AUTOSAR definiert Methoden zur Beschreibung von Software im Fahrzeug, die sicherstellen, dass Softwarekomponenten wieder verwendet, ausgetauscht, skaliert und integriert werden können.

Konzept

Eine der Grundideen der Entwicklungspartnerschaft AUTOSAR lautet *Zusammenarbeit bei Standards – Wettbewerb bei der Umsetzung* (engl. *cooperate on standards – compete on implementation*).

Folgende Fragestellungen und Ziele werden angesprochen:

- Standardisierung wichtiger Systemfunktionen
- Erfüllung zukünftiger Fahrzeuganforderungen bezüglich Verfügbarkeit, Sicherheit und Softwareaktualisierung
- Flexibles Integrieren, Verschieben und Austauschen von Funktionen im Steuergerätenetzwerk
- Unterstützung sog. COTS-Software verschiedener Hersteller
- Beherrschung der gestiegenen Produkt- und Prozesskomplexität
- Kostengünstige Skalierbarkeit
- Wartbarkeit über den gesamten Produktlebenszyklus

Geschichte

Nach ersten Gesprächen der Unternehmen BMW, Daimler, Bosch, Continental und Volkswagen 2002 trat Siemens VDO in die Partnerschaft ein, die 2003 offiziell beschlossen wurde. Weitere, später hinzugestoßene Partner sind Ford, General Motors, Toyota und Peugeot. Diese Unternehmen stellen die Core Partner, die durch Premium Member und Associate Member (weitere 1st Tier) ergänzt werden. Nach der Übernahme von Siemens VDO durch die Continental AG im Jahre 2007 sind noch neun Unternehmen Core Partner bei AUTOSAR: BMW, Bosch, Continental, Daimler, Ford, General Motors, PSA (Peugeot Citroën), Toyota, Volkswagen. Heute (Juli 2010) sind 46 Premium Members bei AUTOSAR vertreten. Dazu zählen Automobilhersteller (z. B. Fiat, Honda, Hyundai, Mazda, Porsche, Renault, TATA, Volvo), Steuergerätehersteller (z. B. Delphi, Denso, Magneti Marelli, Valeo), Hersteller von Entwicklungswerkzeugen (z. B. dSPACE, ETAS, The MathWorks, Vector Informatik) und Halbleiterhersteller (z. B Infineon).

Die Erarbeitung und Verabschiedung der Standards erfolgt in verschiedenen Arbeitsgruppen. Eine gemeinsam erarbeitete Roadmap sichert sowohl die Inhalte als auch den Zeithorizont ab.

AUTOSAR Release	Veröffentlicht	Phase
1.0	08.07.2005	Phase 1
2.0	04.05.2006	
2.1	04.12.2006	
3.0	21.12.2007	Phase 2
3.1	15.08.2008	
4.0	18.12.2009	

Zusätzlich zu diesen Releases gibt es noch eine größere Anzahl (ca. 35) kleinerer Revisions, die hauptsächlich Bugfixes und kleinere Korrekturen beinhalten. Derzeit (Stand Februar 2010) wird der Zeitplan und eine Release-Planung für die Anfang 2010 gestartete Phase III erarbeitet.

AUTOSAR-Architektur

Wesentlich für AUTOSAR ist die logische Aufteilung in die steuergerätespezifische Basis-Software (Basic Software, BSW) und die steuergeräteunabhängige Anwendungs-Software (ASW). Dazwischen liegt ein virtuelles Funktionsbussystem (Virtual Function Bus, VFB). Dieser virtuelle Funktionsbus verbindet alle Softwarekomponenten, auch die, die in unterschiedlichen Steuergeräten implementiert sind. So können diese zwischen verschiedenen Steuergeräten verschoben werden, ohne dass Änderungen in den betreffenden Softwarekomponenten selbst vorgenommen werden müssen. Dies kann zur Optimierung von Rechenleistung, Speicherbedarf oder Kommunikationslast nützlich sein. Die funktionalen Softwarekomponenten (Software Component, SWC) sind strikt voneinander und von der Basis-Software getrennt. Sie kommunizieren über die AUTOSAR-Schnittstelle mit den anderen Funktionen und den Steuergeräteschnittstellen. Diese Schnittstellen (API) sind in SWC-XML-Beschreibungen definiert. Die Basis-Software (Basic Software, BSW) enthält die steuergerätespezifischen Programmteile, wie die Kommunikationsschnittstellen, die Diagnose und das Speichermanagement. Neben der Steuergeräte-Architektur ist auch die Entwicklungsmethodik durch AUTOSAR teilweise standardisiert. Es handelt sich dabei vor allem um die Struktur und die Abhängigkeiten der unterschiedlichen Arbeitsprodukte (z. B. Dateien). Diese werden benötigt, um aus den unterschiedlichen Software-Komponentenbeschreibungen ausführbare Programme für die jeweiligen Steuergeräte zu erzeugen.

Kernstück des Architekturkonzepts ist die AUTOSAR-Laufzeitumgebung (Run-Time Environment, RTE), eine Kommunikationsschicht, die nach dem Prinzip des *Virtual Functional Bus (VFB)* im Sinne einer Middleware von der realen Steuergeräte-Topologie und den daraus resultierenden Kommunikationsbeziehungen abstrahiert. Zwei Funktionen können demnach ohne Kenntnis des Signalpfades Informationen miteinander austauschen, indem sie sog. Kommunikationsports der Laufzeitumgebung verwenden. Dieser Mechanismus macht sich dadurch vorteilhaft bemerkbar, dass Funktionen unabhängig von der später im Fahrzeug existierenden Topologie entwickelt werden können. Die tatsächlichen Signalpfade werden erst spät im Entwicklungsablauf durch einen Konfigurationsmechanismus festgelegt.

Literatur

- Olaf Kindel, Mario Friedrich: *Softwareentwicklung mit AUTOSAR. Grundlagen, Engineering, Management für die Praxis.* dpunkt.verlag, 2009, ISBN 978-3-89864-563-8.
- Werner Zimmermann und Ralf Schmidgall: *Bussysteme in der Fahrzeugtechnik – Protokolle, Standards und Softwarearchitektur*. Vieweg+Teubner, 4. Auflage, 2010, ISBN 978-3-8348-0907-0

Weblinks

- Offizielle Website [1] von AUTOSAR
- AUTOSAR Groupe [2] auf Xing
- Homepage von EB [3] AUTOSAR Basis Software Kern Implementierung von EB (Elektrobit)
- AUTOSAR – A first Glance [4] Video von EB(Elektrobit): gibt eine kurze Einführung zu AUTOSAR
- AUTOSAR Einführung [5] (Vorlesungsskript der Uni Konstanz; pdf, 1,1 MB)
- AUTOMOBIL-ELEKTRONIK – electronica – Sonderausgabe 2008 [6] Die hardware-nahe Seite von Autosar (PDF-Datei; 2,43 MB)
- Fachartikel über die in AUTOSAR definierten Austauschformate [7] in Elektronik automotive
- Fachartikel über typische Änderungsszenarien bei der Entwicklung von AUTOSAR-Steuergerätesoftware [8] in Elektronik automotive

Referenzen

[1] http://www.autosar.org/
[2] http://www.xing.com/net/autosar/
[3] http://www.eb-tresos-blog.com/solutions/tresos/autosar-basic-software/
[4] http://www.eb-tresos-blog.com/technologies/autosar/
[5] http://www.inf.uni-konstanz.de/soft/teaching/ws07/autose/leitner-autosar.pdf
[6] http://imperia.mi-verlag.de/imperia/md/content/ai/ae/fachartikel/ael/2008/21/ael08_21_electronica_022.pdf
[7] http://www.elektroniknet.de/automotive/technik-know-how/test-entwicklungstools/article/1598/0/Neue_Wege_zur_Steuergerxxxaumlxxxte-Software/
[8] http://www.elektroniknet.de/automotive/technik-know-how/bussysteme/article/1646/0/AUTOSAR_in_der_Praxis_Der_Lebenszyklus_von_AUTOSAR-Software_Teil_2/

ENDE

Quelle(n) und Bearbeiter des/der Artikel(s)

Softwaretest *Quelle*: http://de.wikipedia.org/w/index.php?oldid=88039821 *Bearbeiter*: Abubiju, AchimR, Aka, Akropolit, Ammeling, Ano Nym, Avron, Bananen-Joe, BesondereUmstaende, Burmagroup, Bücherwurm16, Carol.Christiansen, ChristianHujer, Clemfix, Comc, Dachris, Darok, Defrenrokorit, Diwas, Dlm, EinStein, Ellyce, Elwe, Entlinkt, Erkan Yilmaz, Fizfaz, Flavia67, Fleasoft, Fristu, Fruchtcocktail, GB, GGlasauer, Gadelor, Gaius L., Gereon K., Ghw, Gnu1742, Hadhuey, Hansruedi.Tremp, HenrikHolke, Himuralibima, Jpp, JustB, Karl-Friedrich Lenz, Kipparj, Kku, Kraymer, LKD, Laetterman, Levin, Liberatus, Lostintranslation, Lustiger seth, Löschfix, Ma-Lik, Mareike jacobshagen, Marsellus, Mcaspari01, Media lib, Michael Kunz, Mikue, Mussklprozz, NSz, Nutzer Eins, Pelz, PeterFrankfurt, QFS, Ramtam, Reallimk, ReneS, Reseka, Rfc, Rolf acker, RonMeier, Rufus46, S.K., STBR, Sebastian.Dietrich, Silberchen, Sinn, SonniWP2, Sparti, Staro1, Stefan Knoepfel, Stefan.qn, StefanMacke, Stephan.Weissleder, Taschenrechner, Test-tools, Theoprakt, Thomaserl, Till.niermann, Traute Meyer, Tsor, Tuthmosis VII, Tönjes, Uncopy, VerwaisterArtikel, Video2005, Vsesb01, VÖRBY, Wiki-piet, Xflupp, Zaungast, Zinnmann, 220 anonyme Bearbeitungen

Testfall *Quelle*: http://de.wikipedia.org/w/index.php?oldid=81816884 *Bearbeiter*: Abubiju, AchimR, Ano Nym, Avron, Bernard Ladenthin, Duesentrieb, Erkan Yilmaz, Fleasoft, Frank Jacobsen, Hadhuey, Hschaefer, ManWing2, Morgenröte, Saehrimnir, Schneeelefant, Sparti, Streifengrasmaus, Uncopy, Uwe Gille, 9 anonyme Bearbeitungen

Bananenprinzip *Quelle*: http://de.wikipedia.org/w/index.php?oldid=85879532 *Bearbeiter*: Aka, Alofok, Archiv, Bananenfalter, BerndB, Chrislb, Cokeser, Enth'ust'eac, Ercas, FWHS, Kirdoran, Krischan111, Manja, Mikue, Nrainer, Ordnung, Panzi, Regiomontanus, Rhino2, Rohieb, Rolf acker, RolfS, Sir Anguilla, Sparti, Speck-Made, Spuerhund, Tau Lambda, Tokikake, Urbanus, VerwaisterArtikel, YMS, 13 anonyme Bearbeitungen

Positivtest *Quelle*: http://de.wikipedia.org/w/index.php?oldid=86838201 *Bearbeiter*: A.Abdel-Rahim, Abubiju, 1 anonyme Bearbeitungen

Negativtest *Quelle*: http://de.wikipedia.org/w/index.php?oldid=86838248 *Bearbeiter*: A.Abdel-Rahim, Abubiju, 1 anonyme Bearbeitungen

Black-Box-Test *Quelle*: http://de.wikipedia.org/w/index.php?oldid=87613809 *Bearbeiter*: Abubiju, AchimR, Astrobeamer, Avoided, Avron, Bernhard55, Bronko, Comc, Dealerofsalvation, Erkan Yilmaz, Gadelor, Gerd Taddicken, Hseffler, Jergen, Kauan, Kku, Lofor, Media lib, Merlin G., Mikue, Rorret, S.K., STBR, Saehrimnir, Sparti, Spuk968, Ste ba, WAH, Whispermane, 44 anonyme Bearbeitungen

Grey-Box-Test *Quelle*: http://de.wikipedia.org/w/index.php?oldid=55499823 *Bearbeiter*: AchimR, Blubbalutsch, ChristianHujer, HolGr, Jpp, S.K., Sebastian Wallroth, Sparti, 10 anonyme Bearbeitungen

White-Box-Test *Quelle*: http://de.wikipedia.org/w/index.php?oldid=88123055 *Bearbeiter*: AchimR, Andreas 06, Avron, Berni, Bronko, Comc, Cpl23, DaB., Der.Traeumer, Erkan Yilmaz, Exil, Gadelor, Gerd Taddicken, Hseffler, JeeAge, Jens m0, Kh80, Luiscantero, Mikue, Mstolt, OD, Oratio, Pixelfire, S.K., Schmiddtchen, Sparti, Ste ba, TomK32, 25 anonyme Bearbeitungen

Code-Walkthrough *Quelle*: http://de.wikipedia.org/w/index.php?oldid=83837088 *Bearbeiter*: Arbraxan, Fleasoft, KaiMartin, SoIssetEben!, Thomas Möller, UlrichJ, WikiCare, 1 anonyme Bearbeitungen

Modellbasiertes Testen *Quelle*: http://de.wikipedia.org/w/index.php?oldid=86221471 *Bearbeiter*: AN, Dealerofsalvation, EXept, Fleasoft, Fruchtcocktail, Govannon, Haflinger, Jaellee, Jens Lüdemann, Jpp, Jschlosser, Jzander, Krawi, LKD, Laetterman, Limasign, Ma-Lik, ManWing2, Michael Kunz, PeeCee, Riepichiep, Steffenjung, Test-tools, Tschäfer, Umweltschützen, WikipediaMaster, Wolf32at, 50 anonyme Bearbeitungen

Dynamisches Software-Testverfahren *Quelle*: http://de.wikipedia.org/w/index.php?oldid=82540569 *Bearbeiter*: Abubiju, Armin.pollak, Avron, Baakoo, Chpfeiffer, Darok, Dealerofsalvation, Gadelor, H005, Heinte, Hubertl, Ixitixel, Jpp, Laetterman, Maggie85, Phloogzoyk, Pittimann, Rabus, Rbendig, STBR, Taschenrechner, Test-tools, Van der Hoorn, 62 anonyme Bearbeitungen

Hardware in the Loop *Quelle*: http://de.wikipedia.org/w/index.php?oldid=85971908 *Bearbeiter*: AHZ, Achimhaag, Aka, Beherbie, BernerMattner, Cruccone, Crux, Elali, Erkan Yilmaz, FelixReimann, Florian.graetz, HaSee, Harro von Wuff, Heinte, Jpp, Jschlosser, LKD, Luiscantero, Lunochod, Ma-Lik, Mintrain, Nameless23, Ninalia007, Scalert, Slillwitz, Sunshine09, Surferskieur, TableSitter, Twts, Vossi12, Wdwd, Werner Adler, Wiggum, Zoosommer, 53 anonyme Bearbeitungen

Model in the Loop *Quelle*: http://de.wikipedia.org/w/index.php?oldid=84980509 *Bearbeiter*: Aves83, BastianVenthur, Blueinf, Drahreg01, GordonKlimm, Inkowik, Jens Lüdemann, Jschlosser, Zölle, 1 anonyme Bearbeitungen

Testautomatisierung *Quelle*: http://de.wikipedia.org/w/index.php?oldid=87121118 *Bearbeiter*: 83nj1, Beherbie, D4sh 0f sunshin3, Du3n5cH, EXept, Elali, GabiS, Gruelz, Gurustefan, HaSee, He3nry, Hermannthomas, Inschenör, Jkoprax, Jpp, Jschlosser, KaiBorgeest, Kaiblankenhorn, Kam Solusar, Kissaki, Korotkiy, Krawi, LKD, Lac-du-Nord, Ma-Lik, ManWing2, Marcustik, Michael Kunz, Michaelt1964, Minderbinder, Mscholze, Nameless23, PDD, PeeCee, Pr0n, QFS, SAPEngel, Schneeelefant, Spille1986, Steffenjung, Stest, TMg, TReichert, Testrob, Tilla, 82 anonyme Bearbeitungen

Modultest *Quelle*: http://de.wikipedia.org/w/index.php?oldid=87925443 *Bearbeiter*: Addicted, AlB, Aleks-ger, AndreasBrodt, Arved, Avron, AwOc, B-e-n, Bbommel, Bhaak, Boris Fernbacher, Cactus26, ChristophDemmer, DDDan, Dishayloo, Dlm, Duesentrieb, Enslin, Erkan Yilmaz, Fleasoft, Fleshgrinder, Florian Adler, Fomafix, G.wolf@klopotek.de, Gaius L., Ghostface, Glauschwuffel, Habakuk, Hades, Harro von Wuff, Hhdw, IGEL, Itangast, JMetzler, Jbuennagel, Juergen861, Kam Solusar, Kaylord, Kek00207, Kissaki, Kku, Klapper, Klausikm, Kockster, Lanzm, Ledermann, Levin, Ma-Lik, Mark Nowiasz, Michelangelo, Mijikenda, Mikegr, Mm pedia, Norro, Philipp Claßen, Rdb, Revvar, Roterraecher, Sebastian.Dietrich, Sparti, Suchenwi, Sympathikus, Sypholux, Test-tools, TheEdge, Thommess, Tiroinmundam, Uwe Hermann, Vrumfondel, W.amadeus, WAH, Yuuki Mayuki, 93 anonyme Bearbeitungen

Liste von Modultest-Software *Quelle*: http://de.wikipedia.org/w/index.php?oldid=87401723 *Bearbeiter*: Aka, Arty, Avron, Bammsi, Bastie, Divbyzero, Don Magnifico, Erkan Yilmaz, Fish-guts, Flindner, FreelancerHamburg, GabiS, Georg0431, Giftpflanze, Gms, Guffi, Habakuk, Hairmare, Harro von Wuff, HenrikHolke, Jens Lüdemann, Jluedem, Jschlosser, Kantor.JH, Karsten11, Krawi, Lichtkind, ManWing2, NoSoftwarePatents, Omi´s Törtchen, Ovhaag, P. Birken, PDD, PerfektesChaos, Phaidros.vie, Provi-neu, RalfHandl, Rioderelfte, Rolf acker, SKr, TheWolf, Theoprakt, Tovomer, Trublu, Willchisum, 31 anonyme Bearbeitungen

Integrationstest *Quelle*: http://de.wikipedia.org/w/index.php?oldid=86766500 *Bearbeiter*: Abdull, Abubiju, AchimR, Dickbauch, Gaius L., Kku, Kwer Wolf, Sparti, Stefan Knoepfel, Stylor, 9 anonyme Bearbeitungen

Programmfehler *Quelle*: http://de.wikipedia.org/w/index.php?oldid=87840553 *Bearbeiter*: AFtec, Adfsdfds, Ahellwig, Alauda, Alex\, AndreasB, Angie, Atlan da Gonozal, Avron, Axel1963, BSDev, Bautsch, Belz, Benzen, Berger77, Bildungsbürger, Björn Bornhöft, Bodo Thiesen, Boonekamp, C-M, Centic, Chiccodoro, Chricho, ChristianHujer, Cosini Zeno, Cruks, Cvk, D, Danimilkasahne, Darok, Der.Traeumer, DerHexer, DerPaul, Dha, Diddi, Dishayloo, Dom0112, Don Magnifico, Duesentrieb, Dundak, ElRaki, ErikDunsing, Fiver, der Hellseher, Fleasoft, Flo 1, Florian.Keßler, FutureCrash, GDK, Georg-Johann, Gidoca, Giftmischer, Guillermo, Gurt, Gustavf, HALsixsixsix, Hadhuey, Hardy Linke, He3nry, Head, Henry Helm, Howwi, Hubi, Hvogelpohl, Hystrix, Häkchen, Jan Giesen, Joergmkam, JonnyJD, Jowereit, Jpp, Jschlosser, JuergenL, Kallistratos, Karla Blomquist, Kdot, Kku, Klaeren, KleinPhi, Koerpertraining, Krawi, LGMuenchen, Leider, LeoVisurgis, Localhost, Lothar Kimmeringer, Lukian, MFM, MGla, Marc van Woerkom, MarkusHagenlocher, Martin Wagenleiter, Matthäus Wander, Matzematik, McB, Media lib, Membeth, Micha L. Rieser, Moros, Mps, Msommerlandt, Mudd1, Mussklprozz, MyBBCoder, NEmai, Nalincah, Norbert, Nrainer, OderWat, Odin, Ohrläpple, PPW, PaulT, PeeCee, Pere Ubu, Peter Thomassen, Peter200, PhilippWeissenbacher, Pinoccio, Pkn, Pruefer, Qman232, Regi51, Reilinger, Romankawe, SCPS, Sfischer, Shoot the moon, Sikilai, Skriptor, Smurf, SonniWP, Spuk968, StYxXx, Stahlkocher, Stefan Kühn, Stefan h, Steffen, Stegosaurus Rex, Stern, SuMMon.KuLT, SuperFLoh, SvGeloven, Tehforsch, The-pulse, ThePeritus, TheSkunk, ThoR, Thogo, Thorny, Tobias Bergemann, TomK32, Tsor, Tsui, Uwe W., Vertigo21, Verwüstung, Wachs, WiESi, WikipediaMaster, Wikipediaphil, Wisi, WissensDürster, Wissling, WolfgangRieger, WvBraun, Xeph, Xo0ox, Yahp, Yohohoho!, YourEyesOnly, Yuuki Mayuki, Zenebona, Zumbo, ZweiZahn, 166 anonyme Bearbeitungen

Proof-Carrying Code *Quelle*: http://de.wikipedia.org/w/index.php?oldid=68055761 *Bearbeiter*: Bautsch, Don Magnifico

Assertion (Informatik) *Quelle*: http://de.wikipedia.org/w/index.php?oldid=84741192 *Bearbeiter*: Abdull, Abevauer, Apulix, Ardo Beltz, Arved, AwOc, Christian2003, DerRaoul, Echoray, Flogger, Fomafix, GottschallCh, Hafenbar, Harald Tribune, Homer Landskirty, Jpp, Kasper4711, Kko, Langec, Merlin G., Mulno, Oefe, PhilippWeissenbacher, S.K., Sparti, Speck-Made, Stefan Majewsky, StefanLoth, Thomas Willerich, Tilmanb, Trustable, Uncopy, Uwe Gille, VictorAnyakin, Wiegels, Wimmerm, Zaungast, 38 anonyme Bearbeitungen

Ausnahmebehandlung *Quelle*: http://de.wikipedia.org/w/index.php?oldid=86148204 *Bearbeiter*: Aka, Ataxa, Axel1963, BSDev, Bdk, BuSchu, Chkorn, ChristianErtl, Danrah, Dishayloo, Don Magnifico, DutiesAtHand, ErikDunsing, Fomafix, HartmutS, Jan Giesen, JanBromberger, Jensw, Jodoform, Joswig, Jpp, JörgP, Karl-Henner, Kound, Leo141, Levin, Liro9000, Mark Nowiasz, Matrixx, Mkleine, Ninjamask, Olei, Philipendula, Revolus, Semper, Sleske, Sparti, Sprachpfleger, Staro1, Stfn, Tinita, Tola69, Uncopy, WiESi, Wisi, WissensDürster, XZise, XenonX3, 87 anonyme Bearbeitungen

Bugtracker *Quelle*: http://de.wikipedia.org/w/index.php?oldid=87657049 *Bearbeiter*: A.Savin, AWendt, Abubiju, Avron, Badenserbub, Chriki, DasBee, Der.Traeumer, DerHerrMigo, ErhardRainer, Gidoca, Gosu, Heinte, Herrick, Hpweecks, Jpp, Konzales, Lee-harvey, Marco2804, MichaelDiederich, Mussklprozz, PPW, PeFu, Peter200, Phrood, Semperor, Speck-Made, Staro1,

Suit, Thornard, TobiasHerp, Trustable, Wikinger08, Yuuki Mayuki, Zerodeath, 32 anonyme Bearbeitungen

Berechenbarkeit *Quelle*: http://de.wikipedia.org/w/index.php?oldid=86216650 *Bearbeiter*: AlfonsGeser, Archiv, Baumfreund-FFM, Bri B., Bunt, ChristophDemmer, Conversion script, Divisor, Duesentrieb, Flo12, FritzG, Gar kein name, Head, Hubertl, Hutschi, Jpp, Jörg Knappen, Karl-Henner, Keboerg, Koethnig, Korelstar, Krawi, Marc van Woerkom, Matthäus Wander, Mik01aj, MikeTheGuru, P. Birken, Peacemaker, Philipendula, Philipp Claßen, Pruefer, Rade Kutil, Riema, Rtc, Sascha Brück, Scr, Sechmet, Stern, TobiasEgg, Trustable, UKoch, WOBE3333, Wiki Gh!, Wuzel, Zeno Gantner, 38 anonyme Bearbeitungen

Code-Freeze *Quelle*: http://de.wikipedia.org/w/index.php?oldid=82521621 *Bearbeiter*: Avron, Bera, Fleasoft, Franz Halac, Joystick, Revolus, Sparti, Trustable, Twisted24de, 3 anonyme Bearbeitungen

Entwicklungsstadium (Software) *Quelle*: http://de.wikipedia.org/w/index.php?oldid=87694331 *Bearbeiter*: AMD-Hammer, Ahoerstemeier, Aka, Amtiss, Andreas 06, Armin P., Avron, BennyJ, Bernard Ladenthin, Carbidfischer, Chaddy, Chiccodoro, Chio, Chrissolon, ChristophDemmer, Daaavid, DanielSHaischt, Drahreg01, Eldred, Ephraim33, Evilboy, Exil, Filzstift, Fleasoft, Gamba, Gnu1742, Grille Chompa, Grindinger, Hasenläufer, HoKi, Iste Praetor, Jacobus21, Jan Giesen, Jello, Jesus Presley, KAMiKAZOW, KaterPeter, Kauan, Kku, Konrad F., Kungfuman, L3XLoGiC, LKD, Lafiestanoesparalosfeos, Lektor, Leo141, Libro, Lirum Larum, LivingShadow, Lofote, Lukaro, Lunochod, Luxo, Magnummandel, ManhattanGuy, Meinungsfreiheit, Meph666, Metalhead64, Micha83i, Micirio, Mthezeroth, Nachbarnebenan, Nivram, Norcas, Nutcracker, Nyks, OBrian, One-eyed pirate, PasO, Peter2, Peter200, PhilippWeissenbacher, Pittimann, Polarlys, Pot, PsY.cHo, Ri st, Richard Huber, Robb, Rohieb, Rolf acker, STBR, Saemikneu, Schattenspieler, Schmiddtchen, Sebi-Book, Sebs, Seewolf, Sparti, Speifensender, Spuk968, Staro1, Stefan Kühn, Suit, Sunbird, Svartskyggen, Taivo, The.Modificator, ThePeritus, TheReincarnator, Trainspotter, Treaki, Tuxman, Tönjes, Uwe Gille, V.R.S., Vitus, W!B:, Wiegels, WikiMax, WissensDürster, Zahnradzacken, Zeichenderzeit, Zinnmann, Zoid, 115 anonyme Bearbeitungen

FMEA *Quelle*: http://de.wikipedia.org/w/index.php?oldid=86516273 *Bearbeiter*: Advokat, AlexM, BJ Axel, Berklas, Brandm, ChristophDemmer, Colin Marquardt, Cyper, Dick Tracy, Dieter66007, Dirk Huber, Dodo von den Bergen, Echoray, Engie, Euphoriceyes, Eynre, Ghw, Goldmember1603, Gustavf, HAL Neuntausend, Hardenacke, Harz4, Herrick, HoHun, Hubi, HurwiczRocks, Ibseteq, Jel, Jergen, Jomai, JuTa, Kaneiderdaniel, Kepr59, LaurensvanLieshout, Lexinho, Ma-Lik, Markus Bärlocher, Misho, Nerd, Ocrho, Oefe, Oxsidian, P. Birken, Patrick Fritz, Rabebo, Rapste, Rax, Reneschmitz, Revvar, Romy2002, Schwalbe, Seewolf, Sinn, Smurftrooper, Sprenger, Stefan Birkner, SteveX, Suricata, TMg, Taeufer, Tapir2008, Thomas Willerich, Thomatronik, Tsor, Ums, Unsterblicher, Uwe Gille, Vren, WikipediaMaster, Wissen, Wolfgang H., Xelor, Yotwen, YourEyesOnly, ß, 222 anonyme Bearbeitungen

Gödelscher Unvollständigkeitssatz *Quelle*: http://de.wikipedia.org/w/index.php?oldid=85323467 *Bearbeiter*: 08-15, 0g1o2i3k4e5n6, AF666, ALOHApano, Aka, AlfonsGeser, Andre Engels, Andreas 06, Andreas S., Arbol01, Arved, Aschwar1, Bdk, Bejo, Ben-Zin, BjKa, Boemmels, Bunt, Calvin-gr, Chef, Chhanser, Christian1985, ChristophDemmer, DWay, Daniel5Ko, Darkking3, Digamma, Dishayloo, Duesentrieb, Dunkelschorsch, Egibiedermann, Enlil2, Ercas, Eulenspiegel1, Flibbo, Fomafix, Frakturfreund, Fristu, Fuzzy, Fxp, GluonBall, Godfatherofpolka, Golfinger, GottschallCh, Gubaer, Gunnar Eberlein, Gunther, Hagman, Hajo Keffer, Hannes Röst, Hans-Jürgen Streicher, Hansele, Hobbes16, Hokanomono, Hubi, Hydrauliker, Hydro, Irene1949, Ixitixel, JohnBrownsBaby, Jpp, Juesch, KaHe, Karl-Henner, Katharina, Kku, Koethnig, Kuebi, Kurt Jansson, Leechuck, Lightbearer, Lupussy, MGla, MKI, Magnus, Makor, Markus Mueller, MarkusRedeker, Martin-vogel, Mathuvw, Media lib, Mekeor, Mh26, Mikue, Millbart, Mkleine, Muffocks, Musicsciencer, Napfton, Necrophorus, Nyse, Odin, P. Birken, Pacogo7, Peter200, Pjacobi, Q. Wertz, Rainer Driesen, Reenpier, Rtc, Rufus46, Seb.froh, Siegwaldt, SirJective, Solid State, Sprachpfleger, StefanWesthoff, Stefanbs, Stern, Szs, Taxiarchos228, Template namespace initialisation script, Tenbergen, Tengai, Tischbeinahe, Tolentino, Torsten Grote, Unyxos, Usc, Uwe Gille, Uwe Lück, Weitwald, Wiegels, Woches, Wst, YMS, Zahnstein, Zeno Gantner, Zooloo, 98 anonyme Bearbeitungen

Hoare-Kalkül *Quelle*: http://de.wikipedia.org/w/index.php?oldid=86375242 *Bearbeiter*: (:Julien:), Blaufisch, Complex, Der Hakawati, Doodee, Enlil2, Expent, FelixD, Floklk, Flying sheep, Fomafix, Hajo Keffer, Hermel, Liberatus, MaBoehm, Mulno, P. Birken, Prochoma, Regnaron, Sebi, Sparti, Steevie, Uncopy, YMS, Zöblitz, 41 anonyme Bearbeitungen

Imperative Programmierung *Quelle*: http://de.wikipedia.org/w/index.php?oldid=84611331 *Bearbeiter*: Aka, Archiv, Arno Matthias, Beek100, Blah, Chrisfrenzel, Der Messer, DerHexer, Fcbaum, Frank Jacobsen, Frank Murmann, Freak 1.5, Frosty79, Gms, Head, JHöcker, Jon Kowal, Jpp, Klaeren, LKD, MH, Maerkl, Matthäus Wander, Nolispanmo, R.K.A.L., Sbeyer, Silophon, Skriptor, Slady, Suhadi Sadono, Tomen, Vwm, WikiNick, 32 anonyme Bearbeitungen

Keyword-Driven Testing *Quelle*: http://de.wikipedia.org/w/index.php?oldid=86693842 *Bearbeiter*: Abevauer, AchimR, AndreasPraefcke, Andycab, Avron, Badenserbub, Cholo Aleman, Fleasoft, Heinte, Jergen, Kinley, Knorxx, Löschfix, Markusbaum, Mnh, STBR, Sae1962, Sparti, Test-tools, WikiCare, 22 anonyme Bearbeitungen

Kontrollflussorientierte Testverfahren *Quelle*: http://de.wikipedia.org/w/index.php?oldid=86972167 *Bearbeiter*: AchimR, Aka, Avron, Bananenfalter, BesondereUmstaende, Bitsandbytes, Blah, CaZeRillo, Cspan64, D, Fake4d, Frank Murmann, Gelbphase, Harro von Wuff, Kaisersoft, Kaneiderdaniel, Kraymer, Krd, Lowar, Lx-s, Mikano, Mstolt, PaterMcFly, Rabus, Sebastian.Dietrich, Stauba, Theonlyandy, 40 anonyme Bearbeitungen

Korrektheit (Informatik) *Quelle*: http://de.wikipedia.org/w/index.php?oldid=81111938 *Bearbeiter*: A2r4e1, Aktionsheld, Blütenzauberer, Duesentrieb, Ephraim33, Fleasoft, Heinte, Liberal Freemason, Liberatus, Mh, Sulai, 5 anonyme Bearbeitungen

Prädikatenlogik *Quelle*: http://de.wikipedia.org/w/index.php?oldid=87397581 *Bearbeiter*: Alexander Sommer, AlfonsGeser, Andreas aus Hamburg in Berlin, Aso, Avoided, Baumfreund-FFM, Belucha, Ben-Zin, Broich, Bru, Ca$e, ChristianErtl, Christianju, ChristophDemmer, CommonsDelinker, Complex, Conversion script, Crux, Dafstyle, Dominik, Entlinkt, ErikDunsing, FerdiBf, Fgb, Fils de la Lumière, Gismatis, GottschallCh, Hafenbar, Hajo Keffer, Hannes Hirzel, Hans-Jürgen Streicher, Hansjörg, Head, Hydro, Ifrost, Ingoschi, JakobVoss, Joey-das-WBF, Johannes Simon, JuTa, Kaisersoft, Kam Solusar, Karl-Henner, Katharina, Kixx, Kku, Lawa, Lipedia, Logik, M. Augustine, Markus Prokott, Martin-vogel, Martinhei, Media lib, Michaelsy, Mindripper, Mrknowitall, Mussklprozz, Nerd, Nevio, Nothere, Otto ter Haar, P. Birken, Pacogo7, Peter Steinberg, Peter200, Phuesicus, Pjacobi, Pjoern, Rentar, RokerHRO, Schewek, Shannon, SirJective, Sparti, Terabyte, Tkarcher, Trooper, TruebadiX, U.jes, Ulrich.fuchs, VerwaisterArtikel, Warumauchimmer, Wasseralm, Weialawaga, Wilfried Neumaier, Xqt, Zeno Gantner, bw2-145pub166.bluewin.ch, 91 anonyme Bearbeitungen

Softwaremetrik *Quelle*: http://de.wikipedia.org/w/index.php?oldid=87071490 *Bearbeiter*: Avron, BenediktM, Berlinschneid, Boemmels, Capaci34, ChristianHujer, Dathomas, Elwood j blues, Erkan Yilmaz, Frank Jacobsen, Froggy, Gaius L., Hans-Jörg Günther, JeeAge, Jpp, Levin, Mboehmer, Mstolt, Niemeyerstein, Olei, Patmuk, PeterVitt, QualiStattQuanti, Rene Mas, RobertAUlm, Sebastian.Dietrich, Solid State, Sparti, Surrounder, Tamaraz, Test-tools, Uncopy, Uwe Hermann, W!B:, Wiegels, 50 anonyme Bearbeitungen

TPT (Software) *Quelle*: http://de.wikipedia.org/w/index.php?oldid=86227648 *Bearbeiter*: Andreas aus Hamburg in Berlin, Avron, Hydro, Jens Lüdemann, Jluedem, Jpp, KommX, Leider, Ma-Lik, Micha.schmiddi, Trublu, YourEyesOnly, 36 anonyme Bearbeitungen

Validierung (Informatik) *Quelle*: http://de.wikipedia.org/w/index.php?oldid=81311306 *Bearbeiter*: Andreas Reif, Anneke Wolf, Avron, C.Löser, DaB., Duesentrieb, Frank Zimmermann, Hafenbar, JakobVoss, Jojoe, Jpp, Kku, Reseka, Stefan2, Thomas Willerich, Trustable, UlrichAAB, Video2005, Wikinator, WolfgangRieger, Zoelomat, 27 anonyme Bearbeitungen

Verifizierung *Quelle*: http://de.wikipedia.org/w/index.php?oldid=88162894 *Bearbeiter*: 20357Hamburg, Andrsvoss, Anneke Wolf, Avron, Balaklava, Baumfreund-FFM, Beek100, Boonekamp, Buecherfresser, Ca$e, Cami de Son Duc, Chrisfrenzel, Church of emacs, DasFliewatüüt, Der.Traeumer, Duesentrieb, Dullnraamer, Ekuah, Emkaer, Empro2, Frank Reinhart, Frank Zimmermann, HHK, HRoestTypo, HaSee, Hadhuey, Hanno Sandvik, Hans J. Castorp, Hjpospie, Jens611, Johannes von Salem und Seborga, Jpp, Kai-Hendrik, Kaneiderdaniel, Karl-Henner, Keigauna, Koethnig, Krawi, Krd, L.M. Morgenroth, Leshonai, Ma-Lik, Marc van Woerkom, Martin-vogel, Michaelsy, Mikullovci11, Mlesniak, Mo4jolo, Nerd, Nockel12, Obersachse, Ojm, P UdK, Paddy, Pinguin.tk, Pittimann, Pruefer, Rangaro, Regi51, Rtc, Sava, Sinuspi, Sokonbud, SonniWP2, Sparti, Spuk968, StG1990, Stefan Kühn, Stern, Studbeefpile, Tsor, Ulrich.fuchs, WAH, Wiska Bodo, Ziko, 94 anonyme Bearbeitungen

Versionsnummer *Quelle*: http://de.wikipedia.org/w/index.php?oldid=87442160 *Bearbeiter*: Aka, Akrause91, Avron, Berntie, Blutfink, Christian Sakowski, DStulle, Deichgraf2010, Dodo von den Bergen, Eke, El Grafo, Elexpress, Engie, Fleasoft, Forestsoft, Frakturfreund, Fuenfundachtzig, G-41614, Gepardenforellenfischer, Gregor Müllegger, HaSee, Jensw, Jergen, Jimini, Laza, Lostintranslation, ManhattanGuy, Mihawk90, MrBurns, Octotron, Peter Wiegel, PhilippWeissenbacher, Pot, Reinhard Kraasch, Revolus, Rohieb, RokerHRO, RolandIllig, Rolf acker, Saehrimnir, Schleinkofer, Sereghir, Sparti, Srbauer, Svartskyggen, Thornard, Trustable, Uncopy, VanGore, W!B:, Xell09, YMS, 26 anonyme Bearbeitungen

Programmierschnittstelle *Quelle*: http://de.wikipedia.org/w/index.php?oldid=87846738 *Bearbeiter*: --, Andre Engels, Basti1302, Benedikt, Blakeks, D, Deki, Eilexe, ErnstA, Felbre, FelixReimann, Fomafix, Fuzz, Heinrich5991, Invisigoth67, JakobVoss, Jan Giesen, Joubsti, Karl-Henner, Klabube, Krawi, Kurt Jansson, Lantash, Lehmi, Lekthor, Len'sche, Lichtkind, MarkusHagenlocher, MarkusKimmerle, Maxb88, Me Snickers, Mps, Norro, OnlineT, Ot, ReneRomann, Rosenzweig, Rowland, S.K., Sae1962, Sascha Hameister, Sascha hameister, Sav, Schandi, Schewek, Schweika, Seth Cohen, Shmia, Sparti, Srbauer, StefanSchlegel, Thornard, Tordans, Tzeh, Urizen, Uwe Gille, V.R.S., VanGore, Wittlaer, Wizzar, YourEyesOnly, Yurik, YvonneM, Zeno Gantner, 89 anonyme Bearbeitungen

Versionsverwaltung *Quelle*: http://de.wikipedia.org/w/index.php?oldid=87156112 *Bearbeiter*: 20percent, ArneBab, Arved, Ashberk, Astrapi, Ath, Avron, Capaci34, ChristianHujer, Clagi, Codeispoetry, Crux, Curtis Newton, DWay, Darkmoon2008, DerAnalyst, Dishayloo, Dr. Henning, Eke, Entlinkt, ErikDunsing, Farino, Fleasoft, Haeber, Hannes Röst, Hb, He3nry, Head, HenrikHolke, Hydro, IGEL, Igrimm12, JARU, JakobVoss, Jed, JensBaitinger, Jnn95, JoBa2282, Joise, Jtxa, Kasper4711, Kerbel, Kockster, Kuli, Langec, Laza, Linuxer, LosHawlos, MarZilein, MarkusKnittig, Media lib, Mfb, Mfx, Mijobe, Minderbinder, Mm pedia, Mps, Mshobohm, Nikai, Oltbaba, Ot, PaterMcFly, Pietz, Pilawa, Polluks, Qwerty84, Rakorn, Raubsaurier, Regression Tester, Reni Tenz, S.K., Schlurcher, Sebastian.Dietrich, Spuk968, Staro2, Stoerte, Stuertz, Tabacha, Thomas Willerich, Thornard, TobiasHerp, Trustable, Tsor, UlrichAAB, Umherirrender, Uncopy, Ungebeten, W!B:, Wikinger08, Wissling, YMS, 71 anonyme Bearbeitungen

wp-Kalkül *Quelle*: http://de.wikipedia.org/w/index.php?oldid=83768029 *Bearbeiter*: Abubiju, Aka, Andy king50, Blütenzauberer, Church of emacs, Doodee, Enlil2, Hajo Keffer, Liberatus, Oxydo, Revolus, Sparti, Stefan Birkner, Stettberger, Tobias Bergemann, WhiteCrow, YMS, 5 anonyme Bearbeitungen

FURPS *Quelle*: http://de.wikipedia.org/w/index.php?oldid=79897948 *Bearbeiter*: Deorma, Der Michl, Emes, Kku, Staro1, Tschäfer, Wst, 6 anonyme Bearbeitungen

Schnittstellentest *Quelle*: http://de.wikipedia.org/w/index.php?oldid=54957930 *Bearbeiter*: AchimR, Biezl, Dodo von den Bergen, Eleazar, MichaelFrey, Trg, 2 anonyme Bearbeitungen

Lasttest (Computer) *Quelle*: http://de.wikipedia.org/w/index.php?oldid=87604795 *Bearbeiter*: Abubiju, AchimR, Andizo, Armadillo-eleven, Avron, Bernedom, Birger Fricke, Cecil, Cjesch, Dressen, Fscheide, Funkruf, Herrick, Hlambert63, Jaellee, Jpp, Kallistratos, Laza, Mareike jacobshagen, Martin.Bemmann, Micha83i, Michelangelo, Saibo, Semper, Sentie77, SonniWP2, Speck-Made, Spence, Stefan Birkner, Test-tools, Thire, Thomro, Tkoeppner, Trinar, Tromla, YMS, 64 anonyme Bearbeitungen

Sicherheitstest (Software) *Quelle*: http://de.wikipedia.org/w/index.php?oldid=87730067 *Bearbeiter*: Abubiju, Appsec, Avron, Bautsch, Borne, Gustavf, Jbo166, Purpleshoe, Rohieb, Rolf acker, SVL, SecurityPatterns, Silberchen, Sparti, Woches, 21 anonyme Bearbeitungen

Äquivalenzklassentest *Quelle*: http://de.wikipedia.org/w/index.php?oldid=77687687 *Bearbeiter*: Fouk, Frank1101, Jerry4100, S.K., Voyager, Wolf32at, 12 anonyme Bearbeitungen

Smoke testing *Quelle*: http://de.wikipedia.org/w/index.php?oldid=84480419 *Bearbeiter*: Collix, Don Magnifico, Jergen, Joey-das-WBF, Lustiger seth, Manecke, Media lib, Ordnung, Pelz, Perot, Revolus, Sukarnobhumibol, 15 anonyme Bearbeitungen

Testgetriebene Entwicklung *Quelle*: http://de.wikipedia.org/w/index.php?oldid=87268931 *Bearbeiter*: AchimR, AndreKR, Archiv, Badenserbub, ChristianHujer, Codeispoetry, Complex, DrHok, Exil, Fleasoft, FrankDopatka, Gorgo, Harfenmusik, Harro von Wuff, Heiko, HerbstSchnee, Inkowik, Jpp, Klausikm, Ma-Lik, Mbroekelmann, MichaelDiederich, Niemeyerstein, Revvar, RobertSonnberger, Sae1962, Sebastian.Dietrich, Silvicola, Sleske, Snooper77, Sparti, Stauba, Stefan Schultz, SvenjaWendler, TMg, Test-tools, Th1979, Till.niermann, Uncopy, Waldgeist, Xasx, YMS, 68 anonyme Bearbeitungen

Regressionstest *Quelle*: http://de.wikipedia.org/w/index.php?oldid=77204297 *Bearbeiter*: 83nj1, Abubiju, Androl, Arved, Avron, Comc, Eehmke, ElRaki, Gadelor, Geisslr, Iiiren, Jpp, Kaiblankenhorn, Karl.Reichert, Lyzzy, Martin Hampl, Media lib, QFS, Schubbay, Slamdank, Srittau, Superbass, Taschenrechner, Vhalstenbach, Zinnmann, 31 anonyme Bearbeitungen

Review (Softwaretest) *Quelle*: http://de.wikipedia.org/w/index.php?oldid=84280011 *Bearbeiter*: Andy king50, Avron, Cilugnedon, Drahreg01, Echtner, Empro2, Erkan Yilmaz, For3st, Gbeckmann, Gerold Broser, Gunggel, Hansruedi.Tremp, Horcrux7, Howwi, J.Ammon, Kku, Myfotoreviewde, Ohrnee, Peter Rösler, RainerB., Randonneur, Rolf acker, STDA, Secular mind, TableSitter, Timekeeper, VÖRBY, WortUmBruch, Århus, 15 anonyme Bearbeitungen

Statische Code-Analyse *Quelle*: http://de.wikipedia.org/w/index.php?oldid=87052020 *Bearbeiter*: Aka, Avron, Cactus26, Captaingrog, Complex, Creando, Eike sauer, Gaius L., Geisslr, Hans Koberger, Herr Th., Hieke, Jens611, Koethnig, Leo141, Liberatus, MarkusHagenlocher, Mef.ellingen, René Schwarz, S.K., Steevie, Tönjes, Uncopy, Wiki4you, 41 anonyme Bearbeitungen

Software Quality Assurance Plan *Quelle*: http://de.wikipedia.org/w/index.php?oldid=87742599 *Bearbeiter*: Aka, Esteiger, Kku, Rdb, Rolf acker, Widewitt, Wiegels, 3 anonyme Bearbeitungen

Software Configuration Management Plan *Quelle*: http://de.wikipedia.org/w/index.php?oldid=69191808 *Bearbeiter*: Appelbuur, Asdert, Esteiger, Hseebauer, JCS, 3 anonyme Bearbeitungen

Software Test Documentation *Quelle*: http://de.wikipedia.org/w/index.php?oldid=79880291 *Bearbeiter*: Anschroewp, Asdert, Avron, Chrisfrenzel, Darok, Edward Montgomery Harrington, Ellyce, Esteiger, ManWing2, Michaelt1964, Mussklprozz, Tröte, 19 anonyme Bearbeitungen

Software Requirements Specification *Quelle*: http://de.wikipedia.org/w/index.php?oldid=86366432 *Bearbeiter*: AN, Abdull, Abubiju, Aka, Arzach, Avron, ChristophDemmer, Der Hakawati, Esteiger, Fleasoft, Ghw, Hans-Jörg Günther, Heinte, Krawi, Michaki, Momesana, Patchworker, Popie, S.K., Soloturn, Sparti, Sterling, WIKImaniac, Wittkowsky, 42 anonyme Bearbeitungen

Software Validation & Verification Plan *Quelle*: http://de.wikipedia.org/w/index.php?oldid=70253623 *Bearbeiter*: Esteiger, Rouven Thimm, Schwallex, Woches

Software Design Description *Quelle*: http://de.wikipedia.org/w/index.php?oldid=65088908 *Bearbeiter*: Cafezinho, Esteiger, He3nry, Hildegund, Ma-Lik, Michaki, S.K., Soloturn, Urs.Waefler, 9 anonyme Bearbeitungen

Software Project Management Plan *Quelle*: http://de.wikipedia.org/w/index.php?oldid=41242266 *Bearbeiter*: Edward Montgomery Harrington

AUTOSAR *Quelle*: http://de.wikipedia.org/w/index.php?oldid=87725520 *Bearbeiter*: Aceop, Aka, Aoide, BesondereUmstaende, Biezl, Cepheiden, Chrispgu, Cove, Ergin66, Florian Adler, Gungnir70, Helmut Zenz, Henning Blatt, Hydro, Jschlosser, Jürgen-Crepin, KGF, Kungfuman, Langermannia, Lax, Manuel.Beck, Millbart, Mischl7, Nameless23, Pessottino, Pittimann, Prometheus27, Rare71, ReqEngineer, RobertDaxxtor, Rudolph Buch, STBR, StephanReichelt, Tilla, Tobias Ambrosi, Toni nietnagel, Tsor, Unenzyklopädisch, WolfgangRieger, YMS, 59 anonyme Bearbeitungen

Quelle(n), Lizenz(en) und Autor(en) des Bildes

Datei:V-Modell.svg *Quelle*: http://de.wikipedia.org/w/index.php?title=Datei:V-Modell.svg *Lizenz*: unbekannt *Bearbeiter*: User:EinStein

Datei:Test Phasenmodell.png *Quelle*: http://de.wikipedia.org/w/index.php?title=Datei:Test_Phasenmodell.png *Lizenz*: unbekannt *Bearbeiter*: VÖRBY. Original uploader was VÖRBY at de.wikipedia

Datei:Test ganzheitlich.png *Quelle*: http://de.wikipedia.org/w/index.php?title=Datei:Test_ganzheitlich.png *Lizenz*: unbekannt *Bearbeiter*: VÖRBY

Datei:IEEE829_Uebersicht_Deutsch.png *Quelle*: http://de.wikipedia.org/w/index.php?title=Datei:IEEE829_Uebersicht_Deutsch.png *Lizenz*: Creative Commons Attribution-Sharealike 3.0 *Bearbeiter*: User:Mussklprozz

Datei:Testen BegriffeZusHang.png *Quelle*: http://de.wikipedia.org/w/index.php?title=Datei:Testen_BegriffeZusHang.png *Lizenz*: unbekannt *Bearbeiter*: VÖRBY. Original uploader was VÖRBY at de.wikipedia

Datei:Testen SchnittstellenThaller.png *Quelle*: http://de.wikipedia.org/w/index.php?title=Datei:Testen_SchnittstellenThaller.png *Lizenz*: unbekannt *Bearbeiter*: VÖRBY. Original uploader was VÖRBY at de.wikipedia

Datei:Testsystem-Architektur.png *Quelle*: http://de.wikipedia.org/w/index.php?title=Datei:Testsystem-Architektur.png *Lizenz*: unbekannt *Bearbeiter*: Benutzer:Korotkiy

Datei:H96566k.jpg *Quelle*: http://de.wikipedia.org/w/index.php?title=Datei:H96566k.jpg *Lizenz*: Public Domain *Bearbeiter*: Courtesy of the Naval Surface Warfare Center, Dahlgren, VA., 1988.

Bild:Ausnahmesituation.png *Quelle*: http://de.wikipedia.org/w/index.php?title=Datei:Ausnahmesituation.png *Lizenz*: GNU Free Documentation License *Bearbeiter*: Mkleine

File:Software dev3 DE.png *Quelle*: http://de.wikipedia.org/w/index.php?title=Datei:Software_dev3_DE.png *Lizenz*: GNU Free Documentation License *Bearbeiter*: FleetCommand, Jacobus21

Datei:Beta-badge.svg *Quelle*: http://de.wikipedia.org/w/index.php?title=Datei:Beta-badge.svg *Lizenz*: GNU Free Documentation License *Bearbeiter*: Benutzer:Connum Original uploader was Connum at de.wikipedia

Bild:Rule-of-ten.png *Quelle*: http://de.wikipedia.org/w/index.php?title=Datei:Rule-of-ten.png *Lizenz*: unbekannt *Bearbeiter*: Benutzer:Reneschmitz

Bild:C0Ueb.png *Quelle*: http://de.wikipedia.org/w/index.php?title=Datei:C0Ueb.png *Lizenz*: unbekannt *Bearbeiter*: Avron, Chaddy, Flominator, Forrester, Jodo, Maggie85

Bild:C1Ueb.png *Quelle*: http://de.wikipedia.org/w/index.php?title=Datei:C1Ueb.png *Lizenz*: unbekannt *Bearbeiter*: Avron, Daniel 1992, Flominator, Forrester, Jodo, Maggie85

Bild:C23Ueb.png *Quelle*: http://de.wikipedia.org/w/index.php?title=Datei:C23Ueb.png *Lizenz*: unbekannt *Bearbeiter*: Avron, Daniel 1992, Maggie85, Mdangers

File:Predicate_logic;_2_variables;_12diag_f.svg *Quelle*: http://de.wikipedia.org/w/index.php?title=Datei:Predicate_logic;_2_variables;_12diag_f.svg *Lizenz*: Public Domain *Bearbeiter*: User:Lipedia

File:Predicate_logic;_2_variables;_12diag_ne.svg *Quelle*: http://de.wikipedia.org/w/index.php?title=Datei:Predicate_logic;_2_variables;_12diag_ne.svg *Lizenz*: Public Domain *Bearbeiter*: User:Lipedia

File:Predicate_logic;_2_variables;_12f.svg *Quelle*: http://de.wikipedia.org/w/index.php?title=Datei:Predicate_logic;_2_variables;_12f.svg *Lizenz*: Public Domain *Bearbeiter*: User:Lipedia

File:Predicate_logic;_2_variables;_A2ne.svg *Quelle*: http://de.wikipedia.org/w/index.php?title=Datei:Predicate_logic;_2_variables;_A2ne.svg *Lizenz*: Public Domain *Bearbeiter*: User:Lipedia

File:Predicate_logic;_2_variables;_A1ne.svg *Quelle*: http://de.wikipedia.org/w/index.php?title=Datei:Predicate_logic;_2_variables;_A1ne.svg *Lizenz*: Public Domain *Bearbeiter*: User:Lipedia

File:Predicate_logic;_2_variables;_E2f.svg *Quelle*: http://de.wikipedia.org/w/index.php?title=Datei:Predicate_logic;_2_variables;_E2f.svg *Lizenz*: Public Domain *Bearbeiter*: User:Lipedia

File:Predicate_logic;_2_variables;_E1f.svg *Quelle*: http://de.wikipedia.org/w/index.php?title=Datei:Predicate_logic;_2_variables;_E1f.svg *Lizenz*: Public Domain *Bearbeiter*: User:Lipedia

File:Predicate_logic;_2_variables;_12ne.svg *Quelle*: http://de.wikipedia.org/w/index.php?title=Datei:Predicate_logic;_2_variables;_12ne.svg *Lizenz*: Public Domain *Bearbeiter*: User:Lipedia

File:Predicate_logic;_2_variables;_implications_(deutsch).svg *Quelle*: http://de.wikipedia.org/w/index.php?title=Datei:Predicate_logic;_2_variables;_implications_(deutsch).svg *Lizenz*: Public Domain *Bearbeiter*: User:Lipedia

Datei:TPT_Logo.png *Quelle*: http://de.wikipedia.org/w/index.php?title=Datei:TPT_Logo.png *Lizenz*: unbekannt *Bearbeiter*: User:Jluedem

Datei:TPT_Screenshot.png *Quelle*: http://de.wikipedia.org/w/index.php?title=Datei:TPT_Screenshot.png *Lizenz*: Creative Commons Attribution 3.0 *Bearbeiter*: User:Jluedem

Bild:IEEE829_Uebersicht_Deutsch.png *Quelle*: http://de.wikipedia.org/w/index.php?title=Datei:IEEE829_Uebersicht_Deutsch.png *Lizenz*: Creative Commons Attribution-Sharealike 3.0 *Bearbeiter*: User:Mussklprozz

Datei:Autosar Logo.svg *Quelle*: http://de.wikipedia.org/w/index.php?title=Datei:Autosar_Logo.svg *Lizenz*: unbekannt *Bearbeiter*: Lukas9950, Sternenfaenger77

Lizenz

Wichtiger Hinweis zu den Lizenzen

Die nachfolgenden Lizenzen bezieht sich auf den Artikeltext. Im Artikel gezeigte Bilder und Grafiken können unter einer anderen Lizenz stehen sowie von Autoren erstellt worden sein, die nicht in der Autorenliste erscheinen. Durch eine noch vorhandene technische Einschränkung werden die Lizenzinformationen für Bilder und Grafiken daher nicht angezeigt. An der Behebung dieser Einschränkung wird gearbeitet.
Das PDF ist daher nur für den privaten Gebrauch bestimmt. Eine Weiterverbreitung kann eine Urheberrechtsverletzung bedeuten.

Creative Commons Attribution-ShareAlike 3.0 Unported - Deed

Diese "Commons Deed" ist lediglich eine vereinfachte Zusammenfassung des rechtsverbindlichen Lizenzvertrages (http://de.wikipedia.org/wiki/Wikipedia:Lizenzbestimmungen_Commons_Attribution-ShareAlike_3.0_Unported) in allgemeinverständlicher Sprache.
Sie dürfen:

- das Werk bzw. den Inhalt **vervielfältigen, verbreiten und öffentlich zugänglich machen**
- **Abwandlungen und Bearbeitungen** des Werkes bzw. Inhaltes anfertigen

Zu den folgenden Bedingungen:

- **Namensnennung —** Sie müssen den Namen des Autors/Rechteinhabers in der von ihm festgelegten Weise nennen.
- **Weitergabe unter gleichen Bedingungen —** Wenn Sie das lizenzierte Werk bzw. den lizenzierten Inhalt bearbeiten, abwandeln oder in anderer Weise erkennbar als Grundlage für eigenes Schaffen verwenden, dürfen Sie die daraufhin neu entstandenen Werke bzw. Inhalte nur unter Verwendung von Lizenzbedingungen weitergeben, die mit denen dieses Lizenzvertrages identisch, vergleichbar oder kompatibel sind.

Wobei gilt:

- **Verzichtserklärung —** Jede der vorgenannten Bedingungen kann aufgehoben werden, sofern Sie die ausdrückliche Einwilligung des Rechteinhabers dazu erhalten.
- **Sonstige Rechte —** Die Lizenz hat keinerlei Einfluss auf die folgenden Rechte:
  - Die gesetzlichen Schranken des Urheberrechts und sonstigen Befugnisse zur privaten Nutzung;
  - Das Urheberpersönlichkeitsrecht des Rechteinhabers;
  - Rechte anderer Personen, entweder am Lizenzgegenstand selber oder bezüglich seiner Verwendung, zum Beispiel Persönlichkeitsrechte abgebildeter Personen.
- **Hinweis —** Im Falle einer Verbreitung müssen Sie anderen alle Lizenzbedingungen mitteilen, die für dieses Werk gelten. Am einfachsten ist es, an entsprechender Stelle einen Link auf http://creativecommons.org/licenses/by-sa/3.0/deed.de einzubinden.

Haftungsbeschränkung
Die „Commons Deed" ist kein Lizenzvertrag. Sie ist lediglich ein Referenztext, der den zugrundeliegenden Lizenzvertrag übersichtlich und in allgemeinverständlicher Sprache, aber auch stark vereinfacht wiedergibt. Die Deed selbst entfaltet keine juristische Wirkung und erscheint im eigentlichen Lizenzvertrag nicht.

GNU Free Documentation License

Version 1.2, November 2002

Copyright (C) 2000,2001,2002 Free Software Foundation, Inc.

51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA

Everyone is permitted to copy and distribute verbatim copies

of this license document, but changing it is not allowed.

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.
This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.
We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.
A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.
A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.
The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.
The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.
A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".
Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.
The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.
A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.
The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.
You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.
If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.
If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.
It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- **A.** Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
- **B.** List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
- **C.** State on the Title page the name of the publisher of the Modified Version, as the publisher.
- **D.** Preserve all the copyright notices of the Document.
- **E.** Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- **F.** Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- **G.** Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- **H.** Include an unaltered copy of this License.
- **I.** Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
- **J.** Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
- **K.** For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
- **L.** Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
- **M.** Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.
- **N.** Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.
- **O.** Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.
You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.
You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.
The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.
The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.
You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.
If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.
If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.
Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright (c) YEAR YOUR NAME.
>
> Permission is granted to copy, distribute and/or modify this document
>
> under the terms of the GNU Free Documentation License, Version 1.2
>
> or any later version published by the Free Software Foundation;
>
> with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts.
>
> A copy of the license is included in the section entitled
>
> "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with...Texts." line with this:

> with the Invariant Sections being LIST THEIR TITLES, with the
>
> Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.
If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.